Model based testing

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Typical development steps and V&V tasks

Requirement analysis

System specification
- Checking completeness, consistency, feasibility, verifiability
- Assuring traceability

Architecture design
- Trade-off analysis, interface analysis, fault effects analysis
- Model based quantitative evaluation

Module design
- Formal verification by (temporal logic based) model checking
- Equivalence checking

Module implementation
- Source code analysis
- Software model checking with abstraction
- Proof of program correctness by theorem proving
- Module testing (unit testing) ← model based techniques

System integration

System delivery

Operation, maintenance

Overview

- Introduction
  - The role of models in testing
  - Use cases for model based testing
- Test case generation for test coverage metrics
  - Using graph-based (direct) algorithms
  - Using model checkers
  - Using bounded model checkers
- Test case generation on the basis of mutations
  - Model mutations
- Conformance and refinement relations for testing
  - May and must preorder, IOCO

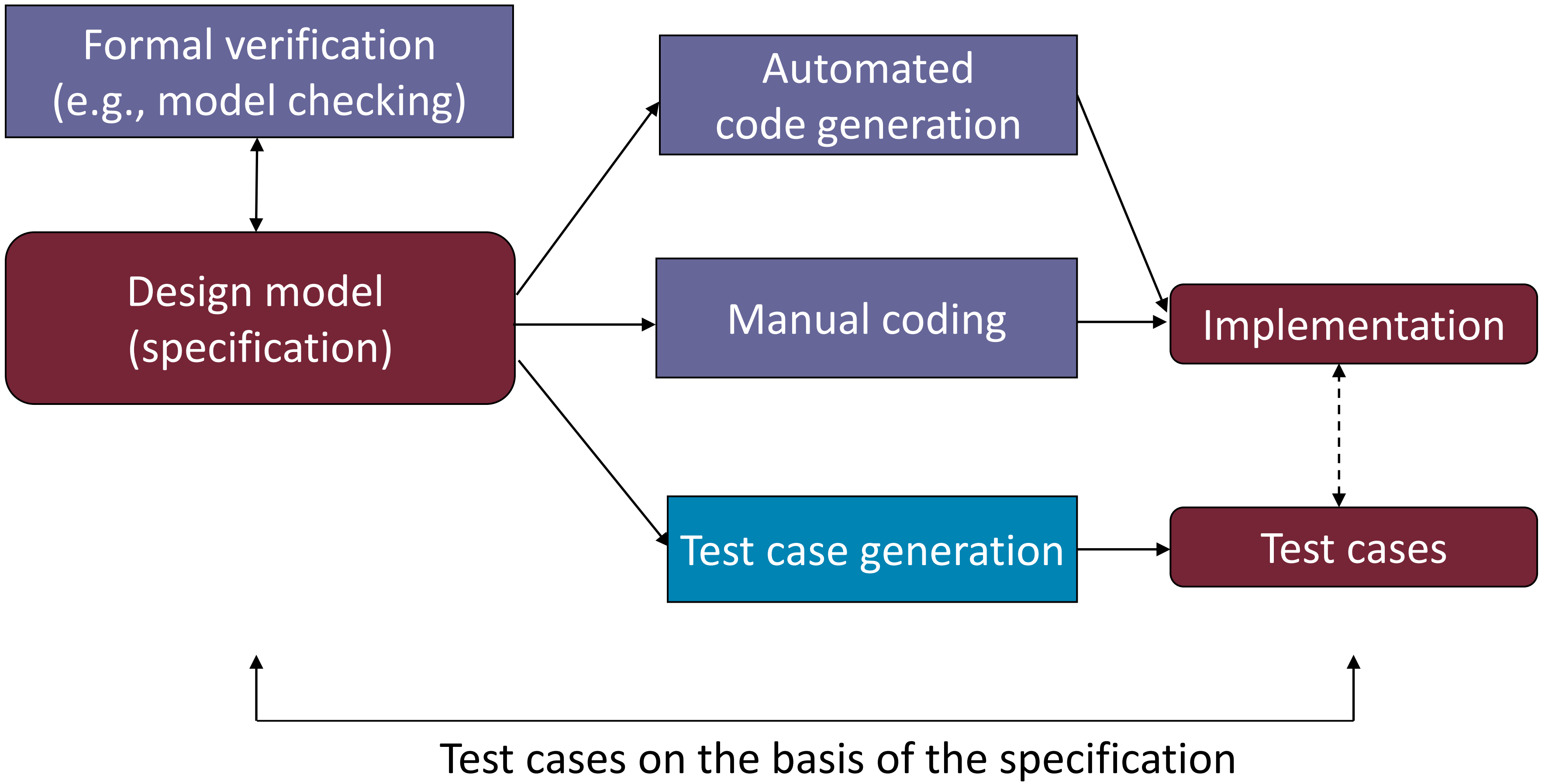
\+ Tools for model based test case generation

Introduction

Common practice: UML models in manual testing

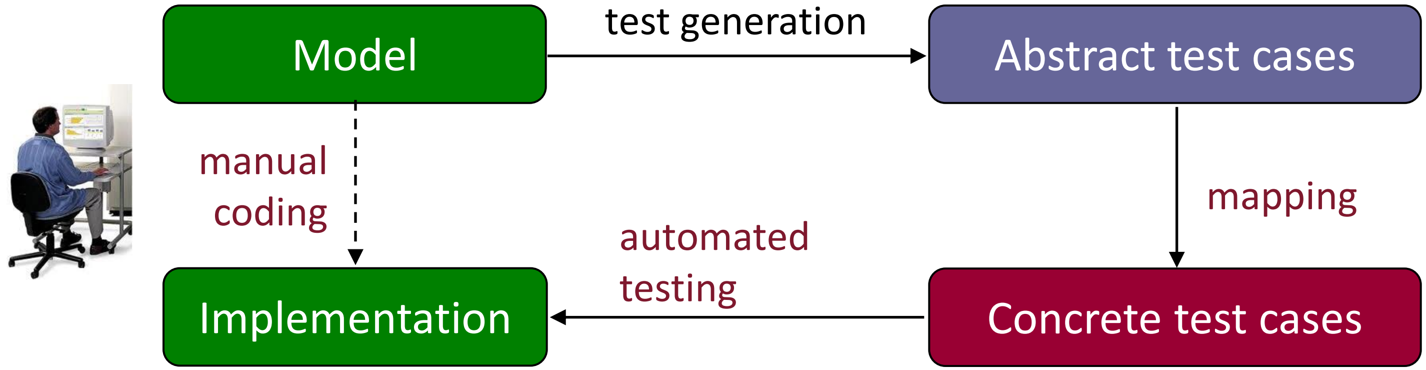
- Use case diagrams:
  - Validation (acceptance) testing: Covering use cases
- Class and object diagrams
  - Module testing: Identifying sw components, interfaces
- State machine and activity diagrams:
  - Module testing: Reference for structure based testing
- Sequence and collaboration diagrams:
  - Integration testing: Identifying scenarios
- Component diagram:
  - System testing: Identifying physical components
- Deployment diagram:
  - System testing: Designing test configuration

