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

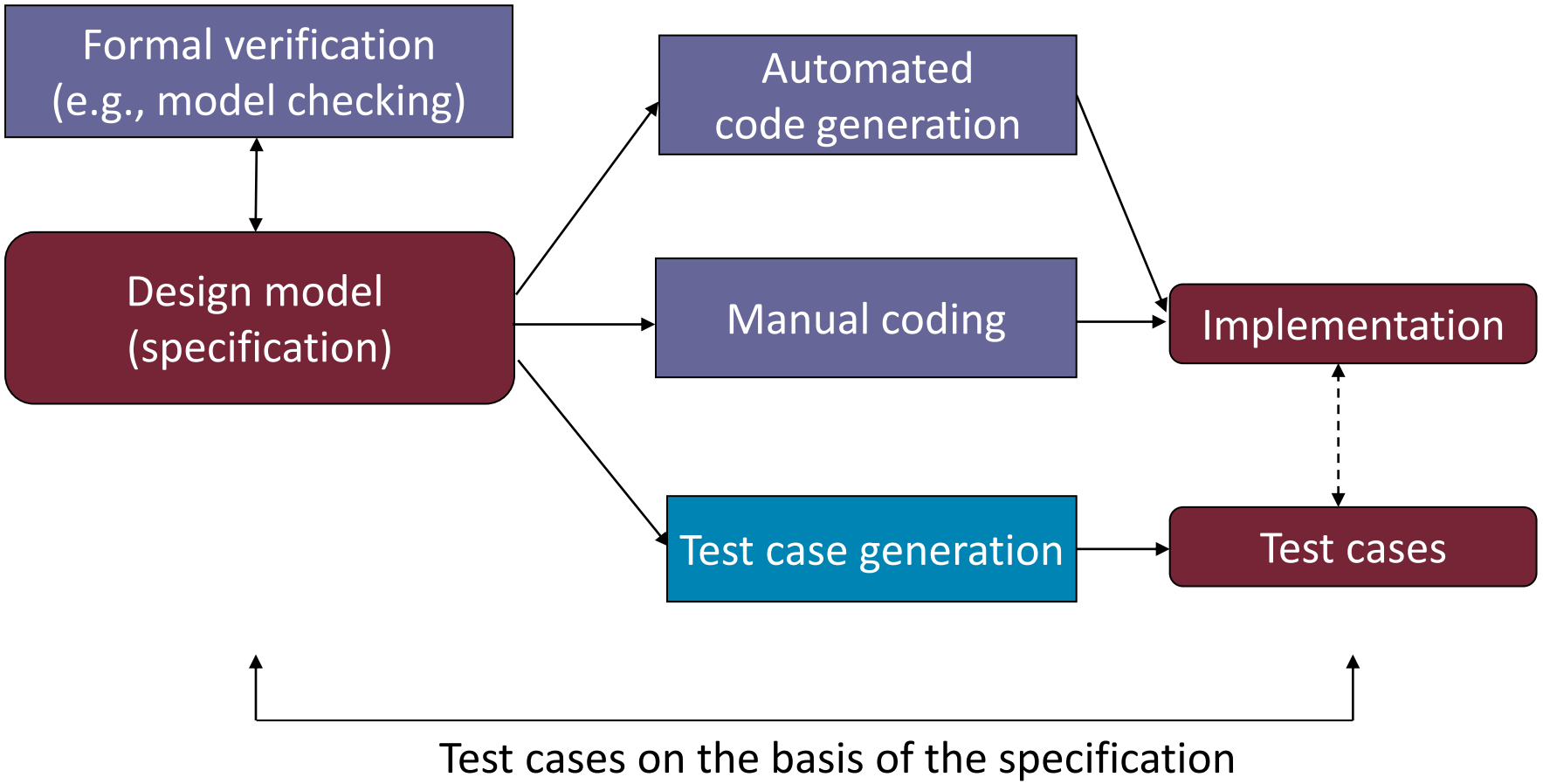
+ Tools for model based test case generation

Introduction

Common practice: UML models in manual testing

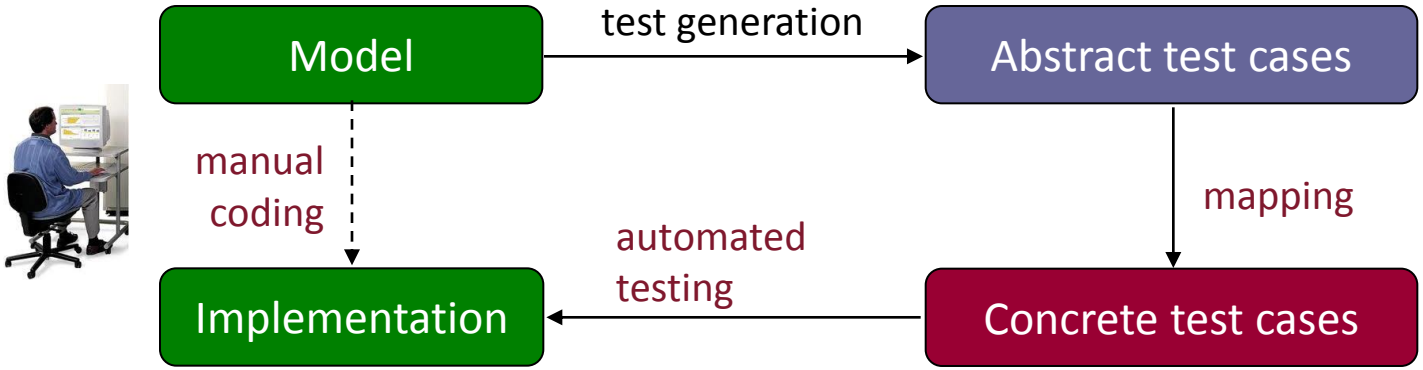
- Use case diagrams:
  - Validation (acceptance) testing: Covering use cases
- Class and object diagrams
  - Module testing: Identifying sw components, interfaces
- State machine and activity diagrams:
  - Module testing: Reference for structure based testing
- Sequence and collaboration diagrams:
  - 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

