IA169 System Verification and Assurance

Course Intro & Fundaments of Testing

Jiří Barnat

Course Organization

Course Coverage

Topics to be covered ...

- Introduction to Testing
- Symbolic Execution
- Model Checking (4 lectures)
- Deductive Verification
- Bounded Model Checking
- Verification of Real-Time and Hybrid Systems
- Verification of Probabilistic Systems
- CEGAR and Abstract Interpretation
- Assurance, Threat Models, Relevant Security Standards

Prerequisites and Follow-Up

Prerequisites

- Formally none, but we expect ...
- ... capability of basic math reasoning and abstractions.
- ... some experience with coding.
- ... you can handle Unix as a user.

Mutual Exclusion with

- IV113 and IV101

Possible Follow-Up

- IA159 Formal Verification Methods

Course Structure and Marking

Structure

- 2/2/2 + 2 ECTS credits
- Lecture/Tutorials/Homework

Marking

- Final exam 70% (written test)
- Assignments 30% (five evaluated tasks)
- 50% for E or Colloquy or Credit
- 60% for D
- 70% for C
- 80% for B
- 90% for A

Seminar schedule

- `https://is.muni.cz/auth/el/1433/jaro2018/IA169/op/IA169_2018_semestr_schedule.html`

Fundaments of Testing

http://www.testingeducation.org/BBST/

Testing

Testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.

Empirical Technical

- Conduct experimental measurements.
- Logic and math.
- Modelling.
- Employs SW tools.

Investigation

- Organised and thorough.
- Self-reflecting.
- Challenging.

Testing

Product or Service

- Software.
- Hardware.
- Data.
- Documentation and specification.
- ... other parts that are delivered.

Information

- Not know before.
- Has some value.

Stakeholders

- Who has interest in the success of testing effort.
- Who has interest in the success of the product.

Fundamental Questions of Testing

Mission

- Why do we test? What we want to achieve?

Strategy

- How to proceed to fulfil the mission efficiently?

Oracle

- How to recognise success of the test.

Incompletness

- Do we realise that testing cannot prove absence of error?

Measure

- How much of of our testing plan has been completed?
- How far we are to complete the mission?

Mission of Testing

Most Typical Mission

- Bug hunting.
- Identification of factors that reduce quality.

Other Missions

- Collect data to support manager decisions, such as: Is the product good enough to be released?
- How much different is the product from product available on market?
- Is the product complete with respect to specification?
- Are individual components logically and ergonomically connected.
- ...

Other Missions

Other Missions – continued

- Support manager decision with empirical results.
- Evaluate the cost of support after release.
- To check compatibility with other products.
- Confirm accordance with the specification.
- Find safe scenarios of product usage.
- To acquire certification.
- Minimise consequences of low quality.
- Evaluate the product for third party.
- ...

Strategy

Strategy

Strategy is a plan, how to fulfil the mission in the given context.

Example: Consider spreadsheet computation in four different contexts.

a) Computer game.

b) Early stage of development of database product.

c) Late stage of development of database product.

d) Driver for medical X-ray scanning device.

Question:

- Will you proceed with the same strategy?

Example – continued

What factors influence strategy selection

- What is the mission?
- How aggressively we need to detect bugs.
- What bugs are less important than others?
- How thoroughly testing will be documented?

Discussion

- Assume that a program has an enter field that is expecting numerical values. Is is meaningful to test the product for situation when we enter non-numeric value? (Not mentioned in specification at all.)

Oracle

Definition of Oracle

Oracle (in the context of testing) is a detection mechanism or principle to learn that the product passed or failed a test.

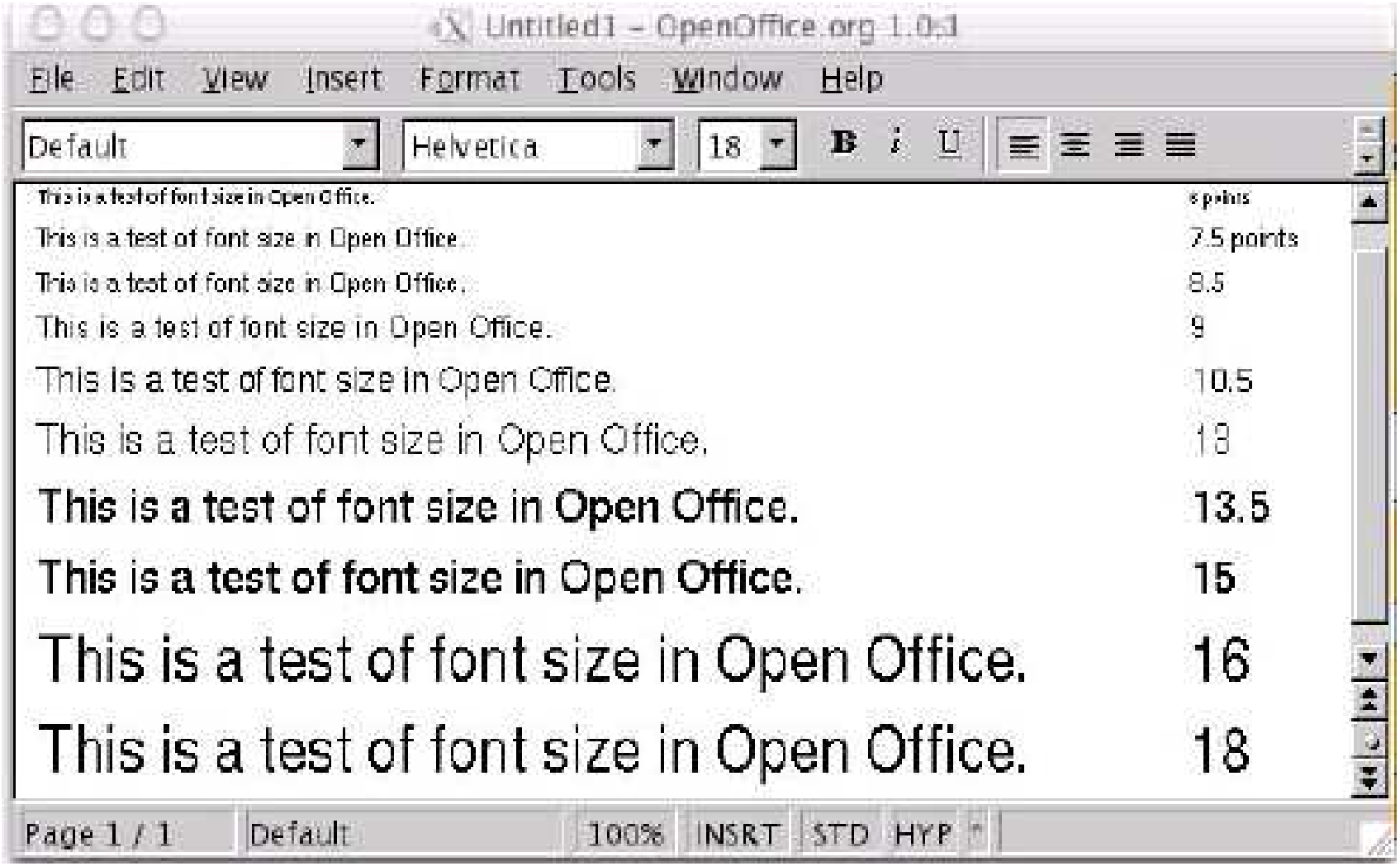
Facts

- If tester claims that the program passed a test, it does not mean the program is correct with respect to the tested property. It depends on the oracle used.
- Basically, any test may fail or pass with a suitable oracle.

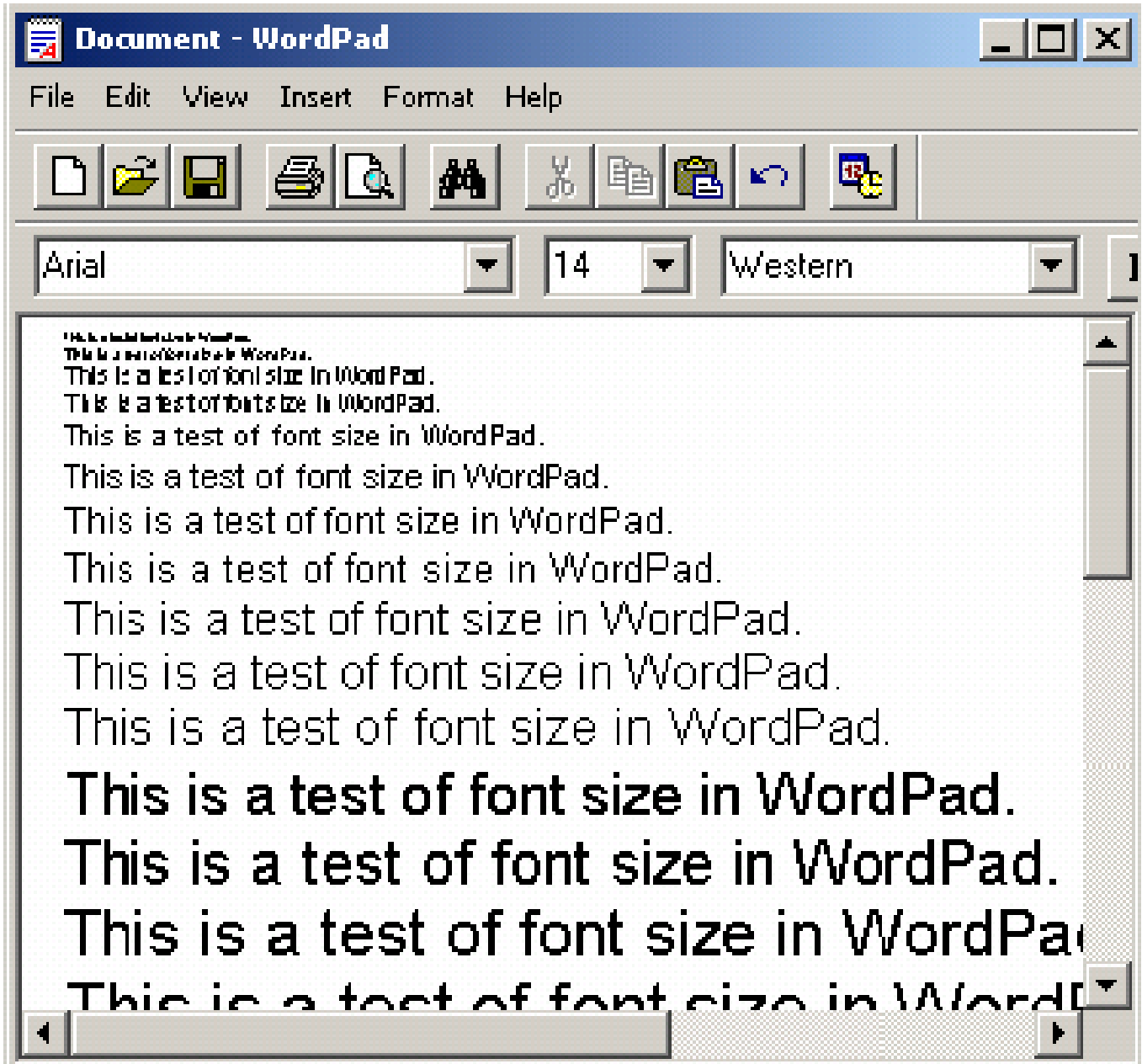
Example

- Does font sizes work properly in OpenOffice, WordPad, and MS Word text editors?

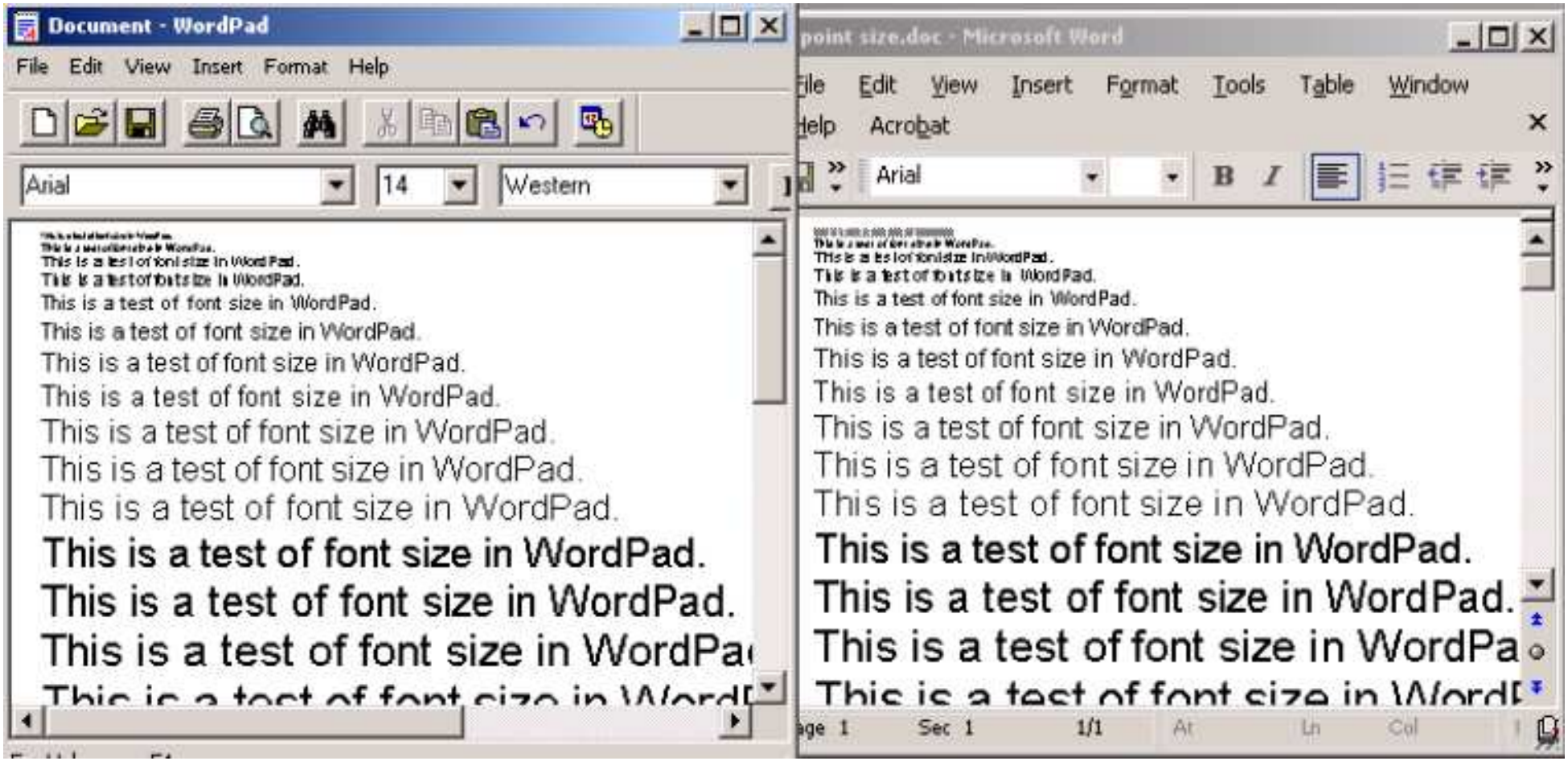
Example – OpenOffice 1.0



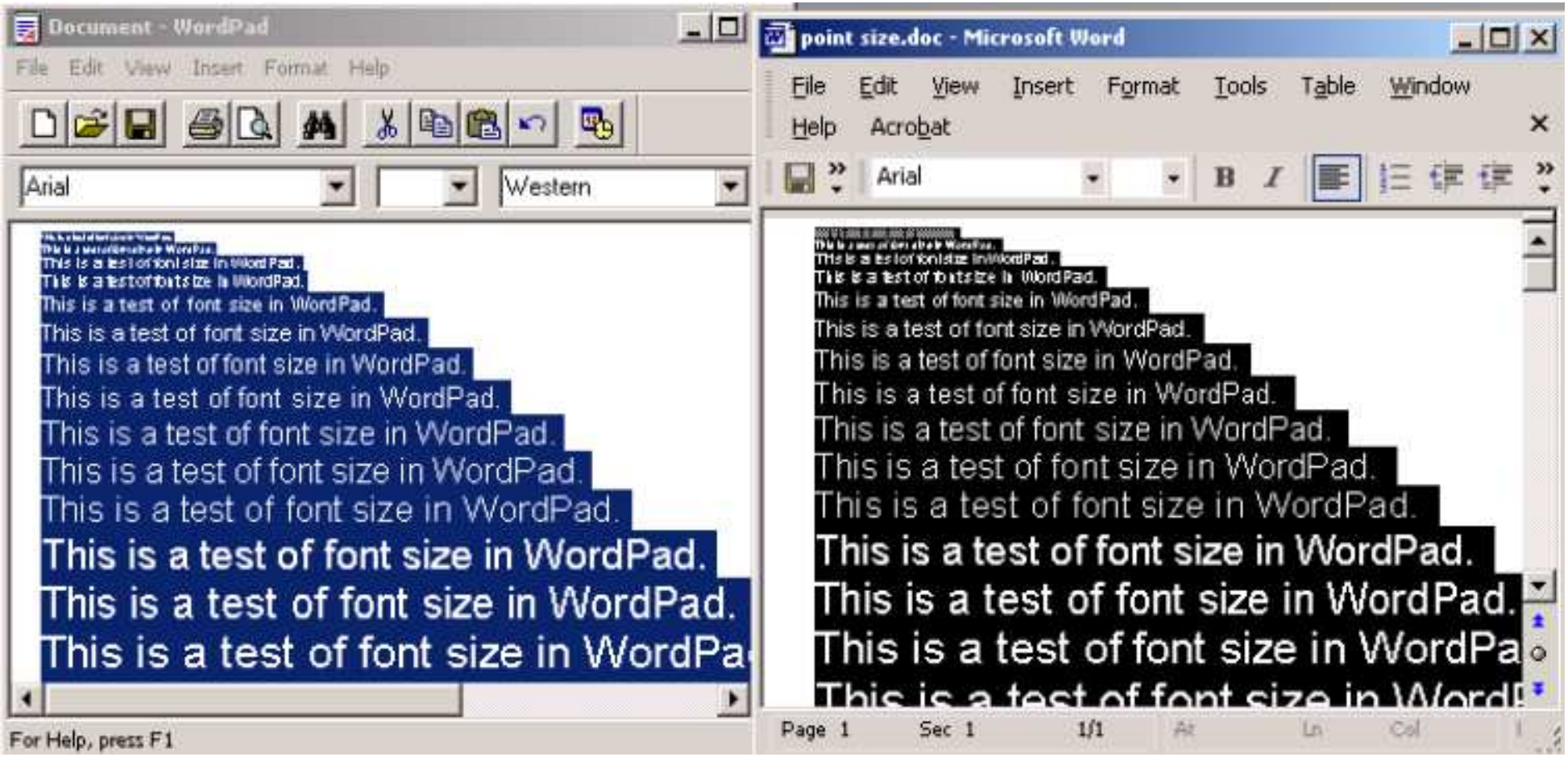
Example – WordPad



Example – WordPad versus MS Word



Example – WordPad versus MS Word (highlighted)



Example – Decisions

Questions

- Is the observed difference in font sizes a bug in WordPad?
- Is the observed difference in font sizes a bug in MS Word?
- Is the observed difference in font sizes a bug at all?

Possible Conclusions

- We do not know if sizes are correct, but we have tendency to believe MS Word rather than to WordPad.
- For WordPad it is not necessary to stick precisely to typographic standards.
- For WordPad it is possibly a bug, but definitely it is not a problem.

Example – Risk-Based Testing

Possible (Pragmatic) Position

- It is/isn't a bug? $\Longrightarrow$ It is/isn't a problem?
- It is necessary to know the context, to guess the metrics that the final consumer will use to judge the issue.
- With some risk we can achieve simplification of the decision.

Simplification in Testing Process

- Avoid tests that obviously does not reveal any problems.
- Avoid tests that obviously reveal only uninteresting problems.

Example – Judge Criteria

How much do we actually know about typography?

- Point definition is unclear.
  (`http://www.oberonplace.com/dtp/fonts/point.htm`)
- Absolute sizes are difficult to measure.
  (`http://www.oberonplace.com/dtp/fonts/fontsize.htm`)

From Uncertainty to Heuristics

- How precisely must the sizes agree in order to declare that the sizes are correct?
- Obtaining complete information and evaluation of all the facts is too complicated and costly.
- **Heuristics** are used instead.

Oracle Problem – Heuristics

Decision Heuristics

- Allows for simplification of decision problem.
- Advice, recommendation, or procedure to be used within the given context.
- Should not build on any hidden knowledge.
- Does not guarantee a good decision.
- Various heuristics may lead to contradictory decisions.

Disadvantages

- Heuristics might be subjective.
- If misused, may cause more harm than good.

Consistency as Heuristics

Consistency

- Good heuristics for decision making.

Consistency with ...

- other functions of the product, similar products, history, producer image, specifications, standards, user expectations, the purpose of the product, etc.

Advantages

- Consistency is objective enough.
- Easily described in bug report.

Imperfection in Decision Making

Unintentional Blindness

- Human tester does not consider any test outputs that he/she does not pay attention to.
- Similarly, mechanical tester does not consider test outputs that it is not told to include into decision.

Uncertainty Principle

- The presence of observer may affect what is observed.

Consequence

- It is impossible to observe all possible outputs from a single test.

Oracle and Automation Process

Motivation

- Automation process eliminates human errors.
- Automation leads to repeatable procedures.
- Automation allows faster test evaluation.

Problems of Automation

- It is necessary to automate the decision making (oracle) principle.
- Can we do it? Only partially.

Standard Way of Oracle Automation

- A file of expected outputs, which is required to match precisely with the outputs of a test being executed.
- Example: MS Word could be used to define a the file of expected outputs for testing WordPad.

Problems of Automated Oracle

Amount of Agreement

- Assume MS Word to serve as the file of expected outputs for testing WordPad.
- How exactly is the expected output stored?
- Is 99% agreement still agreement?
- How is the percentage of agreement defined?

False Alarms

- Using outdated expected output.
- Consequence of simplification of decision making.

Undiscovered Errors

- Expected file exhibits the same error as test output.
- Unintentional Blindness.

Measure Methods in Testing

Coverage as a Measure

Coverage

- A set of source code entities that has been checked with at least one test.
- Source-code entities: lines of code, conditions, function calls, branches, etc.
- Used to identify parts that have not been tested yet.

Coverage as a Measure

- Possible test plan is to achieve a given percentage of coverage.
- The percentage than expresses how much of the final product has been tested.
- Numeric expression for managers to see how much of the product remains to be tested.

Coverage as a Measure – Disadvantages

Problems

- Could avoid testing of interesting input data.
- Does not properly test parts of the product that rely on external services.

Using Coverage as a Measure

- The mission is to test all entities of the product, is that OK?
- Complete coverage does not guarantee quality of the product.
- Stimulates to prefer quantity rather than quality.
- Misleading satisfaction (shouldn't feel safe).

Coverage as a Measure – Disadvantages

Example

```
Input A          // program accepts any
Input B          // integer into A and B
Print A/B
```

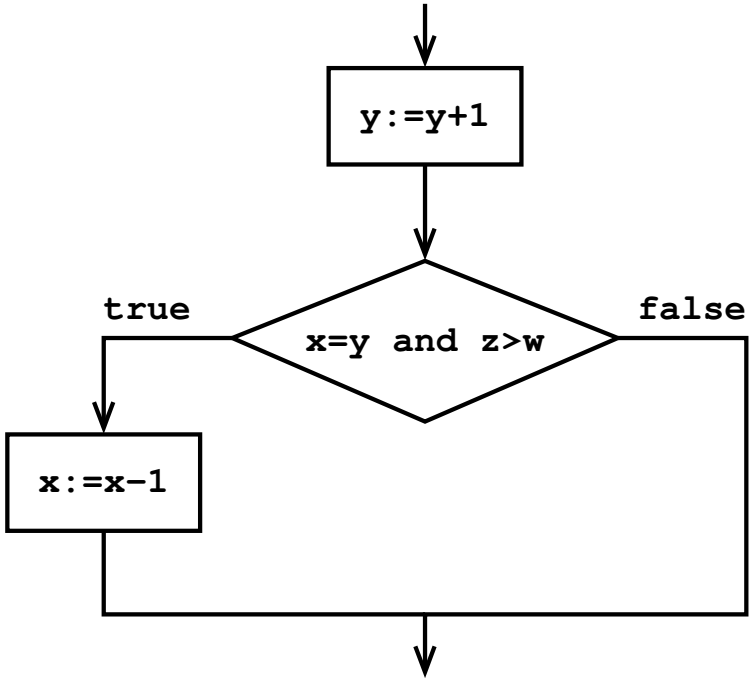
Observation

- Complete coverage is easy achievable.
- For example:

  input: 2,1

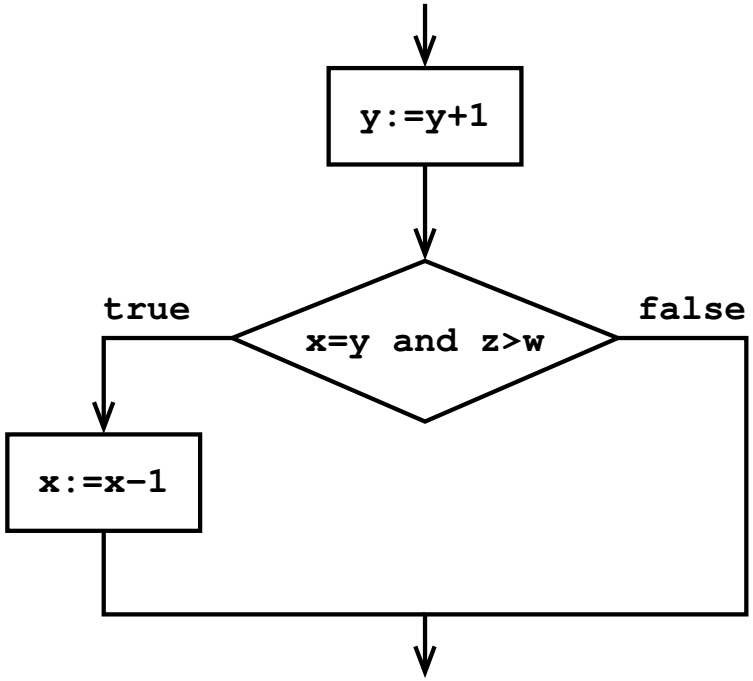
  output: 2
- There is of course a hidden bug in the program!

Coverage Criteria for Control-Flow Graphs



There are various criteria for control-flow graph coverage.

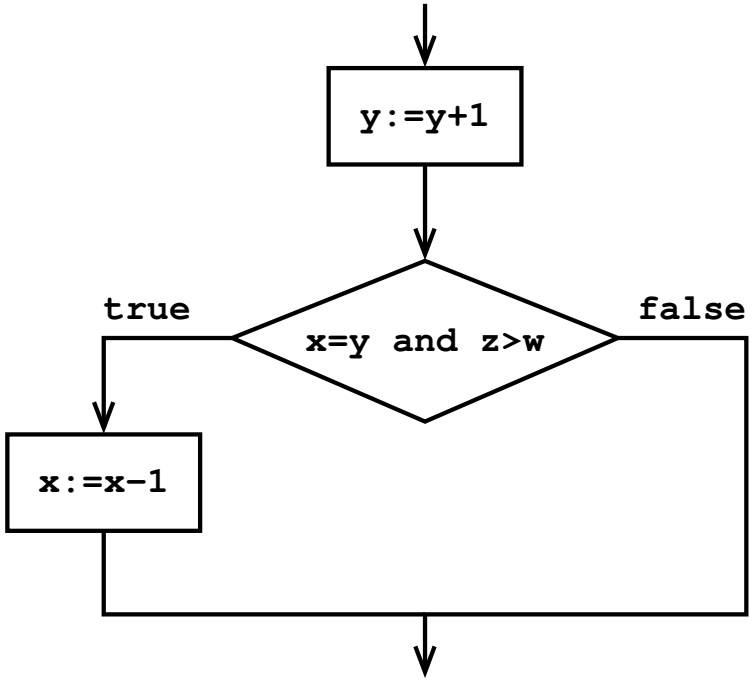
Coverage Criteria for Control-Flow Graphs



Statement coverage

- Every statement (assignment, input, output, condition) is executed in at least one test.
- Set of tests to achieve full coverage:
  $(x = 2, y = 1, z = 4, w = 3)$

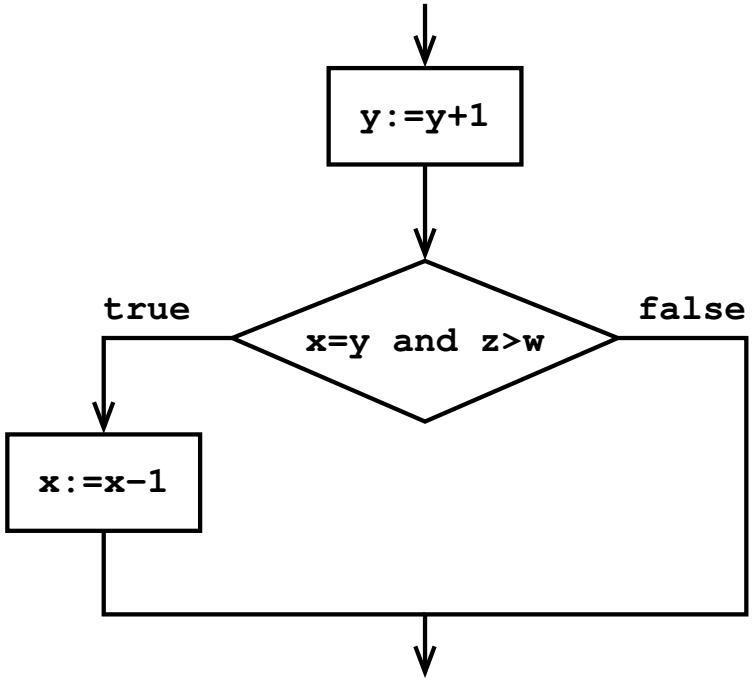
Coverage Criteria for Control-Flow Graphs



Edge coverage

- Every edge of CFG is executed in at least one test.
- Set of tests to achieve full coverage:
  $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 3, z = 5, w = 7)$

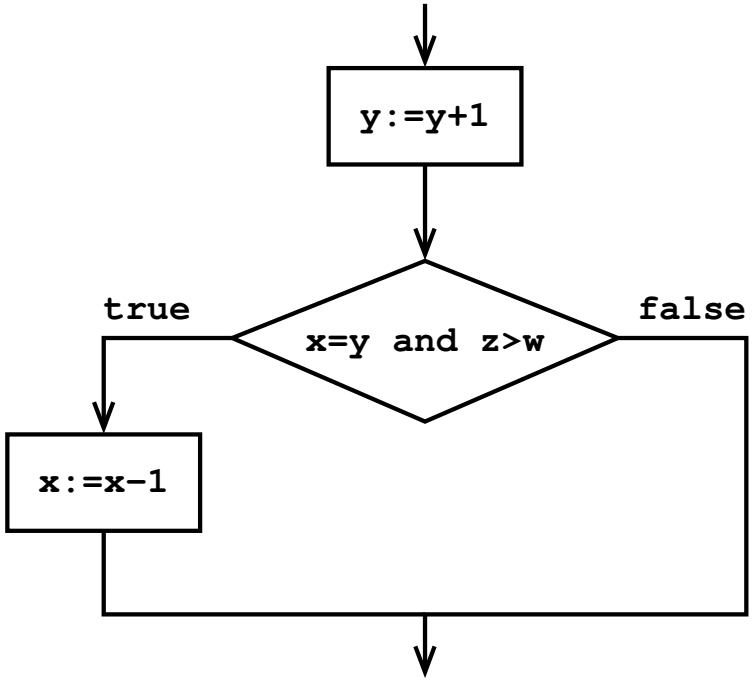
Coverage Criteria for Control-Flow Graphs



Condition coverage

- Every condition is a Boolean combination of elementary conditions, for example $x < y$ or $even(x)$.
- If it is possible, every elementary condition is evaluated in at least one test to TRUE and in at least one test to FALSE.

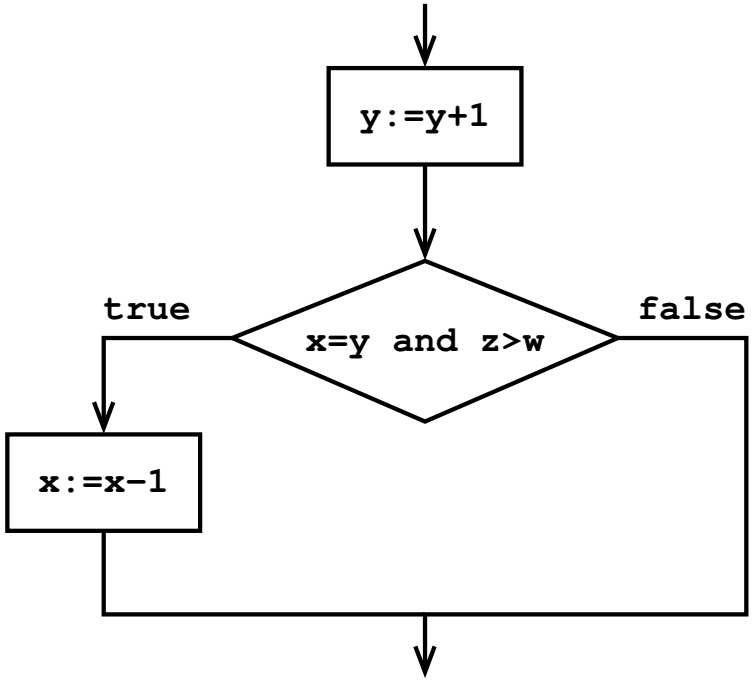
Coverage Criteria for Control-Flow Graphs



Condition coverage

- Set of tests to achieve full coverage:
  $(x = 3, y = 2, z = 5, w = 7)$, $(x = 3, y = 3, z = 7, w = 5)$
- In both cases, only the FALSE branch of IF statement is taken.

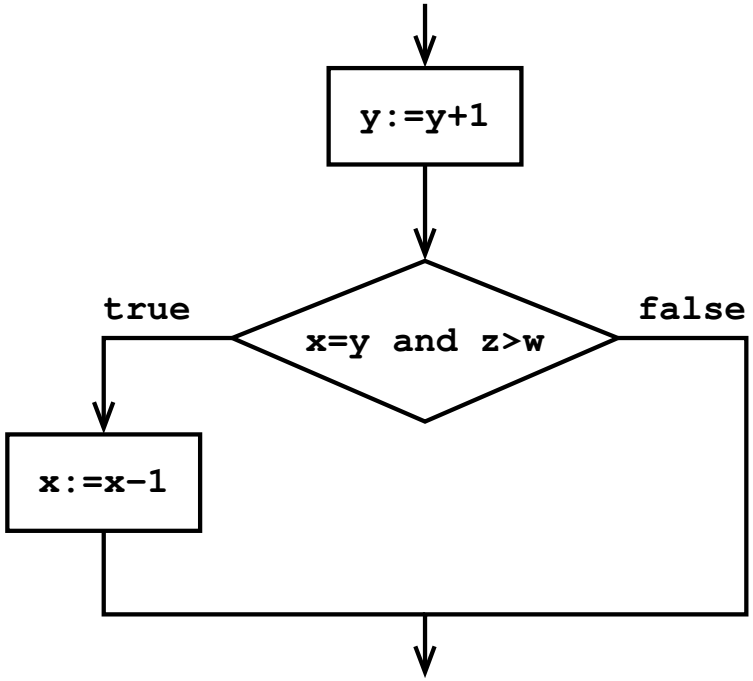
Coverage Criteria for Control-Flow Graphs



Edge/Condition coverage

- Edge and Condition coverage at the same time.
- Set of tests to achieve full coverage:
  $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 2, z = 5, w = 7)$,
  $(x = 3, y = 3, z = 7, w = 5)$
- Is the set the smallest possible one?

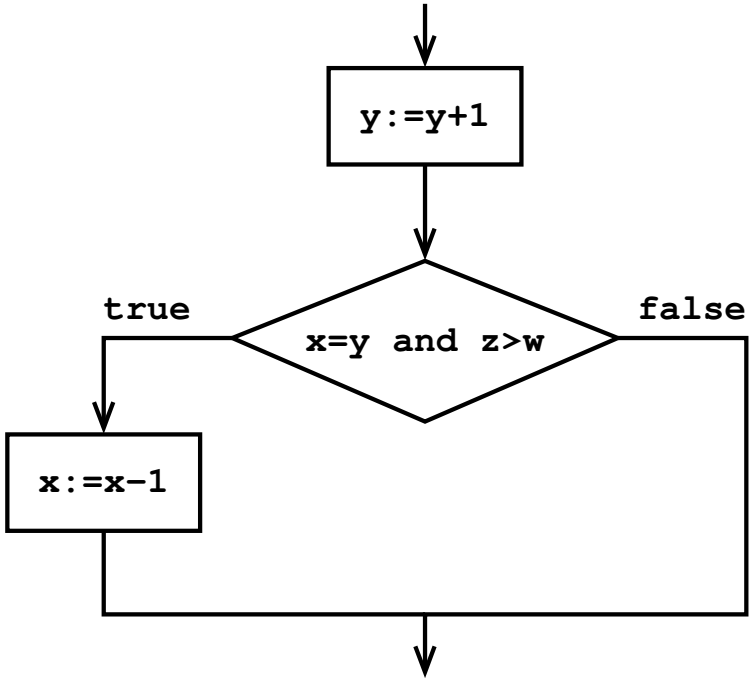
Coverage Criteria for Control-Flow Graphs



Multiple condition coverage

- Every Boolean combination of TRUE/FALSE values that may appear in some decision condition must occur in at least one test.

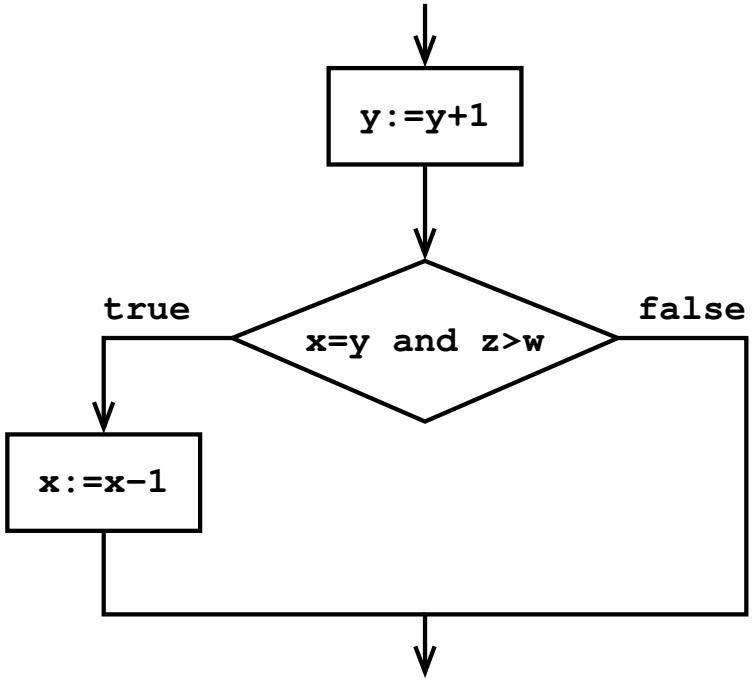
Coverage Criteria for Control-Flow Graphs



Multiple condition coverage

- Set of tests to achieve full coverage:
  $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 2, z = 5, w = 7)$,
  $(x = 3, y = 3, z = 7, w = 5)$, $(x = 3, y = 3, z = 5, w = 6)$
- Exponential grow in the number of tests.

Coverage Criteria for Control-Flow Graphs



Path coverage

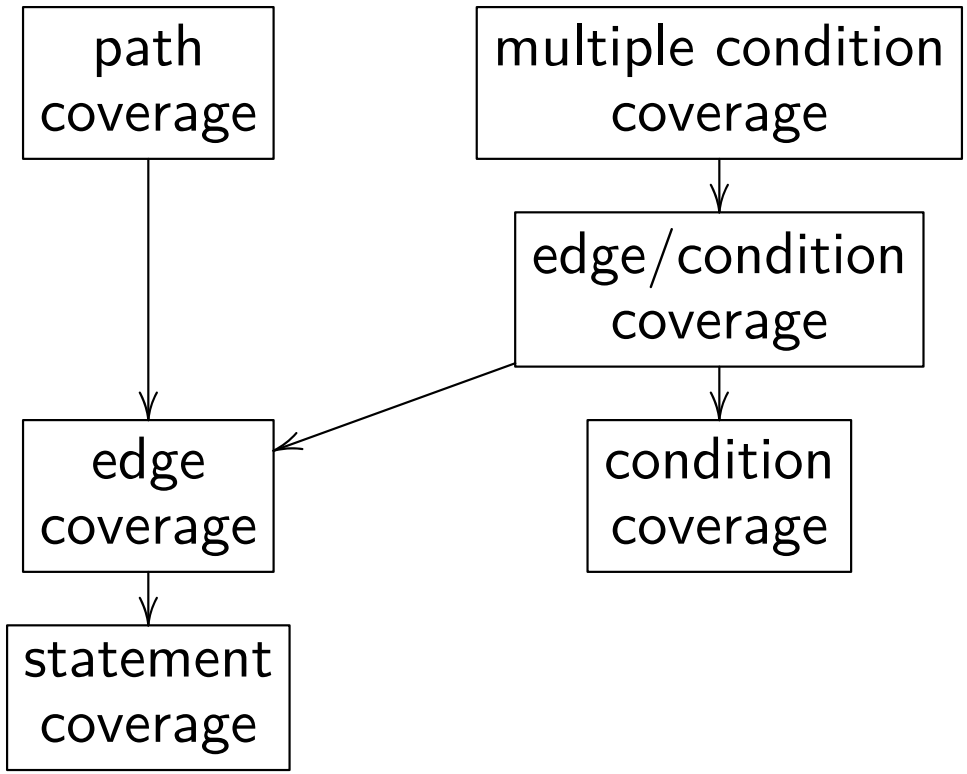
- Every executable path is executed in at least one test.
- The number of paths is big, even infinite in case there is an unbounded cycle in the control-flow graph.

Hierarchy of Coverage Criteria

- Criterion A **includes** criterion B, denoted with A $\rightarrow$ B, if after full coverage of type A we guarantee full coverage of type B.

Hierarchy of Coverage Criteria

- Criterion A **includes** criterion B, denoted with A $\rightarrow$ B, if after full coverage of type A we guarantee full coverage of type B.



Cycle Coverage

Coverage and Number of Cycle Iterations

- All criteria except the path criterion does not reflect the number of iterations over a cycle body.
- In case of nested cycles, systematic testing of all possible executable paths become complicated.

Ad hoc Strategy for Testing Cycles

- Check the case when the cycle is completely skipped.
- Check the case when the cycle is executed exactly once.
- Check the case when the cycle is executed the expected number of times.
- If a boundary n is known for the number of cycle executions, try to design tests where the cycle is executed $n-1$, n, and $n+1$ times.

Coverage for Data-Flow Graphs

Motivation

- Detect usage of undefined variables.
- On some paths, a variable may be set for a specific purpose and later on its value misused for other purpose.
- Control Flow criteria do not guarantee inclusion of tests for above mentioned or likewise situations.

Data Flow Coverage

- Cover paths through control flow graph that go through a location where a variable is used but it is not defined along all incoming paths through control-flow graph.

Support for Code Coverage

C/C++, Linux

- Tools `gcov` and `lcov`.

Example: lcov

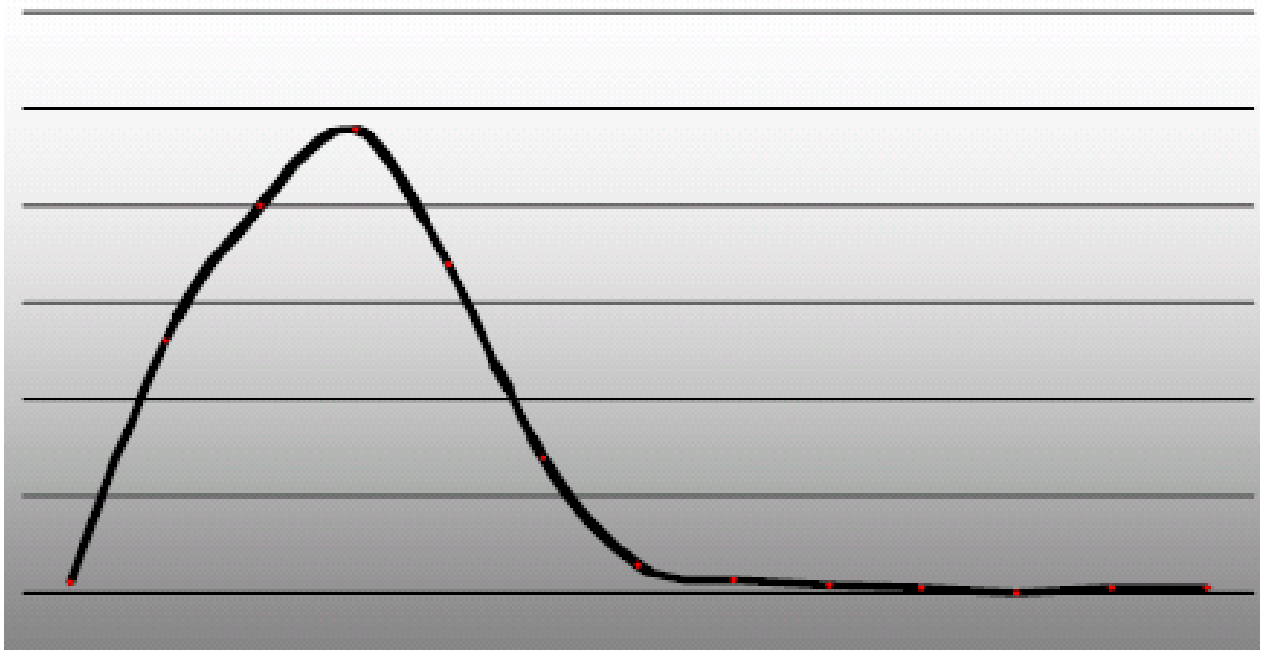
- gcc -fprofile-arcs -ftest-coverage foo.c -o foo
  lcov -d . -z
  lcov -c -i -d . -o base.info
  ./foo
  lcov -c -d . -o collect.info
  lcov -d . -a base.info -a collect.info -o result.info
  genhtml result.info

Statistics on Found/Fixed Bugs per Time Unit

Week statistics

- The number of newly discovered errors.
- The number of fixed errors.
- The ratio of found versus fixed errors.