Model based test case generation: Typical approach

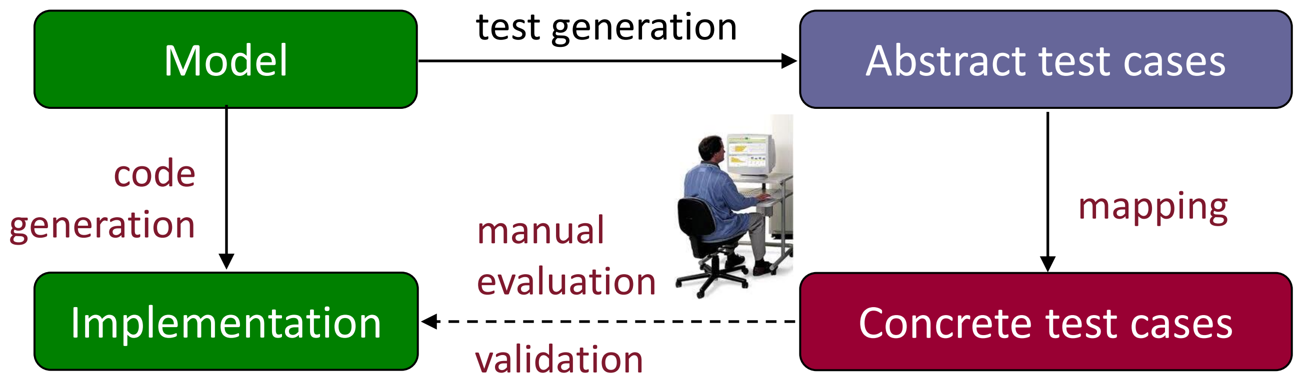


Use cases for model based testing

- In case of manual coding: Conformance checking

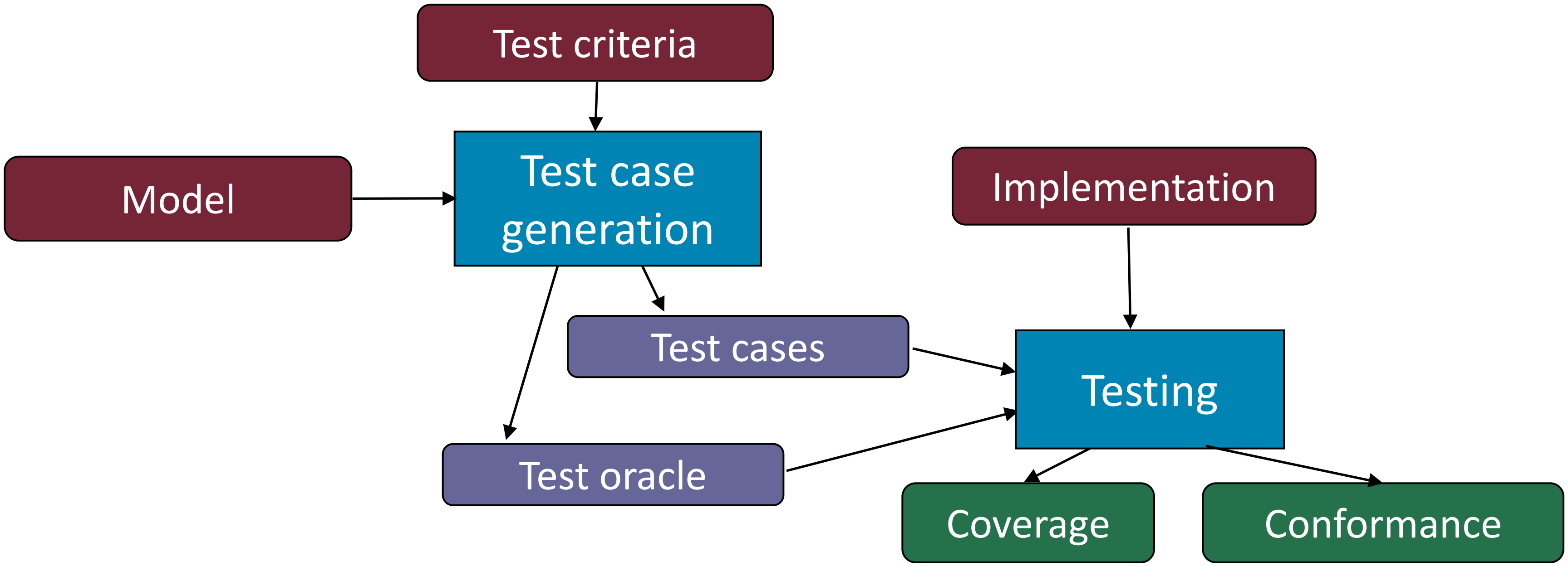


- In case of automated code generation: Validation

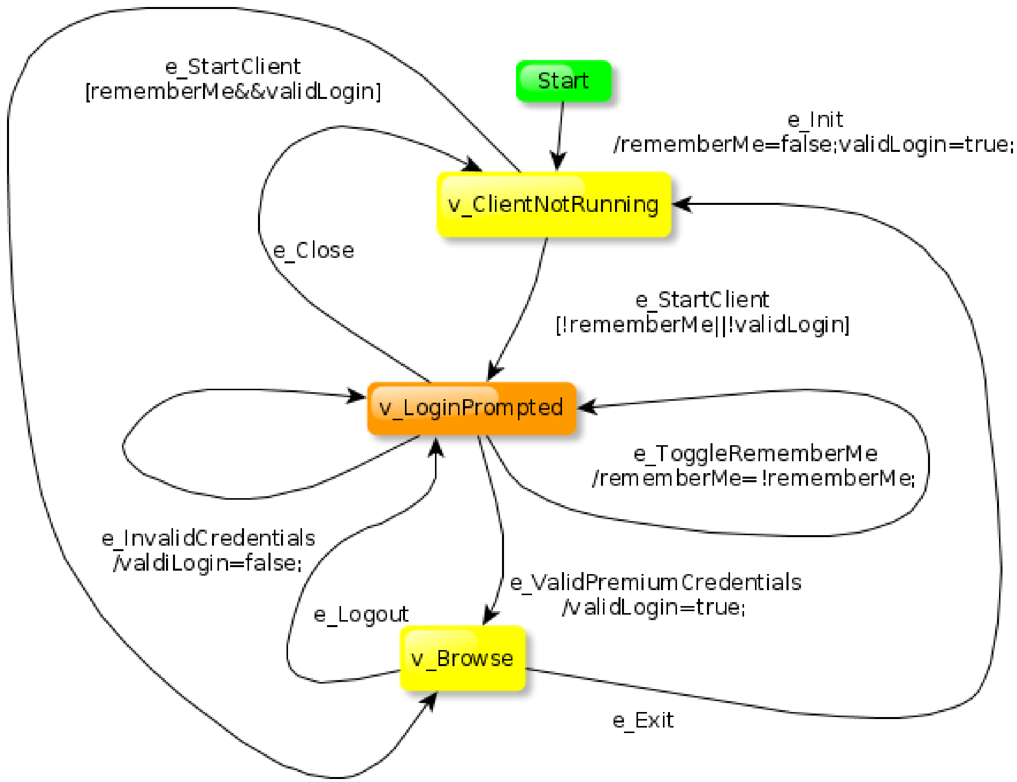


Basic tasks for model based testing (MBT)