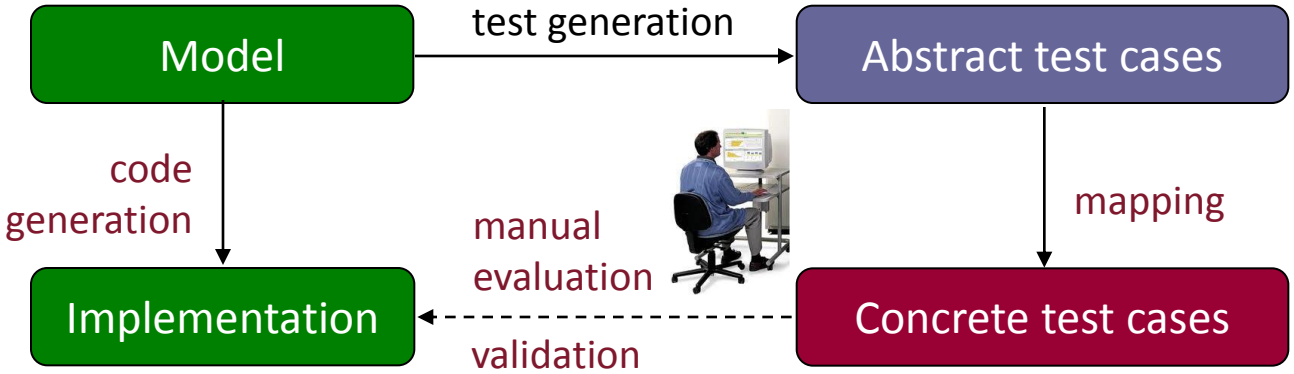


Use cases for model based testing

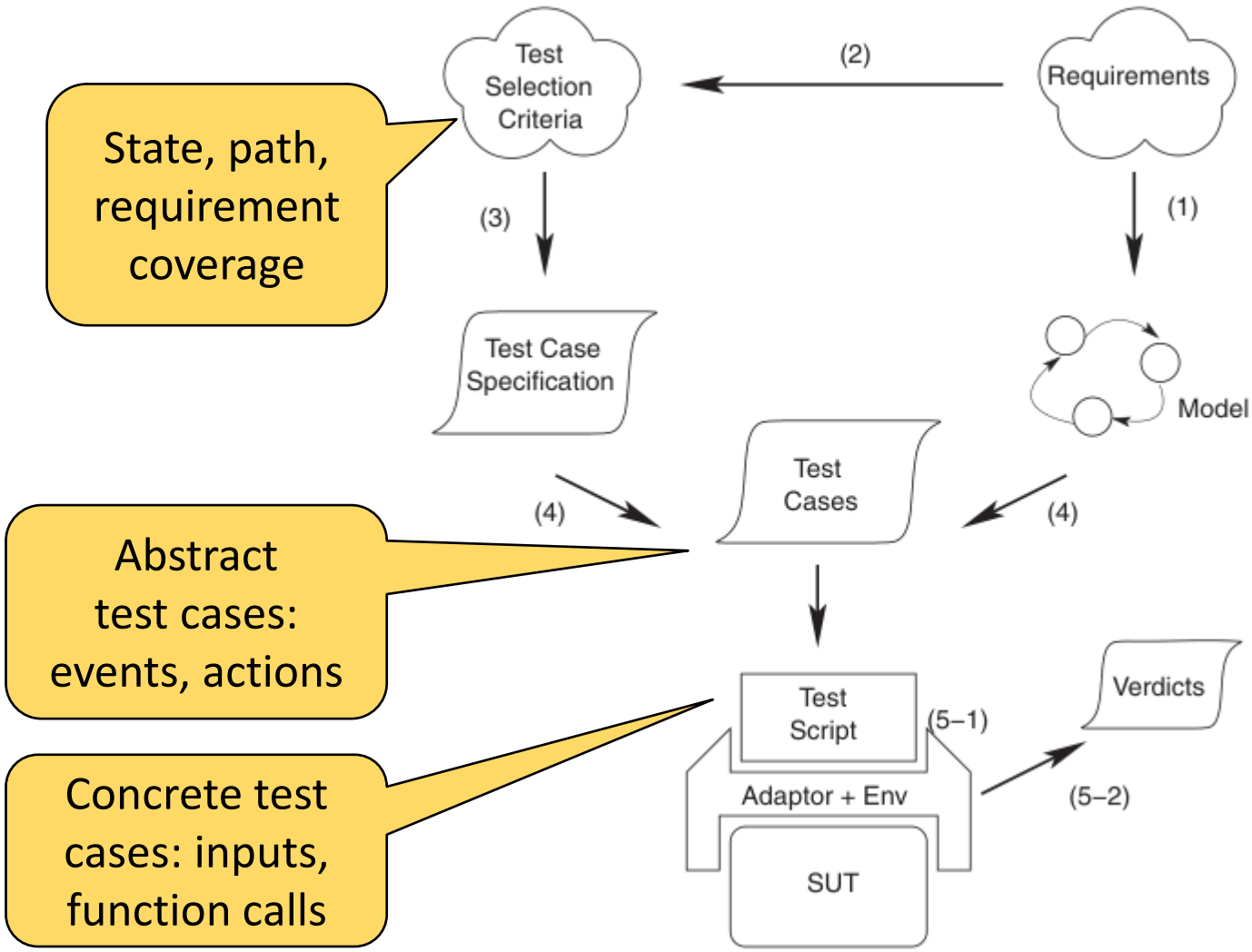
- In case of manual coding: Conformance checking