Visualisation



Weibull Distribution

Observation

- The number of discovered errors per time unit exhibits Weibull Distribution.
- Can be used as a measure for the remaining amount of testing.
- Software Engineering Method to set the release date.

Using Weibull Distribution

- At the moment the curve reaches the peak, the remaining part of the curve may be predicted, hence, a moment in time may be set, when expected number of errors discovered per week drops below a given threshold.
- Parameters of Weibull distribution influence the "width" and "height/slope" of the peak.
- $F(x) = 1 - e^{-ax^{-b}}$ for $x > 0$

Weibull Distribution – Imperfections

Vague Precision

- Testing does not follow the typical usage of the product.
- The probability of error discovery is different for different errors.
- Fix may cause other new errors.
- Bugs are dependent.
- The number of errors in the product changes over time.
- Error insertion exhibits Weibull distribution itself.
- Testing epochs (various testing procedures) are independent.
- ...

Conclusions

- Weibull Distribution is not very reliable.
- Can be used only with large projects for very rough estimation.

Impact of following Weibull Distribution

Assumption

- Software developers are aware of being measured.

Phase one

- Reach the peak as quickly as possible.
- Double reporting of errors.
- Avoid fixing known errors.
- ...

Phase two

- Stick to expected shape of the curve.
- Delay reporting of errors.
- Reporting outside bug-tracking system.
- ...

Incompleteness of Testing

Definition

Observation

- The amount of tests to be run is extremely large.
- Resources for testing are always limited.

What Is Not Complete Testing

- Complete Coverage
  - Every line of code.
  - Every branching point.
  - ...
- When testers do not find more errors.
- Testing plan is finished.

What Is Complete Testing

- There are no hidden or unknown errors in the product.
- If new issue is reported, testing could not be complete.

Reasons for Incompletness of Testing

The number of tests is too large (infinitely many).

To perform all tests means:

- To test all possible input values of every input variable.
- To test all combinations of input variables.
- To test every possible run of a system.
- To test every combination of HW and SW, including future technology.
- To test every way a user may use the product.

Impossibility to Test All Possible Inputs

Data Bus-Width

- The number of tests grows exponentially with respect to bit used for data representation.
- n-bits requires 2^n tests.

Other Reasons

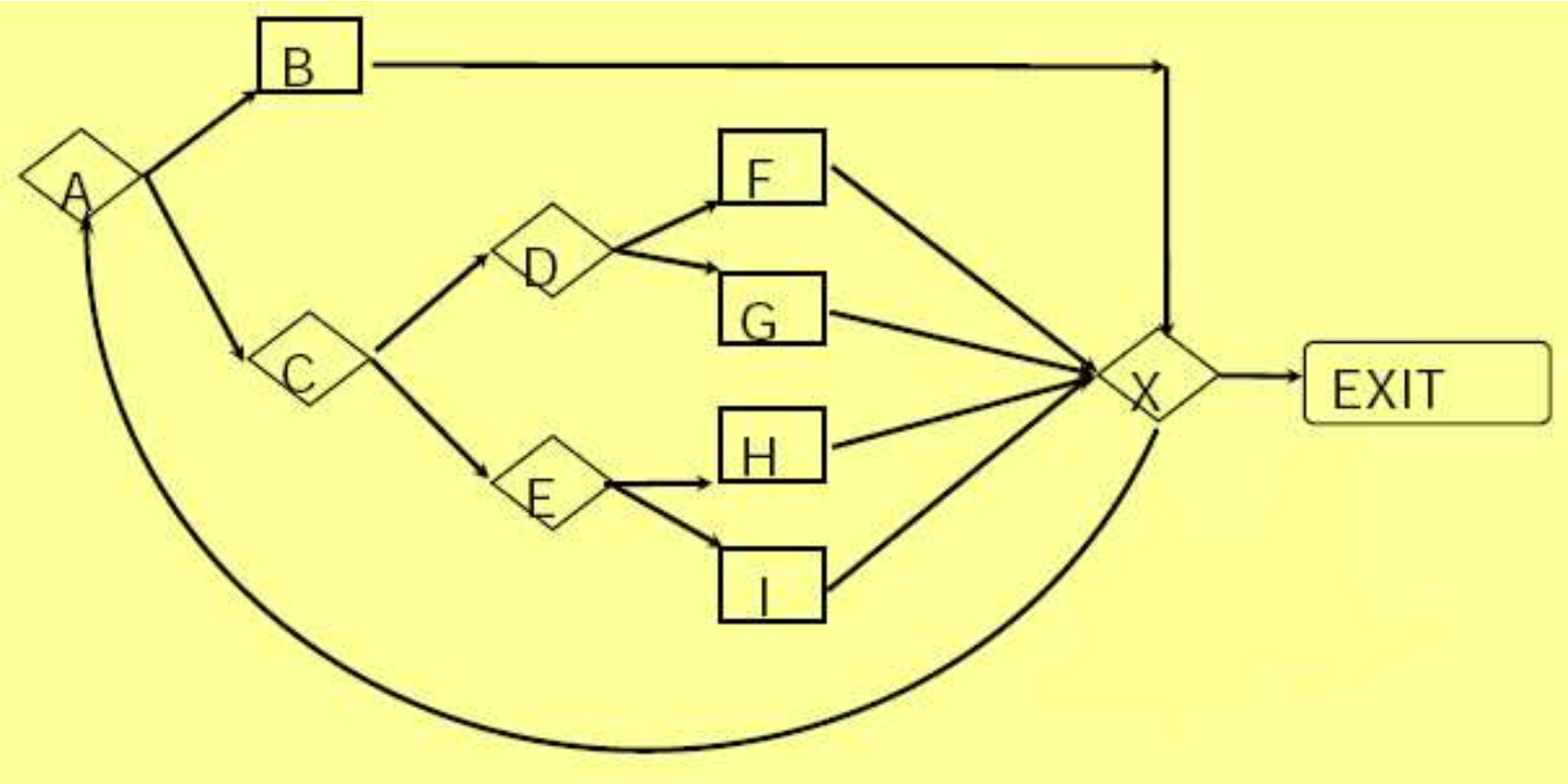
- Timing of actions.
- Invalid or unexpected inputs (buffer overflow).
- Edited inputs
- Easter egg [http://j-walk.com/ss/excel/eastereg.htm]

Common Argumentation

- "This is not what the customer would do with our product."

Incapability to Test All Runs

Assume the following system



Questions

- How many different ways it is possible to reach $\square\text{EXIT}\square$?
- How many different ways it is possible to reach $\square\text{EXIT}\square$, if $\langle A \rangle$ can be visited at most n-times?

Incapability to Test All Runs

Example

- In [F] is a memory leak, in [B] garbage collector.
- System will reach an invalid state, if [B] is avoided long enough.

Observation

- Simplified testing of paths may not discover the error.
- The error manifests in circumstances that cannot be achieved with a simple test.

Summary for Measure and Incompleteness

Incompleteness

- Testing cannot prove absence of error.
- It is impossible to test all valid inputs.
- Existence of testing plan inhibits testing creativity.

Measure

- There are methods to measure progress in testing phase.
- These are unreliable.
- Focusing strongly on a selected measure may influence the effectiveness of testing.

Homework

Homework

- Reading on MC/DC:
  `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf`
- List, and briefly describe as many black-box testing approaches as you can find or are aware of.
  `http://www.testingeducation.org/BBST/`
- Optional: Learn about CMAKE and CTEST systems.

IA169 System Verification and Assurance

Symbolic Execution and Concolic Testing

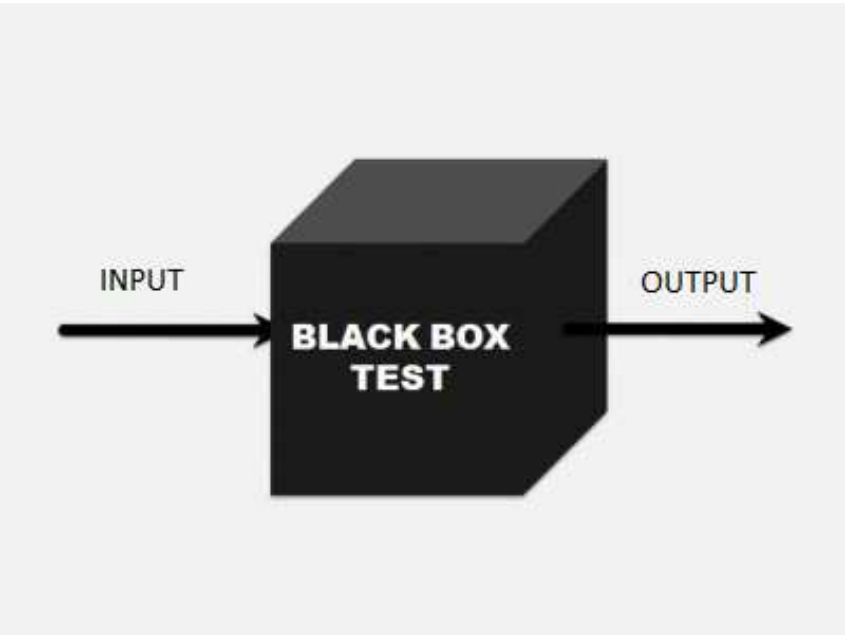
Jiří Barnat

Testing Strategies

Black-box Testing

Black-box

- A product under test is viewed as a **black box**.
- It is analysed through the input-output behaviour.
- Inner details (such as source code) are hidden or not taken into account.



White-box, Gray-box Testing

White-box Testing (Glass-box)

- Inner details are taken into account.
- Tests are selected and executed with respect to the inner details of the product, e.g. code coverage.
- Error insertion, modification of the product for the purpose of testing.
- **Basically only extends any Black-box approach.**

Gray-box Testing

- In between of Black-box and White-box.
- Sometimes the same as White-box, inconsistent terminology.

Testing Techniques

Primary Black-box Strategies

- Domain Testing
- Combinatory Testing
- Scenario Testing
- Risk-based Testing
- Functional Testing
- Fuzz Testing (Mutation Testing)

Primary White-box Extensions

- Model-based Testing
- Unit Testing

Support for Developers

- Regression Testing

Symbolic Execution

Motivation

Problem

- To detect errors that systematically exhibit only for specific input values is difficult.
- Relates to incompleteness of testing.

Still we would like to ...

- test the program on inputs that make program execute differently from what has already been tested.
- test the program for all inputs.

Symbolic Execution

Idea

- Execute a program so that values of input variables are referred to as to symbols instead of concrete values.

Demo

Program	Selected concrete values	Symbolic representation
`read(A)`		
	$A = 3$	$A = \alpha$
`A = A * 2`		
	$A = 6$	$A = \alpha * 2$
`A = A + 1`		
	$A = 7$	$A = (\alpha * 2) + 1$
`output(A)`		

Branching and Path Condition

Observation

- Branching in the code put some restrictions on the data depending on the condition of a branching point.

Example

```
1 if (A == 2)
2   then ...
3   else ...
```

$A = (\alpha * 2) + 1$

$(\alpha * 2) + 1 = 2$

$(\alpha * 2) + 1 \neq 2$

Path Condition

- Formula over symbols referring to input values.
- Encodes history of computation, i.e. cumulative restrictions implied from all the branching points walked-through up to the curent point of execution.
- Initially set to true.

Unfeasible Paths

Observation

- The path condition may become unsatisfiable.
- If so, there are no input values that would make the program execute that way.

Example 1

```
1 if (A == B)
2   then
3     if (A == B)
4       then ...
5       else ...
6   else ...
```

$A = \alpha, B = \beta$

$\alpha = \beta$

$\alpha = \beta \wedge \alpha = \beta$

$\alpha = \beta \wedge \alpha \neq \beta$ *is UNSAT*

$\alpha \neq \beta$

Example 2

% – operation modulo

```
1 A=A%2
2 if (A == 3) then ...
3             else ...
```

$A = \alpha\%2$

$\alpha\%2 = 3$ *is UNSAT*

$\alpha\%2 \neq 3$

Tree of Symbolic Execution

Observation

- All possible executions of program may be represented by a tree structure – **Symbolic Execution Tree**.
- The tree is obtained by unfolding/unwinding the control flow graph of the program.

Symbolic Execution Tree

- Node of the tree encodes program location, symbolic representation of variables, and a concrete path condition.

location	symbolic valuation	path condition
#12	$A = \alpha + 2, B = \alpha + \beta - 2$	$\alpha = 2 * \beta - 1$

- An edge in the tree corresponds to a symbolic execution of a program instruction on a given location.
- Branching point is reflected as branching in the tree and causes updates of path conditions in individual branches.

Example of Symbolic Execution Tree

Program

```
1 input A,B
2 if (B<0) then
3   return 0
4 else
5   while (B > 0)
6   { B=B-1
7     A=A+B
8   }
9 return A
```

Draw Yourself.

Path Explosion

Properties of Symbolic Tree Execution

- No nodes are merged, even if they are the same (the structure is a tree).
- A single program location may be contained in (infinitely) many nodes of the tree.
- Tree may contain infinite paths.

Path Explosion Problem

- The number of branches in the symbolic execution tree may be large for non-trivial programs.
- The number of paths may grow exponentially with the number of branching points visited.

Employing Symbolic Execution Tree for Verification

Analysis of the Tree

- Breadth-first strategy, the tree may be infinite.

Deduced Program Properties

- Identification of feasible and unfeasible paths.
- Proof of reachability of a given program location.
- Error detection (division by zero, out-of-array access, assertion violation, etc.).

Synthesis of Test Input Data

- If the formula encoded as a path condition is satisfiable for a symbolic run, the model of the formula gives concrete input values that make the program to follow the symbolic run.
- Excellent for synthesis of tests that increase code coverage.

Automated Test Generation

Principle

1. Generate random input values (encode some random path).
2. Perform a walk through the Symbolic Execution Tree with the random input values and record the path condition.
3. Generate a new path condition from the recorded one by negating one of the restrictions related to a single branching point.
4. Find input values satisfying the new path condition.
5. Repeat from number 2 until desired coverage is reached.

Practical Notes

- Heuristics for selection of branching point to be negated.
- Augmentation of the code to enable path condition recording.

Limits of Symbolic Execution

Undecidability

- Using complex arithmetic operations on unbounded domains implies general undecidability of the formula satisfaction problem.
- Symbolic Execution Tree is infinite (due to unwinding of cycles with unbound number of iterations).

Computational Complexity

- Path explosion problem.
- Efficiency of algorithms for formula satisfiability on finite domains.

Known Limits

- Symbolic operations on non-numerical variables.
- Not clear how to deal with dynamic data structures.
- Symbolic evaluation of calls to external functions.

Tools for SAT Solving

SAT Problem

Satisfiability Problem – SAT

- Is to decide if there exists a valuation of Boolean variables of propositional logic formula that makes the formula hold true (be valid).

SAT Problem Properties

- Famous NP-complete problem.
- Polynomial algorithm is unlikely to exist.
- Still there are existing SAT solvers that are very efficient and due to a plethora of heuristics can solve surprisingly large instances of the problem.

Tool Z3

ZZZ aka **Z3**

- Developed by Microsoft Research.
- SAT and SMT Solver.
- WWW interface — `http://www.rise4fun.com/Z3`
- Standardised binary API for use within other verification tools.

Decide using Z3

- Is formula $(a \vee \neg b) \wedge (\neg a \vee b)$ satisfiable?

Usage of Z3 – SAT

Reformulate into language of Z3 $(a \vee \neg b) \wedge (\neg a \vee b)$

- ```
  (declare-const a Bool)
  (declare-const b Bool)
  (assert (and (or a (not b)) (or (not a) b)))
  (check-sat)
  (get-model)
  ```

Answer of Z3

- ```
  sat
  (model
    (define-fun b () Bool
      false)
    (define-fun a () Bool
      false)
  )
  ```

Satisfiability Modulo Theory – SMT

Satisfiability Modulo Theory – SMT

- Is to decide satisfiability of first order logic with predicates and function symbols that encode one or more selected theories.
- Typically used theories
  - Arithmetic of integer and floating point numbers.
  - Theories of data structures (lists, arrays, bit-vectors, . . .).

Other view (Wikipedia)

- SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalised approach to constraint programming.

Examples of SMT in Z3

Solve using Z3 `http://rise4fun.com/Z3/tutorial/guide`

- Are there two integer non-zero numbers `x` and `y` such that `y=x*(x-y)`?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x y))))
(assert (not (= y 0)))
(check-sat)
(get-model)
```

- Are there two integer non-zero numbers `x` and `y` such that `y=x*(x-(y*y))`?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x (* y y)))))
(assert (not (= x 0)))
(check-sat)
```

Satisfiability and Validity

Observation

- A formula is valid if and only if its negation is not satisfiable.

Consequence

- SAT and SMT solvers can be used as theorem provers to show validity of some theorems.

Model Synthesis

- SAT solvers not only decide satisfiability of formulae but in positive case also give concrete valuation of variables for which the formula is valid.
- Unlike general theorem provers they provide a counterexample in case the theorem to be proved is invalid (negation is satisfiable).

Concolic Testing

Motivation

Problem

- Efficient undecidability of path feasibility.
- In practice, unknown result often means unsatisfiability (no witness found).
- However, skipping paths that we only think are unfeasible, may result in undetected errors.
- On the other hand, executing unfeasible path may report unreal errors.

Partial Solution

- Let us use concrete and symbolic values at the same time in order to support decisions that are practically undecidable by a SAT or SMT solver.
- Heuristics.
- An interesting case (correct): UNKNOWN $\Longrightarrow$ SAT
- **Con**crete and Symb**olic Testing** = **Concolic Testing**

Hypothetical demo of concolic testing

Program

```
1 input A,B
2 if (A==(B*B)%30) then
3   ERROR
4 else
5   return A
```

Concolic Testing

1. A=22, B=7 (random values), test executed, no errors found.
2. (22==(7*7)%30) is *False*, path condition: $\alpha \neq (\beta * \beta)\%30$
3. Synthesis of input data from negation of path condition: $\alpha = (\beta * \beta)\%30$ – **UNKNOWN**
4. Employ concrete values: $\alpha = (7 * 7)\%30$ – **SAT**, $\alpha = 19$
5. A=19, B=7
6. Test detected error location on program line 3.

SAGE Tool

Systematic Testing for Security: Whitebox Fuzzing

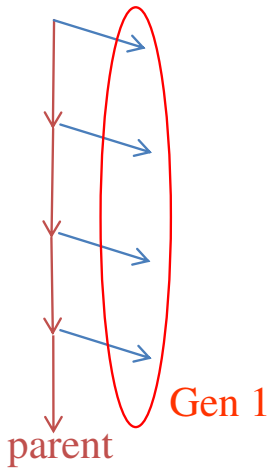
Patrice Godefroid
Michael Y. Levin and David Molnar

http://research.microsoft.com/projects/atg/

Microsoft Research

Whitebox Fuzzing (SAGE tool)

- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution



- Search spaces are huge, the search is partial…
  yet effective at finding bugs !

Example: Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt > 3) crash();
}
```

input = "good"

Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt > 3) crash();
}
```

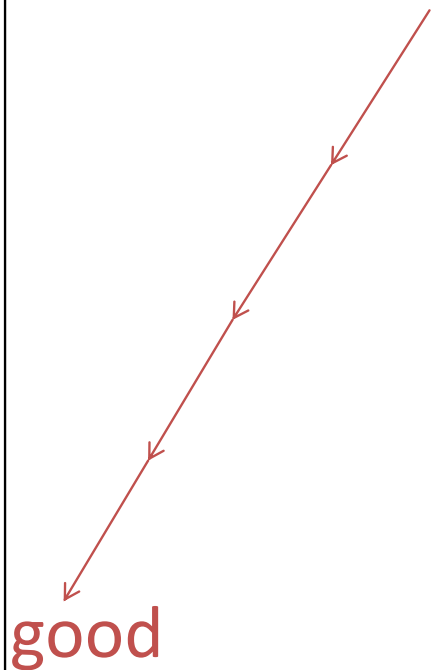
input = "good"

Path constraint:

I_0 != 'b'
I_1 != 'a'
I_2 != 'd'
I_3 != '!'

Negate a condition in path constraint
Solve new constraint → new input

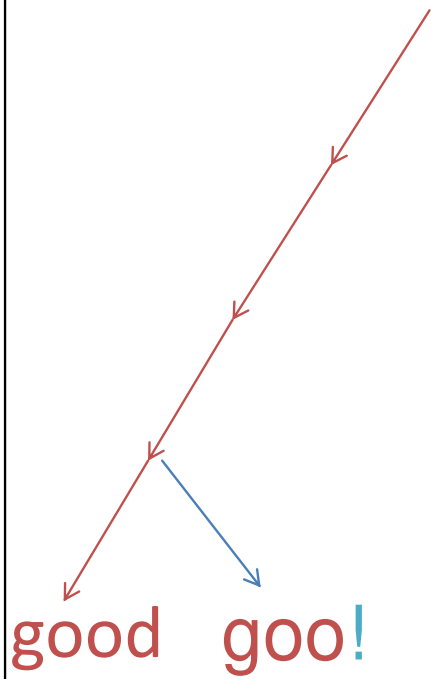
Depth-First Search



input = "good"

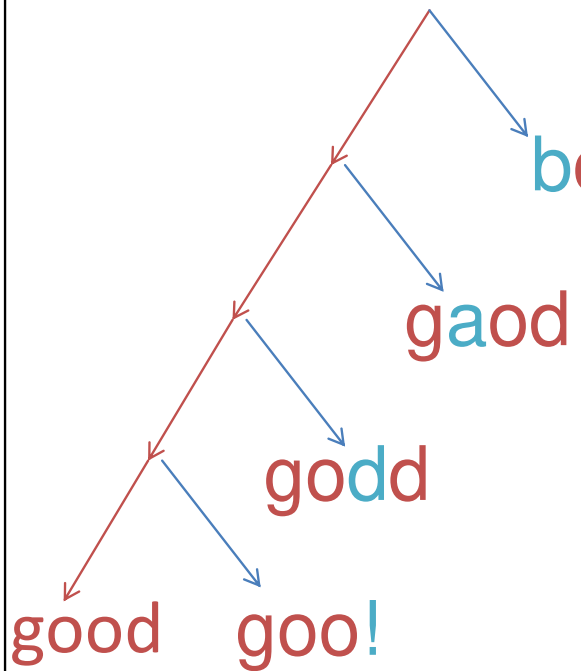
```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;   I0 != 'b'
   if (input[1] == 'a') cnt++;   I1 != 'a'
   if (input[2] == 'd') cnt++;   I2 != 'd'
   if (input[3] == '!') cnt++;   I3 != '!'
   if (cnt > 3) crash();
}
```

Depth-First Search




```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;   I0 != 'b'
   if (input[1] == 'a') cnt++;   I1 != 'a'
   if (input[2] == 'd') cnt++;   I2 != 'd'
   if (input[3] == '!') cnt++;   I3 == '!'
   if (cnt > 3) crash();
}
```

Generational Search

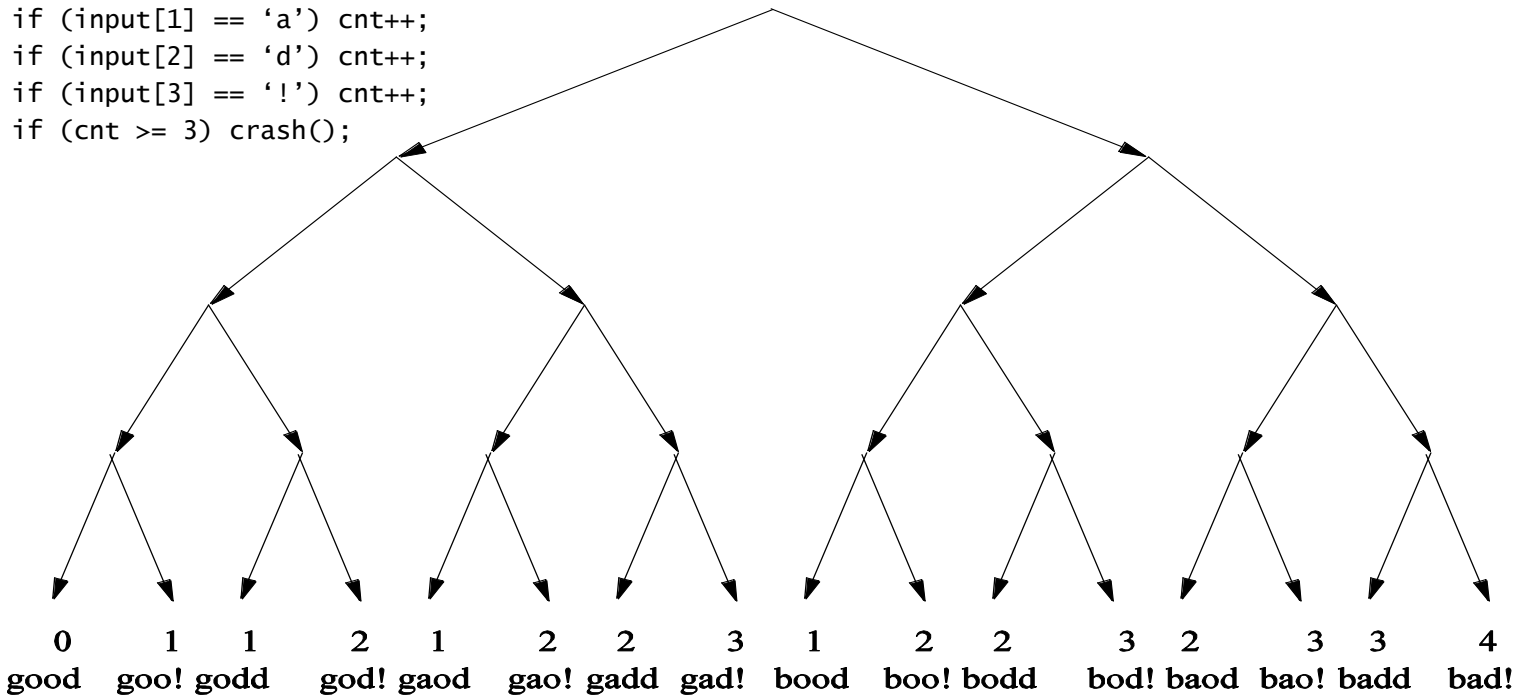


Four “Generation 1” test cases !

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;   I0 == 'b'
   if (input[1] == ‘a’) cnt++;   I1 == 'a'
   if (input[2] == ‘d’) cnt++;   I2 == 'd'
   if (input[3] == ‘!’) cnt++;   I3 == '!'
   if (cnt > 3) crash();
}
```

The Search Space

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```



Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 0 – seed file

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 00 ; RIFF............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 1

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....***.... 
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 2

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 3

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 00 ; ....strh........
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 4

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 5

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; ....strf........
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 6

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 7

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; ............EäN
00000060h: 00 00 00 00                                     ; ....
```

Generation 8

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 9

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 10 – crash bucket 1212954973!

Initial Experiences with SAGE

- Since 1st internal release in April'07: tens of new security bugs found
- Apps: image processors, media players, file decoders,... Confidential !
- Bugs: Write A/Vs, Read A/Vs, Crashes,... Confidential !
- Many bugs found triaged as "security critical, severity 1, priority 1"

Homework

Homework

- Follow Klee tutorials 1 and 2 (`http://klee.github.io/klee/Tutorials.html`)
- Solve The wolf, goat and cabbage problem with Klee
- Solve `http://pex4fun.com/`

IA169 System Verification and Assurance

LTL Model Checking

Jiří Barnat

Motivation

Checking Quality

- Testing is incomplete, gives no guarantees of correctness.
- Deductive verification is expensive.

Typical reasons for system failure after deployment

- Interaction with environment (unexpected input values).
- Interaction with other system components.
- Parallelism (difficult to test).

Model Checking

- Automated verification process for ...
- ... reactive systems.
- ... parallel/distributed systems.

Verification of Reactive and Parallel Programs

Parallel Programs

Parallel Composition

- Components concurrently contribute to the transformation of a computation state.
- The meaning comes from interleaving of actions (transformation steps) of individual components.

Meaning Functions Do Not Compose

- Meaning function of a composition cannot be obtain as composition of meaning functions of participating components.
- The result depends on particular interleaving.

Example of Incomposability

Parallel System

- System: (y=x; y++; x=y) $\|$ (y=x; y++; x=y)
- Input-output variable x
- Meaning function of both processes is λx->x+1.
- The composition is: $(\lambda$x->x+1$)\cdot(\lambda$x->x+1$)$.
- $(\lambda$x->x+1$)\cdot(\lambda$x->x+1$)\ 0 = 2$

Two Different System Runs

- State $= (x, y_1, y_2)$
- (0,-,-) $\xrightarrow{y_1=x}$ (0,0,-) $\xrightarrow{y_2=x}$ (0,0,0) $\xrightarrow{y_1++}$ $\xrightarrow{x=y_1}$ (1,1,0) $\xrightarrow{y_2++}$ $\xrightarrow{x=y_2}$ (**1**,1,1)
- (0,-,-) $\xrightarrow{y_1=x}$ (0,0,-) $\xrightarrow{y_1++}$ $\xrightarrow{x=y_1}$ (1,1,-) $\xrightarrow{y_2=x}$ (1,1,1) $\xrightarrow{y_2++}$ $\xrightarrow{x=y_2}$ (**2**,1,2)

Consequence of Incomposability

Consequence 1

- In general, we cannot deduce correctness of the system from the correctness of its components.

Consequence 2

- In general, we cannot decompose the problem of checking correctness of the system into subproblems, i.e. to reduce it to the verification of correctness of all the participating system components.

Specification for Reactive/Parallel Programs

Problem of Specification

- The number of inputs and the input values are unknown before a particular execution of a reactive system.
- **How to specify the system properties?**

Examples of Specification

- Events A and B happens before event C.
- User is not allowed to enter a new value until the system processes the previous one.
- Procedure X cannot be executed simultaneously by processes P and Q (mutual exclusion).
- Every action A is immediately followed by a sequence of actions B,C and D.

Deductive Verification for Modal and Temporal Logic

Model checking

- Approach to formal verification of systems.
- Based on the system's state-space exploration.
- Requires specification to be given with formulae of temporal logics.

Assumption

- System properties are described formally using formulae of some temporal logic.

Turning into Formal Language

- Use of Modal and Temporal Logics.
- Amir Pnueli, 1977

Model Checking

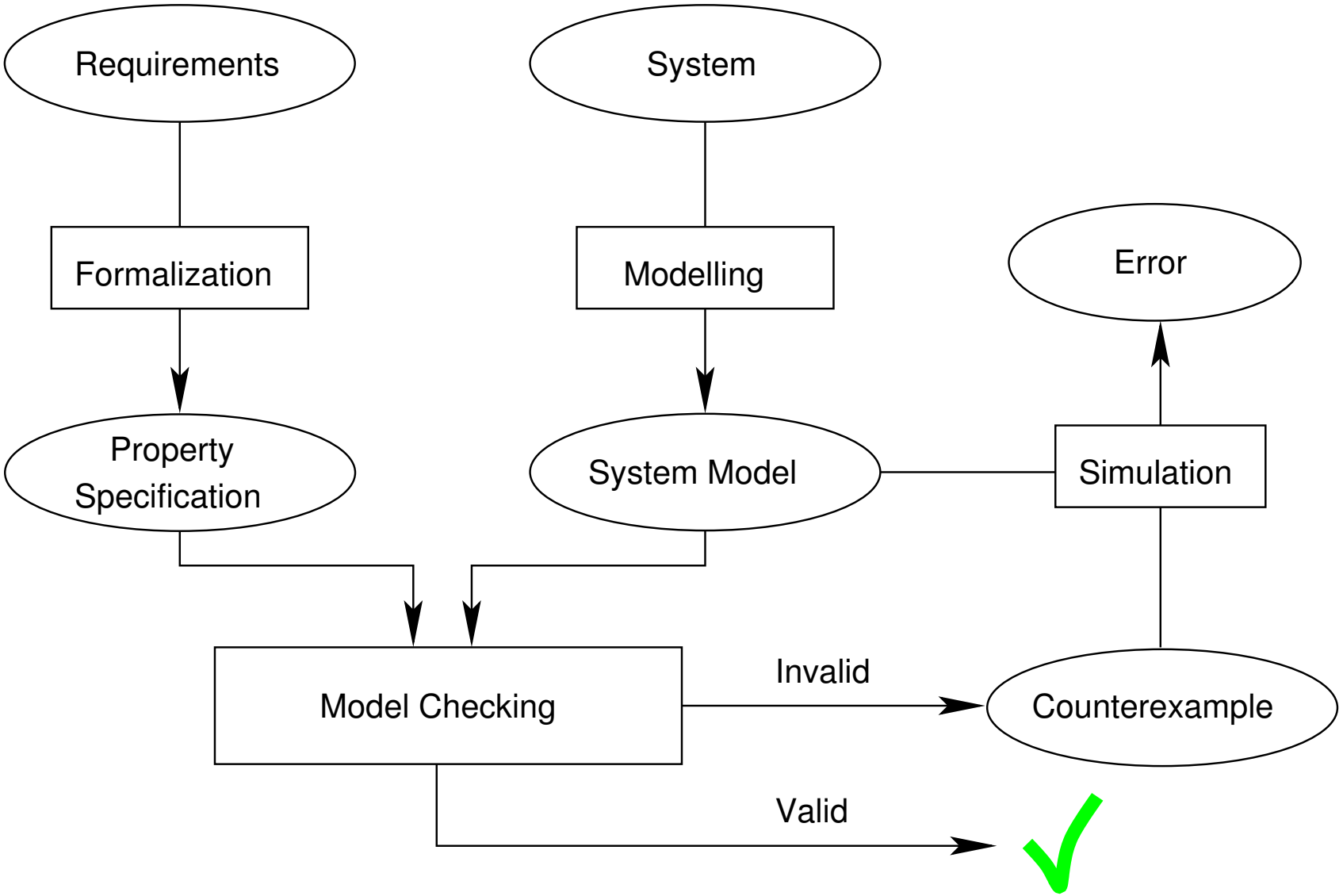
Model Checking – Overview

- Build a formal model $\mathcal{M}$ of the system under verification.
- Express specification as a formula φ of selected temporal logic.
- Decide, if $\mathcal{M} \models \varphi$. That is, if $\mathcal{M}$ is a model of formula φ. (Hence the name.)

Optionally

- As a side effect of the decision a **counterexample** may be produced.
- The counterexample is a sequence of states witnessing violation (in the case the system is erroneous) of the formula.
- **Model checking (the decision process) can be fully automated for all finite (and some infinite) models of systems.**

Model Checking – Schema



Automated Tools for Model Checking

Model Checkers

- Software tools that can decide validity of a formula over a model of system under verification.
- SPIN, UppAal, SMV, Prism, DIVINE ...

Modelling Languages

- Processes described as extended finite state machines.
- Extension allows to use shared or local variables and guard execution of a transition with a Boolean expression.
- Optionally, some transitions may be synchronised with transitions of other finite state machines/processes.

Modelling and Formalization of Verified Systems

Atomic Proposition

Reminder

- System can be viewed as a set of states that are walked along by executing instructions of the program.
- State = valuation of modelled variables.

Atomic Propositions

- Basic statements describing qualities of individual states, for example: $max(x, y) \geq 3$.
- Validity of atomic proposition for a given state must be decidable with information merely encoded by the state.
- Amount of observable events and facts depends on amount of abstraction used during the system modelling.

Kripke Structure

Kripke Structure

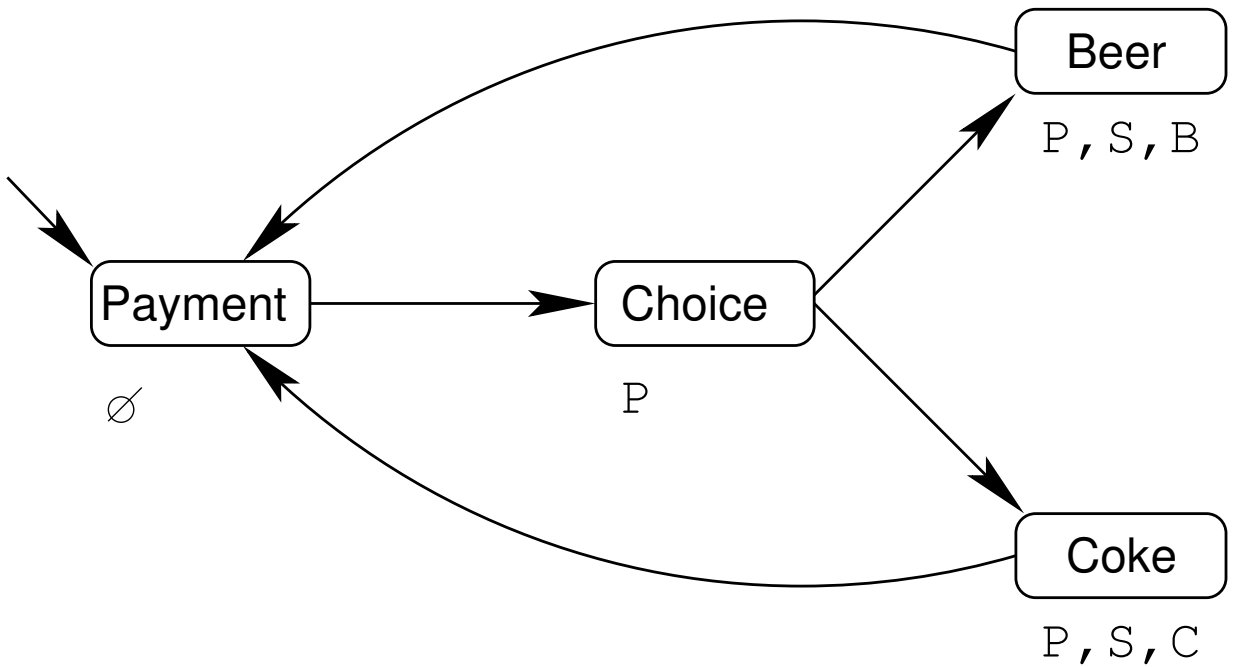
- Let AP be a set of atomic propositions.
- Kripke structure is a quadruple (S, T, I, s_0), where
  - S is a (finite) set of states,
  - $T \subseteq S \times S$ is a transition relation,
  - $I : S \to 2^{AP}$ is an interpretation of AP.
  - $s_0 \in S$ is an initial state.

Kripke Transition System

- Let Act be a set of instructions executable by the program.
- Kripke structure can be extended with transition labelling to form a Kripke Transitions System.
- Kripke Transition System is a five-tuple $(S, T, I, s_0, \mathcal{L})$, where
  - (S, T, I, s_0) is Kripke Structure,
  - $\mathcal{L} : T \to Act$ is labelling function.

Kripke Structure – Example

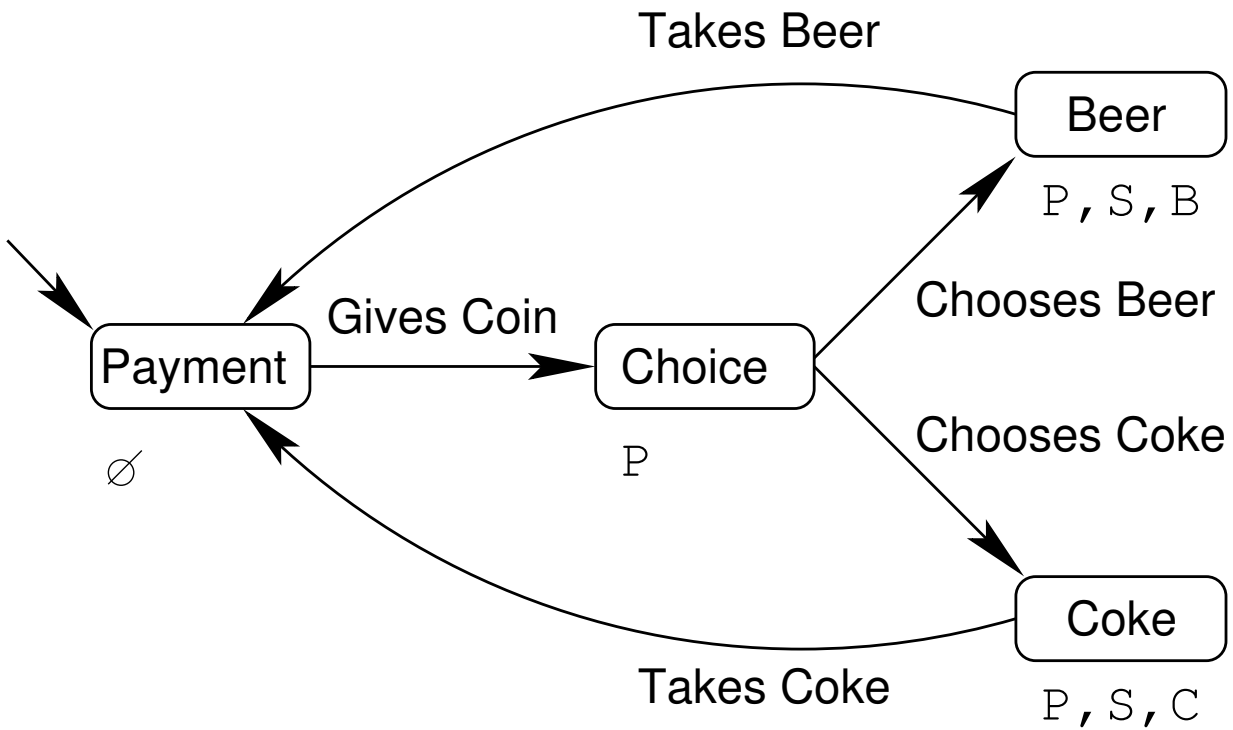
Kripke Structure



AP={P – Paid, S – Served, C – Coke, B – Beer}

Kripke Structure – Example

Kripke Transition System



AP={P – Paid, S – Served, C – Coke, B – Beer}

System Run

Run

- Maximal path (such that it cannot be extended) in the graph induced by Kripke Structure starting at the initial state.
- Let $M = (S, T, I, s_0)$ be a Kripke structure. Run is a sequence of states $\pi = s_0, s_1, s_2, \ldots$ such that $\forall i \in \mathbb{N}_0.(s_i, s_{i+1}) \in T$.

Finite Paths and Runs

- Some finite path $\pi = s_0, s_1, s_2, \ldots, s_k$ cannot be extended if $\nexists s_{k+1} \in S.(s_k, s_{k+1}) \in T$.
- Technically, we will turn maximal finite path into infinite by repeating the very last state.
- Maximal path $s_0, \ldots, s_k$ will be understood as infinite run $s_0, \ldots, s_k, s_k, s_k, \ldots$.

Implicit and Explicit System Description

Observation

- Usually, Kripke structure that captures system behaviour is not given by full enumeration of states and transitions (explicitly), but it is given by the program source code (implicitly).
- Implicit description tends to be exponentially more succinct.

State-Space Generation

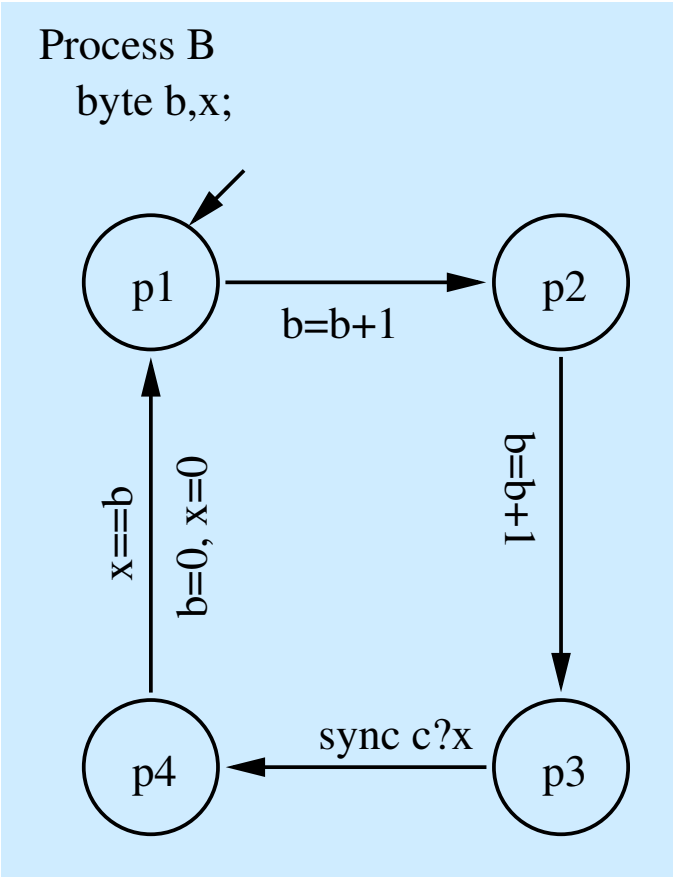
- Computation of explicit representation from the implicit one.
- Interpretation of implicit representation must be formally precise.