- Based on the model and the test criteria:
  - Test case generation (for coverage or behavior conformance)
  - Test oracle generation (synthesis)
  - Test coverage analysis (for the model)
  - Conformance verdict (between model and implementation)



Example open source tool: GraphWalker

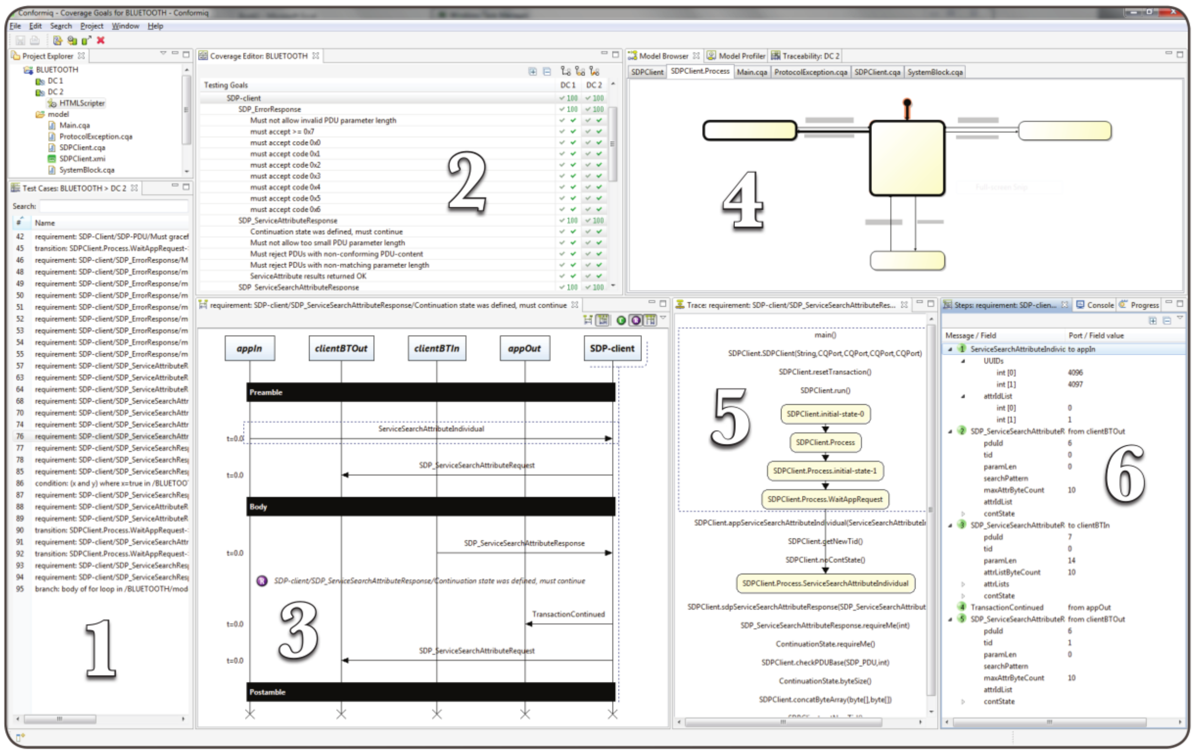


Source: [GraphWalker]

- Input: Finite state machine modell + simple guards
- Output: Tests for state and transition coverage

  + Generating JUnit test stubs (adapter)
- Traversing the graph: random walk, graph based search, shortest path

Example industrial MBT tool: Conformiq



Conformiq Designer IDE for automatic test case generation

Source: Conformiq. „Testing Bluetooth Protocol Stacks with Computer-Generated Tests". Technology brief. 2010

- Input: State machine models + Java action code
- Output: Tests for state, transition, requirement coverage
- Integration with other tools for testing

Overview of algorithms for model based test generation

- Graph-based algorithms
  - Model represented as a graph + traversal/search in this graph
- Application of model checkers
  - Counterexample is a test sequence for specified coverage
  - Symbolic or bounded model checkers
- Mutation based test generation algorithms
  - Test goal: Detect model mutations $\rightarrow$ detect code bugs
- Planner based methods
  - The planner constructs an operation sequence for a test goal
- Evolutionary algorithms (e.g., genetic algorithms)
  - Modifying an initial test suite generated by random walk
  - Optimization: increase coverage, reduce test length, …
- Symbolic execution
  - Control flow automata model

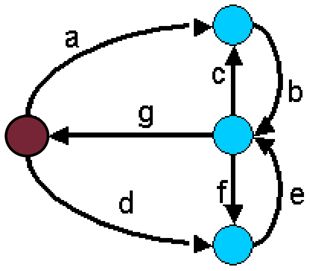
Graph-based algorithms for test generation

Typical applications of graph-based algorithms

- Model: Represents state based, event driven behavior
  - Transitions triggered by input events
  - Actions are given as outputs
- Basic formalisms:
  - Finite state automata (FSM; Mealy, Moore, …)
  - Higher level formalisms mapped to automata (UML statecharts, SCADE Safe Statechart, Simulink Stateflow, …)
- Typical applications
  - User interfaces, web based applications
  - Embedded controllers
  - Communication protocols
- Graph based algorithms
  - Different algorithms for various testing tasks and test criteria
  - Generating optimal test suite: Typically NP-complete

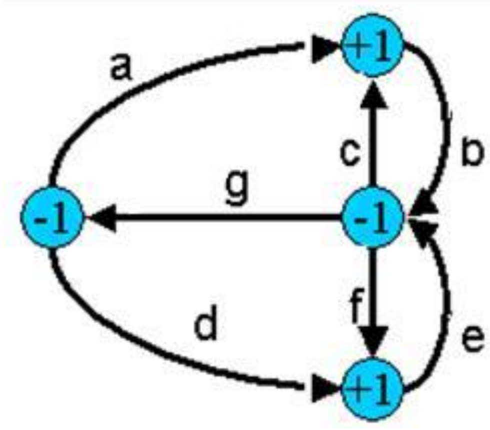
Graph-based algorithm for transition coverage

- Mapping the problem
  - Testing problem: Coverage of transitions
    - All transitions shall be covered by a test sequence
    - The test sequence shall go back to the initial state
  - Graph-based problem: ”New York street sweeper” problem
    - In a directed graph, find the (shortest) path that covers all transitions and goes back to the initial state
    - (The same problem in undirected graphs: ”Chinese postman” problem)