- In case of automated code generation: Validation

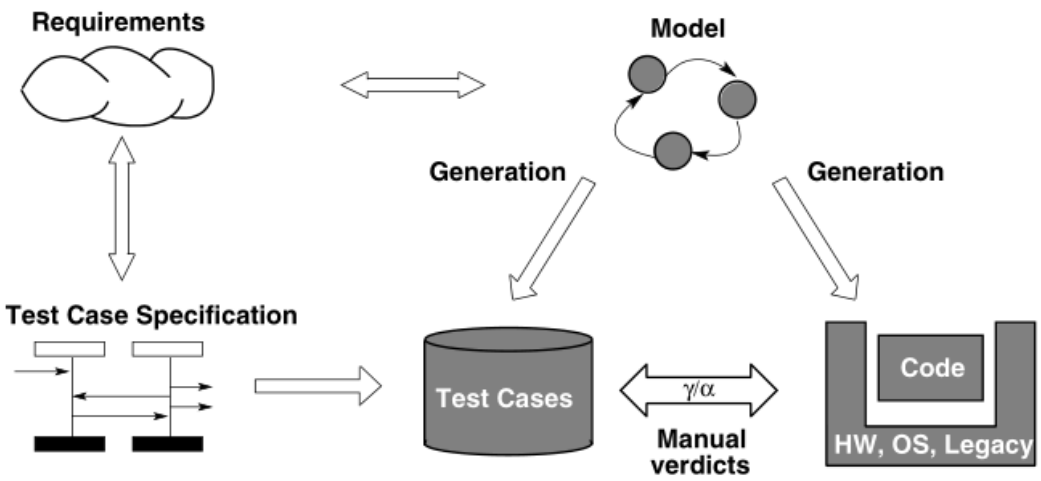


Abstract and concrete test cases



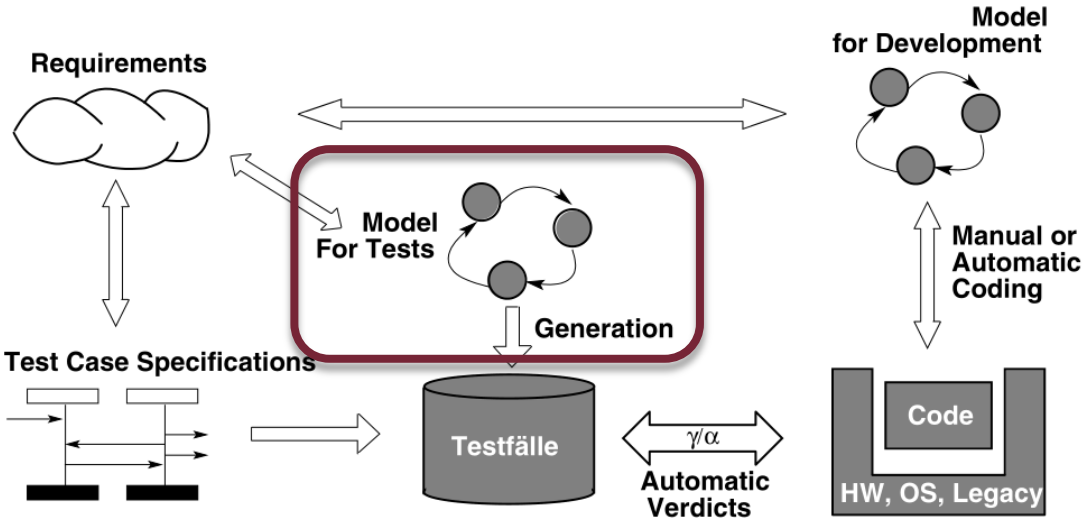
Source: M. Utting, A. Pretschner, B. Legeard. „A taxonomy of model-based testing approaches", STVR 2012; 22:297–312

The role of models



Using the design models as specifications: Testing the conformance of the model and the implementation

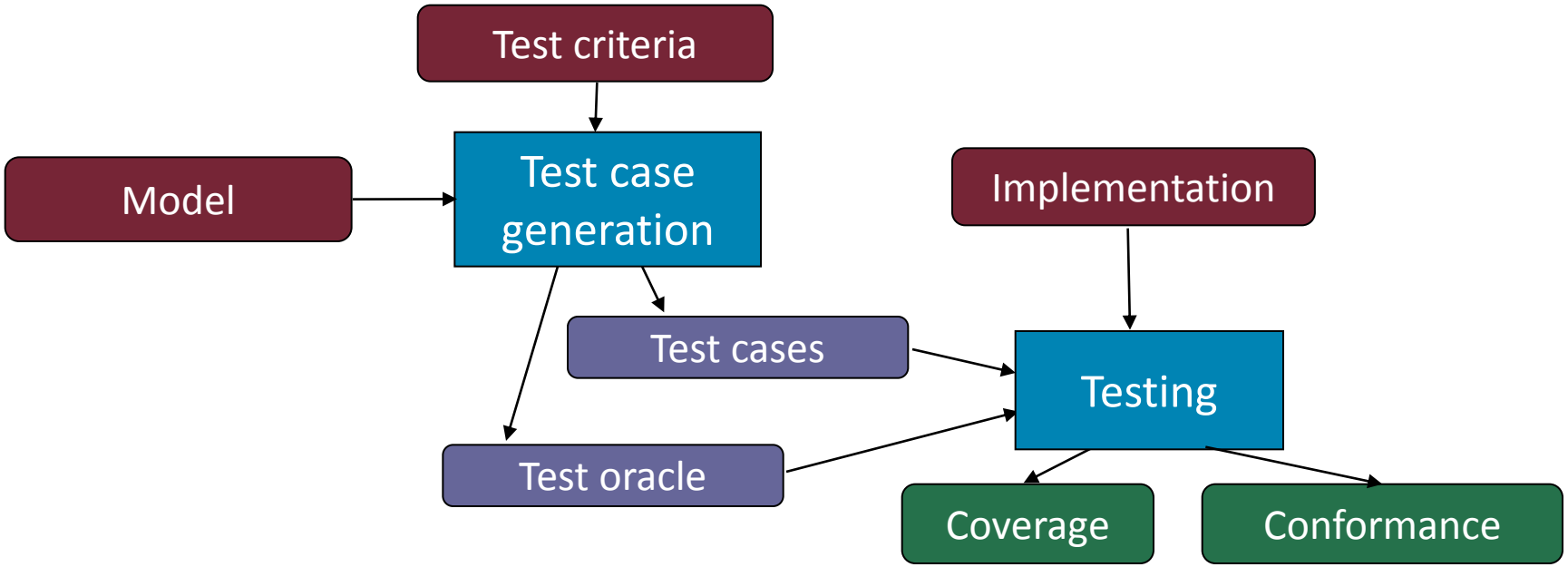
Using separate test models: Specifying what to test, how to test