Practise

- Programming languages do not have precise formal semantics.
- Model checkers often build on top of modelling languages.

An Example of Modelling Language – DVE

- Finite Automaton
  - States (Locations)
  - Initial state
  - Transitions
  - (Accepting states)
- Transitions Extended with
  - Guards
  - Synchronisation and Value Passing
  - Effect (Assignment)
- Local Variables
  - `integer`, `byte`
  - `channel`



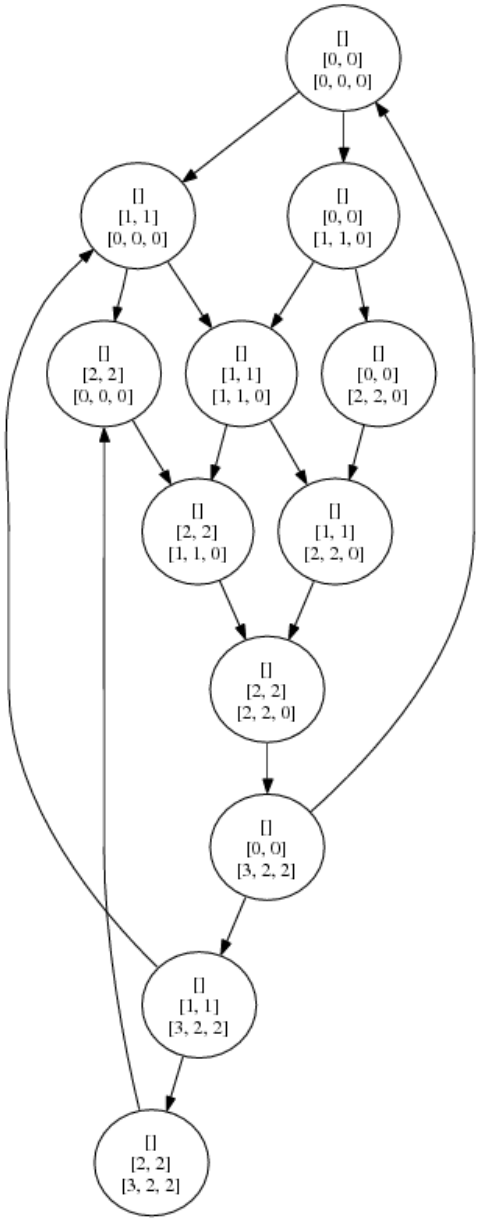
Example of System Described in DVE Language

```
channel {byte} c[0];

process A {
byte a;
state q1,q2,q3;
init q1;
trans
q1→q2 { effect a=a+1; },
q2→q3 { effect a=a+1; },
q3→q1 { sync c!a; effect a=0; };
}

process B {
byte b,x;
state p1,p2,p3,p4;
init p1;
trans
p1→p2 { effect b=b+1; },
p2→p3 { effect b=b+1; },
p3→p4 { sync c?x; },
p4→p1 { guard x==b; effect b=0, x=0; };
}

system async;
```



Semantics Shown By Interpretation

State: []; A:[q1, a:0]; B:[p1, b:0, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
1 ⟨1.0⟩: p1 → p2 { effect b = b+1; }
Command:1

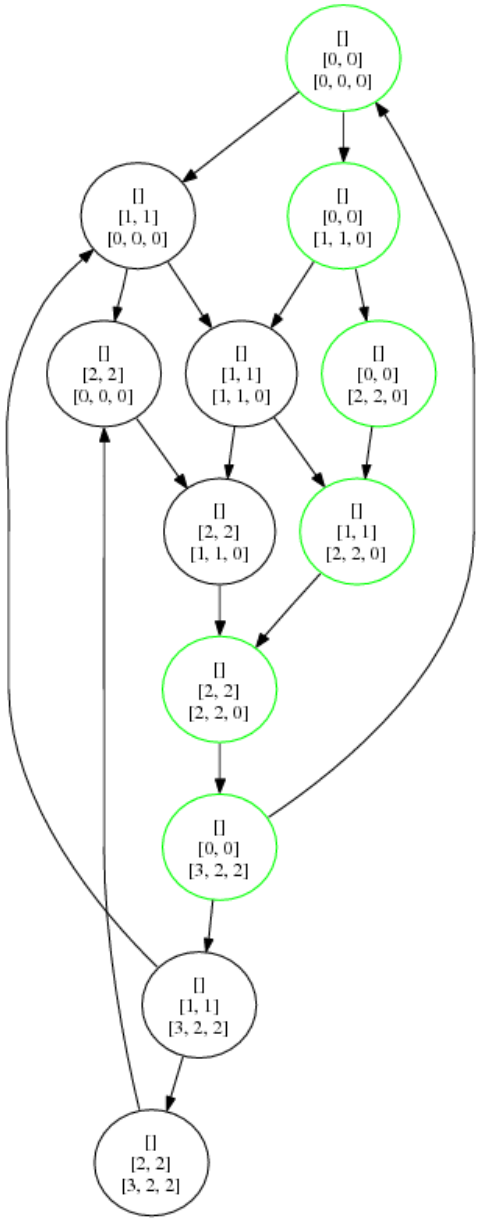
State: []; A:[q1, a:0]; B:[p2, b:1, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
1 ⟨1.1⟩: p2 → p3 { effect b = b+1; }
Command:1

State: []; A:[q1, a:0]; B:[p3, b:2, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
Command:0

State: []; A:[q2, a:1]; B:[p3, b:2, x:0]
0 ⟨0.1⟩: q2 → q3 { effect a = a+1; }
Command:0

State: []; A:[q3, a:2]; B:[p3, b:2, x:0]
0 ⟨0.2&1.2⟩: q3 → q1 { sync c!a; effect a = 0; }
p3 → p4 { sync c?x; }
Command:0

State: []; A:[q1, a:0]; B:[p4, b:2, x:2]



Formalizing System Properties

Specification as Languages of Infinite Words

Problem

- How to formally describe properties of a single run?
- How to mechanically check for their satisfaction?

Solution

- Employ finite automaton as a mechanical observer of run.
- Runs are infinite.
- Finite automata for infinite words (ω-regular languages).
- Büchi acceptance condition – automaton accepts a word if it passes through an accepting state infinitely many often.

Automata over infinite words

Büchi automata

- Büchi automaton is a tuple $A = (\Sigma, S, s, \delta, F)$, where
  - Σ is a finite set of symbols,
  - S is a finite set f states,
  - $s \in S$ is an initial state,
  - $\delta : S \times \Sigma \to 2^S$ is transition relation, and
  - $F \subseteq S$ is a set of accepting states.

Language accepted by a Büchi automaton

- Run ρ of automaton A over infinite word $w = a_1 a_2 \ldots$ is a sequence of states $\rho = s_0, s_1, \ldots$ such that $s_0 \equiv s$ and $\forall i : s_i \in \delta(s_{i-1}, a_i)$.
- $inf(\rho)$ – Set of states that appear infinitely many time in ρ.
- Run ρ is accepting if and only if $inf(\rho) \cap F \neq \emptyset$.
- Language accepted with an automaton A is a set of all words for which an accepting run exists. Denoted as $L(A)$.

Shortcuts in Transition Guards

Observation

- Let AP={X,Y,Z}.
- Transition labelled with $\{X\}$ denotes that X must hold true upon execution of the transition, while Y and Z are false.
- If we want to express that X is true, Z is false, and for Y we do not care, we have to create two transitions labelled with $\{X\}$ and $\{X, Y\}$.

APs as Boolean Formulae

- Transitions between the two same states may be combined and labelled with a Boolean formula over atomic propositions.

Example

- Transitions {X}, {Y}, {X,Y}, {X,Z}, {Y,Z} a {X,Y,Z} can be combined into a single one labelled with $X \vee Y$.
- If there are no restrictions upon execution of the transition, it may be labelled with $\mathit{true} \equiv X \vee \neg X$.

Task: Express with a Büchi automaton

System

- Vending machine as seen before.
- $\Sigma = 2^{\{P,S,C,B\}}$,
- $\mathit{Paid} = \{A \in \Sigma \mid P \in A\}$, $\mathit{Served} = \{A \in \Sigma \mid S \in A\}$, ...

Express the following properties

- Vending machine serves at least one drink.
- Vending machine serves at least one coke.

- Vending machine serves infinitely many drinks.
- Vending machine serves infinitely many beers.

- Vending machine does not serve a drink without being paid.
- After being paid, vending machine always serve a drink.

Linear Temporal Logic

Linear Temporal Logic (LTL) Informally

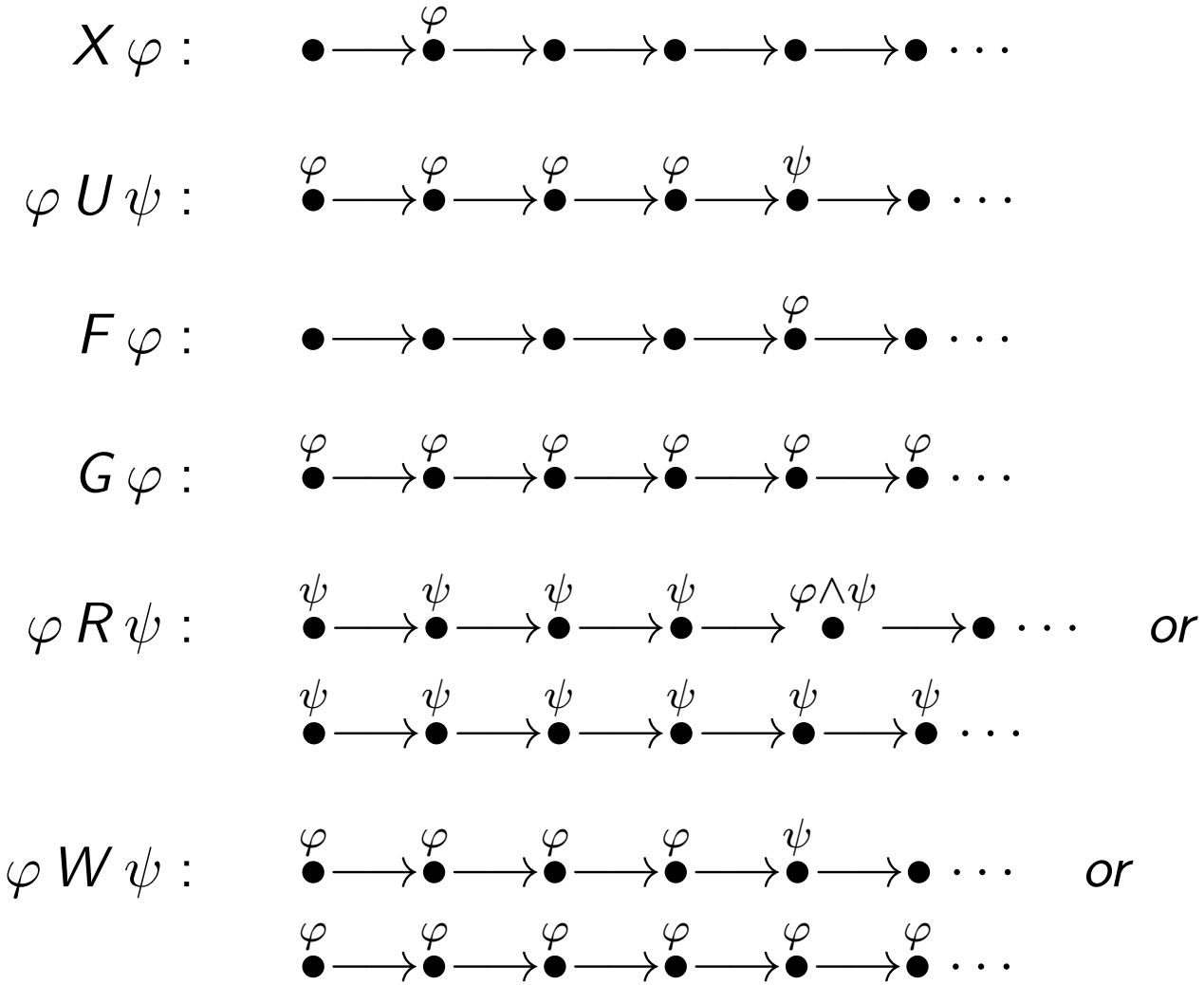
Formula φ

- Is evaluated on top of a single run of Kripke structure.
- Express validity of APs in the states along the given run.

Temporal Operators of LTL

- $F\,\varphi$ — φ holds true eventually (Future).
- $G\,\varphi$ — φ holds true all the time (Globally).
- $\varphi\,U\,\psi$ — φ holds true until eventually ψ holds true (Until).
- $X\,\varphi$ — φ is valid after execution of one transition (Next).
- $\varphi\,R\,\psi$ — ψ holds true until $\varphi \wedge \psi$ holds true (Release).
- $\varphi\,W\,\psi$ — until, but ψ may never become true (Weak Until).

Graphical Representation of LTL Temporal Operators



Syntax of LTL

Let AP be a set of atomic propositions.

- If $p \in AP$, then p is an LTL formula.
- If φ is an LTL formula, then $\neg\varphi$ is an LTL formula.
- If φ and ψ are LTL formulae, then $\varphi \vee \psi$ is an LTL formula.
- If φ is an LTL formula, then $X\,\varphi$ is an LTL formula.
- If φ and ψ are LTL formulae, then $\varphi\,U\,\psi$ is an LTL formula.

Alternatively

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\,\varphi \mid \varphi\,U\,\varphi$$

Syntactic shortcuts

Propositional Logic

- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$
- $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \Leftrightarrow \psi \equiv (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Temporal operators

- $F\,\varphi \equiv \mathit{true}\,U\,\varphi$
- $G\,\varphi \equiv \neg F\,\neg\varphi$
- $\varphi\,R\,\psi \equiv \neg(\neg\varphi\,U\,\neg\psi)$
- $\varphi\,W\,\psi \equiv \varphi\,U\,\psi \vee G\,\varphi$

Alternative syntax

- $F\varphi \equiv \diamond\varphi$
- $G\varphi \equiv \Box\varphi$
- $X\varphi \equiv \circ\varphi$

Models of LTL Formulae

Model of an LTL formula

- Let AP be a set of atomic propositions.
- Model of an LTL formula is a run π of Kripke structure.

Notation

- Let $\pi = s_0, s_1, s_2, \ldots$.
- Suffix of run π starting at s_k is denoted as $\pi^k = s_k, s_{k+1}, s_{k+2}, \ldots$.
- K-th state of the run, is referred to as $\pi(k) = s_k$.

Semantics of LTL

Assumptions

- Let AP be a set of atomic propositions.
- Let π be a run of Kripke structure $M = (S, T, I, s_0)$.
- Let φ, ψ be syntactically correct LTL formulae.
- Let $p \in AP$ denote atomic proposition.

Semantics

$$
\begin{aligned}
\pi \models p \quad &\text{iff} \quad p \in I(\pi(0)) \\
\pi \models \neg\varphi \quad &\text{iff} \quad \pi \not\models \varphi \\
\pi \models \varphi \vee \psi \quad &\text{iff} \quad \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models X\,\varphi \quad &\text{iff} \quad \pi^1 \models \varphi \\
\pi \models \varphi\, U\, \psi \quad &\text{iff} \quad \exists k.0 \leq k, \pi^k \models \psi \text{ and} \\
&\qquad\quad \forall i.0 \leq i < k, \pi^i \models \varphi
\end{aligned}
$$

Semantics of Other Temporal Operators

$$\pi \models F\,\varphi \quad \text{iff} \quad \exists k.k \geq 0, \pi^k \models \varphi$$

$$\pi \models G\,\varphi \quad \text{iff} \quad \forall k.k \geq 0, \pi^k \models \varphi$$

$$\begin{aligned}\pi \models \varphi\,R\,\psi \quad \text{iff} \quad & (\exists k.0 \leq k, \pi^k \models \varphi \wedge \psi \text{ and} \\ & \forall i.0 \leq i < k, \pi^i \models \psi) \\ & \text{or } (\forall k.k \geq 0, \pi^k \models \psi)\end{aligned}$$

$$\begin{aligned}\pi \models \varphi\,W\,\psi \quad \text{iff} \quad & (\exists k.0 \leq k, \pi^k \models \psi \text{ and} \\ & \forall i.0 \leq i < k, \pi^i \models \varphi) \\ & \text{or } (\forall k.k \geq 0, \pi^k \models \varphi)\end{aligned}$$

LTL Model Checking

Verification Employing LTL

- System is viewed as a set of runs.
- System is satisfies LTL formula if and only if all system runs satisfy the formula.
- In other words, any run violating the formula is a witness that the system does not satisfy the formula.

Lemma

- If a finite state system does not satisfy an LTL formula then this may be witnessed with a **lasso-shaped** run.
- Run π is lasso-shaped if $\pi = \pi_1 \cdot (\pi_2)^\omega$, where

  $\pi_1 = s_0, s_1, \ldots, s_k$

  $\pi_2 = s_{k+1}, s_{k+2}, \ldots, s_{k+n}$, where $s_k \equiv s_{k+n}$.
- Note that π^ω denotes infinite repetition of π.

Homework

Homework

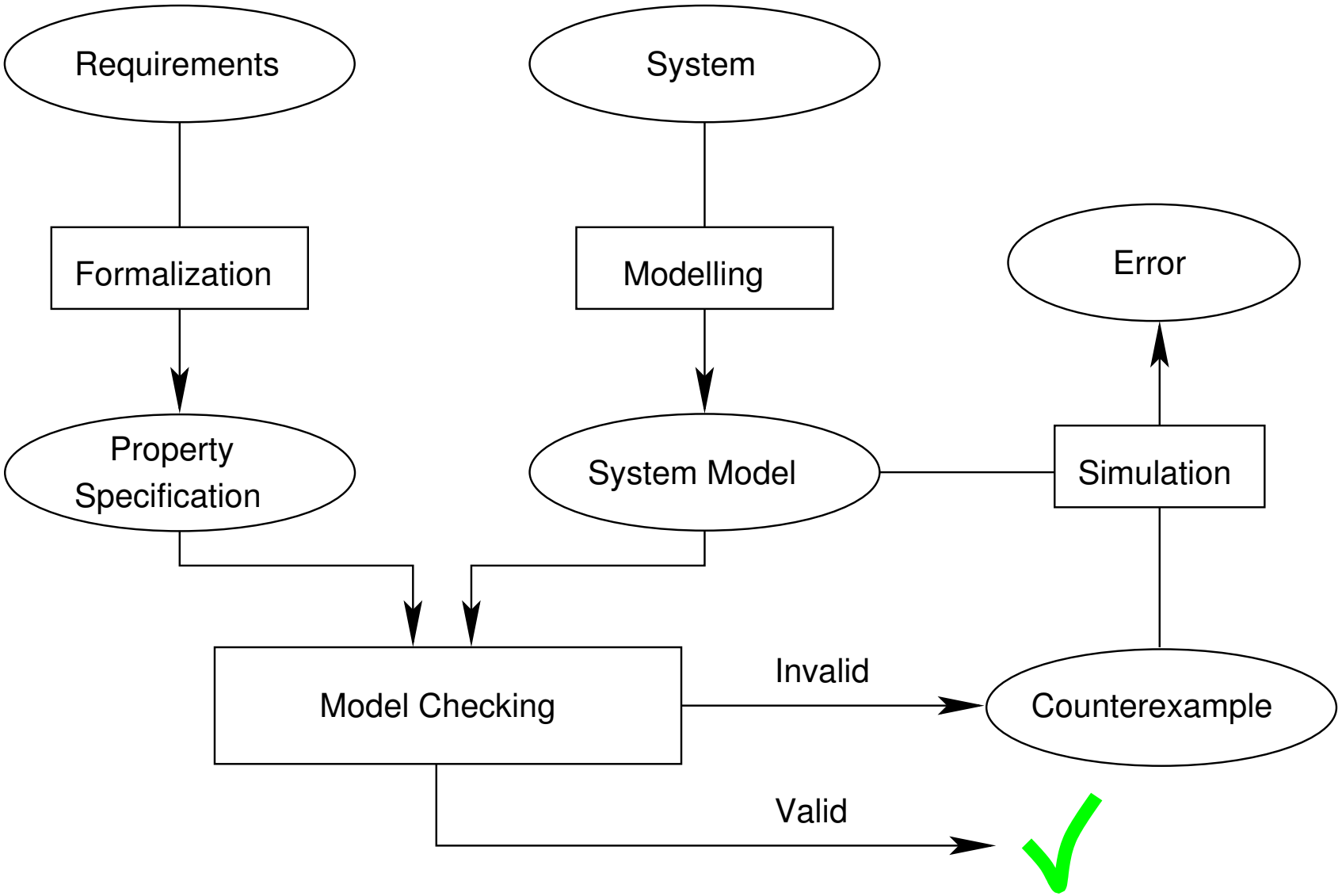
- Model Peterson's mutual exclusion protocol in ProMeLa.
- State expected LTL properties of Peterson's protocol.
- Verify them using SPIN model checker.

IA169 System Verification and Assurance

LTL Model Checking (continued)

Jiří Barnat

Model Checking – Schema



Where are we now?

Property Specification

- English text.
- Formulae of Linear Temporal Logic.

System Description

- Source code in programming language.
- Source code in modelling language.
- Kripke structure representing the state space.

Problem

- Kripke structure M
- LTL formula φ
- $M \models \varphi$?

Automata-Based Approach to LTL Model Checking

Languages of infinite words

Observation One

- System is a set of (infinite) runs.
- Also referred to as formal language of infinite words.

Observation Two

- Two different runs are equal with respect to an LTL formula if they agree in the interpretation of atomic propositions (need not agree in the states).
- Let $\pi = s_0, s_1, \ldots$, then $I(\pi) \stackrel{def}{\Longleftrightarrow} I(s_0), I(s_1), I(s_2), \ldots$.

Observation Three

- Every run either satisfies an LTL formula, or not.
- Every LTL formula defines a set of satisfying runs.

Reduction to Language Inclusion

Reformulation as Language Problem

- Let $\Sigma = 2^{AP}$ be an alphabet.
- Language L_{sys} of all runs of system M is defined as follows.

$$L_{sys} = \{I(\pi) \mid \pi \text{ is a run in } M\}.$$

- Language L_φ of runs satisfying φ is defined as follows.

$$L_\varphi = \{I(\pi) \mid \pi \models \varphi\}.$$

Observation

$$M \models \varphi \iff L_{sys} \subseteq L_\varphi$$

L_{sys} and L_φ expressed by Büchi automaton

Theorem

- For every LTL formula φ we can construct Büchi automaton A_φ such that $L_\varphi = L(A_\varphi)$.

[Vardi and Wolper, 1986]

Theorem

- For every Kripke structure $M = (S, T, I, s_0)$ we can construct Büchi automaton A_{sys} such that $L_{sys} = L(A_{sys})$.

Construction of A_{sys}

- Let AP be a set of atomic propositions.
- Then $A_{sys} = (S, 2^{AP}, s_0, \delta, S)$, where $q \in \delta(p, a)$ if and only if $(p, q) \in T \wedge I(p) = a$.

Where we are now?

Property Specification

- English text.
- Formulae φ of Linear Temporal Logic.
- Buchi automaton accepting L_φ.

System Description

- Source code in programming language.
- Source code in modelling language.
- Kripke structure M representing the state space.
- Buchi automaton accepting L_{sys}.

Problem Reformulation

- $M \models \varphi \iff L_{sys} \subseteq L_\varphi$

Reduction to Büchi Emptiness Problem

Notation

- $co{-}L$ denotes complement of L with respect to Σ^{AP}.

Lemma

- $co{-}L(A_\varphi) = L(A_{\neg\varphi})$ for every LTL formula φ.

Reduction of $M \models \varphi$ to the emptiness of $L(A_{sys} \times A_{\neg\varphi})$

- $M \models \varphi \iff L_{sys} \subseteq L_\varphi$
- $M \models \varphi \iff L(A_{sys}) \subseteq L(A_\varphi)$
- $M \models \varphi \iff L(A_{sys}) \cap co{-}L(A_\varphi) = \emptyset$
- $M \models \varphi \iff L(A_{sys}) \cap L(A_{\neg\varphi}) = \emptyset$
- $M \models \varphi \iff L(A_{sys} \times A_{\neg\varphi}) = \emptyset$

Synchronous Product of Büchi Automata

Theorem

- Let $A = (S_A, \Sigma, s_A, \delta_A, F_A)$ and $B = (S_B, \Sigma, s_B, \delta_B, F_B)$ be Büchi automata over the same alphabet Σ. Then we can construct Büchi automaton $A \times B$ such that $L(A \times B) = L(A) \cap L(B)$.

Construction of $A \times B$

- $A \times B =$
  $(S_A \times S_B \times \{0, 1\}, \Sigma, (s_A, s_B, 0), \delta_{A \times B}, F_A \times S_B \times \{0\})$
- $(p', q', j) \in \delta_{A \times B}((p, q, i), a)$ for all

$$
\begin{array}{ll}
p' \in \delta_A(p, a) & \\
q' \in \delta_B(q, a) & \\
j = (i + 1) \, mod \, 2 & \text{if } (i = 0 \wedge p \in F_A) \vee (i = 1 \wedge q \in F_B) \\
j = i & \text{otherwise}
\end{array}
$$

Synchronous Product of Büchi Automata – Simplification

Observation

- For the purpose of LTL model checking, we do not need general synchronous product of Büchi automata, since Büchi automaton A_{sys} is constructed in such a way that $F_A = S_A$, i.e. it has all states accepting.
- For such a special case the construction of product automata can be significantly simplified.

Construction of $A \times B$ when $F_A = S_A$

- $A \times B = (S_A \times S_B, \Sigma, (s_A, s_B), \delta_{A \times B}, S_A \times F_B)$
- $(p', q') \in \delta_{A \times B}((p, q), a)$ for all
  $p' \in \delta_A(p, a)$
  $q' \in \delta_B(q, a)$

Reduction to Accepting Cycle Detection

Observation

- Any finite automaton may visit accepting state infinitely many times only if it contains a cycle through that accepting state.

Decision Procedure for $M \models \varphi$?

- Build a product automaton $(A_{sys} \times A_{\neg\varphi})$.
- Check the automaton for presence of an accepting cycle.
- If there is a reachable accepting cycle then $M \not\models \varphi$.
- Otherwise $M \models \varphi$.

Detection of Accepting Cycles

Detection of Accepting Cycles

Reachability in Directed Graph

- Depth-first or breadth-first search algorithm.
- $\mathcal{O}(|V| + |E|)$.

Algorithmic Solution to Accepting Cycle Detection

- Compute the set of accepting states in time $\mathcal{O}(|V| + |E|)$.
- Detect self-reachability for every accepting state in $\mathcal{O}(|F|(|V| + |E|))$.
- Overall time $\mathcal{O}(|V| + |E| + |F|(|V| + |E|))$.

Can we do better?

- Yes, with **Nested DFS** algorithm in $\mathcal{O}(|V| + |E|)$.

Depth-First Search Procedure

```
proc Reachable(V, E, v0)
  Visited = ∅
  DFS(v0)
  return (Visited)
end

proc DFS(vertex)
  if vertex ∉ Visited
    then /* Visits vertex */
      Visited := Visited ∪ {vertex}
      foreach { v | (vertex, v) ∈ E } do
        DFS(v)
      od
      /* Backtracks from vertex */
  fi
```

Colour Notation in DFS

Observation

- When running DFS on a graph all vertices can be classified into one of the three following categories (denoted with colours).

Colour Notation for Vertices

- White vertex – Has not been visited yet.
- Gray vertex - Visited, but yet not backtracked.
- Black vertex - Visited and backtracked.

Recursion Stack

- Gray vertices form a path from the initial vertex to the vertex that is currently processed by the outer procedure.

Properties of DFS, $G = (V, E)$ a $v_0 \in V$

Observation

- If two distinct vertices v_1, v_2 satisfy that
  - $(v_0, v_1) \in E^*$,
  - $(v_1, v_1) \notin E^+$,
  - $(v_1, v_2) \in E^+$.
- Then procedure $DFS(v_0)$ backtracks from vertex v_2 before it backtracks from vertex v_1.

DFS post-order

- If $(v, v) \notin E^+$ and $(v_0, v) \in E^*$, then upon the termination of sub-procedure $DFS(v)$, called within procedure $DFS(v_0)$, all vertices u such that $(v, u) \in E^+$ are visited and backtracked.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Observation

- If a sub-graph reachable from a given accepting vertex does not contain accepting cycle, then no accepting cycle starting in an accepting state outside the sub-graph can reach the sub-graph.

The Key Idea

- Execute the inner procedures in a bottom-up manner.
- The inner procedures are called in the same order in which the outer procedure backtracks from accepting states, i.e. the ordering of calls follows a DFS post-order.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

```
proc Detection_of_accepting_cycles
  Visited := ∅
  DFS(v0)
end

proc DFS(vertex)
  if (vertex) ∉ Visited
    then Visited := Visited ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      DFS(s)
    od
    if IsAccepting(vertex)
      then DetectCycle(vertex)
    fi
  fi
end
```

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Assumption On Early Termination

- The inner procedure reports the accepting cycle and terminates the whole algorithm if called for an accepting vertex that lies on an accepting cycle.

Consequences

- If the inner procedure called for an accepting vertex v reports no accepting cycle, then there is no accepting cycle in the graph reachable from vertex v.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Linear Complexity of Nested DFS Algorithm

- Employing DFS post-order it follows that vertices that have been visited by previous invocation of inner procedure may be safely skipped in any later invocation of the inner procedure.

$\mathcal{O}(|V| + |E|)$ **Algorithm**

1) Nested procedures are called in DFS post-order as given by the outer procedure, and are limited to vertices not yet visited by inner procedure.
2) All vertices are visited at most twice.

Detecting Cycles in Inner Procedures

Theorem

- If the immediate successor to be processed by an inner procedure is grey (on the stack of the outer procedure), then the presence of an accepting cycle is guaranteed.

Application

- It is enough to reach a vertex on the stack of the outer procedure in the inner procedure in order to report the presence of an accepting cycle.

$\mathcal{O}(|V| + |E|)$ Algorithm

```
proc Detection_of_accepting_cycles
  Visited := Nested := in_stack := ∅
  DFS(v0)
  Exit("Not Present")
end

proc DFS(vertex)
  if (vertex) ∉ Visited
    then Visited := Visited ∪ {vertex}
    in_stack := in_stack ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      DFS(s)
    od
    if IsAccepting(vertex)
      then DetectCycle(vertex)
    fi
    in_stack := in_stack \ {vertex}
  fi
end

proc DetectCycle (vertex)
  if vertex ∉ Nested
    then Nested := Nested ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      if s ∈ in_stack
        then WriteOut(in_stack)
          Exit("Present")
        else DetectCycle(s)
      fi
    of
  fi
end
```

Time and Space Complexity

Outer Procedure

- Time: $\mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V|)$

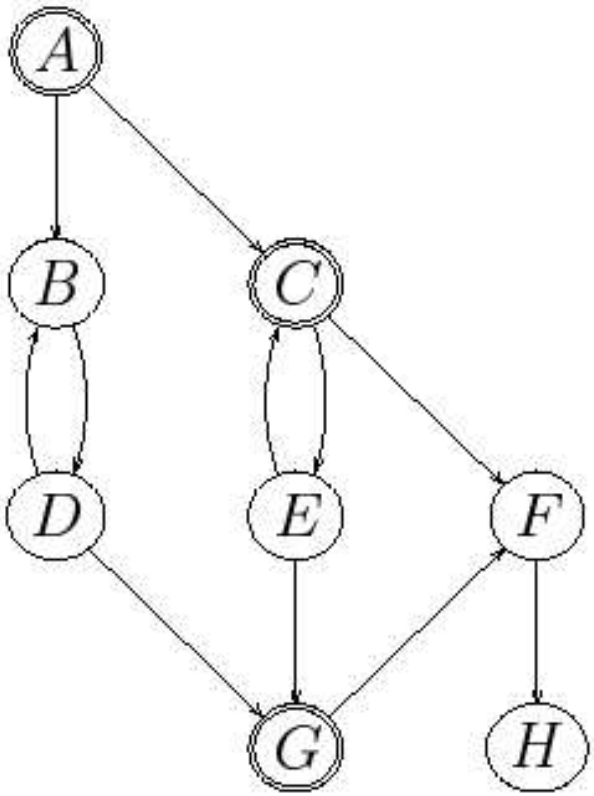
Inner Procedures

- Time (overall): $\mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V|)$

Complexity

- Time: $\mathcal{O}(|V| + |E| + |V| + |E|) = \mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V| + |V|) = \mathcal{O}(|V|)$

Nested DFS – Example



- 1st DFS: A,B,D,B,G,F,H,H,F,G
  1st DFS stack: A,B,D,G
  visited: A,B,D,F,G,H / –
- 2nd DFS: G,F,H,H,F,G
  visited: A,B,D,F,G,H / F,G,H
- 1st DFS: G,D,B,C,E,C,G,E,F,C
  1st DFS stack: A,C
  visited: all / F,G,H
- 2nd DFS: C,E,C
  counterexample: A,C,E,C

visited state backtrack non-accepting state backtrack accepting state

Classification of Büchi Automata

Sub-Classes of Büchi Automata

Terminal Büchi Automata

- All accepting cycles are self-loops on accepting states labelled with `true`.

Weak Büchi Automata

- Every strongly connected component of the automaton is composed either of accepting states, or of non-accepting states.

Impact on Verification Procedure

Automaton $A_{\neg\varphi}$

- For a number of LTL formulae φ is $A_{\neg\varphi}$ terminal or weak.
- $A_{\neg\varphi}$ is typically quite small.
- Type of $A_{\neg\varphi}$ can be pre-computed prior verification.
- Types of components of $A_{\neg\varphi}$
  - Non-accepting – Contains no accepting cycles.
  - Strongly accepting – Every cycle is accepting.
  - Partially accepting – Some cycles are accepting and some are not.

Product Automaton

- The graph to be analysed is a graph of product automaton $A_S \times A_{\neg\varphi}$.
- Types of components of $A_S \times A_{\neg\varphi}$ are given by the corresponding components of $A_{\neg\varphi}$.

Impact on Verification Procedure – Terminal BA

$A_{\neg\varphi}$ is terminal Büchi automaton

- For the proof of existence of accepting cycle it is enough to proof reachability of any state that is accepting in $A_{\neg\varphi}$ part.
- Verification process is reduced to the reachability problem.

„Safety" Properties

- Those properties φ for which $A_{\neg\varphi}$ is a terminal BA.
- Typical phrasing: „Something bad never happens."
- Reachability is enough to proof the property.

Impact on Verification Procedure – Weak BA

$A_{\neg\varphi}$ is weak Büchi automaton

- Contains no partially accepting components.
- For the proof of existence of accepting cycle it is enough to proof existence of reachable cycle in a strongly accepting component.
- Can be detected with a single DFS procedure.
- Time-optimal algorithm exists that does not rely on DFS.

„Weak" LTL Properties

- Those properties φ for which $A_{\neg\varphi}$ is a weak BA.
- Typically, responsiveness: $G\,(a \implies F\,(b))$.

Classification of LTL Properties

Classification

- Every LTL formula belongs to one of the following classes:
  Reactivity, Recurrence, Persistance, Obligation, Safety, Guarantee

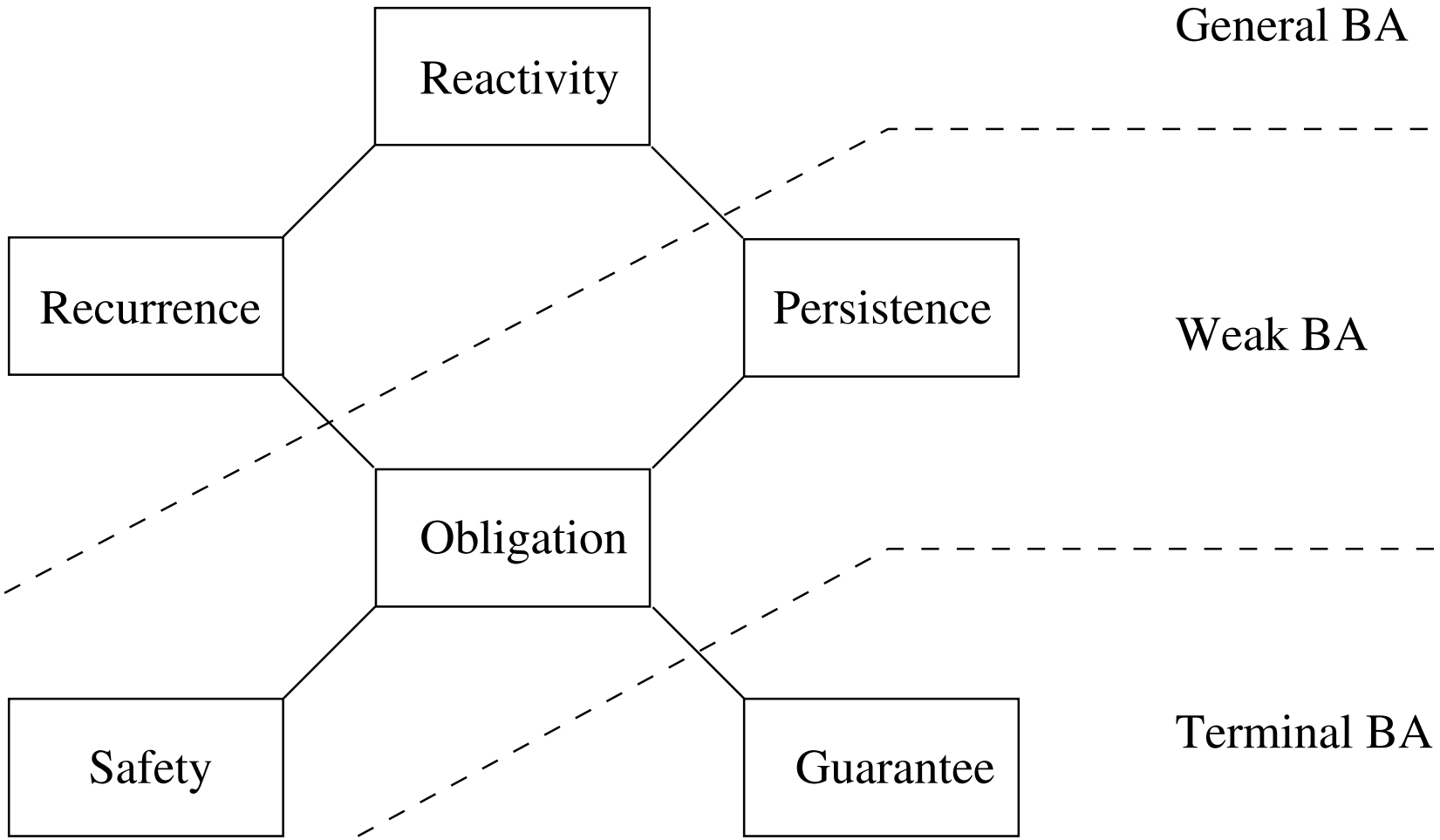
Interesting Relations

- Guarantee class properties can be described with a terminal Büchi automaton.
- Persistance, Obligation, and Safety class properties can be described with a weak Büchi automaton.

Negation in Verification Process ($\varphi \mapsto A_{\neg\varphi}$)

- $\varphi \in$ Safety $\iff$ $\neg\varphi \in$ Guarantee.
- $\varphi \in$ Recurrence $\iff$ $\neg\varphi \in$ Persistance.

Classification of LTL Properties



Fighting State Space Explosion

State Space Explosion Problem

What is State Space Explosion

- System is usually given as a composition of parallel processes.
- Depending on the order of execution of actions of parallel processes various intermediate states emerge.
- The number of reachable states may be up to exponentially larger than the number of lines of code.

Consequence

- Main memory cannot store all states of the product automaton as they are too many.
- Algorithms for accepting cycle detection suffer for lack of memory.

Some Methods to Fight State Space Explosion

State Compression

- Lossless compression.
- Lossy compression – Heuristics.

On-The-Fly Verification

Symbolic Representation of State Space

Reduced Number of States the Product Automaton

- Introduction of atomic blocks.
- Partial order on execution of process actions.
- Avoid exploration of symmetric parts.

Parallel and Distributed Verification

On-The-Fly Verification

Observation

- Product automaton graph is defined implicitly with:
  - $|F|\_init()$ — Returns initial state of automaton.
  - $|F|\_succs(s)$ — Gives immediate successors of a given state.
  - $|Accepting|(s)$ — Gives whether a state is accepting or not.

On-The-Fly Verification

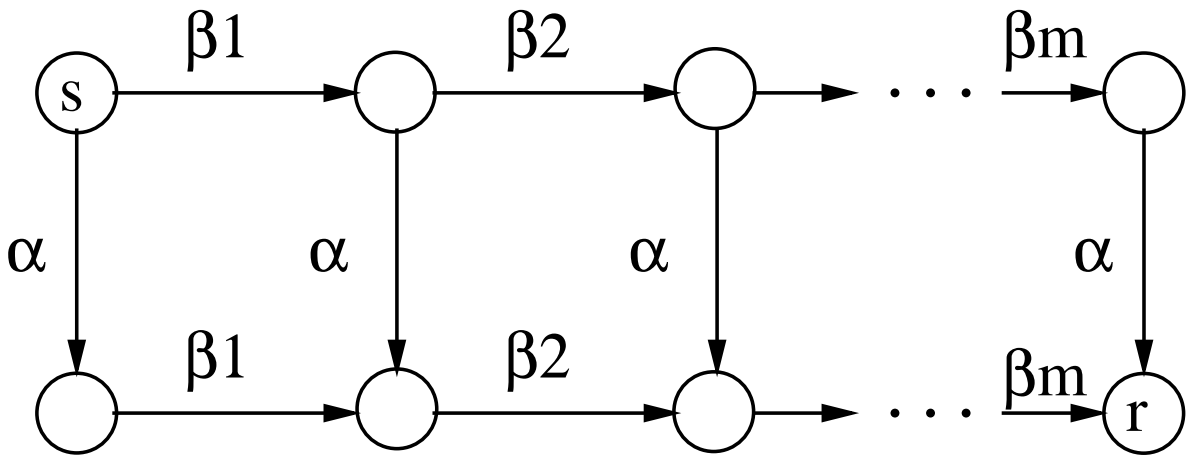
- Some algorithms may detect the presence of accepting cycle without the need of complete exploration of the graph.
- Hence, $\mathcal{M} \models \varphi$ can be decided without the full construction of $A_{sys} \times A_{\neg\varphi}$.
- This is referred to as to on-the-fly verification.

Partial Order Reduction

Example

- Consider a system made of processes A and B.
- A can do a single action α, while B is a sequence of actions β, e.g. $\beta_1, \ldots, \beta_m$.

Unreduced State Space:

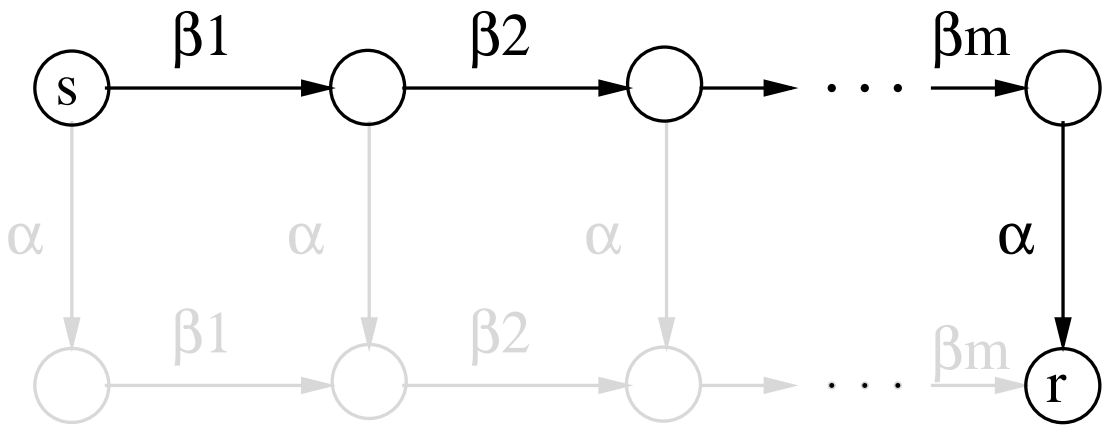


Property to be verifed: Reachability of state r.

Partial Order Reduction

Observation

- Runs $(\alpha\beta_1\beta_2 \ldots \beta_m)$, $(\beta_1\alpha\beta_2 \ldots \beta_m)$, $\ldots$, $(\beta_1\beta_2 \ldots \beta_m\alpha)$ are equivalent with respect to the property.
- It is enough to consider only a representative from the equivalence class, say, e.g. $(\beta_1\beta_2 \ldots \beta_m\alpha)$.



- The representative is obtained by postponing of action α.