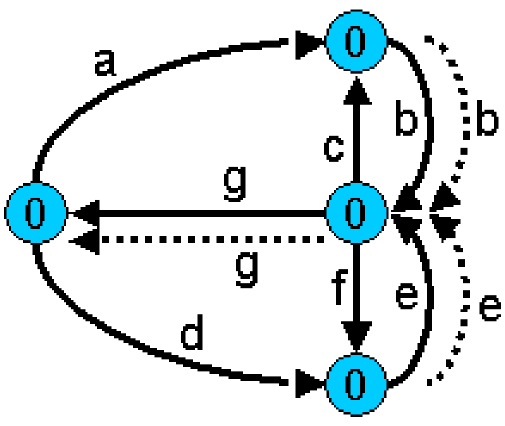


- Basic idea for the algorithm: Euler-graph → Euler-circuit
  - Computing the polarity of vertices: nr. of incoming minus outgoing edges
  - Duplicating edges that lead from a vertex with positive polarity to vertex with negative polarity, until all edges have zero polarity
  - Finding an Euler-circuit in the resulting graph (linear algorithm)
    - Euler-circuit: All edges are covered, it can always be constructed in such graph
  - The traversal of the Euler-circuit defines the test sequence

Example: Transition coverage



Original graph with
polarities of vertices



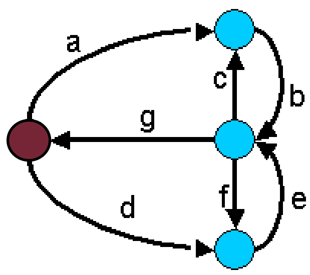
Graph with duplicated edges
(this way having an Euler-graph)

Sequence for traversal (Euler-circuit):

a b c b f e g d e g

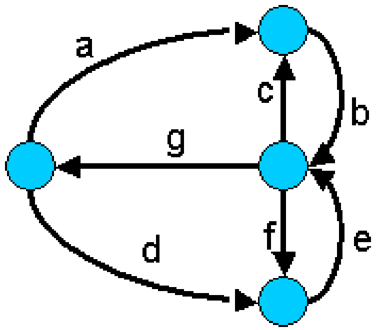
Graph-based algorithm for covering transition pairs

- Mapping the problem
  - Testing problem: Coverage of transition sequences
    - All possible sequences of n subsequent transitions shall be covered by a test sequence
    - The test sequence shall go back to the initial state
    - Simplest case: Covering all transition pairs
  - Graph-based problem: “Safecracker” sequence
    - Find the (shortest) edge sequence that includes all possible sequences of n subsequent edges (simplest case: n=2)

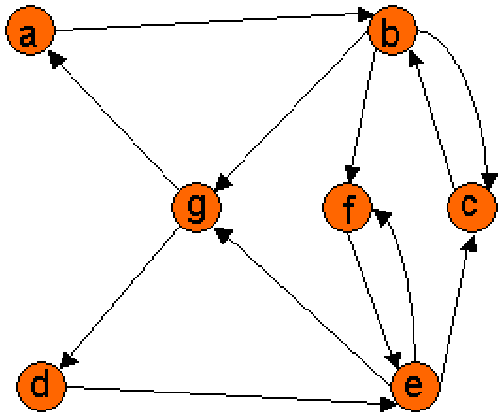


- Basic idea of the algorithm for n=2 (de Bruijn algorithm):
  - Constructing a dual graph
    - Edges of the original graph are mapped to vertices
    - If there is a pair of subsequent edges in the original graph then an edge is drawn in the dual graph between the vertices that represent these edges
  - Forming an Euler-graph (by duplicating edges) from the dual graph
  - Finding an Euler-circuit that defines the test sequence

Example: Covering transition pairs



Original graph



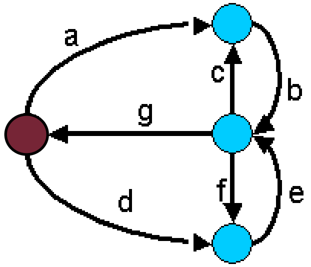
Dual graph with edges representing edge pairs in the original graph

Sequence for traversal that cover all transition pairs:

a b c b f e c b g d e f e g

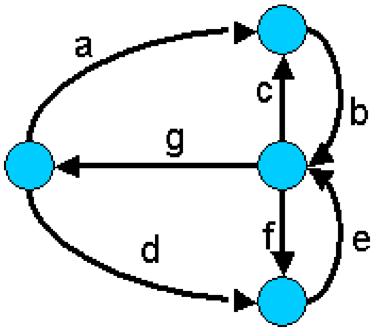
Graph-based algorithm for concurrent testing

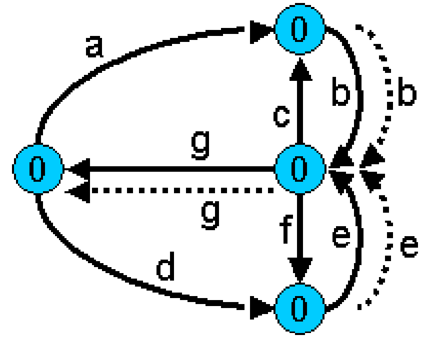
- Mapping the problem
  - Testing problem: Covering all transitions by concurrent testers
    - Goal is complete transition coverage
    - There are several testers that share (preferably equally) the testing task to finish it in the shortest time
    - All testers start in the initial state
    - Condition: The tested system shall be resetable to the initial state
  - Graph-based problems: ”Street sweepers brigade” problem



- Solution with heuristics (not an optimal solution)
  - Giving an upper limit k of the length of the test sequence for each tester
  - Generating an edge sequence in the Euler-graph that contains the highest number of edges that were not covered yet, and consists of at most k edges
  - Generating additional test sequences until uncovered edge exists
  - Trying to lower the limit k until the number of testers can be increased

Example for concurrent transition coverage





Original test sequence (Euler-circuit, for 1 tester):

a b c b f e g d e g

A potential set of concurrent test sequences (k=7):

- Tester 1: a b c b f e g
- Tester 2: d e g

- A better set of concurrent test sequences (k=5):
  - Tester 1: a b c b g
  - Tester 2: d e f e g