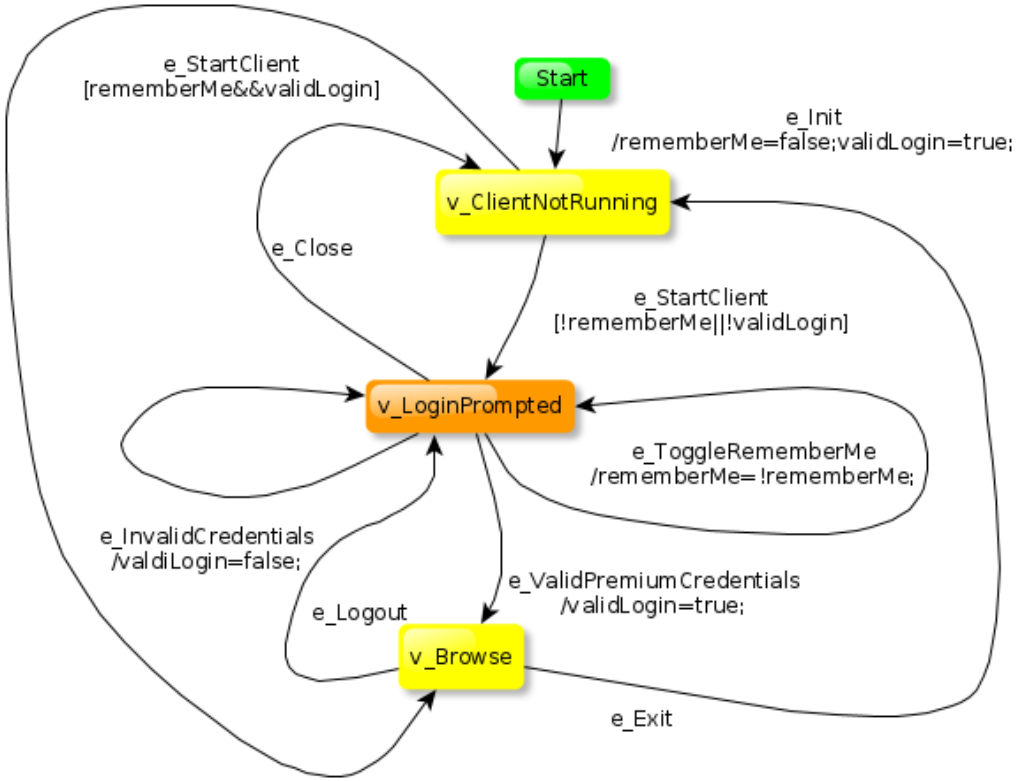
A. Pretschner, J. Philipps. „Methodological Issues in Model-Based Testing", Model-Based Testing of Reactive Systems, 2005.

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)



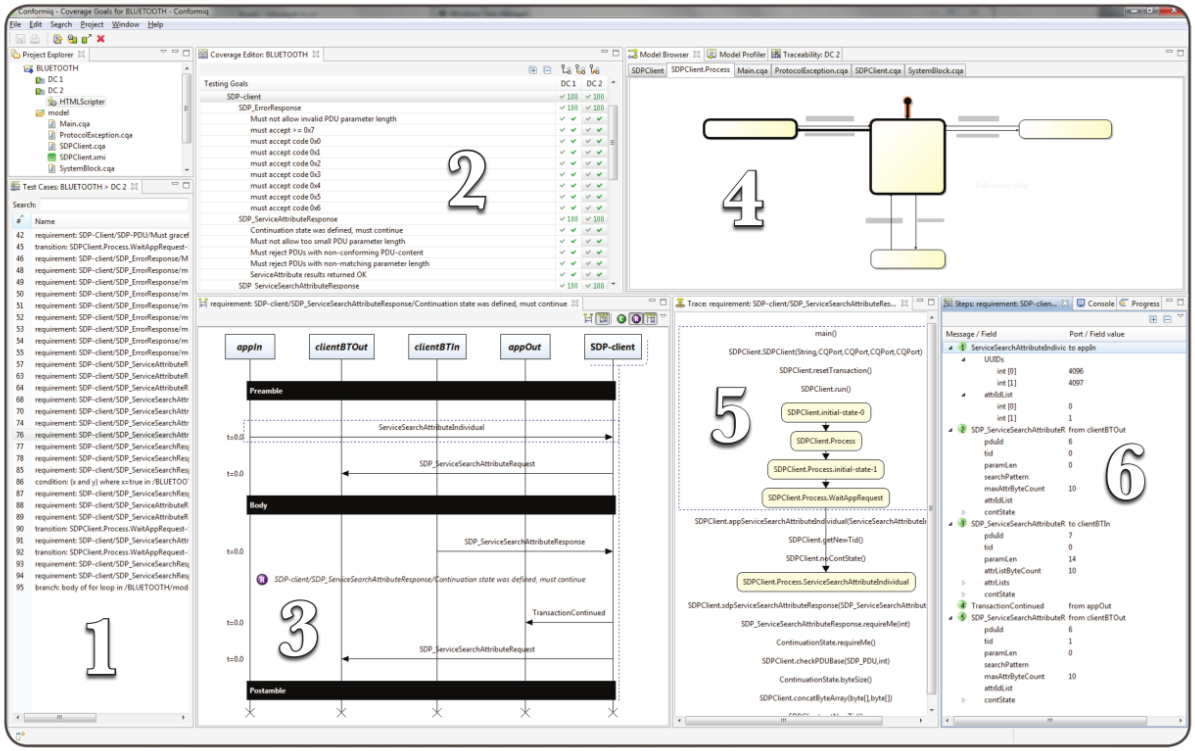
Example open source tool: GraphWalker



Source: GraphWalker

- Finite state machine modell + simple guards
- Tests for state and transition coverage
- Traversing the graph: random walk, graph based search, shortest path
- Generating JUnit test stubs (adapter)