Partial Order Reduction

Reduction Principle

- Do not consider all immediate successor during state space exploration, but pick carefully only some of them.
- Some states are never generated, which results in a smaller state space graph.

Technical Realisation

- To pick correct but optimal subset of successors is as difficult as to generate the whole state space. Hence, heuristics are used.
- The reduced state space must contain an accepting cycle if and only if the unreduced state space does so.
- LTL formula must not use X operator (subclass of LTL).

Distributed and Parallel Verification

Principle

- Employ aggregate power of multiple CPUs.
- Increased memory and computing power.

Problem of Nested DFS

- Typical implementation relies on hashing mechanism, hence, the main memory is accessed extremely randomly. Should memory demands exceeds the amount of available memory, **thrashing** occurs.
- Mimicking serial Nested DFS algorithm in a distributed-memory setting is extremely slow. (Token-based approach).
- It unknown whether the DFS post-order can be computed by a time-optimal scale-able parallel algorithm (Still an open problem.)

Parallel Algorithms for Distributed-Memory Setting

Observation

- Instead of DFS other graph procedures are used.
- Tasks such as breadth-first search, or value propagation can be efficiently computed in parallel.
- Parallel algorithms do not exhibit optimal complexity.

	Complexity	Optimal	On-The-Fly
Nested DFS	O(V+E)	Yes	Yes
OWCTY			
general Büchi automata	O(V.(V+E))	No	No
weak Büchi automata	O(V+E)	Yes	No
MAP	O(V.V.(V+E))	No	Partially
OWCTY+MAP			
general Büchi automata	O(V.(V+E))	No	Partially
weak Büchi automata	O(V+E)	Yes	Partially

Model Checking – Summary

Decision Procedure and State Space Explosion

Properties Validity

- Property to be verified may be violated by a single particular (even extremely unlikely) run of the system under inspection.
- The decision procedure relies on exploration of state space graph of the system.

State Space Explosion

- Unless thee are other reasons, all system runs have to be considered.
- The number of states, that system can reach is up to exponentially larger than the size of the system description.
- Reasons: Data explosion, asynchronous parallelism.

Advantages of Model Checking

General Technique

- Applicable to Hardware, Software, Embedded Systems, Model-Based Development, ...

Mathematically Rigorous Precision

- The decision procedure results with $\mathcal{M} \models \varphi$, if and only if, it is the case.

Tool for Model Checking – Model Checkers

- The so called "Push-Button" Verification.
- No human participation in the decision process.
- Provides users with witnesses and counterexamples.

Disadvantages of Model Checking

Not Suitable for Everything

- Not suitable to show that a program for computing factorial really computes $n!$ for a given n.
- Though it is perfectly fine to check that for a value of 5 it always returns the value of 120.

Often Relies on Modelling

- Need for model construction.
- Validity of a formula is guaranteed for the model, not the modelled system.

Size of the State Space

- Applicable mostly to system with finite state space.
- Due to state space explosion, practical applicability is limited.

Verifies Only What Has Been Specified

- Issues not covered with formulae need not be discovered.

Practicals and Homework – 06

Homework

- Analysis with DIVINE model checker on a more complex example (some homework from previous course on secure coding).

IA169 System Verification and Assurance

CTL Model Checking

Jiří Barnat

Liner vs. Branching Time

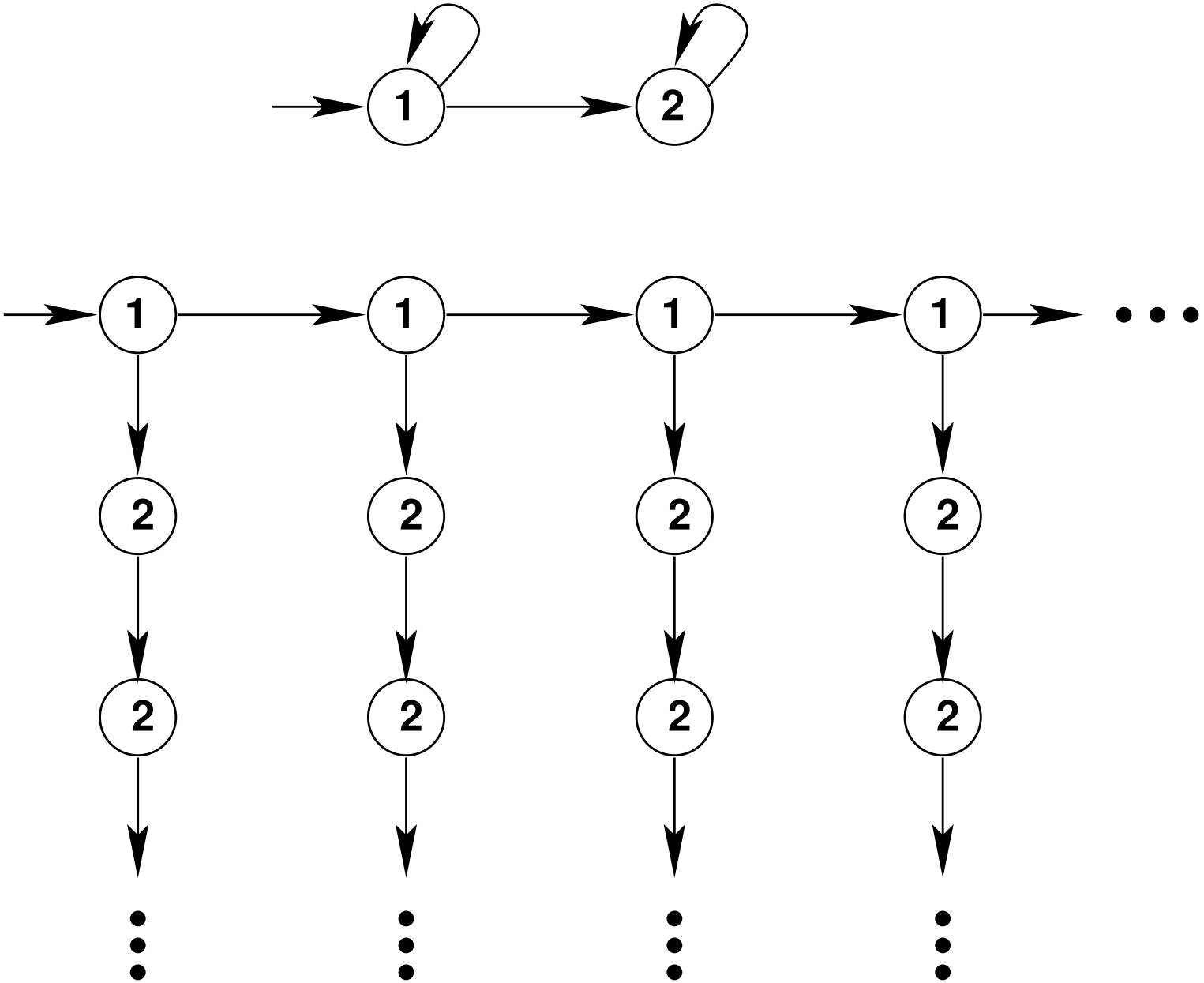
Pnueli, 1977

- System is viewed as a set of state sequences — **Runs**.
- System properties are given as properties of runs,
- ... and can be described with a linear-time logic.

Clarke & Emerson, 1980

- System is viewed as a branching structure of possible executions from individual system states — **Computation Tree**.
- System properties are given as properties of the tree,
- ... and can be described with a branching-time logic.

System and Computation Tree



Computation Tree Logic (CTL)

CTL Informally

Possible Future Computations

- For a given node of a computation tree, the sub-tree rooted in the given node describes all possible runs the system can still take.
- Every such a run is possible future computation.

CTL Formulae Allow For

- Specification of state qualities with atomic propositions.
- Quantify over possible future computations.
- Restrict the set of possible future computations with (quantified) LTL operators.

Example

- $\varphi \equiv EF(a)$
- It is possible to take a future computation such that a will hold true in the computation eventually.

Syntax of CTL

Let AP by a set of atomic propositions.

- If $p \in AP$, then p is a CTL formula.
- If φ is a CTL formula, then $\neg\varphi$ is a CTL formula.
- If φ and ψ are CTL formulae, then $\varphi \vee \psi$ is a CTL formula.
- If φ is a CTL formula, then $EX\,\varphi$ is a CTL formula.
- If φ and ψ are CTL formulae, then $E[\varphi\,U\,\psi]$ is a CTL formula.
- If φ and ψ are CTL formulae, then $A[\varphi\,U\,\psi]$ is a CTL formula.

Alternatively (Backus-Naur Form)

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\,U\,\varphi] \mid A[\varphi\,U\,\varphi]$$

Syntactic Shortcuts

Already Known

- The standard shortcuts from the propositional logic.
- Syntactic shortcuts from LTL
  - $F\,\varphi \equiv true\,U\,\varphi$
  - $G\,\varphi \equiv \neg F\,\neg\varphi$

Deduced CTL Operators

- $EF\,\varphi \equiv E[true\,U\,\varphi]$
- $AF\,\varphi \equiv A[true\,U\,\varphi]$
- $EG\,\varphi \equiv \neg AF\,\neg\varphi$
- $AG\,\varphi \equiv \neg EF\,\neg\varphi$
- $AX\,\varphi \equiv \neg EX\,\neg\varphi$

Models of CTL formulae

Model of a CTL formula

- Let AP be a set of atomic propositions.
- Model of a CTL formula is a state $s \in S$ of Kripke structure $M = (S, T, I, s_0)$.

Reminder

- Run of a Kripke structure is maximal path starting at the initial state of the structure.
- Finite maximal paths are viewed as infinite runs due to infinite repetition of the last state on the path.

Notation

- Let $s \in S$ be a state of Kripke structure $M = (S, T, I, s_0)$.
- $P_M(s) = \{\pi \mid \pi \text{ is a run initiated at state } s\}$

Semantics of CTL

Assumptions

- Let AP be a set of atomic propositions.
- Let $p \in AP$ be an atomic proposition.
- Let $s \in S$ be a state of Kripke structure $M = (S, T, I, s_0)$.
- Let φ, ψ denote syntactically correct CTL formulae.

Semantics

$$
\begin{aligned}
s \models p \quad &\text{iff} \quad p \in I(s) \\
s \models \neg\varphi \quad &\text{iff} \quad \neg(s \models \varphi) \\
s \models \varphi \vee \psi \quad &\text{iff} \quad s \models \varphi \text{ or } s \models \psi \\
s \models EX\,\varphi \quad &\text{iff} \quad \exists \pi \in P_M(s).\pi(1) \models \varphi \\
s \models E[\varphi\,U\,\psi] \quad &\text{iff} \quad \exists \pi \in P_M(s).(\exists k \geq 0.(\pi(k) \models \psi \text{ and} \\
&\qquad\qquad \forall 0 \leq i < k.\pi(i) \models \varphi)) \\
s \models A[\varphi\,U\,\psi] \quad &\text{iff} \quad \forall \pi \in P_M(s).(\exists k \geq 0.(\pi(k) \models \psi \text{ and} \\
&\qquad\qquad \forall 0 \leq i < k.\pi(i) \models \varphi))
\end{aligned}
$$

Task

Atomic Propositions

- AP=$\{a, b, Req, Ack, Restart\}$

Express with CTL Formulae

- A state where a is true, but b is not, is reachable.
- Whenever system receives a request Req, it generates acknowledgement Ack eventually.
- In every run there are infinitely many b's.
- There is always an option to reset the system (reach state $Restart$).

Model Checking CTL

Problem Statements

Model Checking CTL

- Let $M = (S, T, I, s_0)$ be a Kripke structure.
- Let φ be a CTL formula.
- Does initial state of M satisfies φ?

Alternatively

- Let $M = (S, T, I, s_0)$ be a Kripke structure.
- Let φ be a CTL formula.
- Compute a set of states of M satisfying φ.

Above mentioned approaches are also referred to as to

- Local model checking problem — $M, s_0 \models \varphi$.
- Global model checking problem — $\{s \mid M, s \models \varphi\}$.

Algorithm for CTL Model Checking — Idea

Observation

- If the validity of formulae φ and ψ is known for all states, it is easy to deduce validity of formulae $\neg\varphi$, $\varphi \vee \psi$, $EX\,\varphi$, ...

CTL Model Checking – Sketch

- Let $M = (S, T, I)$ be a Kripke structure and φ a CTL Formula.
- A labelling function $label : S \to 2^{2^{\varphi}}$ is computed such that it gives validity of all sub-formulae of φ for all states of Kripke structure M.
- Obviously, $s_0 \models \varphi \iff \varphi \in label(s_0)$.
- Function *label* is computed gradually for individual sub-formulae of φ, starting with the simplest sub-formula and proceeding towards more complex sub-formulae, ending with φ itself.

Sub-formulae of a CTL Formula

Sub-formulae of formula φ

- Let φ be a CTL formula.
- The set of all sub-formulae of formula φ is denoted by 2^{φ}.
- 2^{φ} is defined inductively according to the structure of φ.

Inductive Definition of 2^{φ}

1) $\varphi \in 2^{\varphi}$ (φ is a sub-formula of φ)
2) If $\eta \in 2^{\varphi}$ and
   - $\eta \equiv \neg\psi$, then $\psi \in 2^{\varphi}$
   - $\eta \equiv \psi_1 \vee \psi_2$, then $\psi_1, \psi_2 \in 2^{\varphi}$
   - $\eta \equiv EX\,\psi$, then $\psi \in 2^{\varphi}$
   - $\eta \equiv E[\psi_1\, U\, \psi_2]$, then $\psi_1, \psi_2 \in 2^{\varphi}$
   - $\eta \equiv A[\psi_1\, U\, \psi_2]$, then $\psi_1, \psi_2 \in 2^{\varphi}$
3) Nothing else.

Equivalent Existential Form of CTL

Observation

- It is easier to prove validity of existential quantified modal operators than validity of universally quantified ones.
- For the purpose of verification of CTL-specified properties, it is possible to express the CTL formula in an equivalently expressive existential form of CTL.

Equivalent CTL Syntax

- $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\,U\,\varphi] \mid EG\,\varphi$

Task

- Express formula $EG\,\varphi$ in the original syntax of CTL.
- Give accordingly modified definition of the set of sub-formulae of φ for the above mentioned equivalent syntax.

Algorithm for CTL Model-Checking

INPUT: Kripke structure $M = (S, T, I, s_0)$, CTL formula φ.
OUTPUT: *True*, if $s_0 \models \varphi$; *False* otherwise.

proc CTLMC(φ, M)
  label := I
  Solved := $AP \cap 2^{\varphi}$
  while $\varphi \notin$ Solved do
    foreach ($\eta \in \{\neg\psi_1, \psi_1 \vee \psi_2, EX\,\psi_1, E[\psi_1\,U\,\psi_2], EG\,\psi_1 \mid \psi_1, \psi_2 \in Solved\}$)do
      if ($\eta \in 2^{\varphi}$ and $\eta \notin$ Solved)
        then *label* := *updateLabel*(η, *label*, M)
          Solved := Solved $\cup \{\eta\}$
      fi
    od
  od
  return ($\varphi \in$ *label*(s_0))
end

proc updateLabel(η, *label*, M)
  if ($\eta \equiv E[\psi_1 \, U \, \psi_2]$)
    then return *checkEU*(ψ_1, ψ_2, *label*, M)
  fi
  if ($\eta \equiv EG \, \psi$)
    then return *checkEG*(ψ, *label*, M)
  fi
  foreach (s $\in S$)do
    if ($\eta \equiv \neg\psi$ and $\psi \notin$ *label*(s)) or
      ($\eta \equiv \psi_1 \vee \psi_2$ and ($\psi_1 \in$ *label*(s) $\vee \, \psi_2 \in$ *label*(s))) or
      ($\eta \equiv EX \, \psi$ and ($\exists t \in \{t \mid (s,t) \in T\}$ such that $\psi \in$ *label*(t)))
      then *label*(s) := *label*(s) $\cup \{\eta\}$
    fi
  od
  return *label*
end

Algorithm for CTL Model-Checking $E[\psi_1 \, U \, \psi_2]$

INPUT: Kripke structure $M = (S, T, I)$,
Labelling function $label : S \to 2^{\varphi}$, correct w.r.t validity of ψ_1 and ψ_2
OUTPUT: Labelling function $label : S \to 2^{\varphi}$, correct w.r.t $E[\psi_1 \, U \, \psi_2]$

```
proc checkEU(ψ1, ψ2, label, M)
  Q := {s | ψ2 ∈ label(s)}
  foreach ( s ∈ Q)do
    label(s) := label(s) ∪ {E[ψ1 U ψ2]}
  od
  while (Q ≠ ∅) do
        choose s ∈ Q
        Q := Q ∖ {s}
        foreach ( t ∈ {t | T(t,s)}) do          /* all immediate predecessors */
          if (E[ψ1 U ψ2] ∉ label(t) ∧ ψ1 ∈ label(t))
            then label(t) := label(t) ∪ {E[ψ1 U ψ2]}
                 Q := Q ∪ {t}
          fi
        od
  od
  return label
end
```

Strongly Connected Components

Sub-graph

- Let $G = (V, E)$ be a graph, ie. $E \subseteq V \times V$.
- Graph $G' = (V', E')$ is called sub-graph of G if it holds that $V' \subseteq V$ and $E' = E \cap V' \times V'$.

Sub-graph $C = (V', E')$ of $G = (V, E)$ is called

- Strongly Connected Component, if $\forall u, v \in V'$ it holds that $(u, v) \in E'^*$ and $(v, u) \in E'^*$.
- Maximal Strongly Connected Component (SCC), if C is strongly connected component and for every $v \in (V \setminus V')$ it is the case that $(V' \cup \{v\}, E \cap (V' \cup \{v\} \times V' \cup \{v\}))$ is not.
- Non-trivial SCC, if C is Strongly Connected Component and $E' \neq \emptyset$.

Algorithm for CTL Model-Checking $EG\,\psi$

INPUT: Kripke structure $M = (S, T, I, s_0)$,
Labelling function $label : S \to 2^{\varphi}$, correct w.r.t. ψ
OUTPUT: Labelling function $label : S \to 2^{\varphi}$, correct w.r.t. $EG\,\psi$

proc checkEG(ψ, $label$, M)
  S' := {s | $\psi \in label$(s)}
  SCC := {C | C is non-trivial SCC $G' = (S', T \cap S' \times S')$}
  Q := $\bigcup_{C \in SCC} \{s \mid s \in C\}$
  foreach (s ∈ Q)do
    $label$(s) := $label$(s) ∪ {$EG\,\psi$}
  od
  while Q ≠ ∅ do
      choose s ∈ Q
      Q := Q ∖ {s}
      foreach (t ∈ ($S' \cap \{t \mid T(t, s)\}$))do /* *all immediate predecessors in S'* */
        if $EG\,\psi \notin label$(t)
          then $label$(t) := $label$(t) ∪ {$EG\,\psi$}
               Q := Q ∪ {t}
        fi
      od
  od
end

Complexity of Algorithm for CTL Model Checking

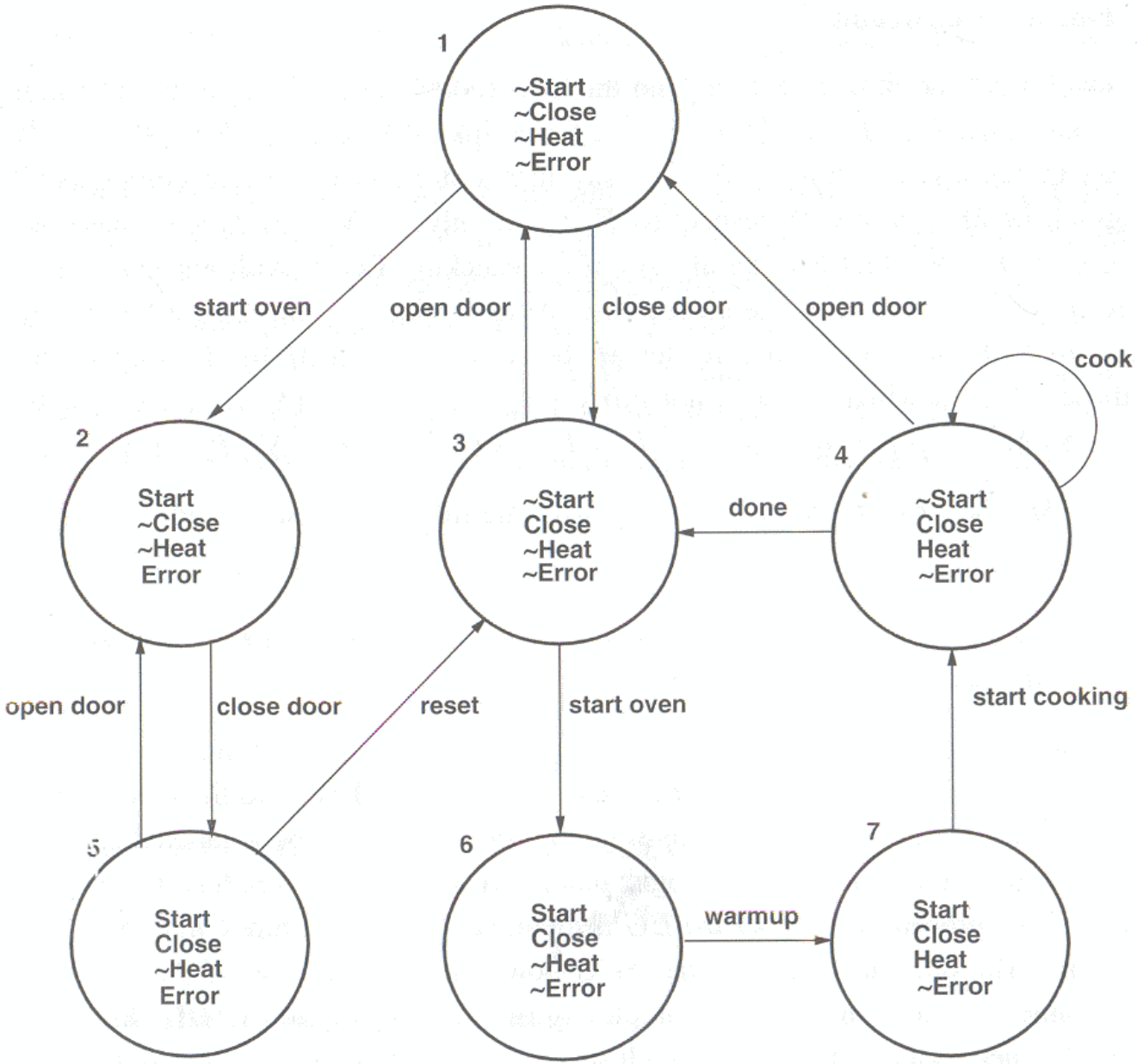
Observation

- Every CTL formula φ is made of at most $\mid \varphi \mid$ sub-formulae.
- Decomposition of every sub-graph of $G = (S, T)$ into SCCs can be done in time $\mathcal{O}(\mid S \mid + \mid T \mid)$.
- Every call to *updateLabel* terminates in time $\mathcal{O}(\mid S \mid + \mid T \mid)$.

Overall complexity

- Algorithm *CTLMC* exhibits $O(\mid \varphi \mid\mid S \mid)$ space and $O(\mid \varphi \mid (\mid S \mid + \mid T \mid))$ time complexity.

Example: Microwave oven $AG(Start \implies AF(Heat))$



Example: Microwave oven $AG(Start \implies AF(Heat))$

Transformation of formula $\varphi \equiv AG(Start \implies AF(Heat))$

- $AG(Start \implies AF(Heat))$
- $AG(\neg(Start \wedge \neg AF(Heat)))$
- $AG(\neg(Start \wedge EG(\neg Heat)))$
- $\neg EF(Start \wedge EG(\neg Heat))$
- $\neg E[true\ U\ (Start \wedge EG(\neg Heat))]$

Validity of sub-formulae $[S(\varphi) = \{s \mid s \models \varphi\}]$

- $S(Start) = \{2, 5, 6, 7\}$
- $S(Heat) = \{4, 7\}$
- $S(\neg Heat) = \{1, 2, 3, 5, 6\}$
- $S(EG(\neg Heat)) = \{1, 2, 3, 5\}$
- $S(Start \wedge EG(\neg Heat)) = \{2, 5\}$
- $S(E[true\ U\ (Start \wedge EG(\neg Heat))]) = \{1, 2, 3, 4, 5, 6, 7\}$
- $S(\neg E[true\ U\ (Start \wedge EG(\neg Heat))]) = \emptyset$

CTL*

CTL* as Extension of CTL

Observation

- Every use of temporal operator in a formula of CTL must be immediately preceded with a quantifier, i.e. use of a modal operator without quantification is not possible.

Logic CTL*

- Branching time logic.
- Similar to CTL.
- Unlike CTL, allows for standalone use of modal operators.

Example

- $A[p \wedge X(\neg p)]$ is CTL*, but is not CTL formula.

Syntax of CTL*

Types of CTL* formulae

- Quantifiers E and A are standalone operators in syntax construction rules. As a result there are two types of formulae in *CTL*: **path** and **state** formulae.
- Application of E and A operators on a path formula (formula of which model is a run of Kripke structure) results in a state formula (formula of which model is a state of Kripke structure)

Syntax of CTL*

state formula	$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid E\,\psi$
path formula	$\psi ::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid X\,\psi \mid \psi\,U\,\psi$

Semantics of CTL*

Assumption

- Let AP be a set of atomic propositions, and $p \in AP$.
- Let $M = (S, T, I)$ be a Kripke structure.
- Let φ_i denote CTL* state formulae, and ψ_i denote CTL* state formulae.

Semantics

$$
\begin{array}{rcl}
M, s \models p & \text{iff} & p \in I(s) \\
M, s \models \neg\varphi_1 & \text{iff} & \neg(M, s \models \varphi_1) \\
M, s \models \varphi_1 \vee \varphi_2 & \text{iff} & M, s \models \varphi_1 \text{ or } M, s \models \varphi_2 \\
M, s \models E\,\psi_1 & \text{iff} & \exists \pi \in P_M(s).\pi \models \psi_1 \\
M, \pi \models \varphi_1 & \text{iff} & M, \pi(0) \models \varphi_1 \\
M, \pi \models \neg\psi_1 & \text{iff} & \neg(M, \pi \models \psi_1) \\
M, \pi \models \psi_1 \vee \psi_2 & \text{iff} & M, \pi \models \psi_1 \text{ or } M, \pi \models \psi_2 \\
M, \pi \models X\,\psi_1 & \text{iff} & M, \pi^1 \models \psi_1 \\
M, \pi \models \psi_1\,U\,\psi_2 & \text{iff} & \exists k \geq 0.(M, \pi^k \models \psi_2 \text{ and} \\
 & & \forall 0 \leq i < k.M, \pi^i \models \psi_1)
\end{array}
$$

Comparison of Expressive Power of LTL, CTL and CTL*

Model Unification

Observation

- Every LTL formula is a CTL* path formula.
- Every CTL formula is a CTL* state formula.
- Model of a path formula is a run of Kripke structure.
- Model of a state formula is a state of Kripke structure.
- Not very suitable for comparison.

Model Unification

- For the purpose of comparison we define how a CTL* path formula is evaluated in a state of Kripke structure.
- Let ψ be CTL* path formula, then

$$M, s \models \psi \quad \text{iff} \quad M, s \models A\,\psi$$

Motivation

Goals

- We intend to find out whether there are properties (formulae) that can be expressed in one of the logic, but cannot be expressed in another one.
- We intend to find out in which logic more properties can be expressed.
- We intend to identify concrete properties, that cannot be expressed in some other logic, i.e. to find out a formula of logic $\mathcal{L}_1$, for which an equivalent formula of logic $\mathcal{L}_2$ does not exist.

Formula Equivalence

- Formulae φ and ψ are equivalent if and only if for any possible Kripke structure $M = (S, T, I, s_0)$ and any state $s \in S$ it is true that

$$M, s \models \varphi \quad \text{iff} \quad M, s \models \psi.$$

Equivalent Expressive Power

Equivalently Expressive

- Temporal logic $\mathcal{L}_1$ and $\mathcal{L}_2$ have the same expressive power, if for all Kripke structures $M = (S, T, I, s_0)$ and states $s \in S$ it holds that

$$\forall \varphi \in \mathcal{L}_1.(\exists \psi \in \mathcal{L}_2.(M, s \models \varphi \iff M, s \models \psi)) \quad (1)$$
$$\wedge\ \forall \psi \in \mathcal{L}_2.(\exists \varphi \in \mathcal{L}_1.(M, s \models \varphi \iff M, s \models \psi)). \quad (2)$$

Less Expressiveness

- If only statement (1) is valid, then logic $\mathcal{L}_1$ is less expressive than logic $\mathcal{L}_2$, and vice versa.

Comparison of LTL, CTL, and CTL*

Theorem

- LTL and CTL are incomparable in expressive power.
  1) $AG(EF(q))$ is a CTL formula that cannot be expressed in LTL.
  2) $FG(q)$ is an LTL formula that cannot be expressed in CTL.

Example – Proof Sketch for 1)

- Find two different Kripke structures and identify two states that can be differentiated with CTL formula $AG(EF(q))$, but cannot be differentiated with any LTL formula (they generate the same set of runs).

Example – Intuition behind 2) [proof is too complex]

- Show that CTL formula $AF(AG(q))$ is not equivalent to LTL formula $FG(q)$.

Comparison of LTL, CTL, and CTL*

Consequence

- CTL* is strictly more expressive than LTL.
  - Every LTL formula is a CTL* formula.
  - CTL* formula $AG(EFq)$ is not expressible in LTL.

Consequence 2

- CTL* is strictly more expressive than CTL.
  - Every CTL formula is a CTL* formula.
  - CTL* formula $FG(q)$ is not expressible in CTL.

Observation

- There are properties expressible on both LTL and CTL.
  - CTL formula $A[p\,U\,q]$ is equivalent to LTL formula $p\,U\,q$.

Homework

Homework

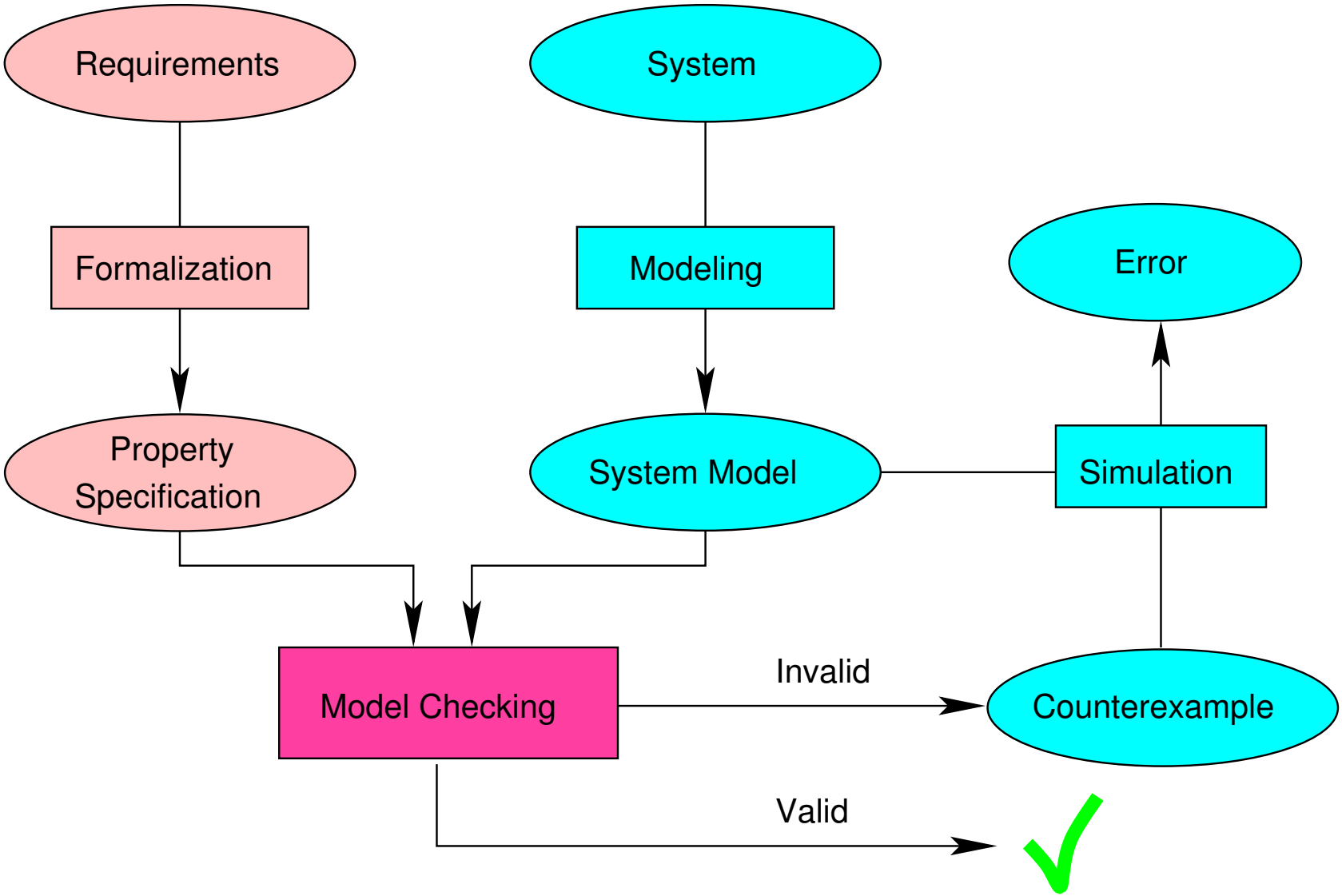
- Solve The wolf, goat and cabbage problem with NuSMV
- Moshe Vardi: *Branching vs. Linear Time: Final Showdown*

IA169 System Verification and Assurance

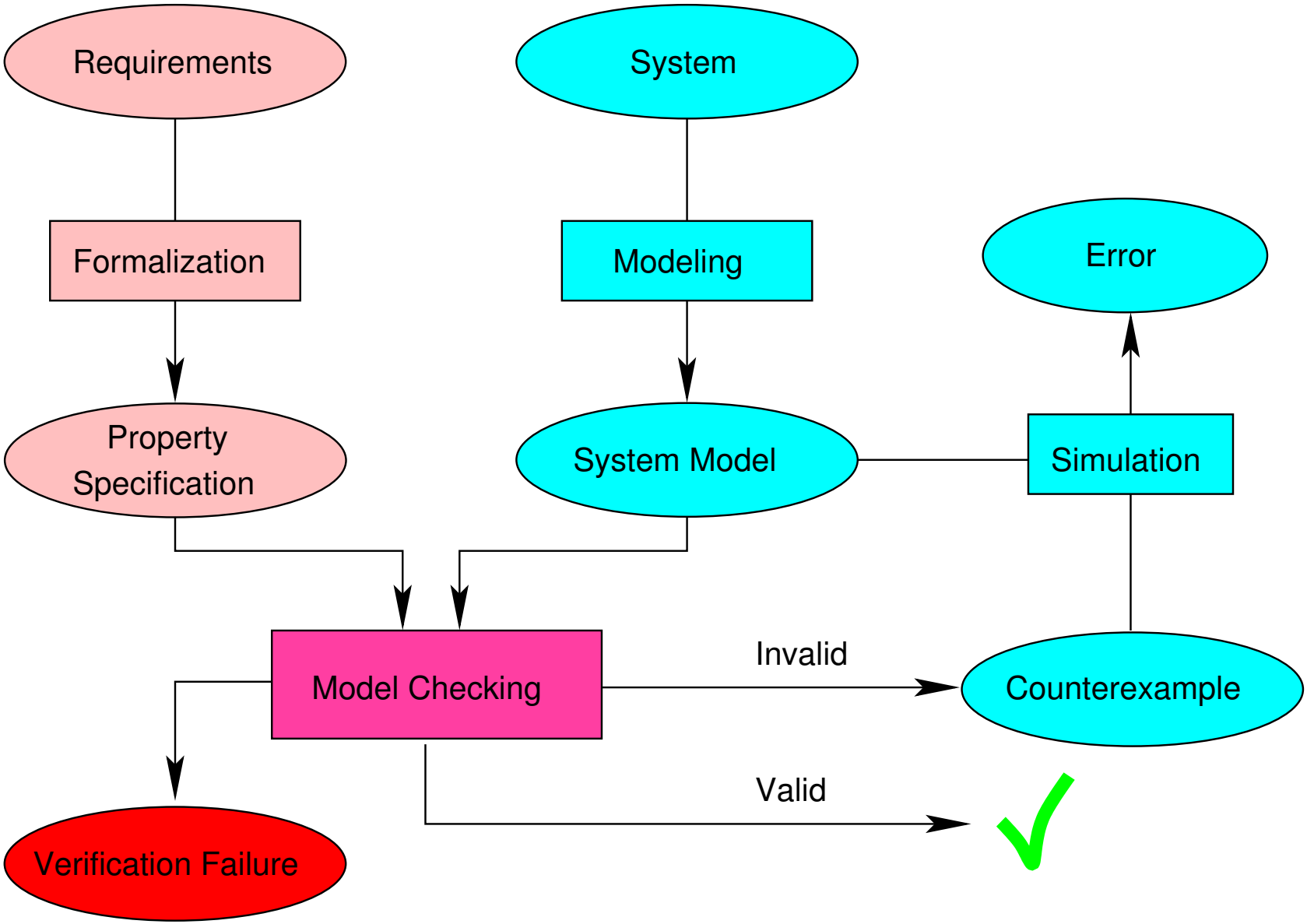
Symbolic Representations in CTL Model Checking

Jiří Barnat

State Space Explosion Problem and Model Checking



State Space Explosion Problem and Model Checking



Motivation

Observation

- Computation state is given by valuation of state variables.
- Every variable has a finite domain, its value may be stored using a fixed number of bits.
- Computation state represented as a bit vector $(a_1, \ldots, a_n)$ of fixed length n.

Set of States

- Algorithms for verification store set of states.
- Set of state can be viewed as a set of binary vectors.
- Set of binary vectors may be described with a **Boolean function**.

Boolean Functions

- These are formulae in propositional logic over a given set of Boolean variables.

Task

- Let system state be given by valuation of four bit variables $(a1, b1, a2, b2)$.
- A state is erroneous if the values of $a1$ and $b1$ and values of $a2$ and $b2$ agree.
- Describe a set of erroneous states with Boolean function.

Some Possible Solutions

Boolean Functions

- These are formulae in propositional logic over a given set of Boolean variables.

Task

- Let system state be given by valuation of four bit variables $(a1, b1, a2, b2)$.
- A state is erroneous if the values of $a1$ and $b1$ and values of $a2$ and $b2$ agree.
- Describe a set of erroneous states with Boolean function.

Some Possible Solutions

- $(a1 \wedge b1 \wedge a2 \wedge b2) \vee (a1 \wedge b1 \wedge \neg a2 \wedge \neg b2) \vee$
  $(\neg a1 \wedge \neg b1 \wedge \neg a2 \wedge \neg b2) \vee (\neg a1 \wedge \neg b1 \wedge a2 \wedge b2)$
- $a1 \Leftrightarrow b1 \wedge a2 \Leftrightarrow b2$

Representation of Boolean Functions

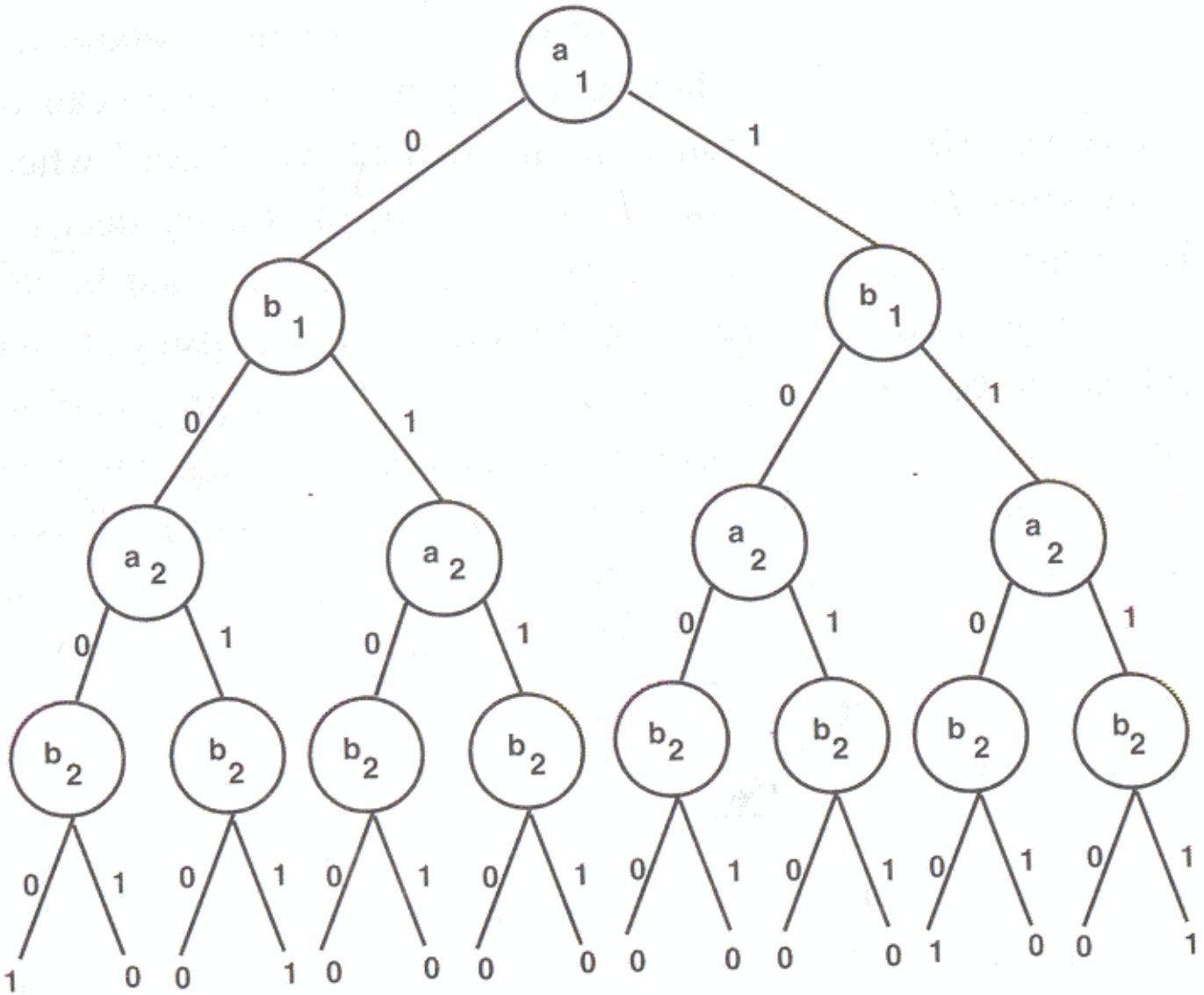
Binary Decision Trees (BDTs)

- Directed tree with a single root state.
- Every inner node is denoted with a Boolean variable (v) and lead to exactly two successors referred to as to ($low(v)$, $high(v)$).
- Every leaf is assigned a binary value, i.e. 0 or 1.

Coding of Boolean Functions with BDTs

- Every combination of values of input variables corresponds to exactly one path from the root of BDT to a leaf.
- Values stored at leaves give the the value of the function for the corresponding input values.

Binary Decision Tree $\psi = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$



Representation of Boolean Functions

Disadvantage of BDTs

- BDTs are uselessly space demanding (contain redundant information).

Task

- Identify isomorphic sub-trees of the BDT from the previous slide.

Binary Decision Diagrams (BDD)

- Acyclic directed graph, of which vertices have output degree either zero (leaf) or two (inner vertex).
- Vertices of BDD have otherwise the same properties as BDT nodes.

Computing (minimal) BDD

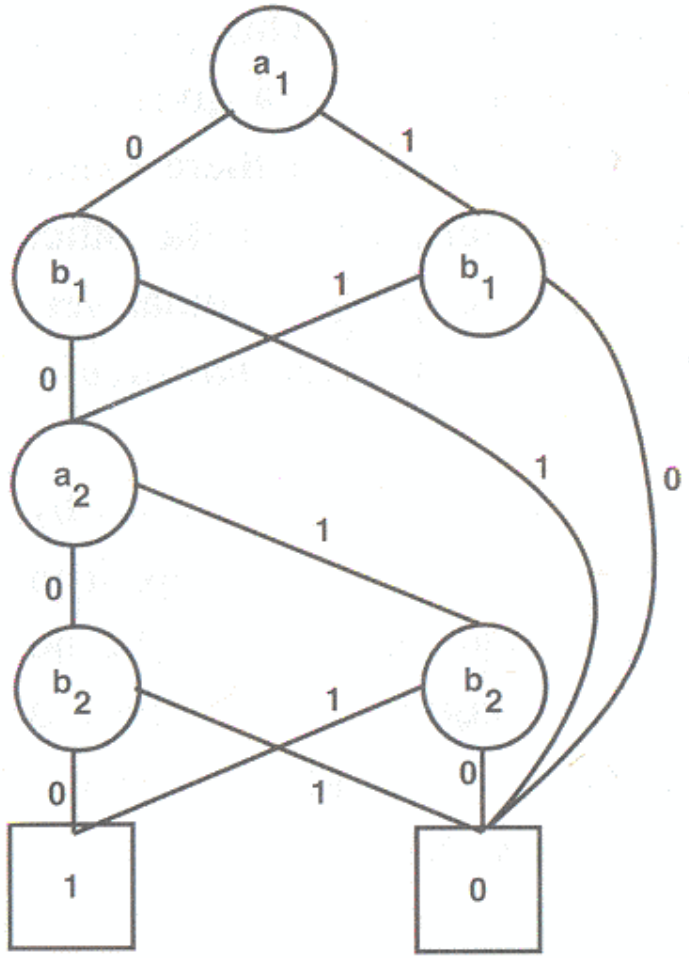
Initialisation

- For a given Boolean function take arbitrary BDD or BDT.
- Eliminate unreachable vertices
- Eliminate duplicate
  1) Remove all but one leaves with the same value.
  2) All edges incident with eliminated leaves reconnect to the the remaining leaf with the same value.