Test generation by model checking

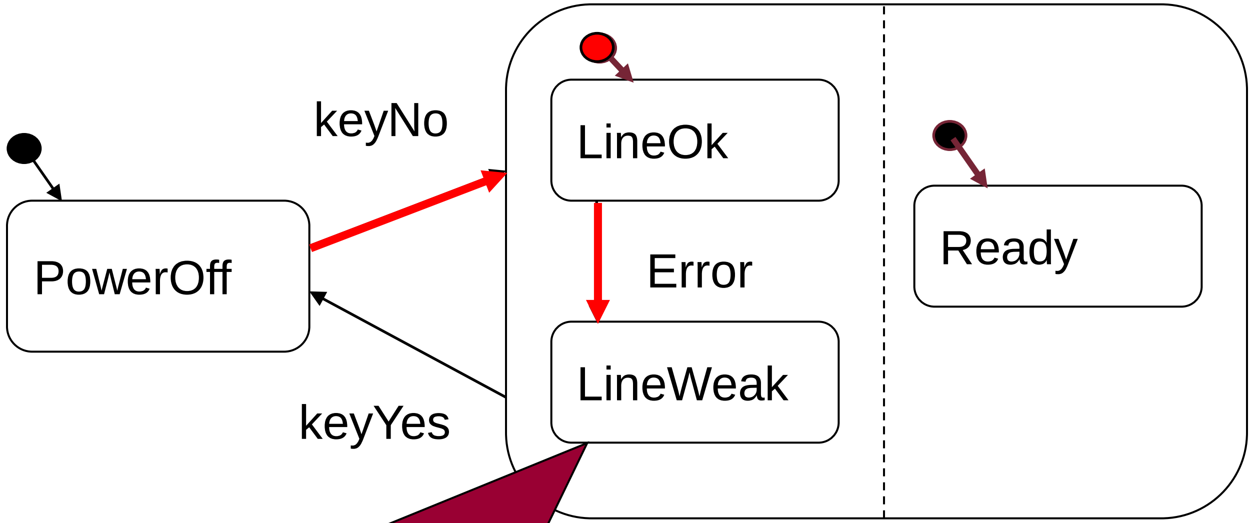
Basic idea

- Typical test coverage criteria (for the model):
  - Control flow based:
    - State coverage, transition coverage
    - Incoming-outgoing transition pairs coverage (for all states)
  - Data flow based:
    - Variable definition and usage coverage (for all variables)
- Required for test generation:
  - Traversal of the state space ← Model checker can perform it
- Basic idea:
  - Let the model checker traverse the state space
  - Let control the model checker in such a way that the counter-examples generated by the model checker form test sequences
  - Proper requirements (temporal logic properties to be checked) are needed – depending on the coverage criteria

Basic idea: Using a model checker for test generation

1. Test sequence to be generated: Coverage of the state LineWeak

3. The counterexample generated by the model checker demonstrates that the given state can be reached

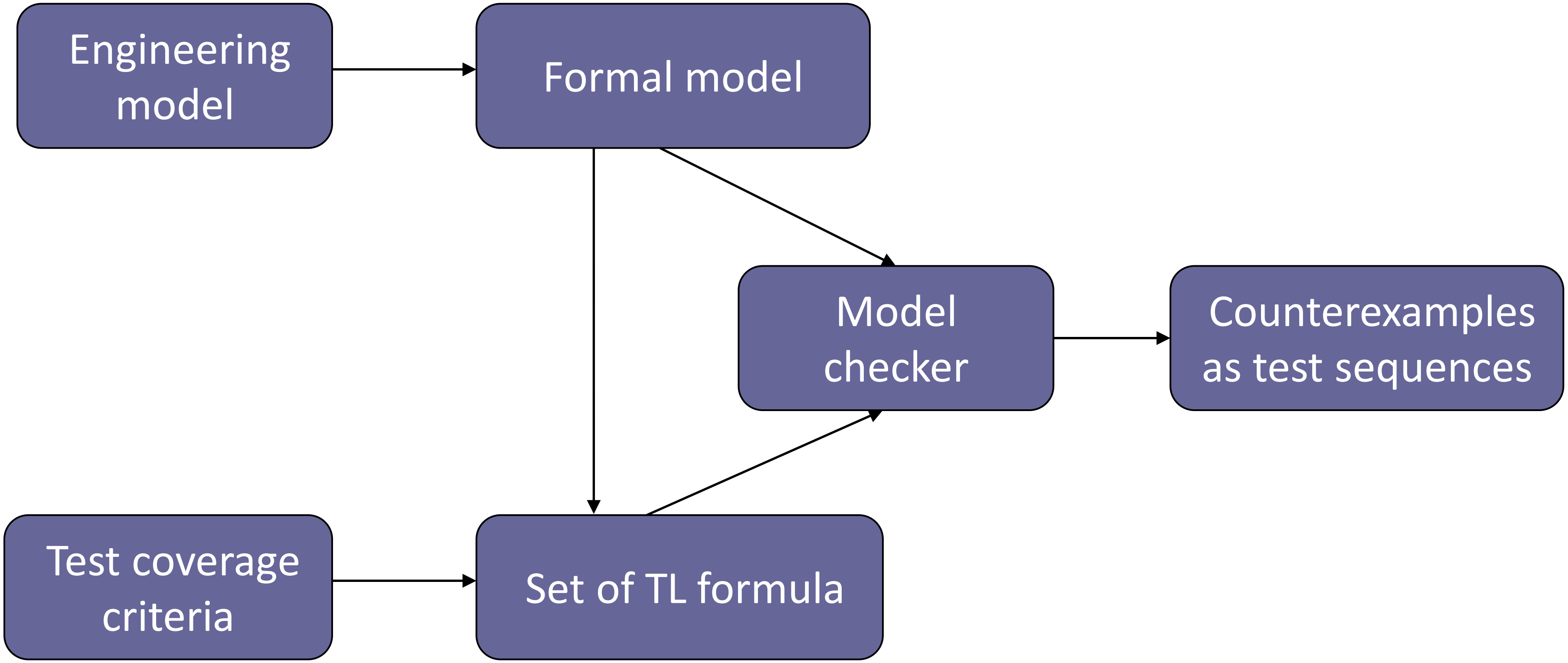


2.Specifying property for the model checker: The state LineWeak cannot be reached:

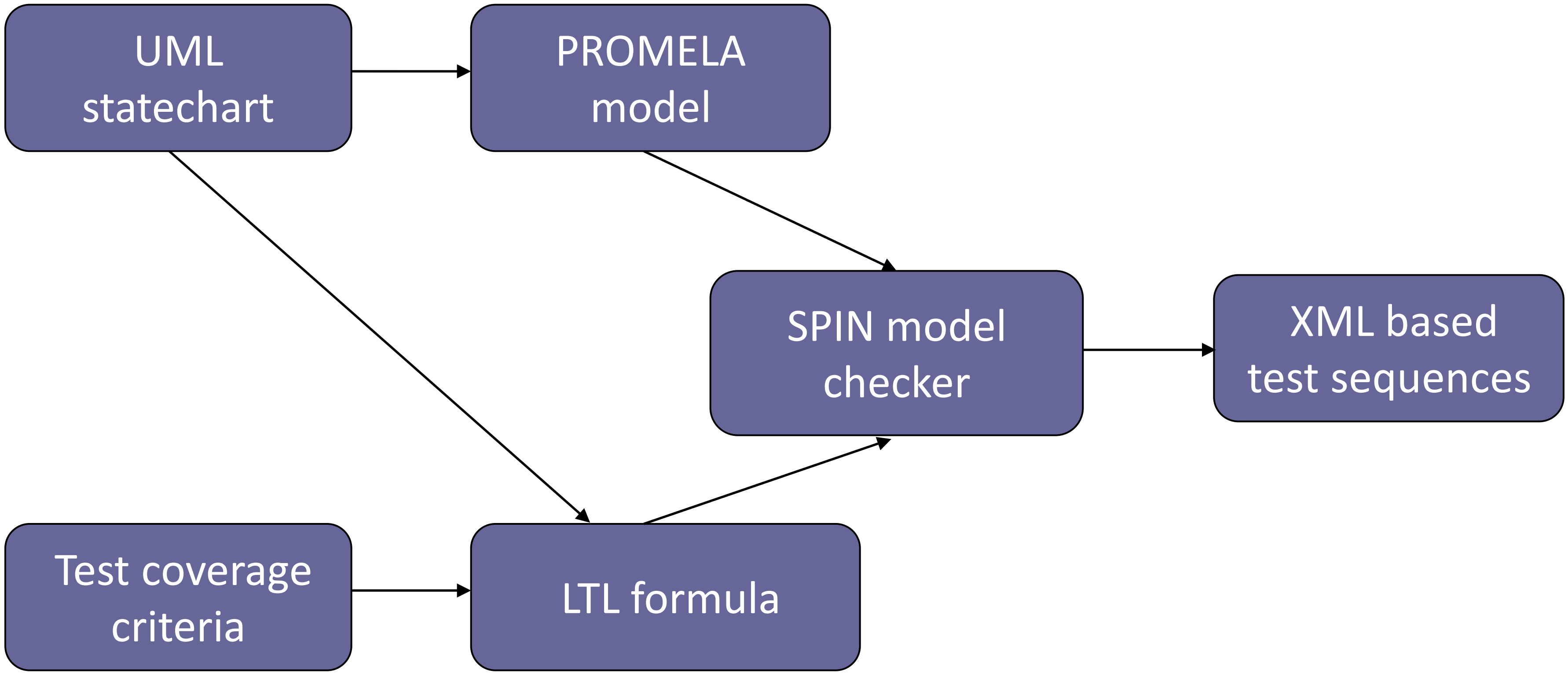
¬ EF LineWeak

4. The counterexample is a test sequence covering the state LineWeak

Framework for automated test generation



A possible implementation of the framework



Representing test coverage criteria by TL formula

- Labels in the model for variable v (predicates):
  - def(v)
  - c-use(v)
  - p-use(v)
  - implicit-use(v)

Using the variable in condition for an implicit transition
Implicit transition: The state does not change if the condition of the implicit transition holds

- Characteristic functions (with state variables):
  - s: being in state s
  - t: executing a given transition t (reaching the target state from the source state)
- State sets (→ represented by characteristic functions):
  - d(v): all def(v)
  - u(v): all c-use(v) or p-use(v)
  - im-u(v): all implicit-use(v)
  - start: state for starting new test (e.g., initial state)

Formula for control flow based coverage criteria

- State coverage:

  $\{\neg EF\ s \mid s \text{ basic state}\}$

  Set of formula is defined

  If a predefined start state shall also be reached for the subsequent test:

  $\{\neg EF\ (s \wedge EF\ start) \mid s \text{ basic state}\}$

  (EF start is omitted from the next formula)

- Weak transition coverage:

  $\{\neg EF\ t \mid t \text{ transition}\}$

- Strong transition coverage:

  $\{\neg EF\ t \mid t \text{ transition}\} \cup \{\neg EF\ it \mid it \text{ implicit transition}\}$

Strong coverage: Implicit transitions (not leaving the given state) are also tested

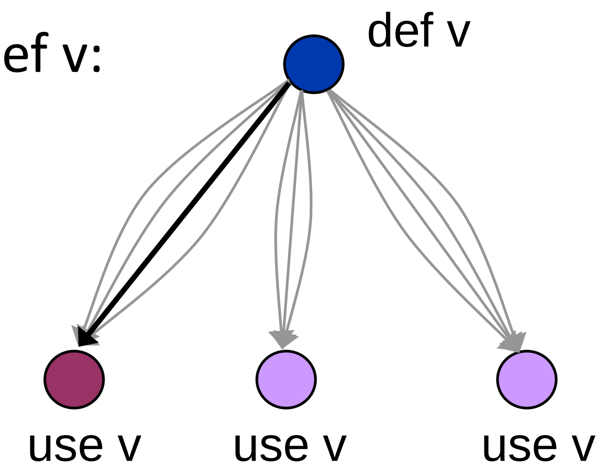
Recap: Data flow based test coverage criteria

- All-defs:

For all v, from all def v:

at least one def-clear path:

to at least one use v:

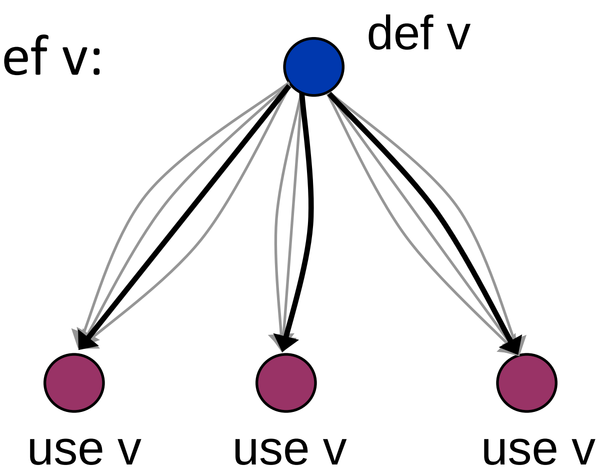


- All-uses:

For all v, from all def v:

at least one def-clear path:

to all use v:



Formula for data flow based test coverage criteria

- Weak all-defs coverage:

  > One def-clear path traversed from all def(v) to one use(v)

  $\{\neg EF\ (t \wedge EX\ E(\neg d(v)\ U\ u(v)))\ |\ v \text{ variable}, t \in d(v)\}$

- Weak all-uses coverage:

  > One def-clear path traversed from all def(v) to all use(v)

  $\{\neg EF\ (t \wedge EX\ E(\neg d(v)\ U\ t'))\ |\ v \text{ variable}, t \in d(v), t' \in u(v)\}$

- Strong all-defs coverage:

  > Implicit variable usage: in transitions not leaving the given state

  $\{\neg EF\ (t \wedge EX\ E(\neg d(v)\ U\ (u(v) \vee im\text{-}u(v))))$
  $|\ v \text{ variable}, t \in d(v)\}$

- Strong all-uses coverage:

  $\{\neg EF\ (t \wedge EX\ E(\neg d(v)\ U\ t'))$
  $|\ v \text{ variable}, t \in d(v), t' \in u(v) \cup im\text{-}u(v)\}$

Features of model checker based test generation

- Capabilities of model checkers:
  - Generating (typically) a single counterexample
  - Test sequences are hard to generate for coverage criteria that require all paths (this way all counterexamples)
    - E.g., all-du-paths criterion
      (all def-clear paths for a given def-use pair shall be tested)
- Abstract test sequences are generated
  - Defining the sequence of inputs
  - Expected outputs shall be determined (e.g., by simulation in the model)
  - Mapping is needed to concrete test sequences: concrete steps (calls) in a concrete test execution environment