Example industrial MBT tool: Conformiq

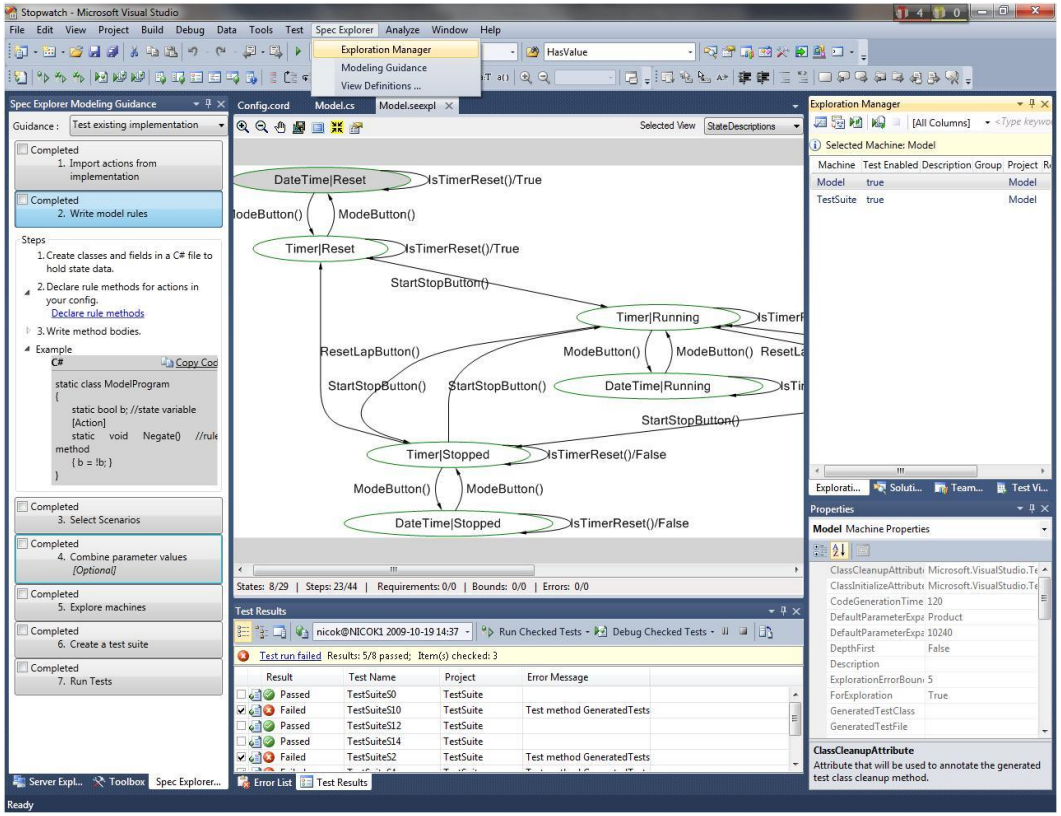


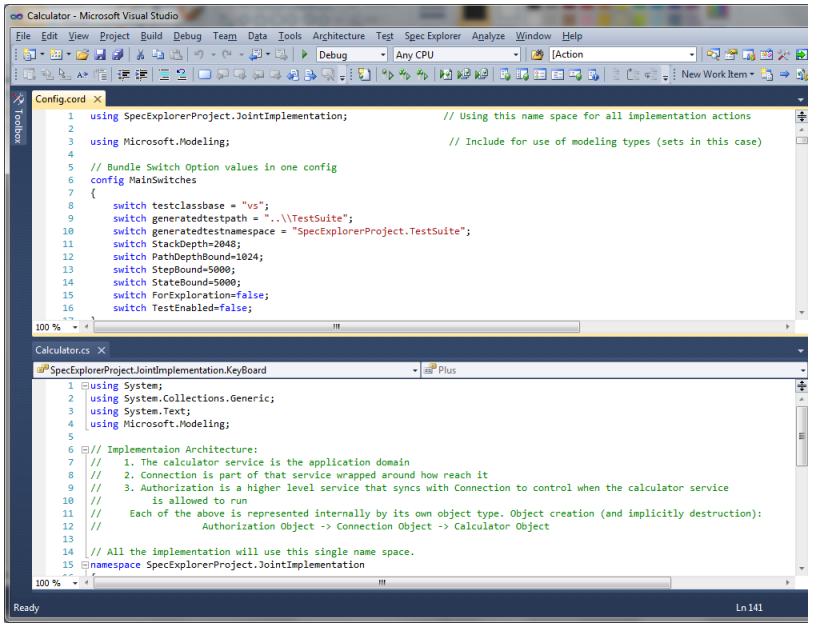
Conformiq Designer IDE for automatic test case generation

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

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

Example industrial MBT tool: SpecExplorer





Source: https://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745

- C# model program + adapter code
- Tests for covering scenarios, action patterns