Repeat Until Fixpoint

- Eliminate duplicate inner vertices.
  - If there are two inner vertices u, v with the same label such that $low(v) = low(u)$ a $high(v) = high(u)$, then remove u and reconnect edges originally leading to u to v.
- Eliminate useless tests
  - Eliminate inner vertex v if $low(v) = high(v)$. Reconnect edges originally leading to v to $low(v)$.

BDD pro $\psi = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$



Coding of Boolean Functions with BDDs

Observation

- Every vertex v of BDD encodes some Boolean function $F_v(x_1, \ldots, x_n)$.

Computing $F_v(x_1, \ldots, x_n)$ for values $h_1, \ldots, h_n$.

- If v is a leaf then
  - $F_v(h_1, \ldots, h_n) = 1$, if v is labelled with value 1.
  - $F_v(h_1, \ldots, h_n) = 0$, if v is labelled with value 0.
- If v is an inner vertex then
  - $F_v(h_1, \ldots, h_n) = F_{low(v)}(h_1, \ldots, h_n)$, if $h_i == 0$.
  - $F_v(h_1, \ldots, h_n) = F_{high(v)}(h_1, \ldots, h_n)$, if $h_i == 1$.

Ordering Variables in BDD — OBDD

Observation

- Some intermediate representation computed during minimisation of a BDD are also valid BDDs.
- A given Boolean function may be represented with multiple different BDDs.

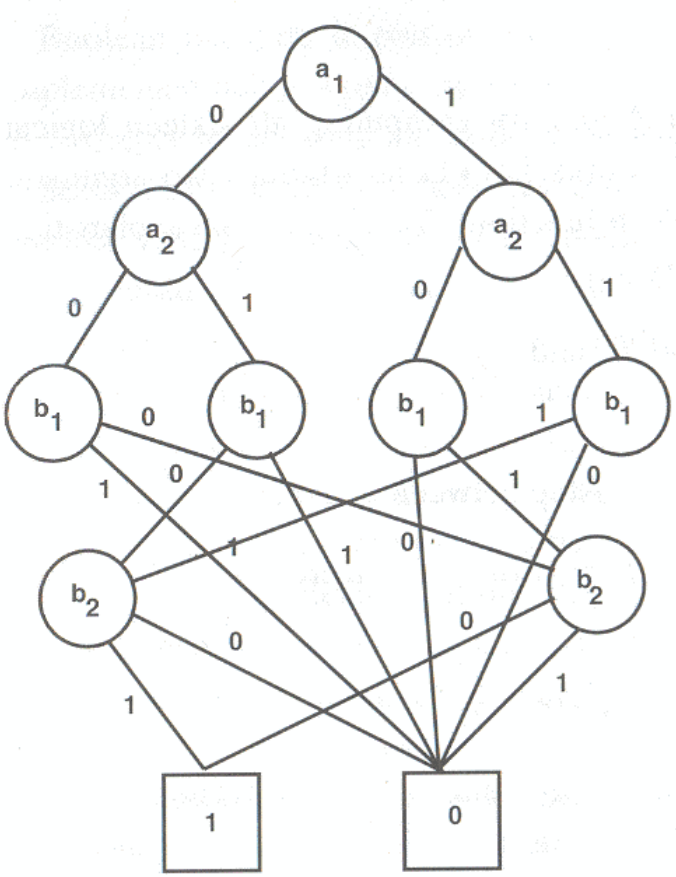
Canonical Form for BDD

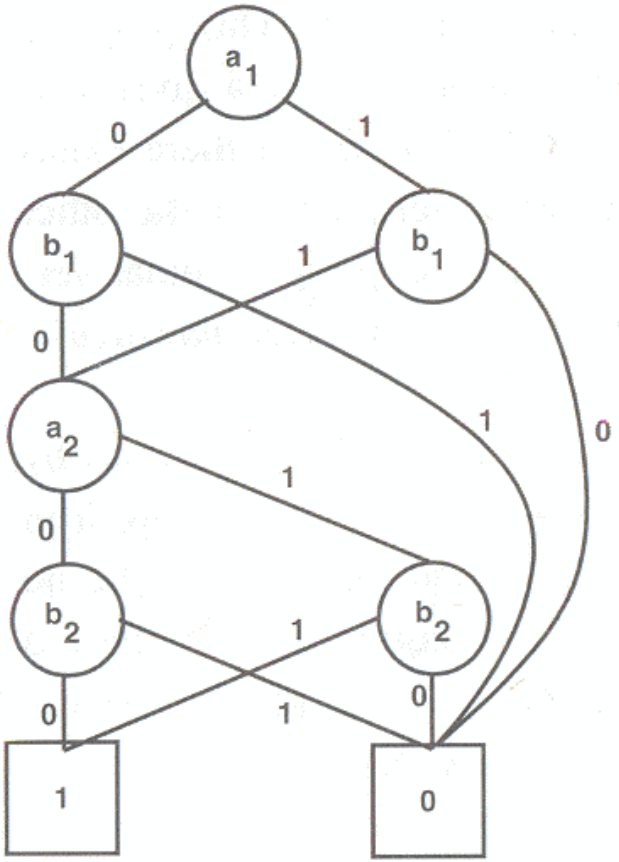
- Minimal BDD computed from a BDD, or BDT with a fixed ordering on variables in inner vertices **is unique**.
- BDD with a fixed variable ordering is referred to as to Ordered BDD (OBDD).

Computing Canonical Form

- Apply algorithm for minimal BDD.
- If performed in a bottom-up manner, obtained in linear time w.r.t. the size of initial BDT or BDD.

OBDDs for Different Variable Ordering





Restriction Operator for OBDDs

Observation

- Every OBDD represents some Boolean function.
- Boolean functions can be combined/composed using unary and binary logic operators such as $\neg, \wedge, \vee, \implies, XOR, \ldots$.
- OBDDs can be composed similarly.

Application of Logic Operators on OBDD

- Let O and O' be OBDDs corresponding to functions f and f', respectively.
- We will refer to function $Apply(O, O', \star)$, as to function that computes OBDD that represents result of application of logic operator $\star$ to functions f and f'.

Operation of Restriction

Operation of Restriction

- $F_{x_i \leftarrow b}(x_1, \ldots, x_n) = F(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$
- Produces Boolean function with all but one free variables.

Realisation for OBDD

- If root r is denoted with the restricted variable x_i, the resulting OBDD will have new root
  - $low(r)$ if $b = 0$
  - $high(r)$ if $b = 1$
- Any edge leading to a inner vertex t that is denoted with the restricted variable x_i is reconnected to
  - $low(t)$ if $b = 0$
  - $high(t)$ if $b = 1$
- OBDD is minimised (contains unreachable nodes).

Shannon expansion

- Any binary logic operator can be applied on OBDDs using **Shannon expansion**:

$$F = (\neg x \wedge F_{x \leftarrow 0}) \vee (x \wedge F_{x \leftarrow 1})$$

- If $F = f \star f'$, for any binary logic operation $\star$, then

$$f \star f' = (\neg x \wedge (f_{x \leftarrow 0} \star f'_{x \leftarrow 0})) \vee (x \wedge (f_{x \leftarrow 1} \star f'_{x \leftarrow 1}))$$

Algorithm for Application of Binary Operators on OBDDs

$Apply(O, O', \star)$

- Let v, v' be root nodes of O, O', denoted with x, x', respectively.
- If v and v' are leaves denoted with values h and h', respectively, then return a leave denoted with $h \star h'$.
- Otherwise, if

$x = x'$ then return a new node w denoted with variable x, where

- $low(w) = Apply(low(v), low(v'), \star)$
- $high(w) = Apply(high(v), high(v'), \star)$

$x < x'$ then return a new node w denoted with variable x, where

- $low(w) = Apply(low(v), O', \star)$
- $high(w) = Apply(high(v), O', \star)$

$x' < x$ then return a new node w denoted with variable x, where

- $low(w) = Apply(O, low(v), \star)$
- $high(w) = Apply(O, high(v), \star)$

Negation Operation and Emptiness Check

Observation

- Let *OBDD X* encodes function F_X, then *OBDD Y* encoding negation function $\neg F_X$ is created as a copy of *OBDD X* in which values of leaves are switched.

Emptiness Check

- OBDDs have canonical form.
- Canonical OBDD representing an empty set is made of a single leaf denoted with 0.

Test for a Presence of Set Member (complicated way)

- Create an OBDD describing the tested member.
- Apply operation $\wedge$ on tested and newly created OBDDs.
- Employ emptiness check on the resulting OBDD.

Symbolic Representation of Kripke Structure

Encoding of Transitions of Kripke Structure

Observation

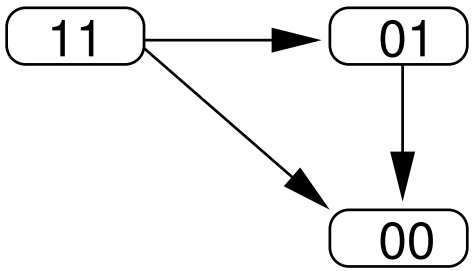
- A state of Kripke structure $M = (S, T, I)$ is given by n binary variables $a_1, \ldots, a_n$.
- Every set of states of Kripke structure can be encoded by an OBDD with n variables.
- Similarly, transition relation $T \subseteq S \times S$ can be encoded by Boolean function with $2n$ variables.

Simplification of OBDD

- Edges leading to zero leaf can be omitted.
- Non-existence of an edge indicates an edge to zero leaf.

Task

- $M = (\{\{00, 01, 11\}\}, \{(11, 00), (11, 01), (01, 00)\}, I)$



- T can be encoded as $F(a, b, a', b')$
- $F(a, b, a', b') =$
  $(a \wedge b \wedge \neg a' \wedge b') \vee (a \wedge b \wedge \neg a' \wedge \neg b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')$
- Assume variable ordering $a < b < a' < b'$ and draw OBDD for F.

Successors of States

Observation

- Assume $M = (S, T, I)$ and $OBDD_T(a, b, a', b')$.
- Let X be a set of states given with $OBDD_X(a, b)$.
- Using $OBDD_T$ and $OBDD_X$, $OBDD_{X'}(a', b')$ representing set of successors of states in X can be computed, i.e.

$$X' = \{v \in S \mid u \in X \wedge (u, v) \in T\}.$$

Successors of States – Algorithm Idea

Computing $OBDD_{X'}$ (intuitively)

- $OBDD_{X'} = Apply(OBDD_T, OBDD_X, \wedge)$
- Modify $OBDD'_X$ so that every path of it contains vertex labelled with a'.
- In $OBDD_{X'}$ erase all vertices labelled with a and b.
- Iterate over all a' vertices, consider them as root and compute respective minimal OBDDs.
- The computed set of OBDDs combine with operation $\vee$.
- Minimise the resulting OBDD.
- Rename primed variables to unprimed.

Task

- Compute OBDD representing successors of states $\{00, 11\}$.

Predecessors of States

Computing Predecessors (intuitively).

- Modify all vertices of $OBDD_X$ to be labelled with primed variables.
- $OBDD_{X'} = Apply(OBDD_T, OBDD_X, \wedge)$
- Modify $OBDD'_X$ so that every path contains vertex labelled with a'.
- Those a' that cannot reach leaf labelled with 1 replace with a new zero leaf.
- Other a' vertices replace with the other leaf.
- Remove all primed nodes and old leaves, and minimise OBDD.

Task

- Compute OBDD representing predecessor of state $\{00\}$.

Symbolic Approach to Model Checking CTL

Reminder

Observation

- If validity of formulae φ and ψ is known for all states of Kripke structure, validity of formulae $\neg\varphi$, $\varphi \vee \psi$, $EX\,\varphi$, etc., can be easily deduced.

Algorithm Idea for Model Checking CTL

- Let $M = (S, T, I)$ be a Kripke structure and φ a CTL formula.
- Labelling function $label : S \to 2^{\varphi}$ is computed, stating which sub-formulae of φ are valid in which states of M.
- Obviously, $s_0 \models \varphi \iff \varphi \in label(s_0)$.
- Function *label* is computed gradually for every sub-formula of φ starting with the simplest sub-formulae (atomic propositions) and terminating after computing the validity of φ.

Symbolic Approach

Idea

- Set of states in which particular sub-formulae hold can be efficiently represented with OBDDs.
- Computation of *label* function for more complex sub-formulae employs manipulation with respective OBDDs.

Realisation

- Set of states represented with OBDDs.
- Boolean functions to evaluate atomic proposition form initial OBDDs.
- According to the structure of the verified formula OBDDs for more complex sub-formulae are computed.
- Test membership of initial state of Kripke structure in the set of states satisfying verified formula.

Atomic Propositions and Logic Operators

Recall Syntax of CTL

- $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\, U\, \varphi] \mid EG\,\varphi$

Computing Set of States Satisfying CTL Formula

- Notation
  - $F(\psi)$ denotes (a function describing) set of states satisfying ψ.
  - $Succ(X)$ denotes immediate successors of states in the set X.
  - $Pred(X)$ denotes immediate predecessors of states in the set X.
- Boolean Functions for Atomic Proposition
  - Atomic propositions describe properties of state variables.
  - Atomic Propositions can be encoded as Boolean functions.
- Computing Boolean Operators $\neg$ and $\vee$
  - $F(\neg\psi_1) = \neg(F\psi_1)$
  - $F(\varphi \vee \psi) = F(\varphi) \vee F(\psi)$

Temporal operators $EX(\varphi)$, $E[\varphi\, U\, \psi]$ and $EG(\varphi)$

Operator $EX(\varphi)$

- $F(EX(\varphi)) = Pred(F(\varphi))$

Operator $E(\varphi\, U\, \psi)$

- $F(E(\varphi\, U\, \psi)) = X$,
  where X is the least fix-point of recursive rule

$$X = F(\psi) \cup (F(\varphi) \cap EX(X))$$

Operator $EG\,(\varphi)$

- $F(EG\,\varphi) = X$,
  where X is the greatest fix-point of recursive rule

$$X = F(\varphi) \cap EX(X)$$

Computing Fix-Points of Function f

The Least Fix-Point

proc $LFP(f)$
  $X = \emptyset$
  $Xold = \emptyset$
  do
    $Xold = X$
    $X := f(X)$
  while $(X \neq Xold)$
end

The Greatest Fix-Point

proc $GFP(f)$
  $X = S$
  $Xold = S$
  do
    $Xold = X$
    $X := f(X)$
  while $(X \neq Xold)$
end

Model Checking – Summary

Model Checking – Summary

Enumerative × Symbolic Approach

- Enumerative – focused on "control-flow"
- Symbolic – focused on "data-flow"

Pros w.r.t. Testing

- No source-code necessary (can be applied on models).
- Suitable for testing of parallel programs.

Pros w.r.t. Static Analysis

- Complete for systems with a finite state space.
- Verification of temporal properties.

Cons

- State space explosion problem.

Practicals and Homework – 08

Homework

- Explore Z3 tutorial (rise4fun.com).

IA169 System Verification and Assurance

Deductive Verification

Jiří Barnat

Verification of Algorithms

Validation and Verification

- A general goal of V&V is to prove correct behaviour of algorithms.

Reminder

- Testing is incomplete.
- Testing can detect errors but cannot prove correctness.
- Model checking is limited.
- Requires finite state space description, suffers from state space explosion, verifies specific program properties only.

Conclusion

- Need for yet another way of verification.

Formal Verification

Goal of formal verification

- The goal is to show that system behaves correctly with the same level of confidence as it is given with a mathematical proof.

Requirements

- Formally precise semantics of system behaviour.
- Formally precise definition of system properties to be shown.

Methods of formal verification

- Model checking
- Deductive verification
- Abstract interpretation

Deductive verification

Notion of Correctness

Program is correct if ...

- ... it **terminates** for a valid input and returns **correct** output.
- There is a need to show two parts – partial correctness and termination.

Partial correctness (Correctness, Soundness)

- If the computation terminates for valid input values (i.e. values for which the program is defined) the resulting values are correct.

Termination (Completness, Convergence)

- If executed on valid input values, the computation always terminates.

Verification of Serial Programs

Serial programs (sequential)

- Input-output-closed programs.
  - All input values are known prior program execution.
  - All output values are stored in output variables.
- Examples: Quick sort, Greatest Common Divider, ...

General Principle

- Program instructions are viewed as state transformers.
- The goal is to show that the mutual relation of input and output values is as expected or given by the specification.
- To verify the correctness of procedure of transformation of input values to output values.

Expressing Program Properties

State of Computation

- State of computation of a program is given by the value of program counter and values of all variables.

Atomic predicates

- Basic statements about individual states of the computation.
- The validity is deduced purely from the values of variables given by the state of computation.
- Examples of atomic propositions: $(x == 0)$, $(x1 >= y3)$.
- Beware of the scope of variables.

Set of States

- Can be described with a Boolean combination of atomic predicates.
- Example: $(x == m) \wedge (y > 0)$

Expressing Program Properties – Assertions

Assertion

- For a given program location defines a Boolean expression that should be satisfied with the current values of program variables in the given location during program execution.
- Invariant of a program location.

Assertions – Proving Correctness

- Assigning properties to individual locations of Control Flow Graph.
- Robert Floyd: Assigning Meanings to Programs (1967)

Error Detection — Assertion Violation

Testing

- Assertion violation serves as a test oracle.

Run-Time Checking

- Checking location invariants during run-time.
- Improved error localisation as the assertion violated relates to a particular program line.

Undetected Errors

- If an error does not manifest itself for the given input data.
- If the program behaves non-deterministically (parallelism).

Hoare Proof System

Hoare Proof System

Principle

- Programs = State Transformers.
- Specification = Relation between input and output state of computation.

Hoare logic

- Designed for showing partial correctness of programs.
- Let P and Q be predicates and S be a program, then

$$\{\mathrm{P}\}\ \mathrm{S}\ \{\mathrm{Q}\}$$

  is the so called *Hoare triple*.

Intended meaning of $\{P\}\ S\ \{Q\}$

- S is a program that transforms any state satisfying *pre-condition* P to a state satisfying *post-condition* Q.

Pre- and Post- Conditions

Example

- $\{z = 5\}\ x = z * 2\ \{x > 0\}$
- Valid triple, though post-condition could be more precise (stronger).
- Example of a stronger post-condition: $\{x > 5 \wedge x < 20\}$.
- Obviously, $\{x > 5 \wedge x < 20\} \implies \{x > 0\}$.

The Weakest Pre-Condition

- P is the weakest pre-condition, if and only if
- $\{P\}S\{Q\}$ is a valid triple and
- $\forall P'$ such that $\{P'\}S\{Q\}$ is valid, $P' \implies P$.
- Edsger W. Dijkstra (1975)

Proving in Hoare System

How to prove {P} S {Q}

- Pick suitable conditions P' a Q'
- Decomposition into three sub-problems:

  $\{P'\}\ S\ \{Q'\}$ $\quad$ $P \implies P'$ $\quad$ $Q' \implies Q$

- Use axioms and rules of Hoare system to prove $\{P'\}\ S\ \{Q'\}$.
- $P \implies P'$ and $Q' \implies Q$ are called proof obligations.
- Proof obligations are **proven in the standard way.**

Hoare System – Axiom for Assignment

Axiom

- Assignment axiom: $\{\phi[\text{x replaced with k}]\}\ x := k\ \{\phi\}$

Meaning

- Triple $\{P\}x := y\{Q\}$ is an axiom in Hoare system, if it holds that P is equal to Q in which all occurrences of x has been replaced with y.

Examples

- $\{y+7>42\}\ x:=y+7\ \{x>42\}$ is an axiom
- $\{r=2\}\ r:=r+1\ \{r=3\}$ is not an axiom
- $\{r+1=3\}\ r:=r+1\ \{r=3\}$ is an axiom

Hoare Logic – Example 1

Example

- Prove that the following program returns value greater than zero if executed for value of 5.
- Program: $out := in * 2$

Proof

1) We built a Hoare triple:
   $\{in = 5\}\ out := in * 2\ \{out > 0\}$
2) We deduce/guess a suitable pre-condition:
   $\{in * 2 > 0\}$
3) We prove Hoare triple:
   $\{in * 2 > 0\}\ out := in * 2\ \{out > 0\}$ (axiom)
4) We prove auxiliary statement:
   $(in = 5) \implies (in * 2 > 0)$

Hoare System – Example of a Rule

Rule

- Sequential composition: $\frac{\{\phi\}S_1\{\chi\} \wedge \{\chi\}S_2\{\psi\}}{\{\phi\}S_1;S_2\{\psi\}}$

Meaning

- If S_1 transforms a state satisfying ϕ to a state satisfying χ and S_2 transforms a state satisfying χ to a state satisfying ψ then the sequence $S_1; S_2$ transforms a state satisfying ϕ to a state satisfying ψ.

In the proof

- Should $\{\phi\}S_1; S_2\{\psi\}$ be used in the proof, an intermediate condition χ has to be found, and $\{\phi\}S_1\{\chi\}$ and $\{\chi\}S_2\{\psi\}$ have to be proven.

Hoare System – Partial Correctness

Axiom for **skip**: $\{\phi\}$ **skip** $\{\phi\}$

Axiom for **:=**: $\{\phi[x := k]\}x\mathbf{:=}k\{\phi\}$

Composition rule: $\dfrac{\{\phi\}S_1\{\chi\}\wedge\{\chi\}S_2\{\psi\}}{\{\phi\}S_1;S_2\{\psi\}}$

Conditional rule: $\dfrac{\{\phi\wedge B\}S_1\{\psi\}\wedge\{\phi\wedge\neg B\}S_2\{\psi\}}{\{\phi\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{\psi\}}$

While rule: $\dfrac{\{\phi\wedge B\}S\{\phi\}}{\{\phi\}\textbf{while } B \textbf{ do } S \textbf{ od } \{\phi\wedge\neg B\}}$

Consequence rule: $\dfrac{\phi \Longrightarrow \phi',\{\phi'\}S\{\psi'\},\psi' \Longrightarrow \psi}{\{\phi\}S\{\psi\}}$

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

```
r = 1;

while (n≠ 0) {
   r = r * n;

   n = n - 1;
}
```

Notes:

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

- $\{ n \geq 0 \wedge t{=}n \}$ $\{P\}$

```
r = 1;

while (n≠ 0) {
   r = r * n;

   n = n - 1;
}
```

$\{ r{=}t! \}$ $\{Q\}$

Notes:

- Reformulation in terms of Hoare logic.
- Note the use of auxiliary variable t.

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

- $\{ \; n \geq 0 \wedge t{=}n \; \}$ $\{P\}$

  r = 1;

  $\{ \; n \geq 0 \wedge t{=}n \wedge r = 1 \; \}$ $\{I_1\}$

  while (n≠ 0) {

  r = r * n;

  n = n - 1;

  }

  $\{ \; r{=}t! \; \}$ $\{Q\}$

Notes:

- $\{n \geq 0 \wedge t{=}n \wedge 1{=}1\}$ r=1 $\{ \; n \geq 0 \wedge t{=}n \wedge r{=}1 \; \}$
- $(n \geq 0 \wedge t{=}n) \implies (n \geq 0 \wedge t{=}n \wedge 1{=}1)$

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

- $\{ \mathsf{n} \geq 0 \wedge \mathsf{t}{=}\mathsf{n} \}$ $\{P\}$

  $\mathsf{r} = 1;$

  $\{ \mathsf{n} \geq 0 \wedge \mathsf{t}{=}\mathsf{n} \wedge r = 1 \}$ $\{I_1\}$

  while $(\mathsf{n} \neq 0)$ $\{ \mathsf{r}{=}\mathsf{t}!/\mathsf{n}! \wedge \mathsf{t} \geq \mathsf{n} \geq 0 \}$ { $\{I_2\}$

  $\quad \mathsf{r} = \mathsf{r} * \mathsf{n};$

  $\quad \mathsf{n} = \mathsf{n} - 1;$

  }

  $\{ \mathsf{r}{=}\mathsf{t}! \}$ $\{Q\}$

Notes:

- Invariant of a cycle: $\{I_2\} \equiv \{ \mathsf{r}{=}\mathsf{t}!/\mathsf{n}! \wedge \mathsf{t} \geq \mathsf{n} \geq 0 \}$
- $I_1 \implies I_2$ $\quad (I_2 \wedge \neg(\mathsf{n}{\neq}0)) \implies Q$

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

- $\{ \; n \geq 0 \wedge t{=}n \; \}$ $\{P\}$
  r = 1;
  $\{ \; n \geq 0 \wedge t{=}n \wedge r = 1 \; \}$ $\{I_1\}$
  while (n≠ 0) $\{ \; r{=}t!/n! \wedge t \geq n \geq 0 \; \}$ { $\{I_2\}$
    r = r * n;
    $\{ \; r{=}t!/(n\text{-}1)! \wedge t \geq n > 0 \; \}$ $\{I_3\}$
    n = n - 1;
  }
  $\{ \; r{=}t! \; \}$ $\{Q\}$

Notes:

- $\{ \; r{*}n = t!/(n\text{-}1)! \wedge t \geq n > 0 \; \}$ r=r*n $\{I_3\}$
- $I_2 \wedge (n{\neq}0) \implies (\; r{*}n = t!/(n\text{-}1)! \wedge t \geq n > 0 \;)$

Hoare Logic – Example 2

Prove that for $n \geq 0$ the following code computes $n!$.

- $\{ n \geq 0 \wedge t{=}n \}$ $\{P\}$
  r = 1;
  $\{ n \geq 0 \wedge t{=}n \wedge r = 1 \}$ $\{I_1\}$
  while (n≠ 0) $\{ r{=}t!/n! \wedge t \geq n \geq 0 \}$ { $\{I_2\}$
    r = r * n;
    $\{ r{=}t!/(n\text{-}1)! \wedge t \geq n > 0 \}$ $\{I_3\}$
    n = n - 1;
  }
  $\{ r{=}t! \}$ $\{Q\}$

Notes:

- $\{ r = t!/(n\text{-}1)! \wedge t \geq (n\text{-}1) \geq 0 \}$ n=n-1 $\{I_2\}$
- $I_3 \implies (r = t!/(n\text{-}1)! \wedge t \geq (n\text{-}1) \geq 0)$

Hoare Logic and Completness

Observation

- Hoare logic allowed us to reduce the problem of proving program correctness to a problem of proving a set of mathematical statements with arithmetic operations.

Notice about correctness's and (in)completeness

- Hoare logic is correct, i.e. if it is possible to deduce $\{P\}S\{Q\}$ then executing program S from a state satisfying P may terminate only in a state satisfying Q.
- If a proof system is strong enough to express integral arithmetics, it is necessarily incomplete, i.e. there exists claims that cannot be proven or dis-proven using the system.
- Hoare system for proving correctness of programs is incomplete due to the proof obligations generated with the consequence rule.

Hoare Logic and Proving Correctness in Practice

Troubles with Proof Construction

- Often pre- and post- condition must be suitable reformulated for the purpose of the proof.
- It is very difficult to identify loop invariants.

Partial Correctness in Practice

- Often reduced to formulation of all the loop invariants, and demonstration that they actually are the loop invariants.
- The proof of being an invariant is often achieved with math induction.

Proving Termination

Well-Founded Domain

- Partially ordered set that does not contain infinitely decreasing sequence of members.
- Examples: $(\mathbf{N},<)$, $(\mathit{PowerSet}(\mathbf{N}),\subseteq)$

Proving Termination

- For every loop in the program a suitable well-founded domain and an expression over the domain is chosen.
- It is shown that the value associated with a location cannot grow along any instruction that is part of the loop.
- It is shown that there exists at least one instruction in the loop that decreases the value of the expression.

Automating Deductive Verification

Principles of Automation of Deductive Verification

Pre-processing

- Transformation of program to a suitable intermediate language.
- Examples of IL: Boogie (Microsoft Research), Why3 (INRIA)

Structural Analysis and Construction of the Proof Skeleton

- Identification of Hoare triples, loop invariants and suitable pre- and post-conditions (some of that might be given with the program to be verified).
- Generation of auxiliary proof obligations.

Solving proof obligations

- Using tools for automated proving.
- May be human-assisted.

Solving Proof Obligations

Tools for Automated Proving

- User guides a tool to construct a proof.
- HOL, ACL2, Isabelle, PVS, Coq, ...

Reduced to the satisfiability problem

- Employ SAT and SMT solvers.
- Z3, ...

Automated Proving

Proof

- A finite sequence of steps that using axioms and rules of a given proof system that transforms assumptions ψ into the conclusion φ.

Observation

- For systems with finitely many axioms and rules, proofs may be systematically generated. Hence, for all provable claims the proof can be found in finite time.
- All reasonable proving systems has infinitely many axioms. Consider, e.g. an axiom $x = x$. This is virtually a shortcut (template) for axioms $1 = 1$, $2 = 2$, $3 = 3$, etc.
- Semi-decidable with dove-tailing approach.

Automated Proofing

Searching for a Proof of Valid Statement

- The number of possible finite sequences of steps of rules and axiom applications is too many (infinitely many).
- In general there is no algorithm to find a proof in a given proof system even for a valid statement.
- Without some clever strategy, it cannot be expected that a tool for automated proof generation will succeed in a reasonable short time.
- The strategy is typically given by an experienced user of the automated proving tool. The user typically has to have appropriate mathematical feeling and education.
- At the end, the tool is used as a mechanical checker for a human constructed proof.

Verification with Tools for Automated Proving

Theorem Provers

- The goal is find the proof within a given proof system.
- the proof is searched for in two modes:
  - Algorithmic mode – Application of rules and axioms
    - Guided by the user of the tool.
    - Application of the general proving techniques, such as deduction, resolution, unification, ...
  - Search mode – Looking for new valid statements
    - Employs brute-force approach and various heuristics.

Existing Tools

- The description of system (axioms, rules) as well as the claim to be proven is given in the language of the tool.

Results of Proof Searching

Possible Outputs

a) Proof has been found and checked.

b) Proof has not been found.

- The statement is valid, can be proven, but the proof has not yet been found.
- The statement is valid, but it cannot be proven in the system.
- The statement is invalid.

Observation

- In the case that no proof has been found, there is no indication of why it is so.

Dafny

http://rise4fun.com/dafny

Homework

- Prove correctness of the following program using Dafny

```
method Count(N: nat, M: int, P: int) returns (R: int) {
  var a := M;
  var b := P;
  var i := 1;
  while (i <= N) {
    a := a+3;
    b := 2*a+b+1;
    i := i+1;
  }
  R := b;
}
```

- Read and repeat:

  Jaco van de Pol: *Automated verification of Nested DFS*

  `http://dx.doi.org/10.1007/978-3-319-19458-5_12`

IA169 System Verification and Assurance

Bounded Model Checking

Jiří Barnat

Reminder – SAT and SMT

Satisfiability – SAT

- Finding a valuation of Boolean variables that makes a given formula of propositional logic true.

Satisfiability Modulo Theory – SMT

- Deciding satisfiability of a first-order formula with equality, predicates and function symbols that encode one or more theories.

Typical SMT Theories

- Unbounded integer and real arithmetic.
- Bounded integer arithmetic (bit-vectors).
- Theory of data structures (lists, arrays, ...).

Reminder – SAT and SMT Solvers

ZZZ aka **Z3**

- Tool developed by Microsoft Research.
- WWW interface — `http://www.rise4fun.com/Z3`
- Binary API for use in other tools and applications.

SMT-LIB

- Standardised language for SMT queries.
- Freely available library with a SMT implementation.

Reminder – Satisfiability and Validity

Observation

- Formula is valid if and only if its negation is not satisfiable.

Consequence

- SAT and SMT solvers can be used as tools for proving validity of formulated statements.

Model Synthesis

- SAT solvers not only decide satisfiability of formulas, but for satisfiable formulas also give the valuation which makes the formula true.
- Unlike theorem provers, they give a "counterexample" in case the statement to be proven is false.

Checking Safety Properties via SAT Reduction

Bounded Model Checking (BMC)

Hypothesis

- If the system contains an error, it can be reproduced with only a small number of controlled steps.

Method Idea

- If we use model checking for error detection, it is sensible to check whether an error (a violation of specification) appears within first k steps of execution.

Literature

- Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: Symbolic Model Checking without BDDs. TACAS 1999: 193-207, LNCS 1579.
- Henry A. Kautz, Bart Selman: Planning as Satisfiability.Proceedings of the 10th European conference on Artificial intelligence (ECAI'92): 359-363, 1992, Kluwer.

Reduction of BMC to SAT

Prerequisites

- The set of prefixes of length k of all runs of a Kripke structure M can be encoded by a Boolean formula $[M]^k$.
- Violation of a *safety* property which can be observed within k steps of the system can be encoded as $[\neg\varphi]^k$.

Reduction to SAT

- We check the satisfiability of $[M]^k \wedge [\neg\varphi]^k$.
- Satisfiability indicates the existence of a counterexample of length k.
- Unsatisfiability shows non-existence of a counterexample of length k.

Kripkeho structure as a Boolean formula

Prerequisites

- Let $M = (S, T, I)$ be a Kripke structure with initial state $s_0 \in S$.
- Arbitrary state $s \in S$ can be represented by a bit vector of size n, that is state $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$.

Encoding M with Boolean Formulae

- $Init(s)$ – formula which is satisfiable for the valuation of variables $a_1, a_2, ..., a_n$ that describe the state s_0.
- $Trans(s, s')$ – a formula which is satisfiable for a pair of state vectors s, s', iff the valuations $a_1, a_2, ..., a_n, a'_1, a'_2, ..., a'_n$ describe states between which a transition $(s, s') \in T$ exists.

Encoding Finite Runs of M

Description of System Runs of Length k

- Run of length k consists of $k+1$ states $s_0, s_1, \ldots, s_k$.
- The set of all runs of size k of the structure M is designated $[M]^k$ and described by the following formula:

$$[M]^k \equiv \mathit{Init}(s_0) \wedge \bigwedge_{i=1}^{k} \mathit{Trans}(s_{i-1}, s_i)$$

Example $[M]^3 \wedge [\neg\varphi]^3$

- $\mathit{Init}(s_0) \wedge \mathit{Trans}(s_0, s_1) \wedge \mathit{Trans}(s_1, s_2) \wedge \mathit{Trans}(s_2, s_3) \wedge \neg\varphi(s_3)$

Completeness of BMC

Completeness of BMC for Detecting Safety Violations

Problem – Undetected Violation of a Safety Property

- The violation is not reachable using a path of length k.
- Paths shorter than k are not encoded in $[M]^k$.

Upper Bound on k

- If $k \geq d$ where d is the graph diameter, all possible error locations are covered.
- The diameter of the graph is bounded by 2^n, where n is the number of bits of the state vector.

Solution

- Executing BMC iteratively for each $k \in [0, d]$.

Automated Detection of Graph Diameter

Facts

- Asking the user is unrealistic.
- Safe upper bounds are extremely overstated.
- We would like the verification procedure itself to detect whether k should be increased further.

Skeleton of an Algorithm for Complete BMC

$k = 0$
while `(true)` **do**
  if (counterexample of length k exists)
    then return `"Invalid"`
  if (all states are reachable within k steps)
    then return `"Valid"`
  $k = k + 1$
od

Notation I

Prerequisites

- Kripke structure $M = (S, T, I)$.
- States are described by bit vectors of fixed length.
- *Trans* is a SAT representation of a binary relation T.

Path of Length n

$$path(s_{[0..n]}) \quad \equiv \quad \bigwedge_{0 \leq i < n} Trans(s_i, s_{i+1})$$

Validity of Statement Q Along the Entire Path

$$all.Q(s_{[0..n]})$$

Notation II

A Loop-Free Path

$$loopFree(s_{[0..n]}) \quad \equiv \quad path(s_{[0..n]}) \ \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

Existence of a Path of Length n From s_0 to s_n

$$path_n(s_0, s_n) \quad \equiv \quad \exists s_1 \ldots s_{n-1}.path(s_{[0..n]})$$

Shortest Path

$$shortest(s_{[0..n]}) \quad \equiv \quad path(s_{[0..n]}) \ \wedge \neg\Big(\bigvee_{0 \leq i < n} path_i(s_0, s_n)\Big)$$

Equivalent Problem Formulation

Verification

- We would like to show that no state that would violate the specification φ is reachable from the initial configuration, i.e. we want to show that

$$\forall i. \forall s_0 \ldots s_i. \Big(\mathit{Init}(s_0) \wedge \mathit{path}(s_{[0..i]}) \implies \varphi(s_i) \Big)$$

Alternatively

- We want to show that from an error state, the initial state is not reachable when going backwards

$$\forall i. \forall s_0 \ldots s_i. \Big(\mathit{Init}(s_0) \impliedby \mathit{path}(s_{[0..i]}) \wedge \neg \varphi(s_i) \Big)$$

Equivalently

$$\forall i. \forall s_0 \ldots s_i. \neg \Big(\mathit{Init}(s_0) \wedge \mathit{path}(s_{[0..i]}) \wedge \neg \varphi(s_i) \Big)$$

Termination of BMC – Acyclic Paths

Termination Condition in the BMC Algorithm Skeleton

- No longer acyclic path from the initial state exists, that is, the following formula is unsatisfiable:

$$Init(s_0) \ \wedge \ loopFree(s_{[0..i+1]})$$

- **Holds symmetrically for backwards reachability from error states.**

Solution 1

- $\texttt{not SAT}\Big(\ loopFree(s_{[0..i+1]}) \ \wedge \ Init(s_0) \ \Big)$
  $\vee$
  $\texttt{not SAT}\Big(\ loopFree(s_{[0..i+1]}) \ \wedge \ \neg\varphi(s_{i+1}) \ \Big)$

Termination of BMC – Acyclic paths II

Higher Efficiency Termination Criterion

- When using backward reachability from $\neg\varphi$ states, paths that visit other $\neg\varphi$ states do not need to be considered.
- Symmetrically holds also for forward reachability with multiple initial states: for completeness detection, paths that visit other initial states do not need to be considered.

Solution 2

- $\texttt{not SAT}\Big(\mathit{loopFree}(s_{[0..i+1]}) \wedge \mathit{Init}(s_0) \wedge \mathit{all}.\neg \mathit{Init}(s_{[1..i+1]}) \Big)$
  $\vee$
  $\texttt{not SAT}\Big(\mathit{loopFree}(s_{[0..i+1]}) \wedge \neg\varphi(s_{i+1}) \wedge \mathit{all}.\varphi(s_{[0..i]}) \Big)$

BMC not starting with $k = 0$

Observation

- For small values of k, SAT queries give neither a counterexample nor enable termination.
- We want to start BMC with $k > 0$.

Reformulating the Counterexample Test

- The original test for counterexample existence for a given k

$$\texttt{SAT}\Big(\mathit{Init}(s_0) \wedge \mathit{path}(s_{[0..k]}) \wedge \neg\varphi(s_k)\Big)$$

  needs to be changed so that we do not miss counterexamples shorter than the initial value of k.
- New test for the existence of a counterexample:

$$\texttt{SAT}\Big(\mathit{Init}(s_0) \wedge \mathit{path}(s_{[0..k]}) \wedge \neg \mathit{all}.\varphi(s_{[0..k]})\Big)$$

k-induction in BMC

Observation

- The tests can be re-formulated so that they resemble the structure of mathematical induction.
- TAUT is a tautology test (unsatisfiability of negation).

Base Case

- Test for counterexample existence.

$$\texttt{SAT}\Big(\neg\big(\ \mathit{Init}(s_o) \wedge \mathit{path}(s_{[0..i]}) \implies \mathit{all}.\varphi(s_{[0..i]})\ \big)\Big)$$

Inductive Step

- Test for completeness.

$$\texttt{TAUT}\Big(\neg\mathit{Init}(s_0) \Longleftarrow \mathit{all}.\neg\mathit{Init}(s_{[1..(i+1)]}) \wedge \mathit{loopFree}(s_{[0..i+1]})\Big)$$
$$\vee$$
$$\texttt{TAUT}\big(\ \mathit{loopFree}(s_{[0..i+1]}) \wedge \mathit{all}.\varphi(s_{[0..i]}) \implies \varphi(s_{i+1})\ \big)$$

Acyclic vs Shortest Paths in BMC

Observation

- Graph diameter (d) is the length of the longest of the shortest paths between each pair of vertices in the graph.
- An acyclic path can be substantially longer than the graph diameter.

BMC with Shortest Paths

- BMC is correct if *loopFree* is replaced with *shortest*.
- The *shortest* predicate, however, needs quantifiers and is as such not a purely SAT application.

For more details, see ...

- Mary Sheeran, Satnam Singh, and Gunnar Stålmarck: Checking Safety Properties Using Induction and a SAT-Solver, FMCAD 2000, 108-125, LNCS 1954, Springer.

LTL and BMC

LTL Verification with BMC

Observation 1

- LTL is only well-defined for infinite runs.
- For evaluating LTL on finite paths, we use three-value logic (true, false, cannot say).
- Validity of some LTL formulas cannot be decided on any finite path (eg. $GF\ a$).

Observation 2

- Cycles that consist of only a few states are unrolled by BMC to acyclic paths of length k.
- We allow encoding lasso-shaped paths.
- That is, (k, l)-cyclic paths.

(k,l)-cyclic paths

(k,l)-cyclic runs

- A run $\pi = s_0 s_1 s_2 \ldots$ of a Kripke structure $M = (S, T, I, s_0)$ is (k, l)-cyclic if

$$\pi = (s_0 s_1 s_2 \ldots s_{l-1})(s_l \ldots s_k)^\omega,$$

where $0 < l \leq k$ a $s_{l-1} = s_k$.

Observation

- If π is (k, l)-cyclic, π is also $(k + 1, l + 1)$-cyclic.
- Treating finite paths as (k, k)-cyclic is incorrect (could create a non-existent run in M).
- Every path of length k is either acyclic or (k, l)-cyclic.

Semantics of LTL on Finite Prefixes of Runs

Semantics of LTL for Finite Prefixes

- Let π be a run of a Kripke structure M.
- k is given.
- $\pi = \pi^0$

$$\begin{aligned} \pi^i \models_{nl} X\,\varphi \quad &\text{iff} \quad i < k \wedge \pi^{i+1} \models_{nl} \varphi \\ \pi^i \models_{nl} \varphi\, U\, \psi \quad &\text{iff} \quad \exists j.\, i \leq j \leq k, \pi^j \models_{nl} \psi \text{ and} \\ &\qquad \forall m.\, i \leq m < j, \pi^i \models_{nl} \varphi \end{aligned}$$

Semantics of $\models_k$ for LTL in BMC

- For (k,l)-cyclic paths, $\pi \models_k \varphi \iff \pi \models \varphi$ holds.
- For acyclic paths, $\pi \models_k \varphi \iff \pi^0 \models_{nl} \varphi$ holds.
- $\models_k \Longrightarrow \models_{k+1}$, $\models_k$ approximates $\models$

BMC for LTL

Goal

- We construct a Boolean formula $[M, \varphi, k]$ which is satisfiable iff Kripke structure M has a run π such that $\pi \models_k \varphi$.
- $[M, \varphi, k] \equiv [M]^k \wedge [\varphi, k]$

Encoding

- $[M]^k$ encodes all paths of length k
- $[\varphi, k] \equiv \_[\varphi, k]_0 \vee \bigvee_{l=1}^{k} {}_l[\varphi, k]_0$
- $\_[\varphi, k]_0$ encodes that the path is acyclic and $\models_{nl} \varphi$
- ${}_l[\varphi, k]_0$ encodes that the path is (k, l)-cyclic and $\models \varphi$

LTL tricks in BMC

Fragment LTL-X

- Reduces the number of transitions (smaller SAT instance).
- Similar to partial order reduction.

For the Interested

- Keijo Heljanko: Bounded Model Checking for Finite-State Systems
  `http://users.ics.aalto.fi/kepa/qmc/slides-heljanko-2.pdf`
- Keijo Heljanko and Tommi Junttila: Advanced Tutorial on Bounded Model Checking
  `http://users.ics.aalto.fi/kepa/acsd06-atpn06-bmc-tutorial/lecture1.pdf`

Conclusions on BMC

Advantages of BMC

General

- Reduces to a standard SAT problem, advances in SAT solving help with BMC.
- Often finds counterexamples of minimal length (not always).
- Boolean formulas can be more compact than OBDD representation.

Verification of HW

- Thanks to k-induction, a very successful method.

Verification of SW

- Currently, according to Software Verification Competition (TACAS 2014), BMC in connection with SMT is currently among the best software verification methods (actually falsification).

Downsides of BMC

General

- Not complete in general.
- Large SAT instances are still unsolvable.

Verification of SW

- Encoding an entire CFG as a SAT instance is currently unrealistic.
- K-induction cannot be used
  (the graph is incomplete, no back edges).
- Problems with dynamic data structure analysis.
- Loop analysis is hard.
- Inefficient for full arithmetic (partially solved by SMT).

Tools and food for thought...