Optimization of test sequences

- Task of model checking:
  - Efficient traversal of the state space: Fast, with low memory needs
- Required for test generation:
  Finding fast a counterexample that is as short as possible
  → Specific settings are needed in the model checker
  - Generating the shortest test sequences: NP-complete problem
- Possible settings (e.g., in case of model checker SPIN):
  - Breadth first search (BFS) in the state space
  - Depth first search, but with limited depth (limited DFS)
  - Finding shorter test sequences in an iterative way
  - Approximate model checking (hash function for storing checked states)
    - Some states (also covered by the hash function) will not be traversed
    - If a counterexample is found then it is a real test sequence for coverage

Example: Results for generating test sequences

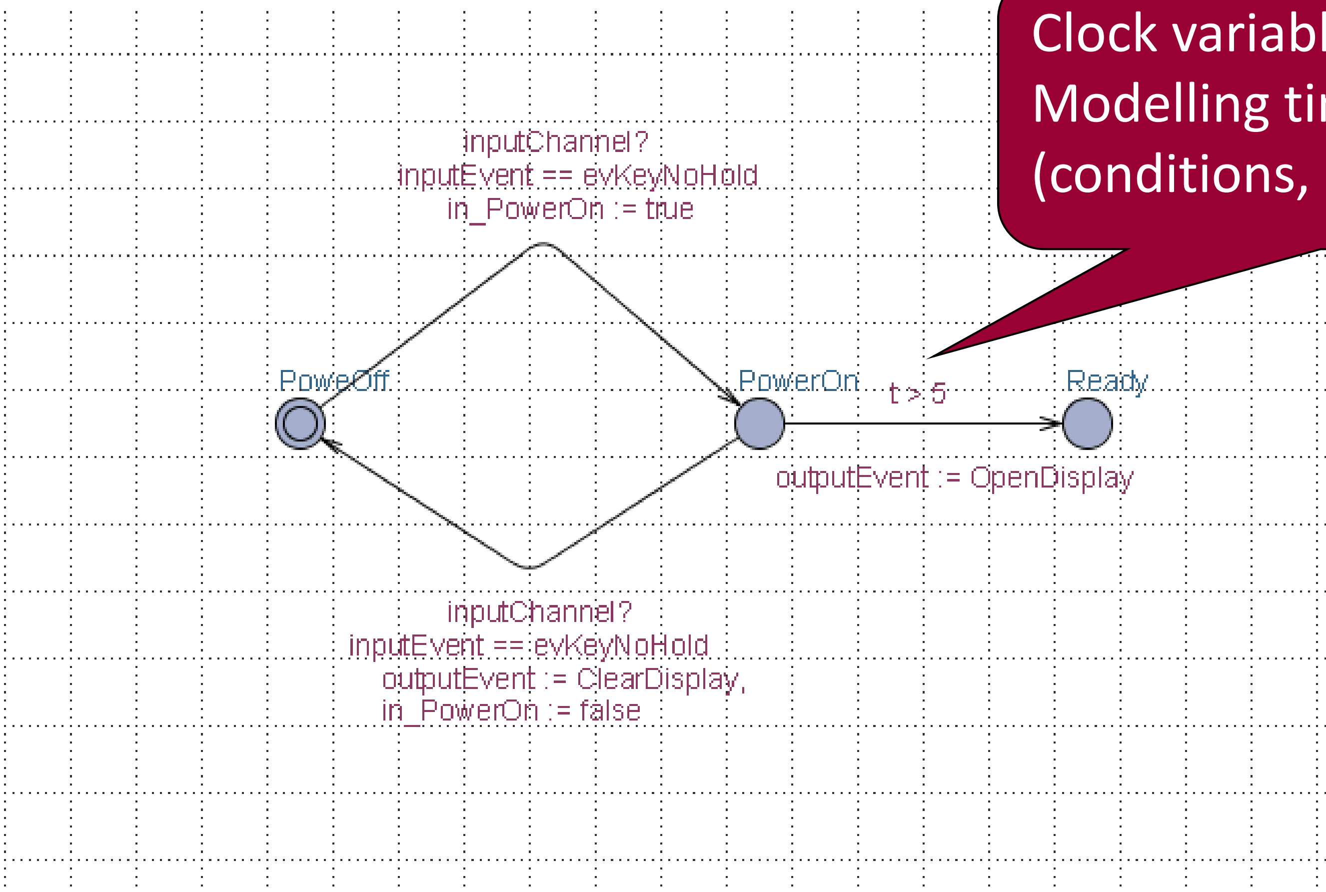
Options (compile time or run-time)	Time required for test generation	Length of all test sequences	Longest test sequence generated
-I	22m 32.46s	17	3
-dBFS	11m 48.83s	17	3
-i -m1000	4m 47.23s	17	3
-I	2m 48.78s	25	6
default	2m 04.86s	385	94
-I -m1000	1m 46.64s	22	4
-m1000	1m 25.48s	97	16
-m200 –w24	46.7s	17	3

Settings:

- -i iterative, -I approx. iterative
- -dBFS breadth first search
- -m limit for depth first search
- -w hash table size

State machine model of the behavior of a mobile phone (10 states, 11 transitions)

Extension of MBT to testing time-dependent behavior



Clock variables:
Modelling time dependency
(conditions, state invariants)

- Timed automata models
- Specific model checker: UPPAAL

Generated counterexamples with timing

State:

(input.sending mobile.PowerOn mobile1.LineOK mobile2.CallWait)

t=0 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=5

Delay: 6

Delays are included between the inputs of the test sequence

State:

(input.sending mobile.PowerOn mobile1.LineOK mobile2.CallWait)

t=6 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=5

Transitions:

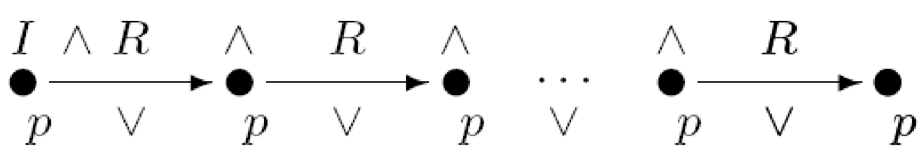
input.sending->input.sendInput { 1, inputChannel!, 1 }

mobile2.CallWait->mobile2.VoiceMail { inputEvent == evKeyYes && t > 5 && in_PowerOn, inputChannel?, 1 }

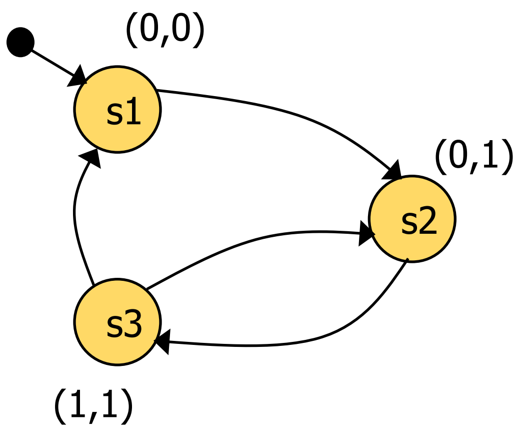
Test generation by bounded model checking