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

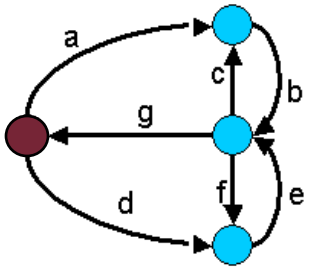
Graph-based algorithms for test generation

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, Büchi, …)
  - 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

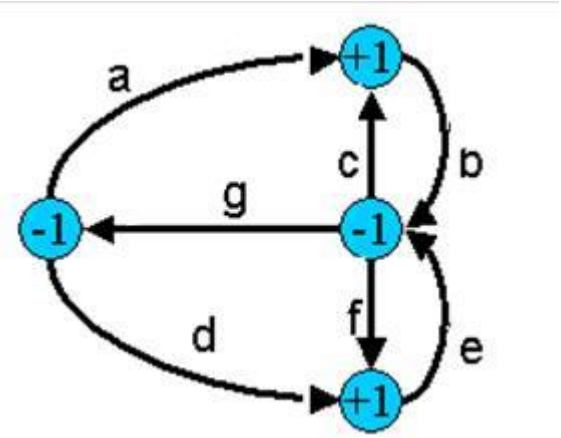
Graph-based algorithm for transition coverage

- 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, what is the (shortest) path that covers all transitions and goes back to the initial state?
    - (The same problem in undirected graphs: ”Chinese postman” problem)

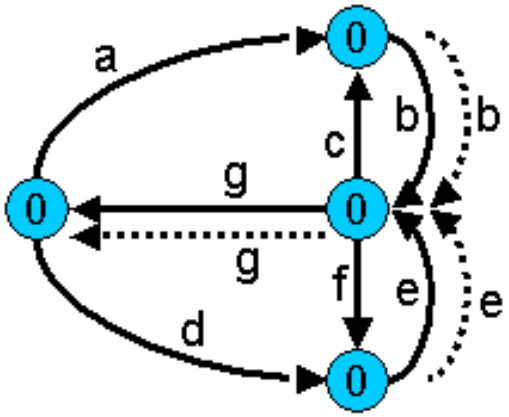


- Basic idea for the algorithm: Euler-graph $\rightarrow$ Euler-circuit
  - Computing the polarity of vertices: nr. of incoming minus outgoing edges
  - Duplicating edges that lead from a vertex with positive polarity to vertex with negative polarity, until all edges have zero polarity
  - Finding an 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



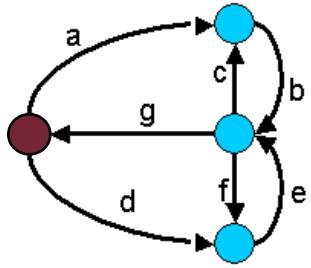
Graph with duplicated edges (this way having an Euler-graph)

Sequence for traversal (Euler-circuit):

a b c b f e g d e g

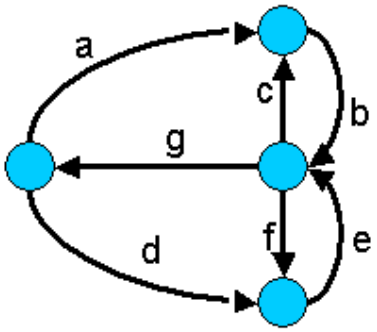
Graph-based algorithm for covering transition pairs

- 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

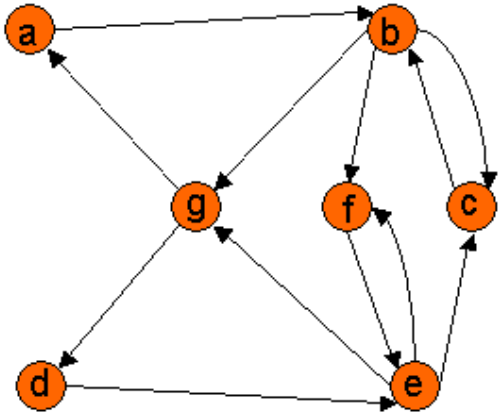


  - Graph-based problem: "Safecracker" sequence
    - (Shortest) edge sequence that includes all possible sequences of n subsequent edges (simplest case: n=2)
- Basic idea of the algorithm for n=2 (de Bruijn algorithm):
  - Constructing a dual graph
    - Edges of the original graph are mapped to vertices
    - 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)
  - Finding an Euler-circuit that defines the test sequence

Example: Covering transition pairs



Original graph



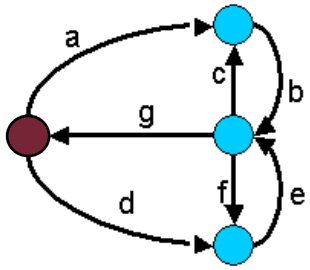
Dual graph with edges representing edge pairs in the original graph

Sequence for traversal that cover all transition pairs:

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

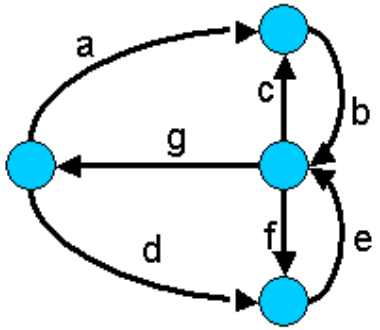
Graph-based algorithm for concurrent testing

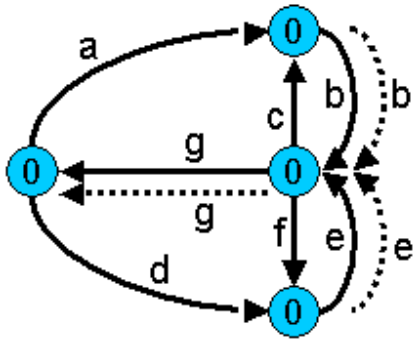
- Mapping the problem
  - Testing problem: Covering all transitions by concurrent testers
    - Goal is complete transition coverage
    - There are several testers that share (preferably equally) the testing task to finish it in the shortest time
    - All testers start in the initial state
    - Condition: 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