Tools

- CBMC – BMC for ANSI-C.
- ESBMC – uses SMT, built on top of CBMC.
- LLBMC – BMC for LLVM bitcode.

Food for Thought...

- What differentiates modern SMT-BMC from symbolic execution?
- Boundaries are not clear.

Homework

- Study structure and results of Software Verification Competition (TACAS).

IA169 System Verification and Assurance

Verification of Real-Time and Hybrid systems

Jiří Barnat

Verification in Model-Based Development

Software Engineering Experience

- Employing V&V techniques too late in the development process significantly increases the cost of poor quality.
- The sooner a bug is detected the cheaper is the fix.
- Model-Based Development
- Model-Based Verification

Model-Based Development

- Consider models of the target system in order to ,e.g., simulate its behaviour in the design phase prior implementation.
- Behavioural models can be used for verification.

Hybrid Systems

Hybrid Systems

- Systems that combine multiple kinds of dynamics.
- Continuous systems driven by discrete events.

Areas of existence

- Mechanical systems
  - Continuous movement and contact with physical obstacle.
- Electrical systems
  - Continuous nature of electric charge in circuit driven by discrete switches.
- **Embedded systems**
  - Computer-driven systems in analogue environment.

Example – Bouncing Ball

System Description

- A ball released at height h bounces on a hard surface. The ball is under continuous influence of the gravity (9.8m/s^2). When bounces some energy is consumed by friction and elasticity and turns into heat.

Physics

- Acceleration = First derivative of speed with respect to time.
- Speed = First derivative of height with respect to time.

Abstraction and simplification

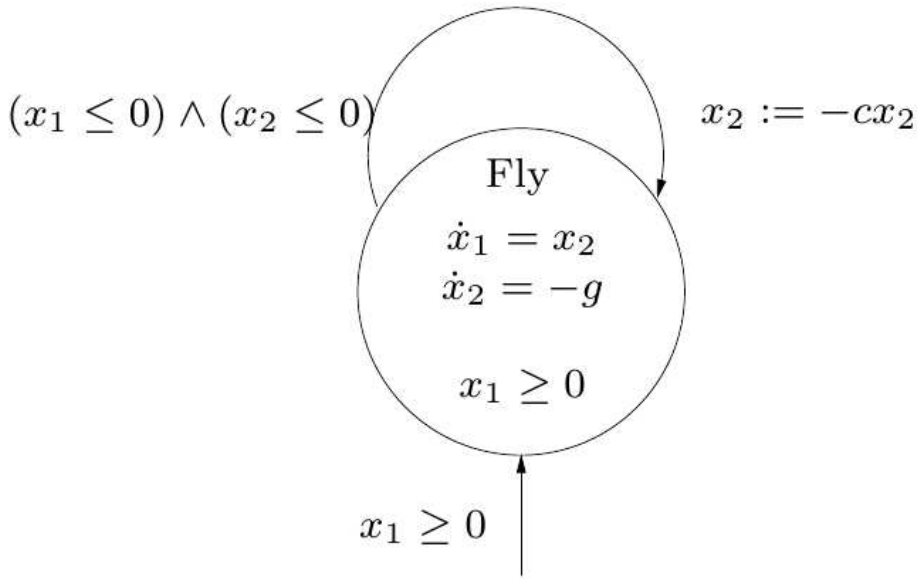
- Modelled with a mass point.
- Instant (time-less) bounce.

Bouncing Mass Point – Hybrid Automaton

Automaton Description

- $x1$ — height
- $x2$ — vertical speed ($+$ means up, $-$ means down)
- $c \in [0, 1]$ — loss of energy (elasticity and heat)

Schema



Analyses and Control of Hybrid Systems

Questions

- What time elapses between the fourth and fifth bounce?
- If given horizontal speed, will the ball jump over an obstacle?
- ...

Searching for Answers

- Need for precise formal description of hybrid system.
- Algorithmic analysis of properties of hybrid systems and controller synthesis.

Hybrid Automata

Hybrid Automata

Hybrid Automaton is a tuple

- $Q = \{q1, q2, \ldots\}$ — Set of discrete states.
- $X = \mathbf{R}^n$ — Set of continuous states.
- $f : Q \times X \to \mathbf{R}^n$ — System dynamics.
- $Init \subseteq Q \times X$ — Set of initial states.
- $Dom : Q \to PowerSet(X)$ — State invariants.
- $E \subseteq Q \times Q$ — Set of discrete transitions
- $G : E \to PowerSet(X)$ — Map of transition guards.
- $R : E \times X \to PowerSet(X)$ — Map of transition resets.

State of Hybrid Automaton

State of Hybrid Automaton

- Given by the discrete state and the current value of continuous variables: $(q, \overrightarrow{x}) \in Q \times X$.

Initial State

- Set of initial states in both the discrete and continuous part.
- $(q_0, \overrightarrow{x_0}) \in I$

Transitions of Hybrid Automaton

Transition by Time Passing

- Let $(q, \overrightarrow{x})$ be origin state.
- Continuous part for every variable x follows the system dynamics

$$\frac{dx(t)}{dt} = f(q, x), \text{ where } x(0) = x$$

- Discrete part does not change:

$$q(t) = q$$

- Time may pass only if the state invariant is valid:

$$x(t) \in Dom(q)$$

Transitions of Hybrid Automaton

Discrete Transition

- Let $(q, \overrightarrow{x})$ be origin state.
- It is possible (but not necessary) to perform a transition

$$(q, q') \in E,$$

- if transition guard is valid, i.e.

$$\overrightarrow{x} \in G(q, q').$$

- If the transition is taken, the continuous part of the state is updated accordingly:

$$\overrightarrow{x'} := R((q, q'), \overrightarrow{x})$$

- The target state after a discrete transition is $(q', \overrightarrow{x'})$.

Reasonable restrictions of Hybrid Automata

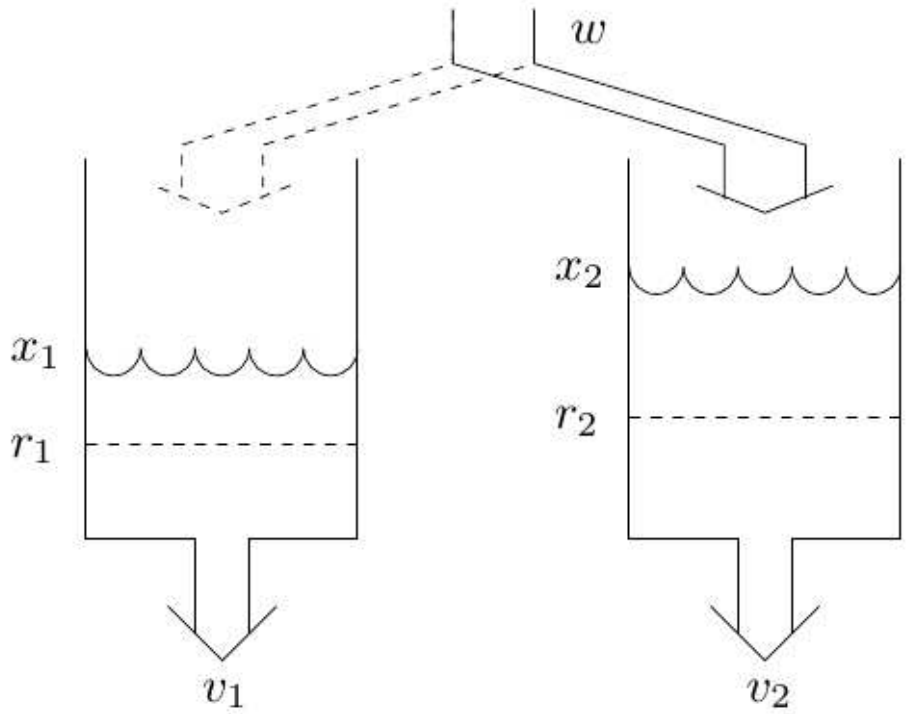
Restrictions in Continuous Part

- $f(q, \overrightarrow{x})$ is Lipschitz continuous for $\forall q \in Q$, (solution of differential equations is well defined)
- $\forall e \in E$ we assume non-empty $G(e)$
- $\forall e \in E$ and $\forall x \in Q$ we assume non-empty $R(e, x)$

Restrictions in Discrete Part

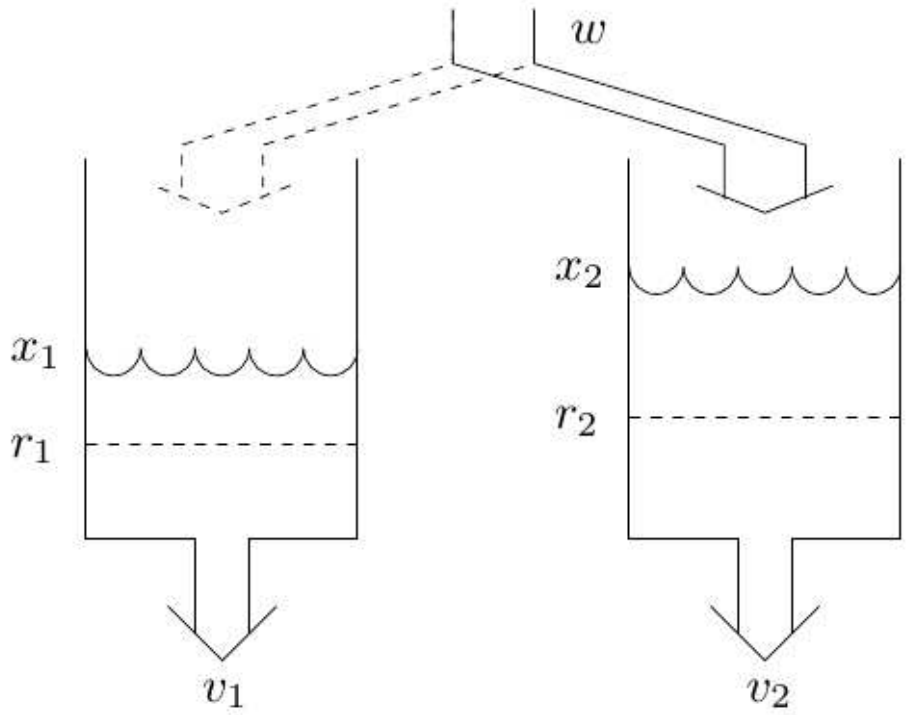
- The set of discrete state is finite.

Example 2 – Water Tank



- Two water tanks, volume of water denoted with x_1 and x_2.
- There is a constant speed leak from both tanks, v_1 and v_2.
- A hose can fill one of the tanks with speed w.
- The hose is always in exactly one of the tanks.

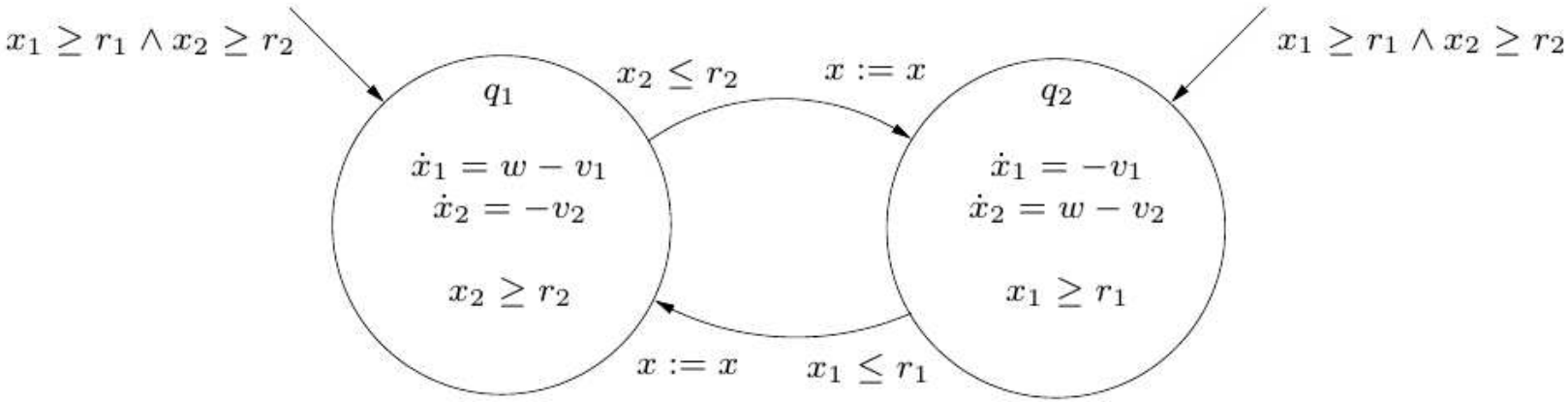
Example 2 – Water Tank



Goal

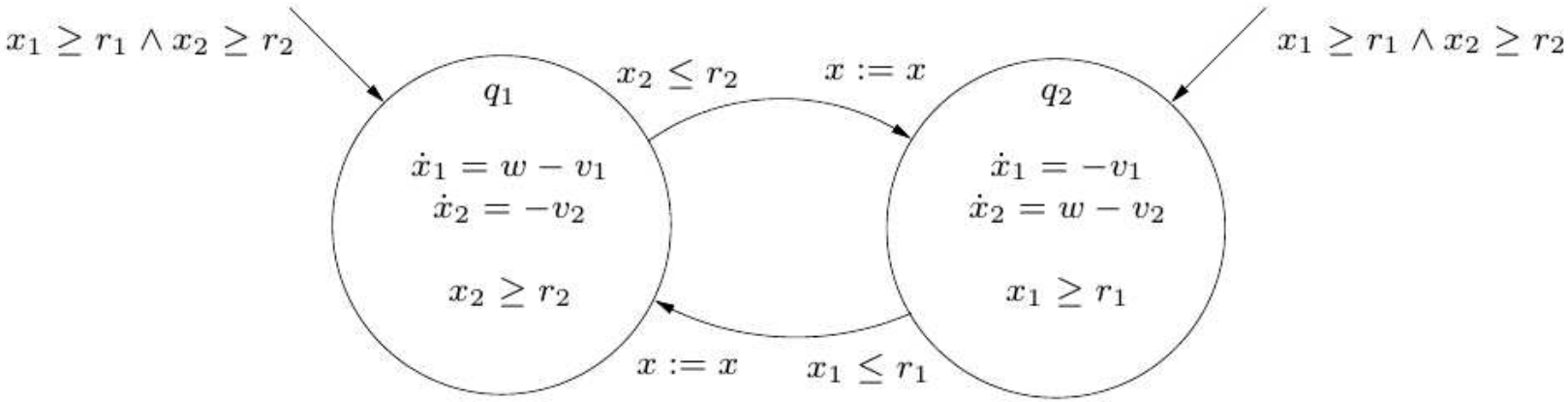
- Keep water level above the necessary minimum r_1 and r_2.
- Initially, there is enough water in both tanks.
- The hose is switched to a tank at the moment the water level in the tank drops to the required minimum.

Water Tanks — Formal Definition of the System



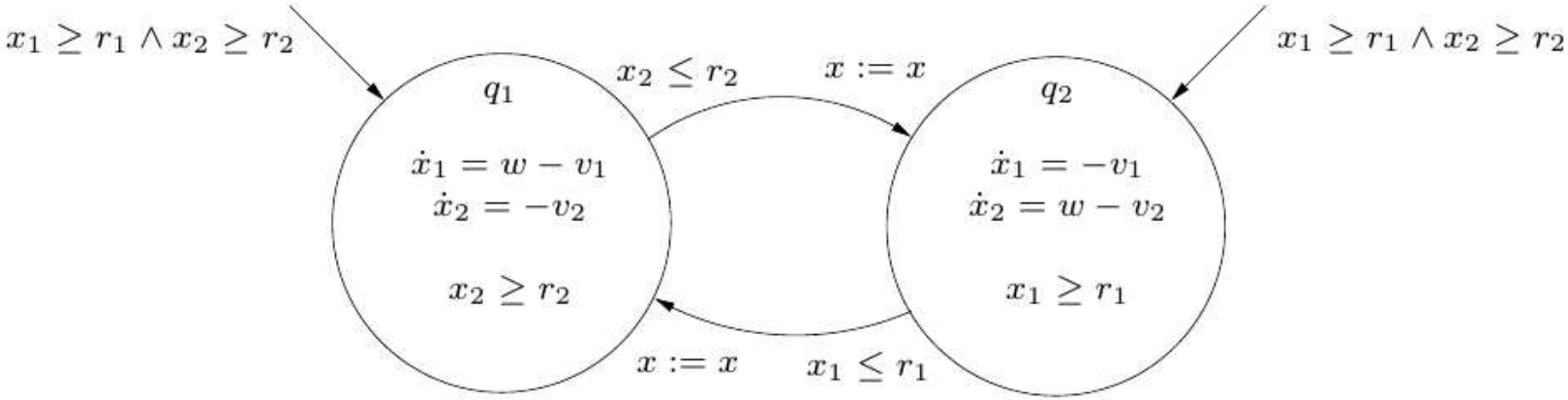
- $Q = \{q_1, q_2\}$
- $X = \mathbf{R} \times \mathbf{R}$
- $f(q_1, x) = \begin{bmatrix} w - v_1 \\ -v_2 \end{bmatrix} \quad f(q_2, x) = \begin{bmatrix} -v_1 \\ w - v_2 \end{bmatrix}$

Water Tanks — Formal Definition of the System



- $\mathit{Init} = \{q_1, q_2\} \times \{x \in \mathbf{R} \times \mathbf{R} \mid x_1 \geq r_1 \wedge x_2 \geq r_2\}$
- $\mathit{Dom}(q_1) = \{x \in \mathbf{R} \times \mathbf{R} \mid x_2 \geq r_2\}$
  $\mathit{Dom}(q_2) = \{x \in \mathbf{R} \times \mathbf{R} \mid x_1 \geq r_1\}$

Water Tanks — Formal Definition of the System



- $E = \{(q_1, q_2), (q_2, q_1)\}$
- $G(q_1, q_2) = \{x \in \mathbf{R} \times \mathbf{R} \mid x2 \leq r2\}$
  $G(q_2, q_1) = \{x \in \mathbf{R} \times \mathbf{R} \mid x_1 \leq r_1\}$
- $R(q_1, q_2, x) = R(q_2, q_1, x) = \{x\}$

Hybrid Time Sequence (HTS)

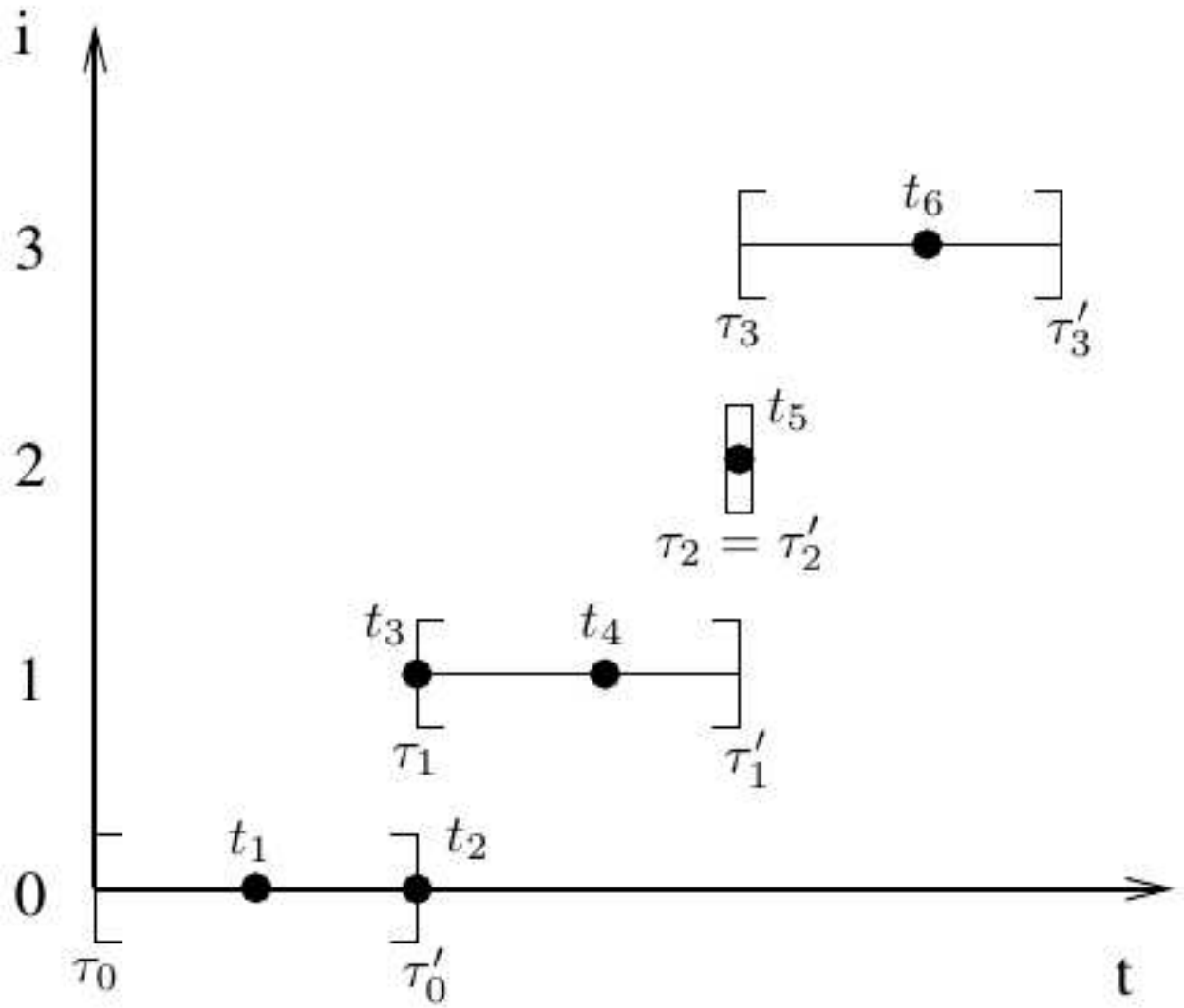
Informally

- A run of hybrid automaton proceeds in a sequence of continuous time intervals. Discrete transitions happen on the boundaries of the intervals in instant time.
- The time characteristic of a run of hybrid automaton is formalised with the usage of the so called **Hybrid Time Sequence**.

Definitions

- Hybrid Time Sequence is a (finite or infinite) sequence of intervals $\tau = \{I_0, I_1, \ldots, I_N\} = \{I_i\}_{i=0}^{N}$ such that:
  - $I_i = [\tau_i, \tau_i']$ for all $i < N$
  - If $N < \infty$ then either $I_N = [\tau_N, \tau_N']$ or $I_N = [\tau_N, \tau_N')$
  - $\tau_i \leq \tau_i' = \tau_{i+1}$ for all $0 \leq i < N$.

Graphical Representation of Hybrid Time Sequence



Ordering of Time Moments

Observation

- If every time moment is related with an interval of HTS ...
- ... then time moments can be linearly ordered.

Ordering $\prec$

- $t_1 \in I_i,\ t_2 \in I_j$
- $t_1 \prec t_2 \stackrel{def}{=} (t_1 < t_2) \vee (t_1 = t_2 \wedge i < j)$

Generalisation

- Every hybrid time sequence is linearly ordered with $\prec$ relation.

Ordering of Hybrid Time Sequence

Prefix Of Hybrid Time Sequence

- $\tau = \{I_i\}_{i=0}^{N}$
- $\hat{\tau} = \{\hat{I}_i\}_{i=0}^{M}$
- We say that τ is a prefix of $\hat{\tau}$ (denoted with $\tau \sqsubseteq \hat{\tau}$), if

  $\tau = \hat{\tau}$, or

  N is finite $\wedge\ I_N \subseteq \hat{I}_N \wedge \forall i \in [0, N) : I_i = \hat{I}_i$

Proper Prefix

- $\tau \sqsubset \hat{\tau} \quad \equiv \quad \tau \sqsubseteq \hat{\tau} \ \wedge \ \tau \neq \hat{\tau}$

Relation $\sqsubseteq$ is a Partial Ordering

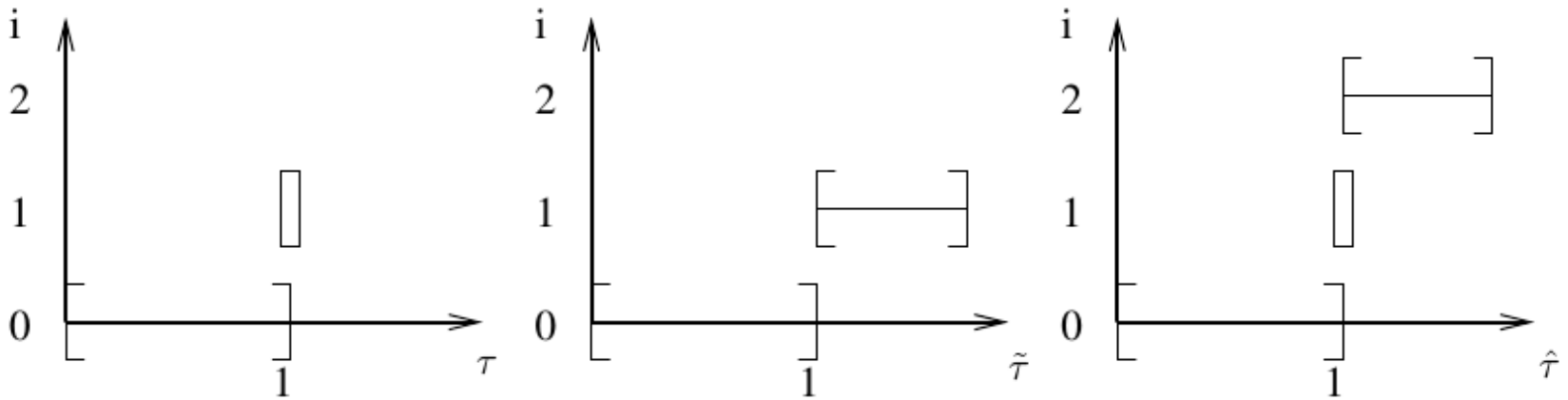
- There exist τ and $\widehat{\tau}$ such that $\tau \not\sqsubseteq \widehat{\tau}$ and $\widehat{\tau} \not\sqsubseteq \tau$.

Task

Task – Find $\tau, \widetilde{\tau}$ and $\widehat{\tau}$ such that

- $\tau \sqsubseteq \widetilde{\tau}$
- $\tau \sqsubseteq \widehat{\tau}$
- $\widetilde{\tau} \not\sqsubseteq \widehat{\tau} \wedge \widehat{\tau} \not\sqsubseteq \widetilde{\tau}$

Solution



Hybrid Trajectories

Definition

- Hybrid trajectory is a triple (τ, q, x), where τ is hybrid time sequence $\tau = \{I\}_0^N$ and q, x are two sequences of functions $q = \{q_i\}_0^N$ and $x = \{x_i\}_0^N$ such that $q_i : I_i \to Q$ and $x_i : I_i \to \mathbf{R}^n$, respectively.

Intuition

- Continuous part flows within individual time intervals of hybrid time sequence.
- Discrete state within a single interval does not change.
- Discrete transitions realise transitions from the end of one interval to the beginning of the succeeding interval.

Run of Hybrid Automaton

Run of Hybrid Automaton

- Let $\mathcal{H} = (Q, X, f, \mathit{Init}, \mathit{Dom}, E, G, R)$ be hybrid automaton.
- Let (τ, q, x) be hybrid trajectory.
- Trajectory (τ, q, x) is a run of automaton $\mathcal{H}$, if it is compliant with $\mathcal{H}$ in: initial condition, discrete behaviour and continuous behaviour.

Initial Condition

- $(q_0(0), x_0(0)) \in \mathit{Init}$

Discrete Behaviour – For all $i < N$ it holds that

- $(q_i(\tau_i'), q_{i+1}(\tau_{i+1})) \in E$
- $x_i(\tau') \in G(q_i(\tau_i'), q_{i+1}(\tau_{i+1}))$
- $x_{i+1}(\tau_{i+1}) \in R(q_i(\tau_i'), q_{i+1}(\tau_{i+1}), x_i(\tau_i'))$

Run of Hybrid Automaton

Run of Hybrid Automaton

- Let $\mathcal{H} = (Q, X, f, Init, Dom, E, G, R)$ be hybrid automaton.
- Let (τ, q, x) be hybrid trajectory.
- Trajectory (τ, q, x) is a run of automaton $\mathcal{H}$, if it is compliant with $\mathcal{H}$ in: initial condition, discrete behaviour and continuous behaviour.

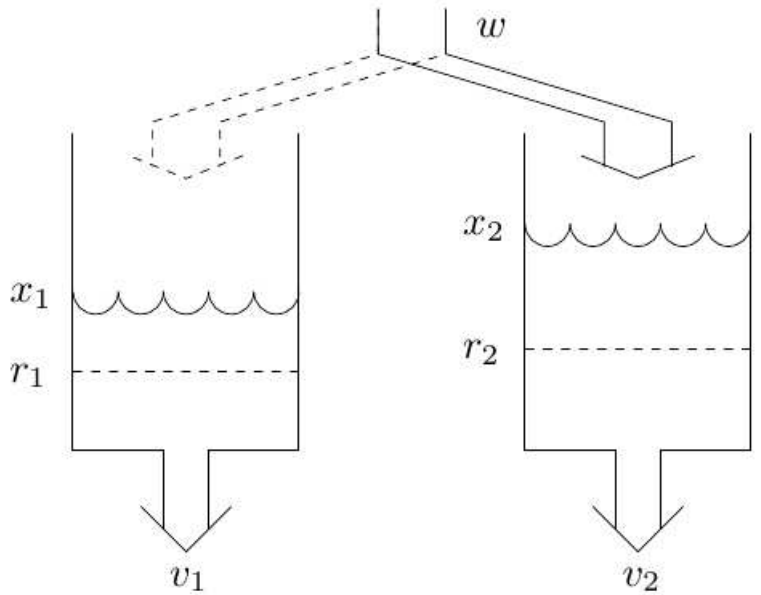
Continuous Behaviour – For all $i \leq N$ it holds that

- $q_i : I_i \to Q$ is constant over $t \in I_i$,
- $x_i : I_i \to X$ is a solution to differential equation

$$\frac{dx_i(t)}{dt} = f(q_i(t), x_i(t))$$

  over I_i beginning in $x_i(\tau_i)$,
- For all $t \in [\tau_i, \tau_i')$ it holds that $x_i(t) \in Dom(q_i(t))$.

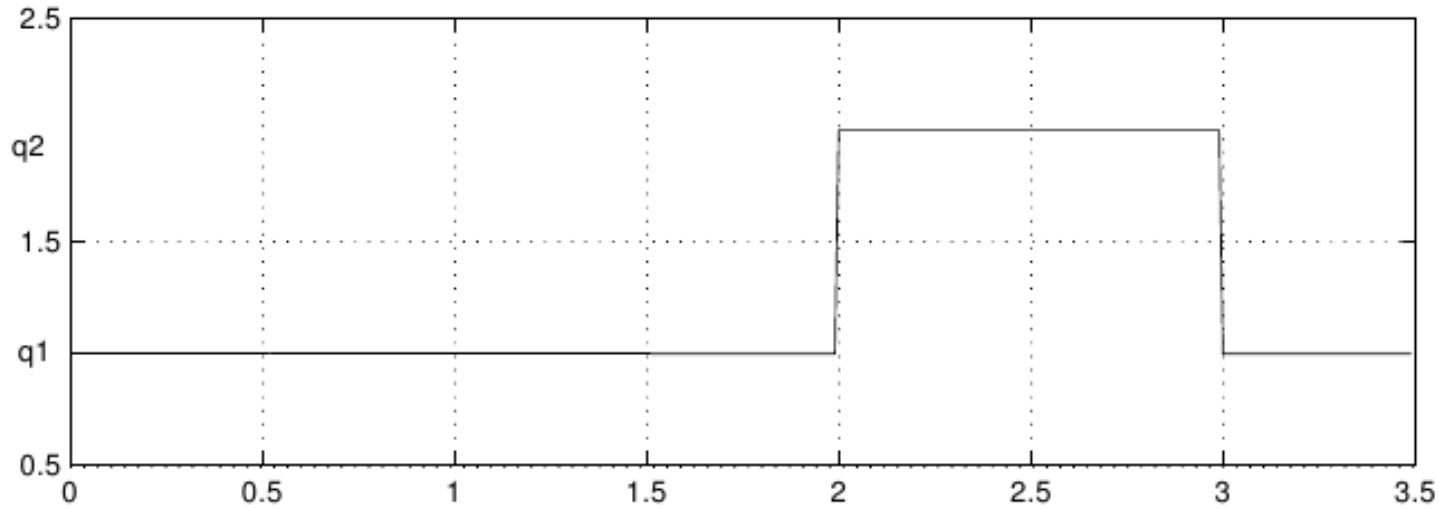
Water Tanks – Example

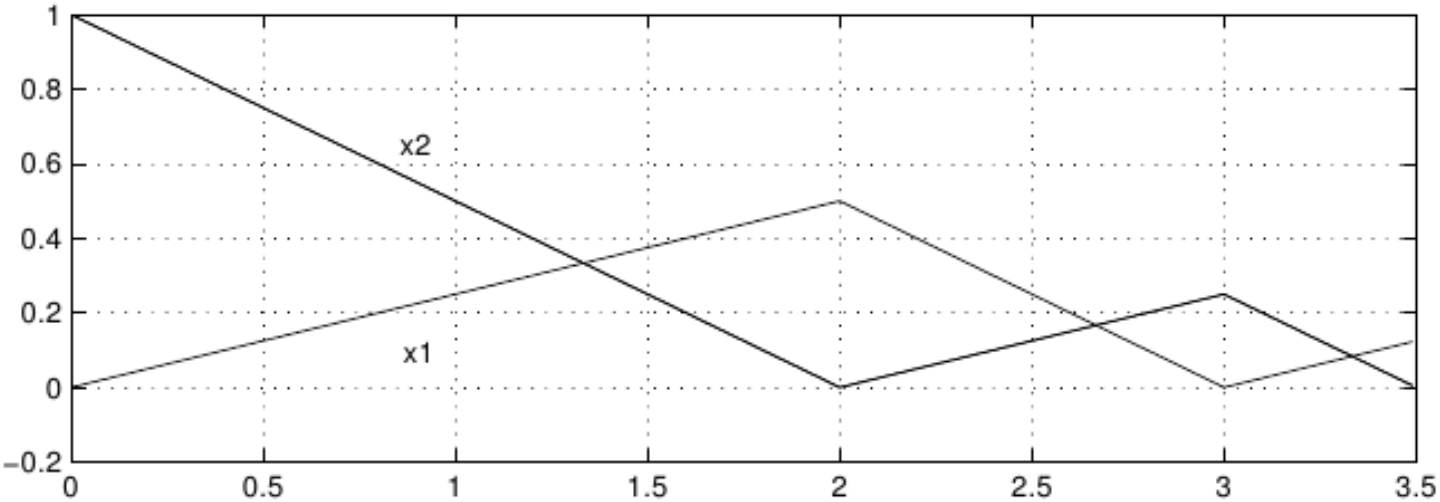


Specification

- $\tau = \{[0, 2], [2, 3], [3, 3.5]\}$
- Constants $r_1 = r_2 = 0$, $v_1 = v_2 = 0.5$, $w = 0.75$
- Initial state $q = q_1$, $x_1 = 0$, $x_2 = 1$.

Water Tanks – Trajectories





Classification of Runs (τ, q, x)

Finite

- If τ is finite and the last interval of τ is closed.

Infinite

- If τ is infinite sequence, or the sum of time intervals in τ is infinite, i.e.

$$\Sigma_{i=0}^{N}(\tau_i' - \tau_i) = \infty.$$

Zeno

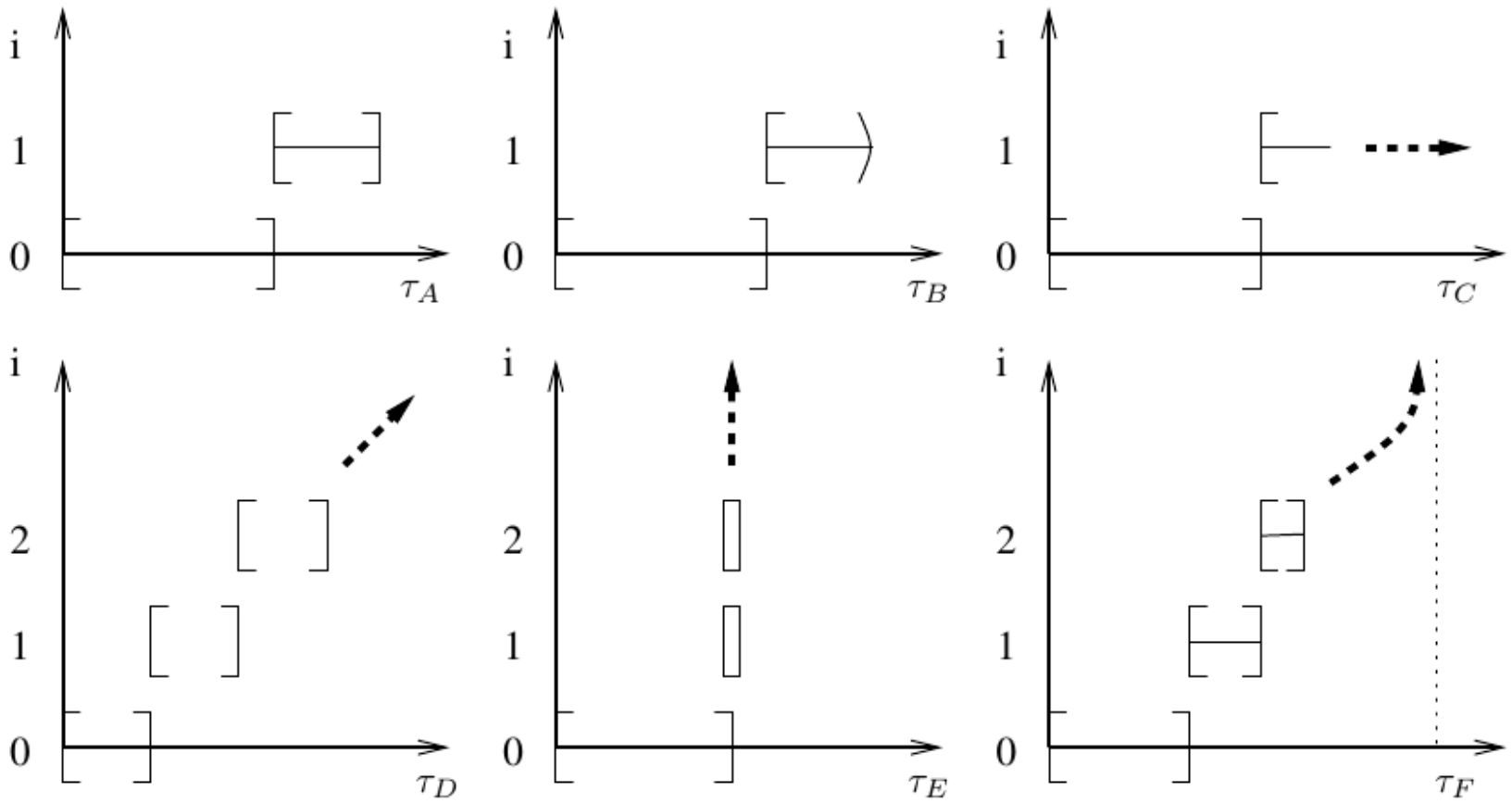
- If τ is infinite, but

$$\Sigma_{i=0}^{N}(\tau_i' - \tau_i) < \infty.$$

Maximal

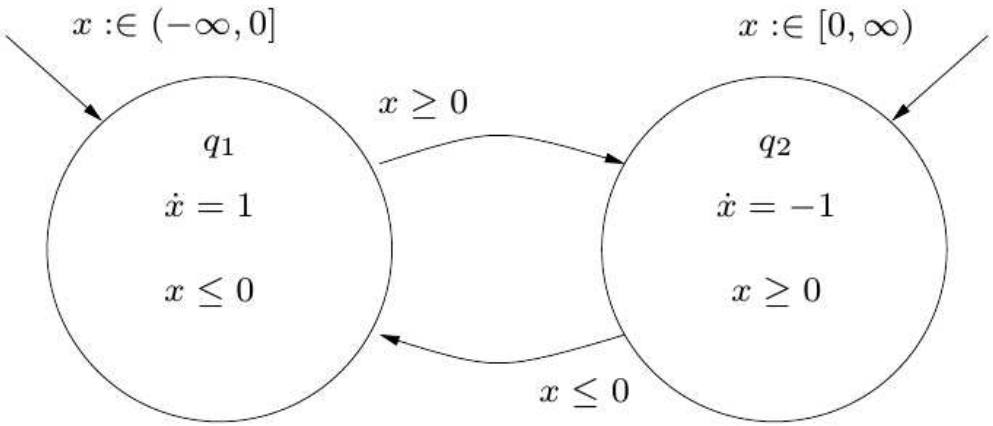
- If τ is no proper suffix of any other run τ' of $\mathcal{H}$.

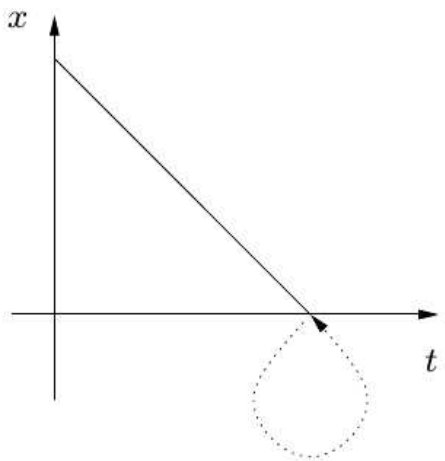
Classification of Runs



τ_A finite; τ_C and τ_D infinite; τ_E and τ_F Zeno.
What class is run τ_B?

Examples of ZENO Runs





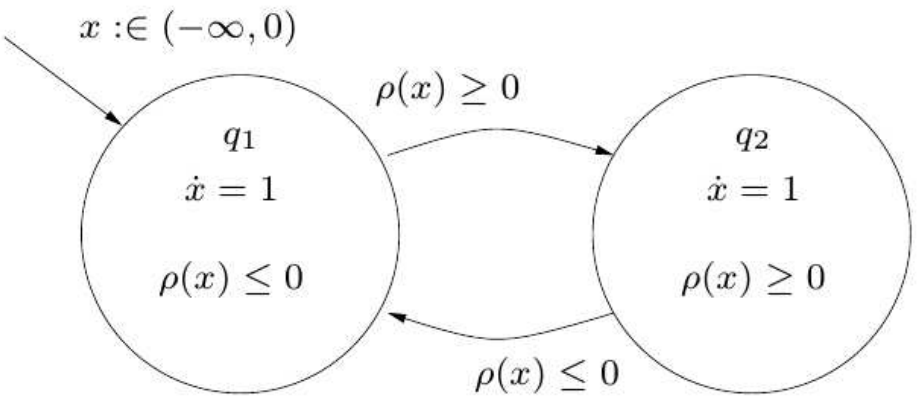
Examples of ZENO Runs

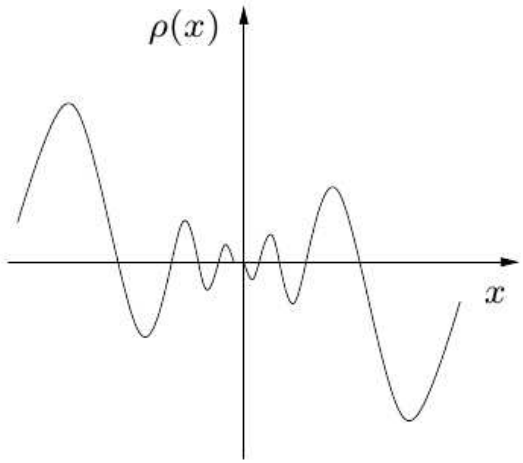
Let

$$\rho(x) = \begin{cases} \sin\left(\frac{1}{x^2}\right) \exp\left(-\frac{1}{x^2}\right) & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases}$$

Then

- the following hybrid system has infinitely many intersections with x axis in the interval $(-\epsilon, 0]$.





Modelling Hybrid Systems

Observation

- Hybrid automata are meant to describe real hybrid systems.
- Due to abstraction and simplification, it is possible to specify unrealistic situation.

Risk of Modelling

- Can create system that have no solutions.
- Can create system that have only unrealistic solutions.
- Can create system that have non-deterministic solutions.

Terminology

- System that has no solution (no run exist) is called *blocking* system.

Unrealistic Runs

Observation

- Non-blocking system does not guarantee that some runs are realistic.
- Non-blocking system does not guarantee that some runs are time divergent.

Unrealistic Runs

- Runs that perform infinitely many discrete transitions in finite time are called **ZENO** runs.
- Created by abstraction and simplification in modelling.

Discussion

- Why the ball does not bounce forever?
- It is important to see which simplification lead to ZENO runs.

Non-determinism

Non-determinism

- In general can be described as absence of unique solutions, i.e. that a hybrid automaton accepts multiple different runs emanating from the same initial conditions.
- When limited to Lipschitz continuous functions with unique solution, reason for non-determinism comes from discrete transitions.

Non-deterministic on Purpose

- Can be used to model uncertainty.
- Modeller should make difference between non-determinism due to simplification, and non-determinism used on purpose.

Observation

- Non-determinism is a real cause of troubles in both analysis and controller synthesis of hybrid systems.