Recap: Bounded model checking

- Using SAT solvers for checking reachability of specific states
  - Given a Boolean formula (Boolean function), SAT solver generates a variable assignment (substitution) that makes the formula true
- Mapping the verification problem to Boolean function:
  - Characteristic function for initial states: I(s)
  - Characteristic function for specified “bad” states: p(s)
  - Characteristic function of the state transition relation: $C_R(s, s')$
  - “Stepping forward” along the state transitions: $C_R(s^i, s^{i+1})$
- The characterization of a counterexample (with conjunction):
  - Starting from the initial state: I(s)
  - „Stepping” along the transition relation: $C_R(s,s')$
  - Specifying that $p(s^i)$ holds somewhere along the path



Recap: Encoding a model

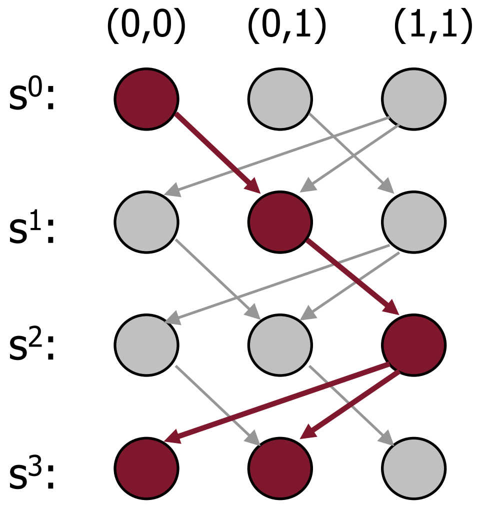


Initial state:

$I(x,y) = (\neg x \wedge \neg y)$

Transition relation:

$$C_R(x,y,x',y') = (\neg x \wedge \neg y \wedge \neg x' \wedge y') \vee \vee (\neg x \wedge y \wedge x' \wedge y') \vee \vee (x \wedge y \wedge \neg x' \wedge y') \vee \vee (x \wedge y \wedge \neg x' \wedge \neg y')$$



Paths with 3 steps from the initial state:

$$I(s^0) \wedge path(s^0,s^1,s^2,s^3) = $$
$$= I(x^0,y^0) \wedge C_R(x^0,y^0,x^1,y^1) \wedge C_R(x^1,y^1,x^2,y^2) \wedge C_R(x^2,y^2,x^3,y^3)$$

SAT based test generation for coverage criteria

- Constructing the Boolean function:
  - Encoding paths with k steps from the initial state
  - Specifying test criterion: In general, a TG formula
    - Reaching (covering) a state
    - Executing (covering) a transition
    - Traversing (covering) a part of the model, …

$$\underbrace{I(s^0) \wedge \bigwedge_{i=0}^{k-1} C_R(s^i, s^{i+1})}_{\text{Model paths of k length}} \wedge \underbrace{TG}_{\text{Test goal}}$$

- If this formula can be satisfied, then the substitution gives a test
  - This test is according to TG and limited to k steps
  - If there is no substitution then there is no test for TG in k steps

Features of BMC based test generation

- Limitations for test generation
  - Test of max. k steps can be generated
  - The length of paths can be increased iteratively
  - If a test sequence is found then it can be used
  - If there is no test found then a longer test sequence may exist
- Mapping the test generation problem to SAT problem can be made automatically
- The specification of test goals can be simplified
  - For C programs: FQL language for test goals (FSHELL tool)

    in /code.c/ cover @line(6),@call(f1) passing @file(code.c) \ @call(f2)
  - Specifying pre- and postconditions: Is there a test when the postcondition is not satisfied (although the precondition holds)?

Test generation based on mutations

Using fault sets for test generation

- Experience in software testing:
  - Coupling effect: Test cases that are efficient to find simple faults are also efficient for finding more complex faults
  - Competent programmer hypothesis: The programs are typically good, and the majority of faults are often occurring typical faults
- Basic idea:
  - Generating "mutant" models that contain typical simple faults, and generate tests for detecting these faults
  - There tests are expected to be more efficient in detecting more complex faults than random tests
- Typical "mutations":
  - Changing arithmetic operations in conditions
  - Changing the ordering of actions, messages
  - Omission of actions, messages, function calls
  - …

Equivalence relation for BMC based test generation

- Inputs and outputs are distinguished in the model
  - in(s) – inputs (events) in state s
  - out(s) – observable outputs (actions) in state s
  - δ action: lack of observable output
- Definition of the k-equivalence for the behaviour of two models:

  For the first k steps, providing identical input sequences, the outputs of the two models are the same

- Notation:

	Original model M:	Mutated model M':
Predicate for initial state:	$I(s^0)$	$I'(s'^0)$
State transition relation:	$C_R(s^i, s^{i+1})$	$C_R'(s'^i, s'^{i+1})$

Paths of length k from the initial state: $I(s^0) \wedge \bigwedge_{i=0}^{k-1} C_R(s^i, s^{i+1})$

Mutation based test generation using k-equivalence

- Construction of a SAT formula for detecting a mutation:
  - Providing the same input sequence for the two models
  - Traversing paths of length k in the original model
  - Traversing paths of length k in the mutant model
  - At least one output shall be different in the two models

$$\bigwedge_{i=0}^{k}(\text{in}(s^i)=\text{in}(s'^i)) \wedge I(s^0)\wedge\bigwedge_{i=0}^{k-1}C_R(s^i,s^{i+1}) \wedge I'(s'^0)\wedge\bigwedge_{i=0}^{k-1}C'_R(s'^i,s'^{i+1}) \wedge \bigvee_{i=0}^{k}(\text{out}(s^i)\neq\text{out}(s'^i))$$

The same inputs	Original model	Mutant model	At least one output is different

- If this formula can be satisfied then the substitution defines a test
  - The test detects the mutation: An output is different if the mutation is included in the model
  - If there is no substitution then there is no test for k steps

More general problem: Conformance in testing

- Test generation for mutations:
  - Construction of test input sequences that result in different behavior in the original (fault-free) and in the mutated model
  - Expected output sequence of the mutation test: Belongs to the mutation
  - These are so-called negative tests (failed test: no mutation)
- How to define the “difference” between two behaviors: What are the faults/mutations that are allowed?
  - Additional behavior besides the specified behavior?
  - Omission of some output?
- Typical solutions
  - Safety critical systems: Equivalent behavior, strictly according to the specification (complete specification and implementation are assumed)
  - “Common” systems: Conformant behavior, the specification provides the frame (limits) for acceptable behavior (incomplete specification and incomplete implementation are allowed)