Problems of Simulations and Analysis of Hybrid Systems

Existence of Solution

- How to detect existence of non-blocking run?
- How to detect ZENO behaviour?

Uniqueness

- How to perform simulation of non-deterministic system?
  - Discrete transition vs. continuous time evolution.
  - Discrete transition vs. discrete transition.
- As-soon-as semantics.

Discontinuity

- How to detect satisfiability of transition guards?
- What if state invariant ends with open interval $[a, b)$ and the succeeding transition is allowed to execute at time $[b]$?

Composition

- How to compose hybrid automata?

Non-blocking and Deterministic Hybrid Automaton

Non-blocking Hybrid Automaton

- Hybrid automaton H is called non-blocking if for all initial states $(\hat{q}, \hat{x}) \in \mathit{Init}$ there exist an infinite run emanating from $(\hat{q}, \hat{x})$.

Deterministic Hybrid Automaton

- Hybrid automaton H is called deterministic, if for all initial states $(\hat{q}, \hat{x}) \in \mathit{Init}$ there exist at most one maximal run emanating from $(\hat{q}, \hat{x})$.

Using Hybrid Automata

Analysis and Synthesis Hybrid Systems (HS)

Motivation for Modelling

- The goal of modelling of HS is to deduce properties of, or synthesise controllers for real HS from properties of, or controllers for modelled HS.

Verification

- Does hybrid system exhibits declared behaviour (does it satisfy specification)?

Synthesis

- There are number of choices to build a HS, using models it is possible to decide which choices are good and which are bad prior the construction of the real HS.

Validation

Validation

- Check that the design described as a hybrid automaton and the real product produced behave accordingly.
- Some system modelled with hybrid automata may be unrealistic (and cannot be built) due to simplifications and abstractions used during modelling phase.

Usual Work-flow

- Synthesis (of model)
- Verification (of model)
- Validation (equivalence of model and real product)

Specification

Stability

- Typical property of purely continuous systems.
- To request stability for hybrid systems requires to specify what does the stability means with respect to the discrete part of the system.

Specification by Set of States

- Specification of safety and forbidden areas.

Specification by Set of Trajectories

- Properties of hybrid systems that can be expressed as properties of runs.
- Set of allowed runs of a hybrid automaton.
- Formally described using modal and temporal logic, such as (CTL, LTL, CTL*).

Methods of Analysis of HS

Deductive Methods

- Using math reasoning, such as math induction, to deduce properties of hybrid systems.

Model Checking

- Algorithmic procedure for deciding formally specified properties of hybrid systems.
- Decidable only for limited sub-classes of hybrid automata.

Simulations

- Used to estimate the set of reachable states.
- The precision of estimation is difficult to say.

Deductive Methods – Invariant Set

Typical Goal

- Typical goal for deductive methods is to set boundaries of the *reach* set using the so called **Invariant Set.**
- Invariant set is a set of states for which it holds that if a run of hybrid system is initiated at the state of the set it only visits states that are in the set (i.e. never leaves invariant set).

Formal Definition of Invariant Set

- Set of state $M \subseteq Q \times X$ of hybrid automaton H is called *invariant* if for all $(\hat{q}, \hat{x}) \in M$, all solutions (τ, q, x) starting from $(\hat{q}, \hat{x})$, all $I_i \in \tau$ and all $t \in I_i$ it holds that $(q_i(t), x_i(t)) \in M$.

Deductive Methods – Properties of Invariant Set

Observation

- Union and Intersection of Invariant Sets of hybrid automaton H is also an invariant set for H.
- If M is an invariant set and $Init \subseteq M$, then $Reach \subseteq M$.

Consequence

- To approximate the $Reach$ set it is possible to deduce a number of invariant sets that contain initial state and are at the same time below the set of all states of hybrid automaton (here denoted by F)

$$Init \subseteq M \subseteq F$$

and intersect them.

Model Checking

Simplification

- For hybrid automata we restrict ourselves in the course to algorithmic test of reachability of a given state.

Considered Sub-classes of Hybrid Automata

- Timed Automata (TA).
- Rectangular Hybrid Automata (RHA).
- Linear Hybrid Automata (LHA).

Software Tools

- UPPAAL – Timed Automata
- PHaVer – RHA, partially LHA (HyTech)

Timed Automata

Restriction

- All derivations to drive continuous evolution of the automaton has the form of:

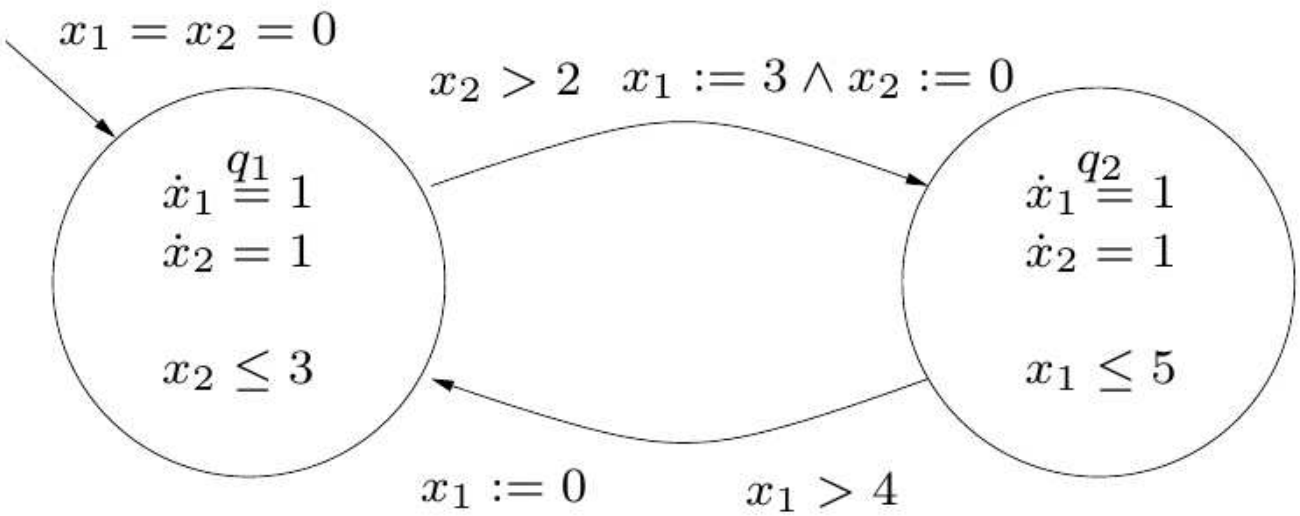
$$\frac{dx_i(t)}{dt} = 1$$

- Resets R of discrete transitions are allowed either to keep the value of the continuous variable, or to reset it to 0.
- *Dom* and G are defined only using relations $\leq$ and $\geq$ with respect to integral values.

Intuition

- Finite automaton with a set of continuous variables to measure elapsed time.
- Measured time values may be reset to 0 using discrete transition.

Example of Timed Automaton



Exercise

- In two-dimensional graph with axes x_1 and x_2 show how the values of continuous variables change.

Region Abstraction

Key Observation

- With respect to the restriction that comparisons are made only against integral values, two floating point values that have the same integral part cannot be differentiated.

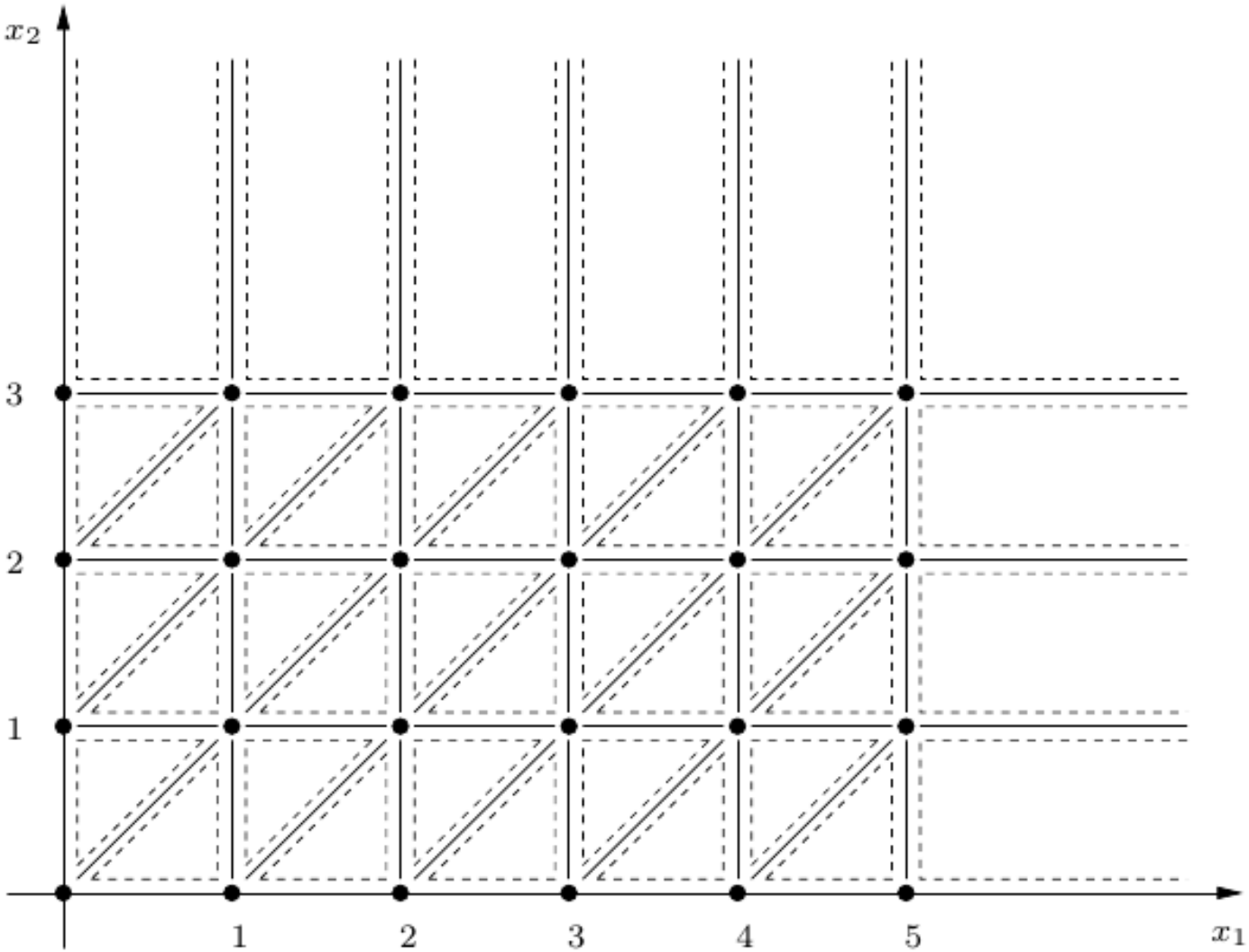
Equivalence Classes on the Continuous Domain

- If c is the greatest integral number used in a guard of timed automaton then the continuous domain can be represented with a finite set of intervals as follows:

$$[0], (0,1), [1], (1,2), [2], \ldots [c-1], (c-1, c), [c], (c, \infty)$$

- Abstracted domain is finite for every continuous variable.
- It is possible to construct finite-state automaton that faithfully simulates behaviour of the timed automaton.
- This can be used for verification purposes.

Region Abstraction



Rectangular Hybrid Automata (RHA)

Restriction

- All derivations to drive continuous evolution of the automaton has the form of:

$$a \leq \frac{dx_i(t)}{dt} \leq b,$$

  where a and b are rational constants.
- When specifying RHA no derivation equations are given, just the boundary constants a and b.

Exercise

- Consider a RHA with two continuous variables x_1 and x_2.
- On two-dimensional graph with axes x_1 and x_2 show the evolution of values of the continuous variables.
- Guess the origin of the name of this particular sub-class of hybrid automata.

Reachability is Decidable for RHA

Reachability

- Reachability problem for RHA is decidable if there are only finitely many values to which a continuous variable may be reset by a discrete transition.
- The most general sub-class of hybrid automata for which reachability is still decidable.

Going Beyond Means Undecidability

- Relaxation from restriction of resets is known to result in sub-class of hybrid automata for which the reachability problem is undecidable.

Linear Hybrid Automata (LHA)

Definition

- Let $k_0, \ldots, k_m$ be numeric constants and $x_1, \ldots, x_m$ variables. An expression in the form of $k_0 + k_1 x_1 + k_2 x_2 + \cdots + k_m x_m$ is called a *linear expression*.
- Let t_1, t_2 be linear expressions. An expression of the form $t_1 \leq t_2$ is called linear inequality.
- Hybrid automaton H is called **linear hybrid automaton** (LHA), if *Init*, *Dom*, G and f are defined as Boolean combinations of linear inequalities.

Undecidability

- The reachability problem for LHA is undecidable.
- Algorithms implemented for the LHA sub-class are incomplete (HyTech).

Homework

Homework

- Will Lake Mead go dry? (SPACEex tool).
  `http://spaceex.imag.fr/documentation/tutorials`

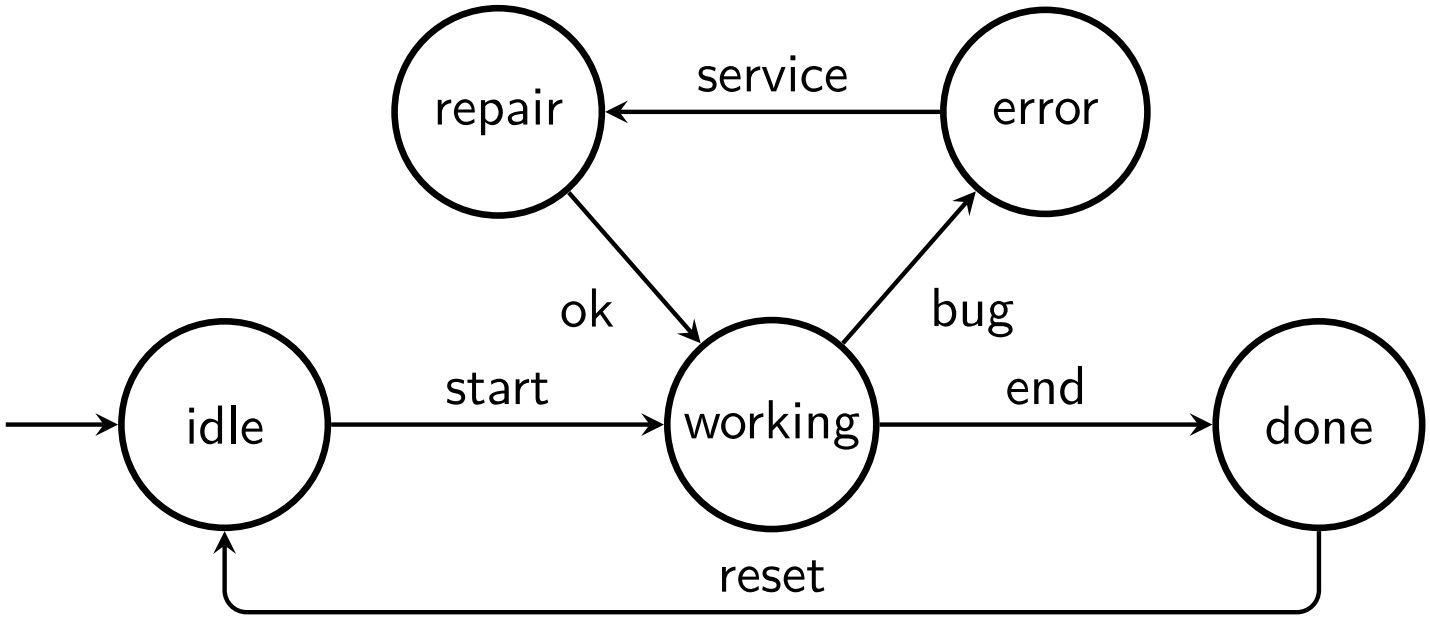
IA169 System Verification and Assurance

Verification of Systems with Probabilities

Vojtěch Řehák

Motivation example

Fail-repair system

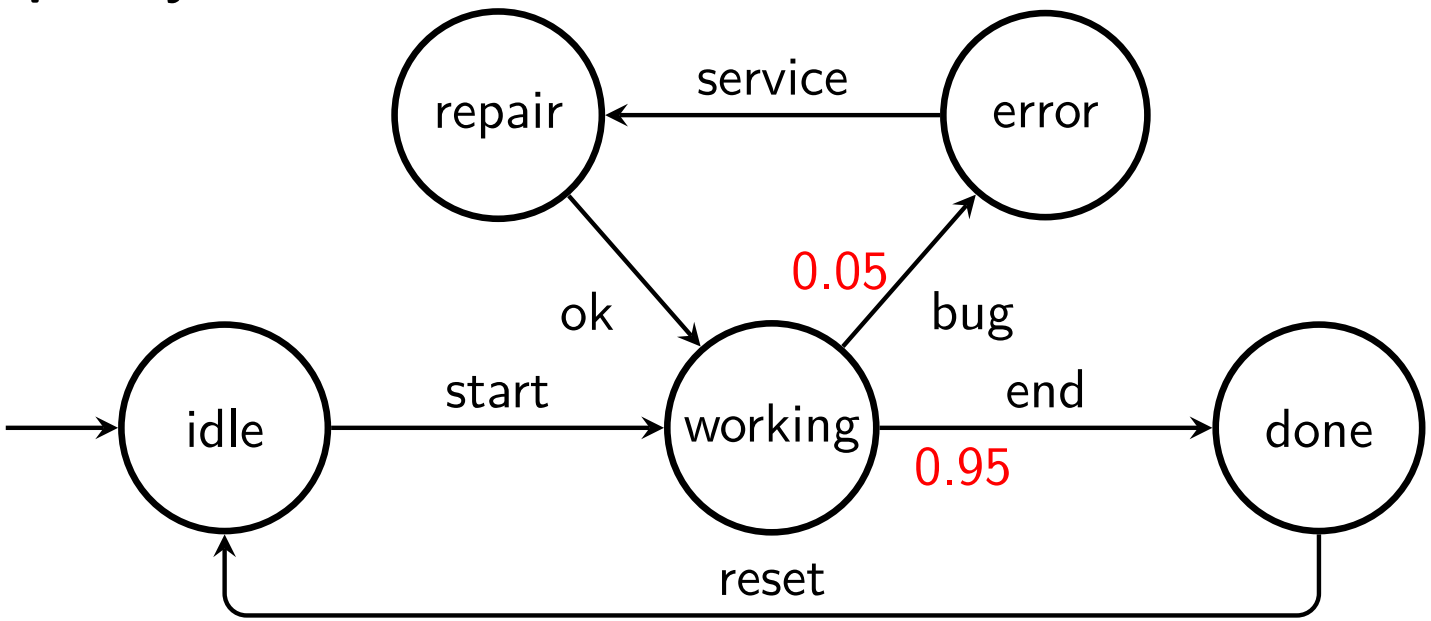


What are the properties of the model?

- $G(\text{working} \implies F\ \text{done})$ **NO**
- $G(\text{working} \implies F\ \text{error})$ **NO**
- $FG(\text{working} \vee \text{error} \vee \text{repair})$ **NO**

Motivation example

Fail-repair system



- What is the probability of reaching "done" from "working" with no visit of "error"?
- What is the probability of reaching "done" from "working" with at most one visit of "error"?
- What is the probability of reaching "done" from "working"?

Discrete-time Markov Chains (DTMC)

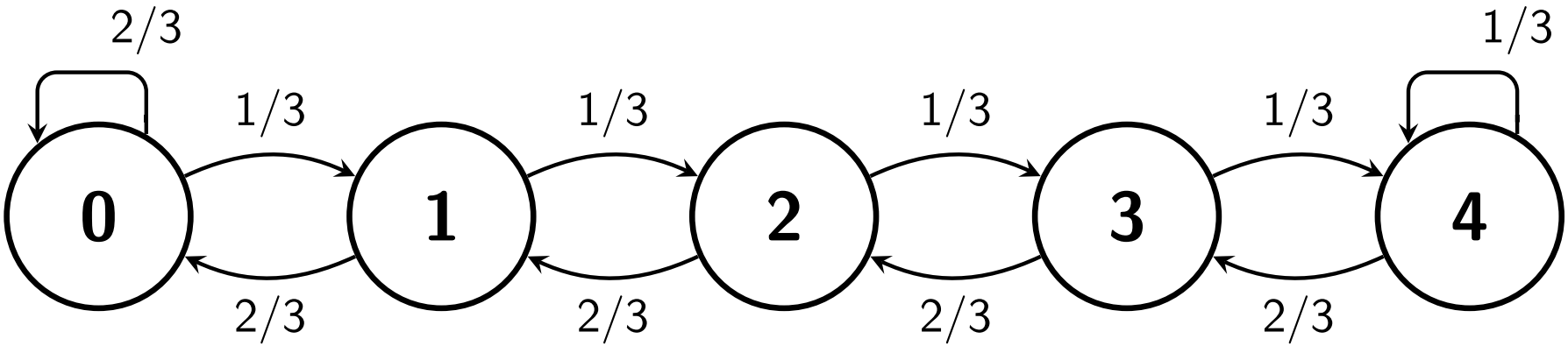
Probabilistic models

Discrete-time Markov Chains (DTMC)

- Standard model for probabilistic systems.
- State-based model with probabilities on branching.
- Based on the current state, the succeeding state is given by a discrete probability distribution.
- Markov property ("memorylessness") — only the current state determines the successors (the past states are irrelevant).
- Probabilities on outgoing edges sums to 1 for each state.
- Hence, each state has at least one outgoing edge ("no deadlock").

DTMC examples

Model of a queue

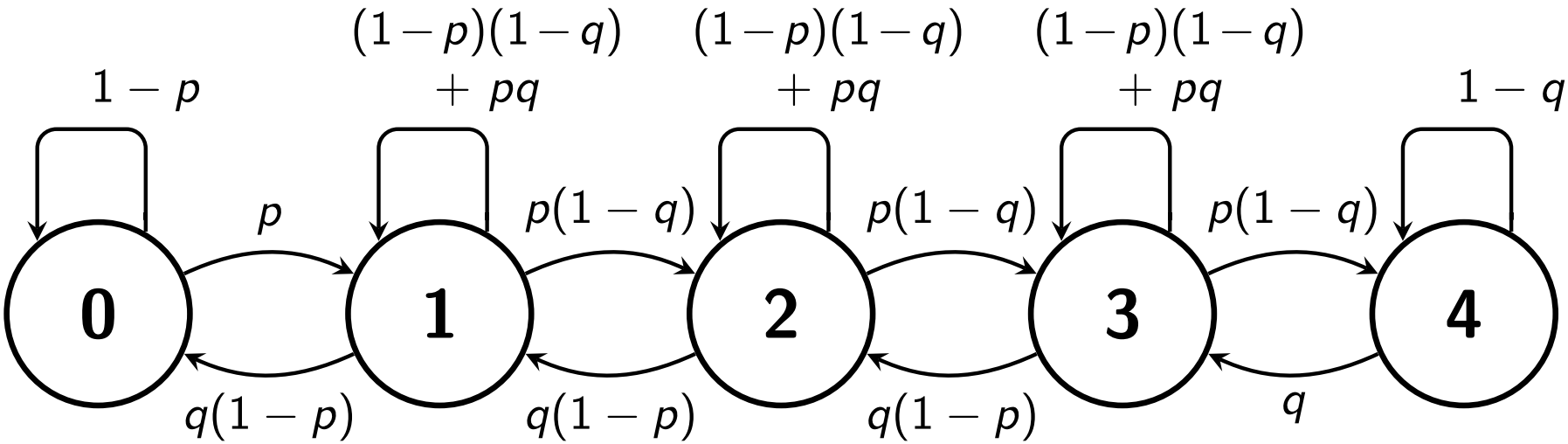


Queue for at most 4 items. In every time tick, a new item comes with probability $1/3$ and an item is consumed with probability $2/3$.

What if a new items comes with probability $p = 1/2$ and an item is consumed with probability $q = 2/3$?

DTMC examples

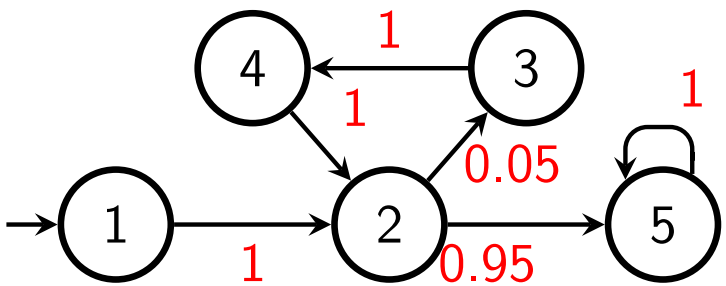
Model of the new queue



DTMC - formal definition

Discrete-time Markov Chain is given by

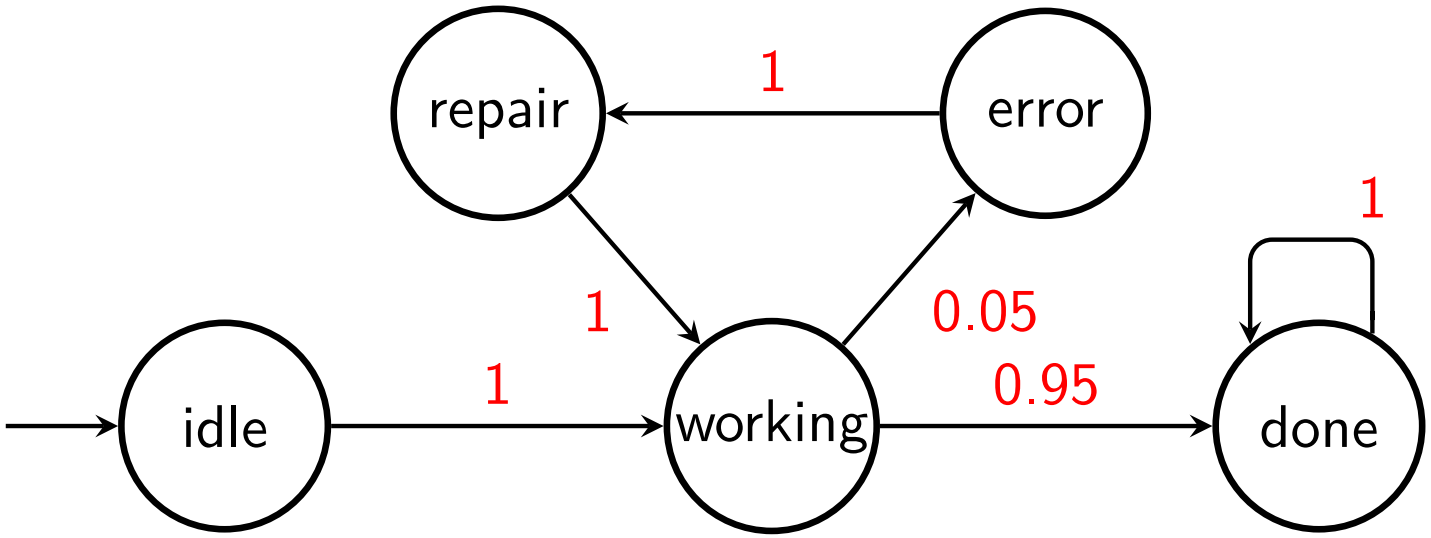
- a set of states S,
- an initial state s_0 of S,
- a probability matrix $P : S \times S \to [0, 1]$, and
- an interpretation of atomic propositions $I : S \to AP$.



$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.05 & 0 & 0.95 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Back to our questions

Fail-Repair System



- What is the probability of reaching "done" from "working" with no visit of "error"?
- What is the probability of reaching "done" from "working" with at most one visit of "error"?
- What is the probability of reaching "done" from "working"?

Markov chain analysis

Transient analysis

- distribution after k-steps
- reaching/hitting probability
- hitting time

Long run analysis

- probability of infinite hitting
- stationary (invariant) distribution
- mean inter visit time
- long run limit distribution

Property Specification

Property specification languages

Recall some non-probabilistic specification languages:

LTL formulae

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\,\varphi \mid \varphi\,U\,\varphi$$

CTL formulae

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\,U\,\varphi] \mid EG\,\varphi$$

Syntax of CTL*

state formula	$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid E\,\psi$
path formula	$\psi ::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid X\,\psi \mid \psi\,U\,\psi$

Property specification languages

We need to quantify probability that a certain behaviour will occur.

Probabilistic Computation Tree Logic (PCTL)

Syntax of PCTL

$$\begin{array}{ll} \text{state formula} & \varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid P_{\bowtie b}\psi \\ \text{path formula} & \psi ::= X\,\varphi \mid \varphi\, U\, \varphi \mid \varphi\, U^{\leq k}\, \varphi \end{array}$$

where

- $b \in [0, 1]$ is a probability bound,
- $\bowtie \in \{\leq, <, \geq, >\}$, and
- $k \in \mathbf{N}$ is a bound on the number of steps.

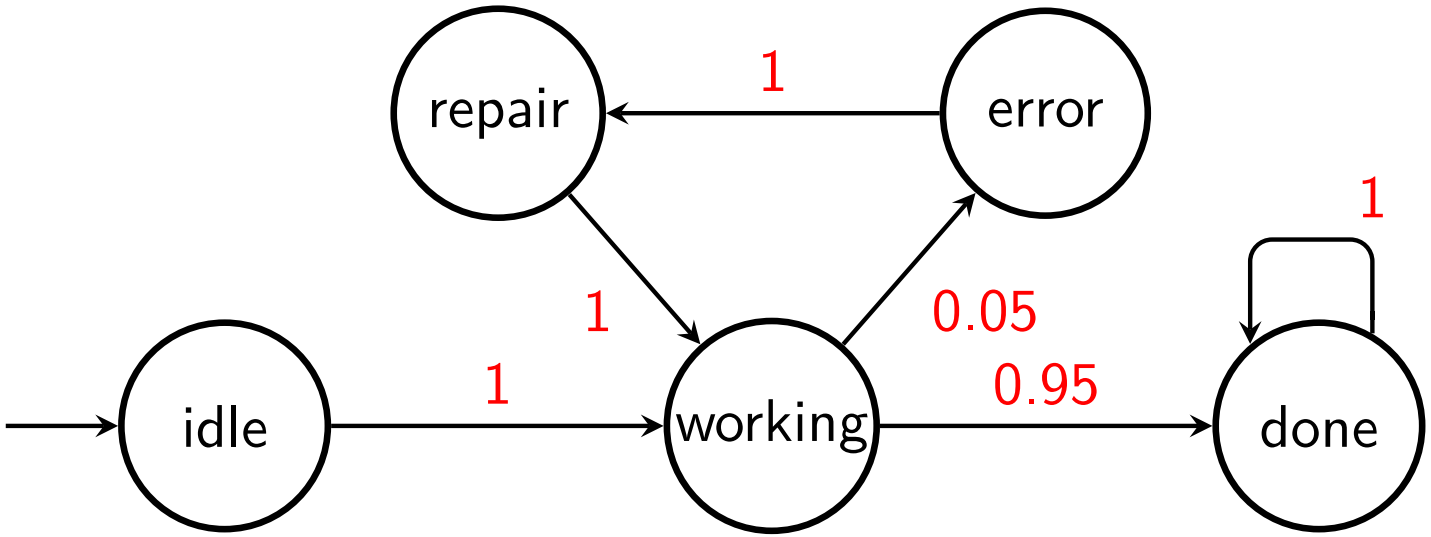
A PCTL formula is always a state formula.

$\alpha\, U^{\leq k}\, \beta$ is a bounded until saying that α holds until β within k steps.
For $k = 3$ it is equivalent to $\beta \vee (\alpha \wedge X\, \beta) \vee (\alpha \wedge X\, (\beta \vee \alpha \wedge X\, \beta))$.

Some tools also supports $P_{=?}\psi$ asking for the probability that the specified behaviour will occur.

PCTL examples

We can also use derived operators like G, F, $\wedge$, $\Rightarrow$, etc.



Probabilistic reachability $P_{\geq 1}(F\, done)$

- probability of reaching the state *done* is equal to 1

Probabilistic bounded reachability $P_{>0.99}(F^{\leq 6}\, done)$

- probability of reaching the state *done* in at most 6 steps is > 0.99

Probabilistic until $P_{<0.96}((\neg error)\, U\, (done))$

- probability of reaching *done* with no visit of *error* is less than 0.96

Qualitative vs. quantitative properties

Qualitative PCTL properties

- $P_{\bowtie b}\,\psi$ where b is either 0 or 1

Quantitative PCTL properties

- $P_{\bowtie b}\,\psi$ where b is in $(0, 1)$

Qualitative properties

In DTMC where zero probability edges are erased, it holds that

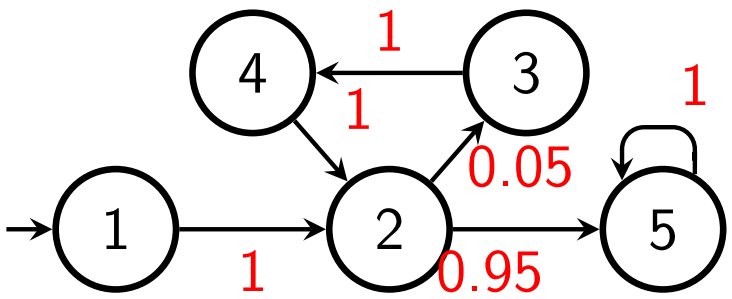
- $P_{>0}(\, X\, \varphi)$ is equivalent to $EX\, \varphi$
  - there is a next state satisfying φ
- $P_{\geq 1}(\, X\, \varphi)$ is equivalent to $AX\, \varphi$
  - the next states satisfy φ
- $P_{>0}(\, F\, \varphi)$ is equivalent to $EF\, \varphi$
  - there exists a finite path to a state satisfying φ

but

- $P_{\geq 1}(\, F\, \varphi)$ is **not** equivalent to $AF\, \varphi$
  (see, e.g., AF *done* on our running example)

There is no CTL formula equivalent to $P_{\geq 1}(\, F\, \varphi)$,
and no PCTL formula equivalent to $AF\, \varphi$.

Quantitative - forward reachability



$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.05 & 0 & 0.95 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Probability distribution after k steps when starting in 1

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \times P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

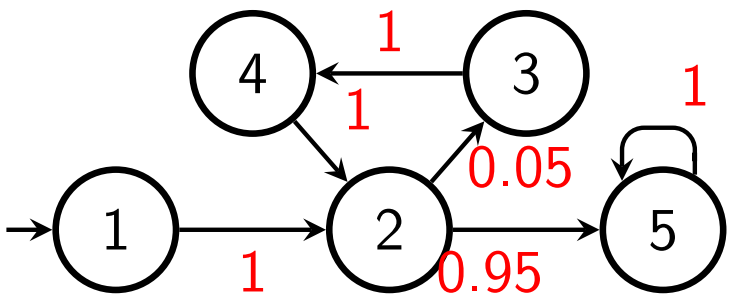
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \times P^2 = \begin{bmatrix} 0 & 0 & 0.05 & 0 & 0.95 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \times P^3 = \begin{bmatrix} 0 & 0 & 0 & 0.05 & 0.95 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \times P^4 = \begin{bmatrix} 0 & 0.05 & 0 & 0 & 0.95 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} \times P^5 = \begin{bmatrix} 0 & 0 & 0.0025 & 0 & 0.9975 \end{bmatrix}$$

Quantitative - backward reachability



$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.05 & 0 & 0.95 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Prob. of being in states 2 or 5 after k steps, i.e. $P_{=?}F^{=k}(2 \vee 5)$

$$P \times \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 1 & 0.95 & 0 & 1 & 1 \end{bmatrix}^T$$

$$P^2 \times \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 0.95 & 0.95 & 1 & 0.95 & 1 \end{bmatrix}^T$$

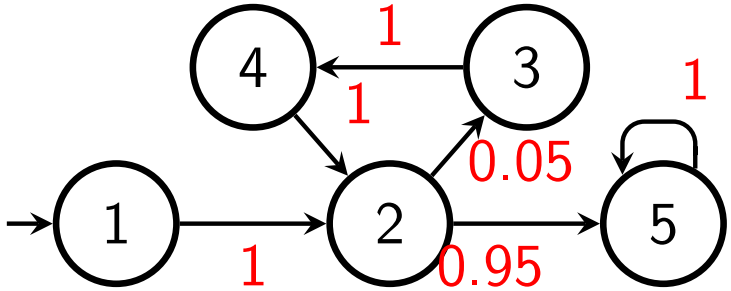
$$P^3 \times \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 0.95 & 1 & 0.95 & 0.95 & 1 \end{bmatrix}^T$$

$$P^4 \times \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 1 & 0.9975 & 0.95 & 1 & 1 \end{bmatrix}^T$$

$$P^5 \times \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 0.9975 & 0.9975 & 1 & 0.9975 & 1 \end{bmatrix}^T$$

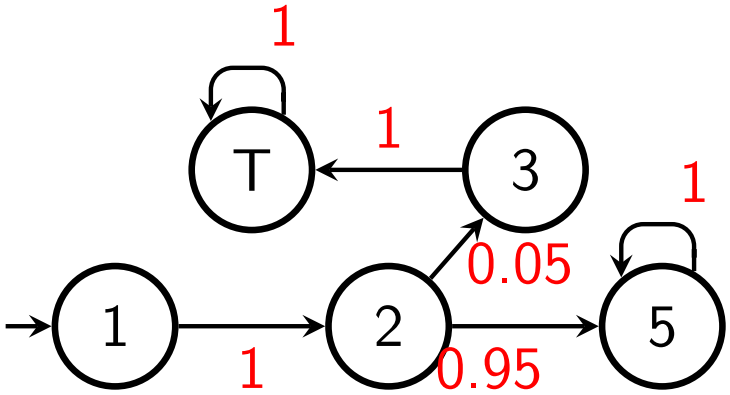
"Up to" reachability

Computing $P_{=?}\ F^{\leq 6}\ 3$.

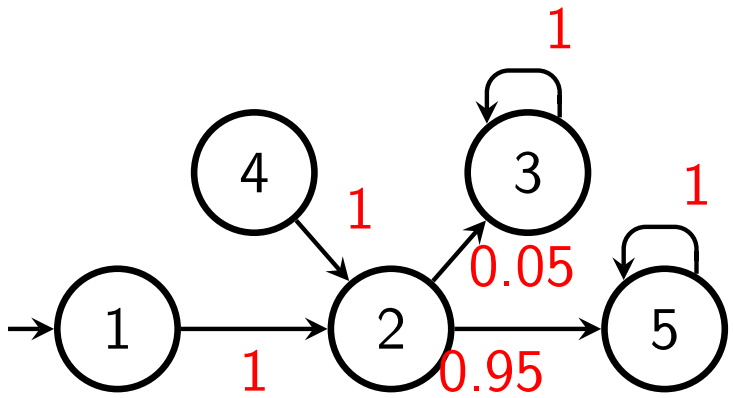


Is it $\sum_{i=0}^{6} P_{=?}\ F^{=i}\ 3$?

No, we are summing probabilities of repeated visits.
It is true when the model is changed such that repeated visits are not possible. Alternativelly we can make the target state is absorbing.



and it is $\sum_{i=0}^{6} P_{=?}\ F^{=i}\ 3$



and it is $P_{=?}\ F^{=6}\ 3$

Unbounded reachability - optional slide

Unbounded reachability

Let $p(s, A)$ be the probability of reaching a state in A from s.

It holds that:

- $p(s, A) = 1$ for $s \in A$
- $p(s, A) = \sum_{s' \in succ(s)} P(s, s') * p(s', A)$ for $s \notin A$

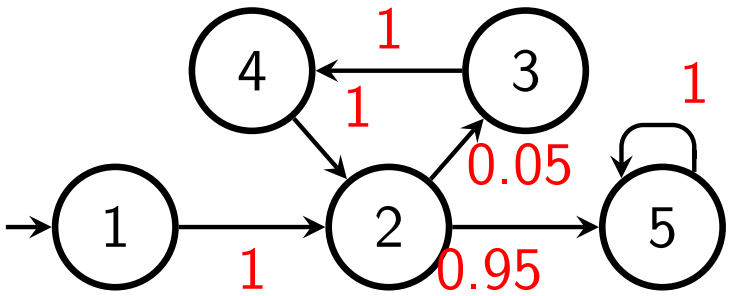
where $succ(s)$ is a set of successors of s and $P(s, s')$ is the probability on the edge from s to s'.

Theorem

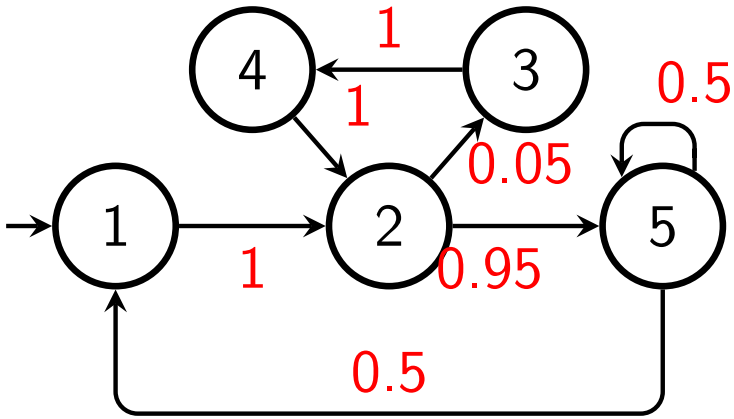
- The minimal non-negative solution of the above equations equals to the probability of unbounded reachability.

Long Run Analysis

Long run analysis



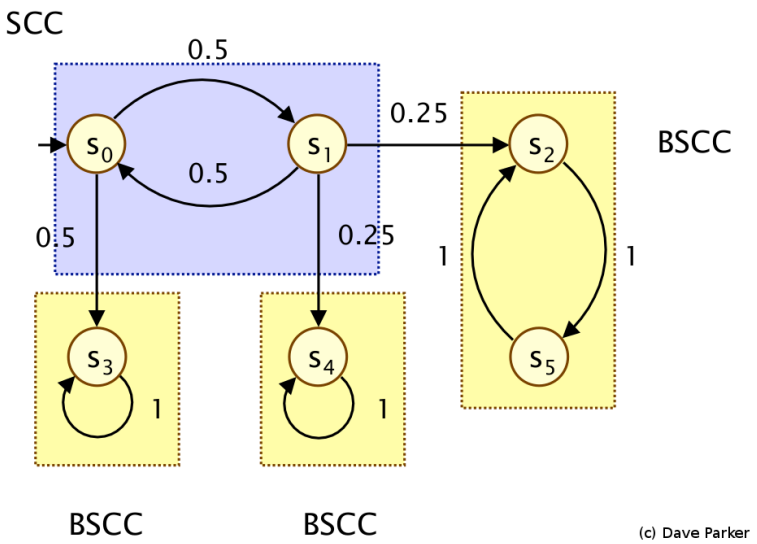
Recall that we reach the state 5(*done*) with probability 1.



What are the states visited infinitely often with probability 1?

States visited infinitely often

Decompose the graph representation onto strongly connected components.



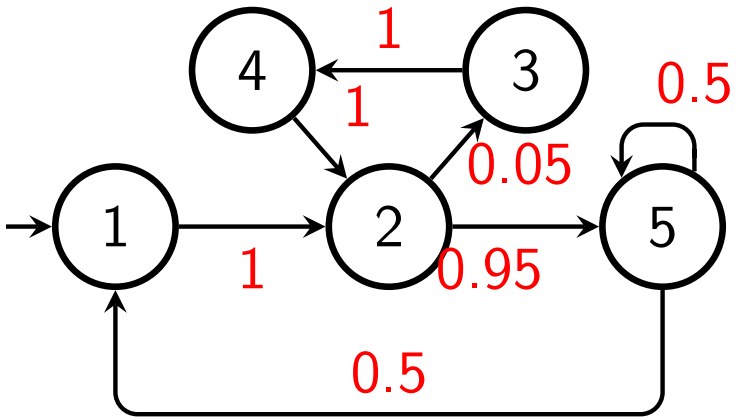
Theorem [1]

- A state is **visited infinitely often** with probability 1 if and only if it is in a **bottom strongly connected component**.
- All other states are **visited finitely many times** with probability 1.

[1] This holds only in DTMC models with finitely many states.

Frequency of visits

How often is a state visited among the states visited infinitely many times?



Theorem

$$lim_{n\to\infty} E\left(\frac{\#\text{ visits of state } i \text{ during the first } n \text{ steps}}{n}\right) = \pi_i$$

where π is a so called **stationary** (or **steady-state** or **invariant** or **equilibrium**) **distribution** satisfying $\pi \times P = \pi$.

DTMC extensions - communication and nondeterminism

Last remark on some DTMC extensions.

Modules and synchronisation

- modules
- actions
- rewards

Decision extension

- Markov Decision Processes (MDP)
- **Pmin** and **Pmax** properties

IA169 System Verification and Assurance

CEGAR and Abstract Interpretation

Jiří Barnat

Program Analysis

Goals of Program Analysis

- Deduce program properties from the program source code and ...
- ... employ them for program optimisation.
- ... employ them for **program verification**.

Undecidability

- Validity of all interesting program properties written in general programming language is algorithmically undecidable.
- Henry Gordon Rice (1953) – Rice's Theorems.
- Alan Turing (1936) – Halting Problem.

Undecidable – So what now?

Abstraction

- To hide details in order to simplify the analysis.
- Aims at correct, even if incomplete solution.

Using Abstraction

- To build (typically finite state) model of system under verification. (Followed, e.g., by model checking approach to verification.)
- System execution within the context of abstraction – **Abstract Interpretation**.

Recall Other Approaches

- Fixed input set of values (Testing).
- Limited exploration of the full state space (Bounded MC).
- Practical undecidability (Symbolic Execution).

Data and Predicate Abstraction

Data Abstraction

Motivation

- State space explosion due to large data domains.
- Reduction employing principle of domain testing, i.e. replacing original data domain with a data domain with less number of members.

Terminology

- Abstraction: mapping concrete states to abstract ones.
- Concretisation: mapping abstract states to set of concrete states.

Example of Data Abstraction

- `Int -> { Even, Odd }`
- Concrete state: $\langle$ `PC:12, A:15, B:0` $\rangle$
- Abstract state: $\langle$ `PC:12, A:Odd, B:Even` $\rangle$

Abstract Transition System

Transitions in Concrete and Abstract Semantics

- Statement in program code, line 12: `A := A+A`
- In concrete semantics:
  $\langle$`PC:12, A:15, B:0`$\rangle \longrightarrow \langle$`PC:13, A:30, B:0`$\rangle$
- in abstract semantics:
  $\langle$`PC:12, A:Odd, B:Even`$\rangle \longrightarrow \langle$`PC:13, A:Even, B:Even`$\rangle$

Non-Determinism in Abstract Transition System

- Abstract state: $\langle$`PC:13, A:Even, B:Even`$\rangle$
- Statement in program code, line 13: `A := A div 2`
- $\langle$`PC:13, A:Even, B:Even`$\rangle \longrightarrow$

  $\langle$`PC:14, A:Even, B:Even`$\rangle$

  $\langle$`PC:14, A:Odd, B:Even`$\rangle$

Relation of Abstract and Concrete Transition Systems

Over-Approximation

- Every run of concrete system is present in concretisation of some abstract run (run of abstracted transition system).
- There may exist runs that are present in concretisation of some abstract run, but are not allowed in concrete transition system.

Under-Approximation

- Every run present in concretisation of any abstract run is an existing run of concrete transition system.
- There may exist runs of concrete transition systems that are not present in concretisation of any abstract run.

Verification of Approximated Transition Systems

Notation

- ATS – Abstract Transition System
- CTS – Concrete Transition System

Verification of Over-Approximated Systems

- Absence of error in ATS proves absence of error in CTS.
- Error in ATS may, but need not indicate error in CTS.
- Error in ATS that is not an error in CTS, is referred to as false positive (spurious error, false alarm).

Verification of Under-Approximated Systems

- Error in ATS proves presence of error in CTS.
- Absence of error in ATS does not prove absence of error in CTS.
- Error in CTS that is not present in ATS is referred to as false negative.

Example – Concrete Semantics

Task

- Is error state reachable in the following program?
- % denotes modulo operation, `A` in an integral variable

Source-code	Value of `A` in concrete semantics after execution of program statement	
1 `read(A);`	`[int]`	
2 `A = A % 2;`	`[0]`	`[1]`
3 `A = A + 1;`	`[1]`	`[2]`
4 `if (A==0)`	`<false>`	`<false>`
5 **`error;`**		
6 `else`		
7 `return;`	`<ret>`	`<ret>`

Example – Data Abstraction

Task

- Is error state reachable in the following program?
- `A` is abstracted into parity domain `{even,odd}`.

Source-code		Value of `A` in abstract semantics after execution of program statement	
1	`read(A);`	`[even]`	`[odd]`
2	`A = A % 2;`	`[even]`	`[odd]`
3	`A = A + 1;`	`[odd]`	`[even]`
4	`if (A==0)`	`<false>`	`<true/false>`
5	**`error;`**		**`<error>`**
6	`else`		
7	`return;`	`<ret>`	`<ret>`

Predicate Abstraction

Predicate Abstraction

- Predicates – Boolean expressions about variable values.
- Example of definition of abstract transition system:
  $\langle$Program Counter, Validity of selected predicates$\rangle$

Amount of Abstraction

- Amount of predicates influences the precision of abstraction.
- Less predicates $\rightsquigarrow$ big ambiguity, smaller state space.
- More predicates $\rightsquigarrow$ increased precision, bigger state space.

Task

- For the given program code and set of predicates, draw the abstract transition system formed using predicate abstraction.
- Check if there is path in your ATS that is not a spurious run and leads to an error state.

```
1   read(A);
2   A = A % 2;
3   A = A + 1;
4   if (A==0)
5      error;
6   else
7      return;
```

a) $P1 \equiv A = 0$

b) $P1 \equiv A = 0$,
$P2 \equiv A \geq 0$

Remarks on Predicate Abstraction

Analysis of Abstract Runs Leading to Error

- Decision about validity of the run (Is it a false alarm?)
- Deduction of new predicates to make abstraction more precise.

Size of Abstract Transition System

- The size of the state space grows exponentially with the number of predicates.

Possible Solution

- Predicates are bound to particular program locations.

CEGAR Approach

Counter-Example Guided Abstraction Refinement

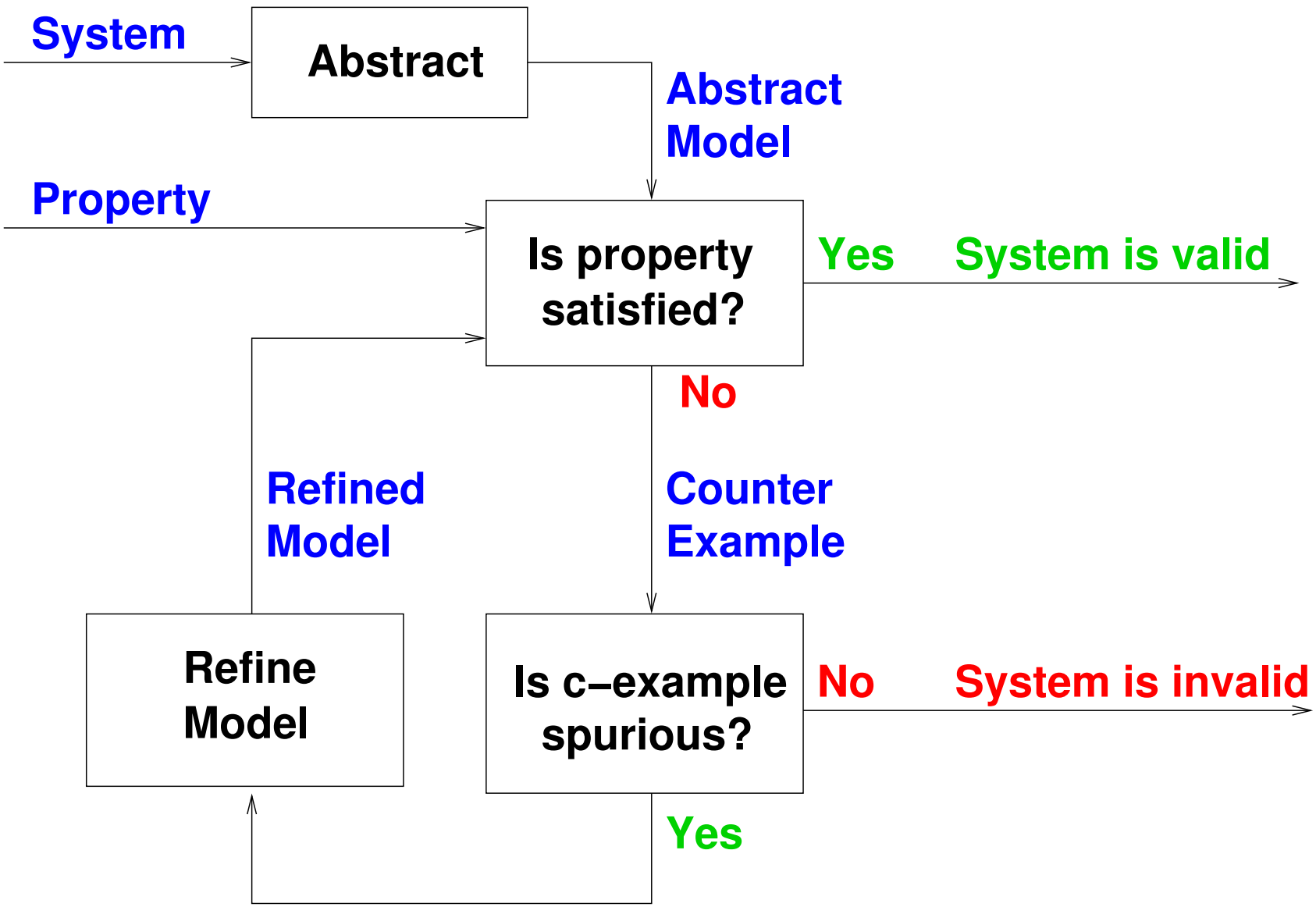
Principle of CEGAR Approach

- Given an initial set of predicates, system is abstracted with predicate abstraction.
- Abstract transition system (over-approximation) is verified with a model checking procedure.
- In the case a reported counterexample is find spurious, it is analysed in order to deduce new predicates and refine predicate abstraction.
- Procedure repeats until real counterexample is find or system is successfully verified.

Notes

- Deducing new predicates is very difficult.
- Often done within model checking procedure (on-the-fly).
- Berkeley Lazy Abstraction Software Verification Tool (BLAST).

Schema of CEGAR Approach



Basics of Abstract Interpretation and Static Analysis of Programs

Program Representation – Flow Graph

- "Special version" of Control-Flow Graph.
- Every edge is either guarded with a single guard or defines a single assignment.

Goal

- Compute properties of individual vertices of the flow-graph.

Goal Examples

- Deduce range of values a particular variable may take in a given program location.
- Compute a set of live variables in a given program location.
- ...

A General View on Abstract Interpretation

Property Decomposition

- The property to be verified by general program analysis procedure can be decomposed into local data values assigned to individual vertices of the (control-)flow graph.
- The result of verification is compound or deduced from the values local to the flow-graph vertices.

Value Improvement

- It is defined how an edge of the (control-)flow graph improves (locally updates) the decomposed value of the property to be verified.
- Application of local update functions gradually (monotonously) improve (approximate) the overall solution.

General Algorithm for Computing Abstract Interpretation

Initialisation

- Initially, a suitable value is assigned to every vertex.
- Every edge of the graph is marked as unprocessed.

Computation

- Pick an unprocessed edge of the graph and perform the local update relevant to the edge and associated vertices. If the update improved the solution, mark the edge as unprocessed again.
- Repeat until there are unprocessed edges, i.e. until overall fix-point is reached.

Example – Computing Live Variables

Setup

- Initial value associated with graph vertices is $\emptyset$.
- Every edge from vertex u to vertex v updates value associated with the vertex u as follows:

$$V(u) = V(u) \cup \Big(V(v) \setminus \mathit{assigned}(u, v) \cup \mathit{used}(u, v) \Big),$$

where $V(x)$ denotes value associated with vertex x, $\mathit{assigned}(u, v)$ and $\mathit{used}(u, v)$ denote variables redefined and used along the edge (u, v), respectively.

Observation

- In every moment of computation there is some (approximating) solution to the verified property.
- Reaching a fix-point indicate, no more information can be deduced by program analysis in the current setup.

Abstract Interpretation

Observation

- The procedure presented on previous slides is quite general. Choosing proper setup may result in verification (computation) of many interesting program properties.
- Often this is combined with some data abstraction for variables.
- May be performed on partially unwinded graphs.
- Generally referred to as to **abstract interpretation**.

Parameters

- What abstract domain is used.
- Direction of update function (forward, backward, both).
- What does update function do.
- How are the values merged over multiple incoming edges.
- Order of processing of unprocessed edges.
- Termination detection.

Relevant Questions of Abstract Interpretation

Is there a fix-point?

- Often the composition of domains of values associated to vertices forms a complete lattice.
- Knaster-Tarski theorem says that every monotonous function over such a domain has a fix point.

Does computation terminates?

- If there is no infinitely ascending sequence of possible solutions in the composition of local domains, then yes.
- Otherwise, need not terminate.

Widening

- Auxiliary transformation of intermediate results such that it preserves correctness, and at the same times prevents existence of long (possibly infinite) ascending sequence of values in the corresponding domain.

Example of widening

- Let be given boundaries of precision in which we want to know the range of possible values of some variable.
- Beyond the precision boundaries values will be extended to infinity, i.e. $+\infty$ or $-\infty$.
- Sequence

  $$[0,1] \subset [0,2] \subset [0,3] \subset [0,4] \subset [0,5] \subset [0,6] \ldots$$

  will for precision bound [0,3] turn into:

  $$[0,1] \subset [0,2] \subset [0,3] \subset [0,+\infty]$$

Abstract Interpretation and Narrowing

Narrowing

- Using widening may lead to very imprecise results.
- Widening can be used to accelerate analysis of cycles.
- After widening-based analysis of cycle, the values are made more precised with narrowing (similar but dual technique).

Example

- Interval $[0,+\infty]$ will after narrowing shrink to $[0,n]$.

Commentary

- Precise usage of widening and narrowing is beyond the scope of this lecture.

Other Notes Related to Abstract Interpretation

Other Corners of Program Analysis

- Inter-procedural analysis.
- Analysis of parallel programs.
- Generation of invariants.
- Pointer analysis and analysis of dynamic memory structures.
- ...

CPA checker

- The Configurable Software-Verification Platform
- `http://cpachecker.sosy-lab.org/`
- Very successful competitor in Software Verification Competition.

Homework

Wanna Chalenge?

- Get acquainted with CPAchecker (`https://github.com/dbeyer/cpachecker/blob/trunk/README.txt`)
- Comment counterexample report (`http://cpachecker.sosy-lab.org/counterexample-report/ErrorPath.0.html`)

Homework

- Install and try BLAST verification tool.

Information technology security evaluation – standards, assurance

IA169 – System verification and assurance

Jiří Barnat, Vojtěch Řehák, Vashek Matyáš

CROCS
Centre for Research on Cryptography and Security

Course coverage

- assurance, threat models, relevant security standards,
- testing, simulations, advance testing, symbolic execution,
- abstract interpretation, static analysis, theorem proving,
- automated formal verification & introduction to model-based verification
- concrete software verification tools for analysis of sequential and concurrent systems, real-time systems, and systems with probabilities.

Position within other courses

- PA193 – Secure coding principles and practices
  - Very useful
- PA018 – Advanced Topics in IT Security
  - No strict dependence
- IA159 – Formal Verification Methods
  - Useful follow-up

Course topics I

- Need for verification in the security area – Common Criteria, FIPS 140-2, threat models, assurance
- Introduction to formal verification and testing
- Deductive verification
- LTL (Linear Temporal Logic) Model Checking
- CTL (Computation Tree Logic) Model Checking
- Symbolic execution

Course topics II

- Bounded Model Checking
- CEGAR and Abstract Interpretation
- Verification of Real-Time and Hybrid systems
- Verification of systems with probabilities

Course structure, credits

- 2/2/2 credits, +2 for the final exam
  - Lecture: 2 hours weekly
  - Seminar: 2 hours weekly
  - Homework: 5+ hours weekly (on average)

Course marking

- Final exam: 70%
- Assignments: 30%

- Marking scheme:
  A for 90% or higher, then
  B for 80% or higher, then
  C for 70% or higher, then
  D for 60% or higher, then
  E for 50% or higher, then
  F(ail) for less than 50%.

Colloquy or credit – at least 44%.

Resources used – first lecture

- *Common Criteria for Information Technology Security Evaluation*, v 3.4, rev. 4, Sep 2012
  - http://www.commoncriteriaportal.org/
- *Separation Kernel Protection Profile Revisited: Choices and Rationale*, T.E. Levin et al., 4th Annual Layered Assurance Workshop, 2010
- *Common Criteria Certification in the UK – UK IT security evaluation & certification scheme*, CESG
- *Understanding the Windows EAL4 evaluation*, J.S. Shapiro, IEEE Computer 03/2003
- Security Requirements for Cryptographic Modules, FIPS PUB 140-2
  - http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

Security threat model

- Two views:
  1) Description of security threats considered when designing a (security) solution/system.
  2) Definition of (all) possible threats to consider.
- Usual security notion:
  - *Assets* to be protected
  - *Vulnerabilities* of assets and relevant systems
  - *Threats* exploiting the vulnerabilities
  - *Countermeasures* (aim to) mitigate the threats

Threat modelling – approaches

- Attacker centric
  - Popular in the research community (following two slides)
- System centric (a.k.a. design/SW centric)
  - Taking over in the past decade or so, e.g. used in the Microsoft Security Development Lifecycle
- Asset centric
  - Business logic

- Defender/owner view getting more prominent

Attacker models – Needham & Schroeder

- attacker can eavesdrop and interfere all communication
  - record/modify/replay/inject messages
- node internal processes are safe
  - secret keys, encryption process, …

- Comms security classics, paper from 1978 – paper Using encryption for authentication in large networks of computers, ACM Communications

Attacker models – Dolev & Yao

- Network = set of abstract machines exchanging messages.
- Message = formal terms. Terms reveal some of the message internal structure to the adversary, but not all.
- Adversary can overhear, intercept, and synthesize any message, is limited by the constraints of the cryptographic methods used.
  - Sometimes put as “the attacker carries the message.”
- Paper “On the security of public key protocols”, IEEE Trans. on Information Theory, 1983

Trusted system/product

- Such one that behaves in a way we expect it to behave
- Can be trusted to only such a functionality that adheres to the relevant security policy
- Trust
  - Belief that (a system…) satisfies given (security) requirements and specifications
  - Chance that (a system…) can breach the (security) policy without leaving any trace of evidence ☺

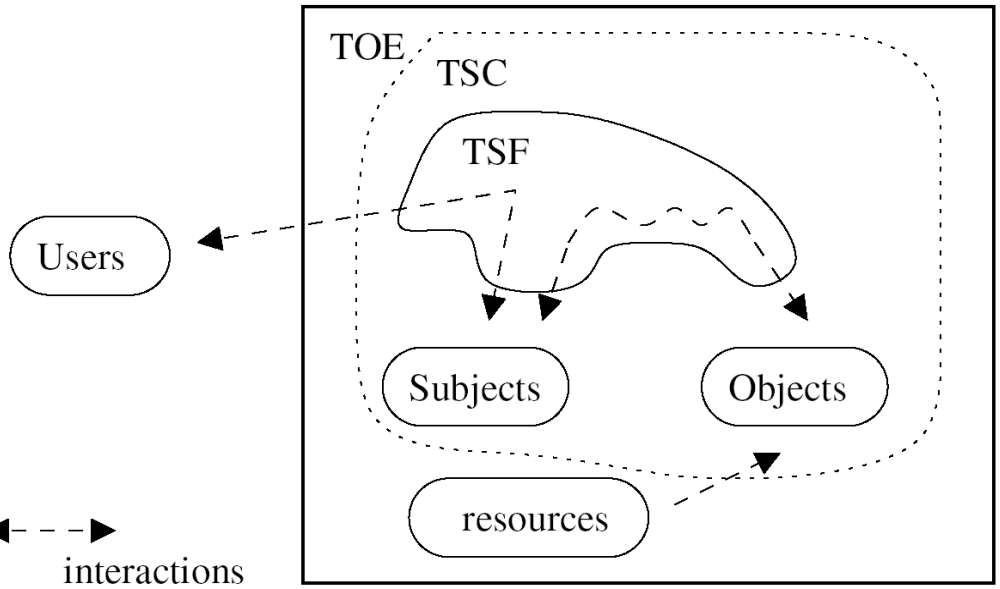
Common Criteria

- Interests of users, manufacturers, evaluators
- *Target of evaluation* (TOE) – what is (to be) evaluated
- *Protection profile (PP)* (smartcards, biometrics, etc.)
  – Catalogued as a self-standing evaluation document
- *Security target* (ST) – theoretical concept/aim
- *Security Functional Requirements* (SFRs) – individual security functions provided by the TOE

- Evaluation of TOE – is the reality corresponding to theory (ST)?

Common Criteria model

- TOE: Target of Evaluation – the evaluated system
- TSF: TOE Security Functions – HW, SW, FW used by the TOE
- TSC: TSF Scope of Control – interactions under the TOE security policy



Common Criteria – two catalogues

- Two catalogues of components for specification of assurance and functionality requirements, with a standard terminology.
- Functionality – rules governing access to & use of TOE resources, and thus information and services controlled by the TOE
- Assurance
  - grounds for confidence that an entity meets its security objectives (CC v2.3)
  - grounds for confidence that a TOE meets the SFRs (CC v3.1)

Assurance is not robustness

- Assurance
  - grounds for confidence that an entity meets its security objectives (CC v2.3)
  - grounds for confidence that a TOE meets the SFRs (CC v3.1)
- Robustness
  - characterization of the strength of a security function, mechanism, service or solution, and the assurance (or confidence) that it is implemented and functioning correctly (US DoD definition)

CC evaluation in a nutshell

1. Define the product/system for evaluation
2. Specify its functionality
3. Specify the assurance level claimed
4. See details of evaluation with a certification body
5. Prepare evidence

CC: Functional & Assurance Requirements

Security Functional Requirements (SFRs)

- The core – CC is in a major part a catalogue of security functions
- Same product with different ST => different SFRs
- Correctness of one function can depend on another function

Security Assurance Requirements (SARs)

- Measures taken to assure compliance with the claimed functionality
- Design, development, evaluation/verification
- CC provides a catalogue of SARs

CC functional classes

- FAU: SECURITY AUDIT
- FCO: COMMUNICATION
- FCS: CRYPTOGRAPHIC SUPPORT
- FDP: USER DATA PROTECTION
- FIA: IDENTIFICATION AND AUTHENTICATION
- FMT: SECURITY MANAGEMENT
- FPR: PRIVACY
- FPT: PROTECTION OF THE TSF
- FRU: RESOURCE UTILISATION
- FTA: TOE ACCESS
- FTP: TRUSTED PATH/CHANNELS

CC assurance classes

- APE: PROTECTION PROFILE EVALUATION
- ASE: SECURITY TARGET EVALUATION
- ADV: DEVELOPMENT
- AGD: GUIDANCE DOCUMENTS
- ALC: LIFE-CYCLE SUPPORT
- ATE: TESTS
- AVA: VULNERABILITY ASSESSMENT
- ACO: COMPOSITION

CC assurance paradigms

- *assurance based upon an evaluation* (active investigation)

- measuring the validity of the documentation and of the resulting IT product by expert evaluators with increasing emphasis on scope, depth, and rigour
- CC does not exclude, nor does it comment upon, the relative merits of other means of gaining assurance

Assurance through evaluation I

a) analysis and checking of process(es) and procedure(s);
b) checking that process(es) and procedure(s) are being applied;
c) analysis of the correspondence between TOE design representations;
d) analysis of the TOE design representation against the requirements;
e) verification of proofs;

Assurance through evaluation II

f) analysis of guidance documents;
g) analysis of functional tests developed and the results provided;
h) independent functional testing;
i) analysis for vulnerabilities (including flaw hypothesis);
j) penetration testing.

CC evaluation assurance scale

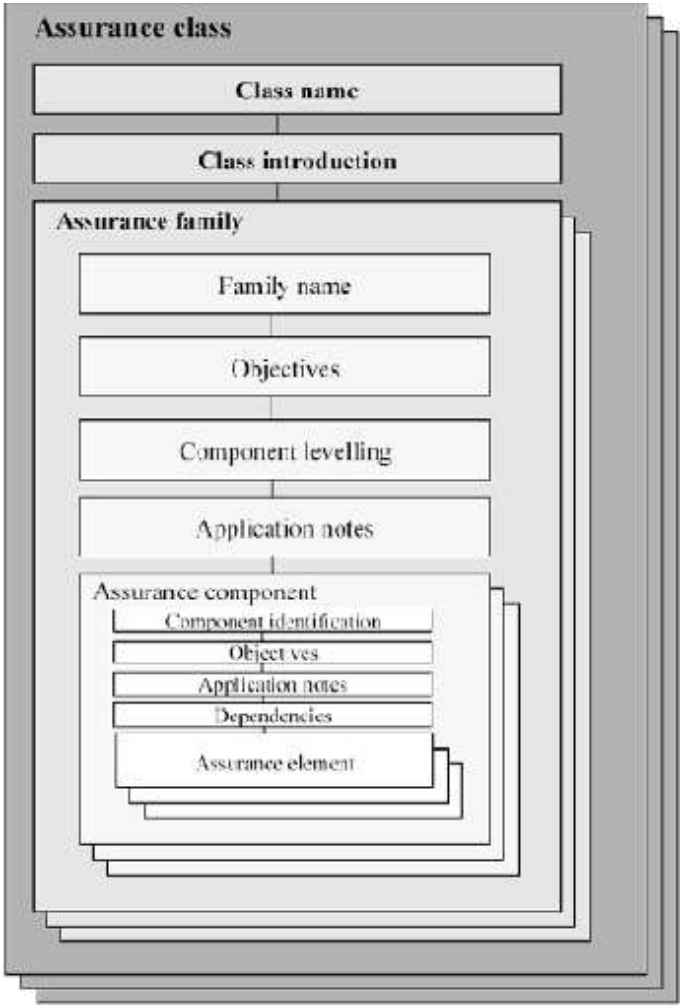
The increasing level of effort is based upon:

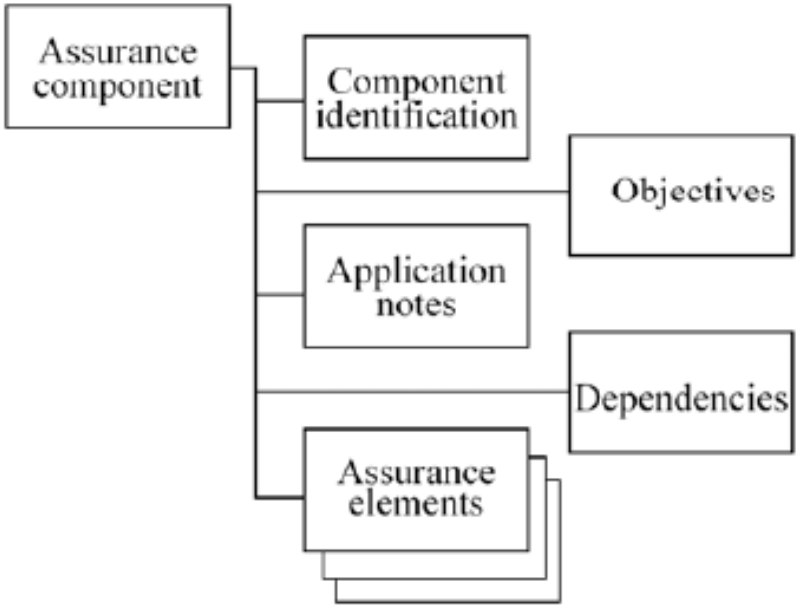
a) *scope* – the effort is greater because a larger portion of the IT product is included;

b) *depth* – the effort is greater because it is deployed to a finer level of design and implementation detail;

c) *rigour* – the effort is greater because it is applied in a more structured, formal manner.

CC – assurance hierarchy & component structure





Assurance elements – 3 exclusive classes

1. *Developer action elements*: activities that shall be performed by the developer. Further qualified by evidential material referenced in the following set of elements. Req's marked by "D" at the element No.
2. *Content and presentation of evidence* elements: the evidence required, what the evidence demonstrates, what the evidence shall convey. Marked by "C".
3. *Evaluator action elements*: activities that shall be performed by the evaluator. Marked by "E".

Assurance class	Assurance Family	Assurance Components by Evaluation Assurance Level						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		1	1	1	1	1	1
	ADV_FSP	1	2	3	4	5	5	6
	ADV_IMP				1	1	2	2
	ADV_INT					2	3	3
	ADV_SPM						1	1
	ADV_TDS		1	2	3	4	5	6
Guidance documents	AGD_OPE	1	1	1	1	1	1	1
	AGD_PRE	1	1	1	1	1	1	1
Life-cycle support	ALC_CMC	1	2	3	4	4	5	5
	ALC_CMS	1	2	3	4	5	5	5
	ALC_DEL		1	1	1	1	1	1
	ALC_DVS			1	1	1	2	2
	ALC_FLR							
	ALC_LCD			1	1	1	1	2
	ALC_TAT				1	2	3	3
Security Target evaluation	ASE_CCL	1	1	1	1	1	1	1
	ASE_ECD	1	1	1	1	1	1	1
	ASE_INT	1	1	1	1	1	1	1
	ASE_OBJ	1	2	2	2	2	2	2
	ASE_REQ	1	2	2	2	2	2	2
	ASE_SPD		1	1	1	1	1	1
	ASE_TSS	1	1	1	1	1	1	1
Tests	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
	ATE_IND	1	2	2	2	2	2	3
Vulnerability assessment	AVA_VAN	1	2	2	3	4	5	5

Certified Products by Assurance Level and Certification Date																	
EAL	**1999**	**2000**	**2001**	**2002**	**2003**	**2004**	**2005**	**2006**	**2007**	**2008**	**2009**	**2010**	**2011**	**2012**	**2013**	**2014**	**2015**	**Total**
EAL1	0	0	0	0	0	0	1	1	10	3	1	0	1	10	2	4	1	34
EAL1+	1	0	0	0	0	0	0	0	17	0	2	11	2	0	1	2	0	36
EAL2	0	0	1	0	0	0	1	2	29	5	10	2	11	3	10	13	0	87
EAL2+	0	0	0	1	1	1	2	2	27	14	18	15	21	38	35	33	0	208
EAL3	0	0	0	0	0	3	0	0	18	7	3	33	33	23	11	10	0	141
EAL3+	0	0	0	0	0	2	1	1	40	11	16	20	36	48	27	11	0	213
EAL4	0	1	0	1	0	0	1	0	35	6	9	5	6	3	7	2	0	76
EAL4+	0	1	1	2	2	3	5	3	149	63	74	65	72	107	71	46	3	667
EAL5	0	0	0	0	0	0	0	0	6	3	2	0	1	0	0	0	0	12
EAL5+	0	0	0	0	0	0	3	0	50	27	31	43	36	29	58	54	2	333
EAL6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
EAL6+	0	0	0	0	0	0	0	0	0	0	2	3	0	4	8	6	0	23
EAL7	0	0	0	0	0	0	0	0	0	0	1	0	0	0	4	0	0	5
EAL7+	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
Basic	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Medium	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
US Standard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
None	0	0	0	0	0	0	0	0	0	0	0	0	0	1	10	43	0	54
Totals:	**1**	**2**	**2**	**4**	**3**	**9**	**14**	**9**	**381**	**139**	**169**	**198**	**219**	**266**	**244**	**224**	**6**	**1890**

7 evaluation assurance levels (EALs)

- Hierarchical system – higher or new components
  - bold faced text in the description for the **added components**

- The following slides present first the EALs in the language of the CC and then from a practical perspective.

EAL1 – functionally tested

- some confidence in correct operation is required, but the threats to security are not viewed as serious
  - sufficient to simply state the SFRs that the TOE must meet, rather than deriving them from threats, etc. through security objectives;
  - analysis is supported by a search for potential vulnerabilities in the public domain and independent testing (functional and penetration) of the TSF;
  - This EAL provides a meaningful increase in assurance over unevaluated IT.

EAL2 – structurally tested

- assurance by a full security target (with given SFRs);
- analysis of the SFRs, using *functional* and *interface specs*, *guidance documentation* and *basic TOE architecture* description to understand the security behaviour;
- configuration management system and evidence of secure delivery procedures;
- independent confirmation of the developer test results, vulnerability analysis (based upon the *above in italics*) demonstrating resistance to penetration attackers with a basic attack potential.

EAL3 – methodically tested and checked

- architectural description of the TOE design;
- development environment controls
- improved testing coverage of the security functionality and mechanisms and/or procedures that provide some confidence that the TOE was not tampered with during development.

EAL4 – methodically designed, tested, and reviewed

- complete interface specification, description of the basic modular design of the TOE, implementation representation for the entire TSF;
- demonstrating resistance to penetration attackers with an Enhanced-Basic attack potential;
- additional TOE configuration mgmt incl. automation;

EAL5 – semiformally designed and tested

- modular TSF design;
- comprehensive TOE configuration management;
- semiformal design descriptions, a more structured (and hence analysable) architecture.

EAL6 – semiformally verified design and tested

- formal model of select TOE security policies;
- semiformal presentation of the functional specification and TOE design;
- modular layered and simple TSF design;
- structured development process, development environment controls, and comprehensive TOE configuration mgmt incl. complete automation;
- more comprehensive analysis, more architectural structure (e.g. layering), more comprehensive independent vulnerability analysis.

EAL7 – formally verified design and tested

- structured presentation of the implementation;
- implementation representation, complete independent confirmation of the developer test results;
- comprehensive analysis using formal representations and formal correspondence, and comprehensive testing.

CC certified products by country & EAL

Certified Products by Scheme and Assurance Level

Scheme	EAL1	EAL1+	EAL2	EAL2+	EAL3	EAL3+	EAL4	EAL4+	EAL5	EAL5+	EAL6	EAL6+	EAL7	EAL7+	B	M	S	N	Total
Australia	2	1	13	8	4	5	8	14	0	0	0	0	1	0	0	0	0	4	60
Canada	6	0	21	103	12	28	6	46	0	0	0	0	0	0	0	0	0	4	226
Germany	9	4	7	22	14	53	15	273	8	140	0	9	0	0	0	0	0	2	556
Spain	7	8	5	4	3	9	0	22	0	1	0	0	0	0	0	0	0	0	59
France	2	18	1	14	0	28	4	231	2	169	0	8	4	0	0	0	0	0	481
India	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	3
Italy	1	5	0	0	2	0	1	2	0	0	0	0	0	0	0	0	0	0	11
Japan	0	0	6	6	90	61	0	0	0	0	0	0	0	0	0	0	0	0	163
Republic of Korea	0	0	1	1	9	15	24	14	0	9	0	0	0	0	0	0	0	0	73
Malaysia	6	0	6	0	0	2	1	2	0	0	0	0	0	0	0	0	0	0	17
Netherlands	0	0	0	1	1	1	1	14	0	7	0	6	0	1	0	0	0	0	32
Norway	0	0	1	11	0	7	12	8	2	3	0	0	0	0	0	0	0	0	44
New Zealand	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sweden	0	0	4	0	2	0	2	3	0	0	0	0	0	0	0	0	0	0	11
Turkey	0	0	3	1	1	0	0	12	0	2	0	0	0	0	0	0	0	0	19
United Kingdom	0	0	1	10	1	3	0	10	0	1	0	0	0	0	0	0	0	1	27
United States	1	0	17	27	1	1	2	15	0	1	0	0	0	0	0	0	0	43	108
Totals:	**34**	**36**	**87**	**208**	**141**	**213**	**76**	**667**	**12**	**333**	**0**	**23**	**5**	**1**	**0**	**0**	**0**	**54**	**1890**

EAL1 – functionally tested

- analysis supported by independent testing of a sample of the security functions;
- applicable where confidence in correct operation is required but the security threat assessment is low.

- This EAL is particularly suitable for legacy systems as it should be achievable without the assistance of the developer.

EAL2 – structurally tested

- analysis exercises a functional and interface specification and the high-level design of the subsystems of the TOE;
- independent testing of the security functions;
- evidence required of developer 'black box' testing and development search for obvious vulnerabilities.

- EAL2 is applicable where a low to moderate level of independently assured security is required.

EAL3 – methodically tested and checked

- analysis supported by 'grey box' testing, selective independent confirmation of the developer test results and evidence of a developer search for obvious vulnerabilities;
- development environment controls and TOE configuration management are also required.

- EAL3 for a moderate level of independently assured security, with a thorough investigation of the TOE and its development, without incurring substantial re-engineering costs.

EAL4 – methodically designed, tested, and reviewed

- analysis supported by the low-level design of TOE modules and a subset of the implementation;
- testing supported by an independent search for obvious vulnerabilities;
- development controls supported by a life-cycle model, identification of tools and automated configuration management.

- EAL4 for a moderate to high level security, where some additional security-specific engineering costs may be incurred.

EAL5 – semiformally designed and tested

- analysis includes all of the implementation;
- supplemented by a *formal model*, a *semiformal presentation of the functional specification* and high level design and a *semiformal demonstration of correspondence*;
- search for vulnerabilities must ensure resistance to penetration attackers with a moderate attack potential;
- covert channel analysis and modular design required.
- EAL5 for a high level of security in a planned development coupled with a rigorous development approach.

EAL6 – semiformally verified design and tested

- analysis supported by a *modular approach to design* and a structured presentation of the implementation;
- independent search for vulnerabilities must ensure resistance to penetration attackers with a high attack potential;
- a systematic search for covert channels;

- EAL6 where a specialised security TOE is required for high risk situations.

EAL7 – formally verified design and tested

- the formal model is supplemented by a *formal presentation of the functional specification and high level design, showing correspondence*;
- evidence of developer 'white box‘ testing and complete independent confirmation of developer test results.

- EAL7 where a specialised security TOE is required for extremely high risk situations.

CC certified products by country & EAL

Certified Products by Scheme and Assurance Level

Scheme	EAL1	EAL1+	EAL2	EAL2+	EAL3	EAL3+	EAL4	EAL4+	EAL5	EAL5+	EAL6	EAL6+	EAL7	EAL7+	B	M	S	N	Total
Australia	2	1	13	8	4	5	8	14	0	0	0	0	1	0	0	0	0	4	60
Canada	6	0	21	103	12	28	6	46	0	0	0	0	0	0	0	0	0	4	226
Germany	9	4	7	22	14	53	15	273	8	140	0	9	0	0	0	0	0	2	556
Spain	7	8	5	4	3	9	0	22	0	1	0	0	0	0	0	0	0	0	59
France	2	18	1	14	0	28	4	231	2	169	0	8	4	0	0	0	0	0	481
India	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	3
Italy	1	5	0	0	2	0	1	2	0	0	0	0	0	0	0	0	0	0	11
Japan	0	0	6	6	90	61	0	0	0	0	0	0	0	0	0	0	0	0	163
Republic of Korea	0	0	1	1	9	15	24	14	0	9	0	0	0	0	0	0	0	0	73
Malaysia	6	0	6	0	0	2	1	2	0	0	0	0	0	0	0	0	0	0	17
Netherlands	0	0	0	1	1	1	1	14	0	7	0	6	0	1	0	0	0	0	32
Norway	0	0	1	11	0	7	12	8	2	3	0	0	0	0	0	0	0	0	44
New Zealand	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sweden	0	0	4	0	2	0	2	3	0	0	0	0	0	0	0	0	0	0	11
Turkey	0	0	3	1	1	0	0	12	0	2	0	0	0	0	0	0	0	0	19
United Kingdom	0	0	1	10	1	3	0	10	0	1	0	0	0	0	0	0	0	1	27
United States	1	0	17	27	1	1	2	15	0	1	0	0	0	0	0	0	0	43	108
Totals:	**34**	**36**	**87**	**208**	**141**	**213**	**76**	**667**	**12**	**333**	**0**	**23**	**5**	**1**	**0**	**0**	**0**	**54**	**1890**

Famous issue – Windows 2000

- Windows 2000 operating system was certified (Common Criteria) at EAL-4 in 2002.
  - with SP3 and one patch;
  - EAL-4, augmented with ALC_FLR.3 (Systematic Flaw Remediation);
  - Microsoft invested millions of dollars and three years of effort to gain the certification. (S. Bekker, Redmond Magazine).
- Controlled Access Protection Profile (CAPP)

CAPP assumption A.PEER

"Any other systems with which the TOE communicates are assumed to be under the same management control and operate under the same security policy constraints.

The TOE is applicable to networked or distributed environments only if the entire network operates under the same constraints and resides within a single management domain.

There are no security requirements that address the need to trust external systems or the communications links to such systems."

Controlled Access Protection Profile

- Level of protection appropriate for an assumed non-hostile and well-managed user community
  - requiring protection against threats of inadvertent or casual attempts to breach the system security.
- The profile is not intended to be applicable to circumstances in which protection is required against determined attempts by hostile and well funded attackers to breach system security.
- CAPP does not fully address the threats posed by malicious system development or administrative personnel.

Windows 2000 EAL-4 certification

- EAL4 rating means that you did a lot of paperwork related to the software process, but says absolutely nothing about the quality of the software itself. (J.S. Shapiro)
- System disconnected from networks (at different security level), disabled media drives, etc.
- Don't hook this to the internet, don't run email, don't install software unless you can 100 percent trust the developer, and if anybody who works for you turns out to be out to get you, you are toast. (J.S. Shapiro)

And Now for Something Completely Different… about Assurance viewed by…

- Customer – what level of guarantee I get that security has been implemented in the product?
- Developer – what (inputs and cooperation) will my team have to provide for the evaluation?
- Evaluator – did I get all required inputs and did all tests run OK to confirm the claim?
- Operator – what assumptions can I build on when preparing for my actions?

Security Requirements for Cryptographic Modules

- Federal Information Processing Standard (FIPS) Publication 140-2 (FIPS PUB 140-2)
- published May 2001 and last updated Dec 2002
  - FIPS 140-3 (Draft) – proposed revision, hanging in the air since 2009 (!)
- 4 levels, hierarchical levelling
- 11 functions (requirements):
  - Cryptographic module specification; Cryptographic module ports and interfaces; Role, services, and authentication; Physical security; Operational environment; Cryptographic key management; Mitigation of other attacks; …

FIPS 140-2 Annexes (drafts)

- Annex A: Approved Security Functions (Draft 2011)
- Annex B: Approved Protection Profiles (Draft 2007)
- Annex C: Approved Random Number Generators (Draft 2010)
- Annex D: Approved Key Establishment Techniques (Draft 2011)

FIPS 140-2 levels I

Level 1

- basic security requirements (e.g., certified algorithm);
- no specific physical security mechanisms.

Level 2

- features that show evidence of tampering – physical access to the plaintext cryptographic keys and critical security parameters (CSPs) within the module
  - including tamper-evident coatings or seals that must be broken to attain,
- or pick-resistant locks on covers or doors to protect against unauthorized physical access.

FIPS 140-2 levels II

Level 3

- high probability of detecting and responding to attempts at physical access, use or modification of the cryptographic module;
- may include the use of strong enclosures and tamper detection/response circuitry that zeroes all plain text CSPs when the removable covers/doors of the cryptographic module are opened.

FIPS 140-2 levels III

Level 4

- physical security mechanisms provide a complete envelope of protection around the cryptographic module with the intent of detecting and responding to all unauthorized attempts at physical access;
- protects a cryptographic module against a security compromise due to environmental conditions or fluctuations outside of the module's normal operating ranges for voltage and temperature;
- for operation in physically unprotected environments.

FIPS 140-2

- Level 2 – operating system at EAL2+
- Level 3 – operating system at EAL3+
  - and additional req.: Security Policy Model (ADV_SPM.1)
- Level 4 – operating system at EAL4+

Nice standards and theory, but…

- OpenSSL derivative FIPS-certified, found flawed
  - that particular one de-certified, others including the flaw not
- Dual EC DRBG defective by design mandated for FIPS 140-2
- IBM 4758 (with CCA API) – level 4
  - easy/fast logical attacks on CCA API
- Safenet Luna CA^3
  - disassembling showed no potting material
  - undocumented API functions
  - functionality in breach of security policy

Study of a particular PP

- PP BSI-PP-0025 – German (BSI) Common Criteria Protection Profile for USB Storage Media
  - https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/ReportePP/pp0025b_engl_pdf.pdf?__blob=publicationFile
- PP organisation:
  - the TOE description,
  - the TOE security environment,
  - the security objectives,
  - the IT security requirements and
  - the rationale.

PP BSI-PP-0025 – roles in the TOE

- Authorised user (S1)
  - Holds the authentication attribute required to access the TOE's protected memory area, in which the confidential data is stored.
  - Can modify the authentication attribute.

PP BSI-PP-0025 – roles in the TOE, cont'd

- Non-authorised user (S2)
  - Wishes to access S1's confidential data in the USB storage medium's memory (examples of confidential data are given in Section 2.5).
  - Does not have the authentication attribute to access the protected data.
  - Can obtain a USB storage medium of the same type. Can try out both logical and physical attacks on this USB storage medium.
  - Can gain possession of the TOE relatively easily since the TOE has a compact form.

PP BSI-PP-0025 – threats (countered)

- T.logZugriff – Assuming that S2 gains possession of the TOE, he/she accesses the confidential data on the TOE. S2 gains logical access by, for example, connecting the TOE to the USB interface of a computer system.
- T.phyZugriff – Assuming that S2 gains possession of the TOE, he/she accesses the TOE's memory by means of a physical attack. Such an attack could take the following form, for example: S2 removes the TOE's memory and places it into another USB storage medium which he/she uses for the purpose of logical access to the memory.

PP BSI-PP-0025 – threats, cont'd

- T.AuthÄndern – Assuming that S2 gains possession of the TOE, he/she sets a new authentication attribute, with the result that the data becomes unusable for S1.
- T.Störung – A failure (e.g. power failure or operating system error) stops the TOE operating correctly. As a result, confidential data remains unencrypted or the TOE's file system is damaged.

Study of another PP development

- Security kernel – used to simulate a distributed environment, introduced by J Rushby (1981) as a solution to the difficulties and problems that had arisen in the development and verification of large, complex security kernels that were intended to “provide multilevel secure operation on general-purpose multi-user systems.”
- U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, v 1.03
- Study paper “Separation Kernel Protection Profile Revisited: Choices and Rationale”, T.E. Levin et al., URL: http://fm.csl.sri.com/LAW/2010/law2010-03-Levin-Nguyen-Irvine.pdf