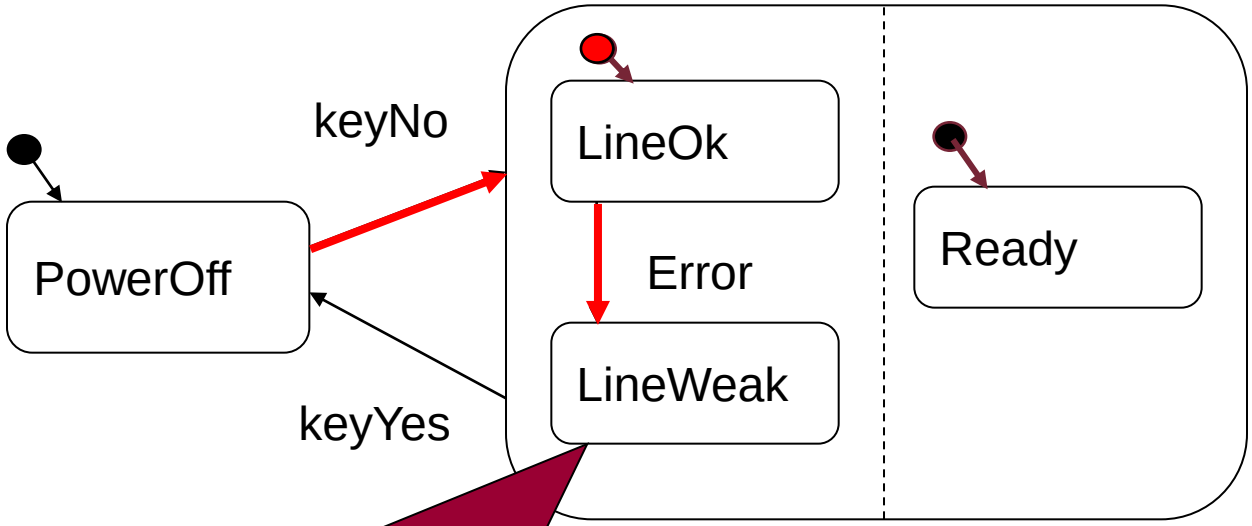
Basic idea

- Typical test coverage criteria (for the model):
  - Control flow based:
    - State coverage, transition coverage
    - Incoming-outgoing transition pairs coverage
  - 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

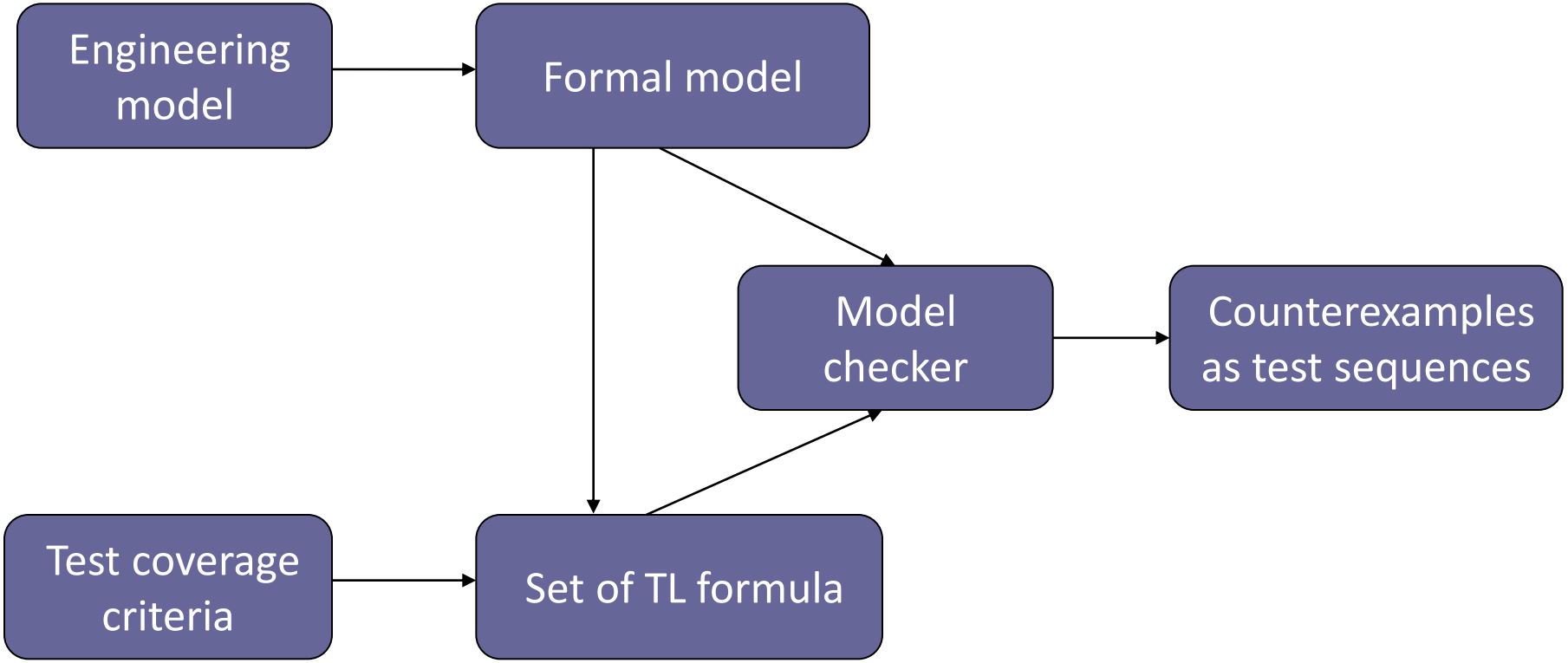


2.Specifying property for the model checker: The state LineWeak cannot be reached:

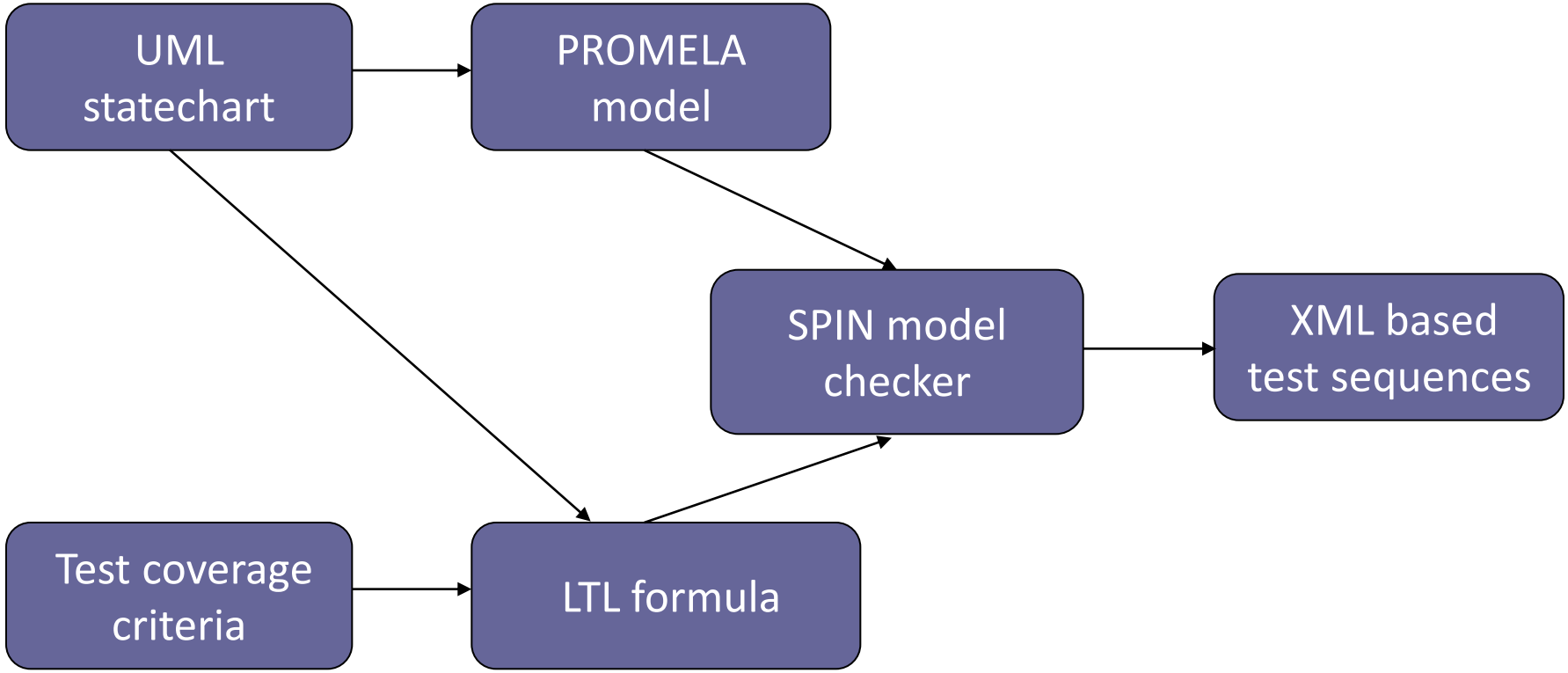
$\neg$ **EF LineWeak**

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 ($\rightarrow$ 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 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 coverage: Implicit transitions (not leaving the given state) are also tested

- Strong transition coverage:

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

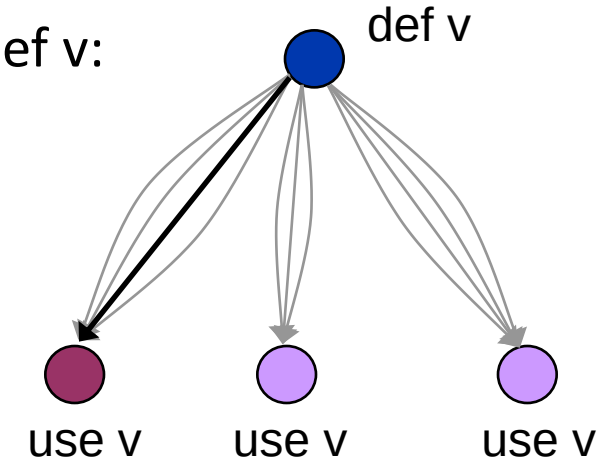
Recap: Data flow based test coverage criteria

- All-defs:

For all v, from all def v:

at least one def-clear path:

to at least one use v:

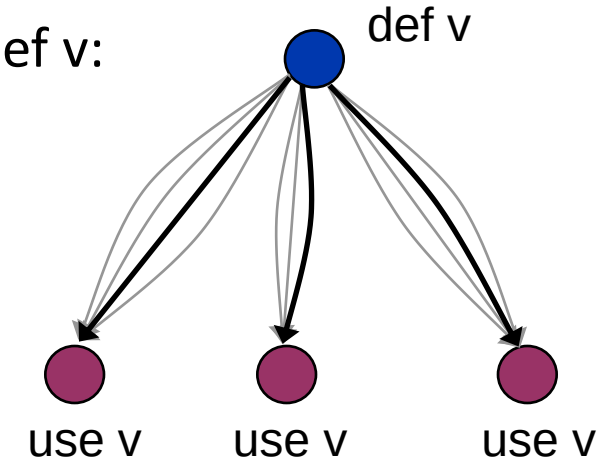


- All-uses:

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:

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

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

- Weak all-uses coverage:

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

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

- Strong all-defs coverage:

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

  Implicit variable usage: in conditions for not leaving the state

- Strong all-uses coverage:

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

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 path for a given def-use pair)
- 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

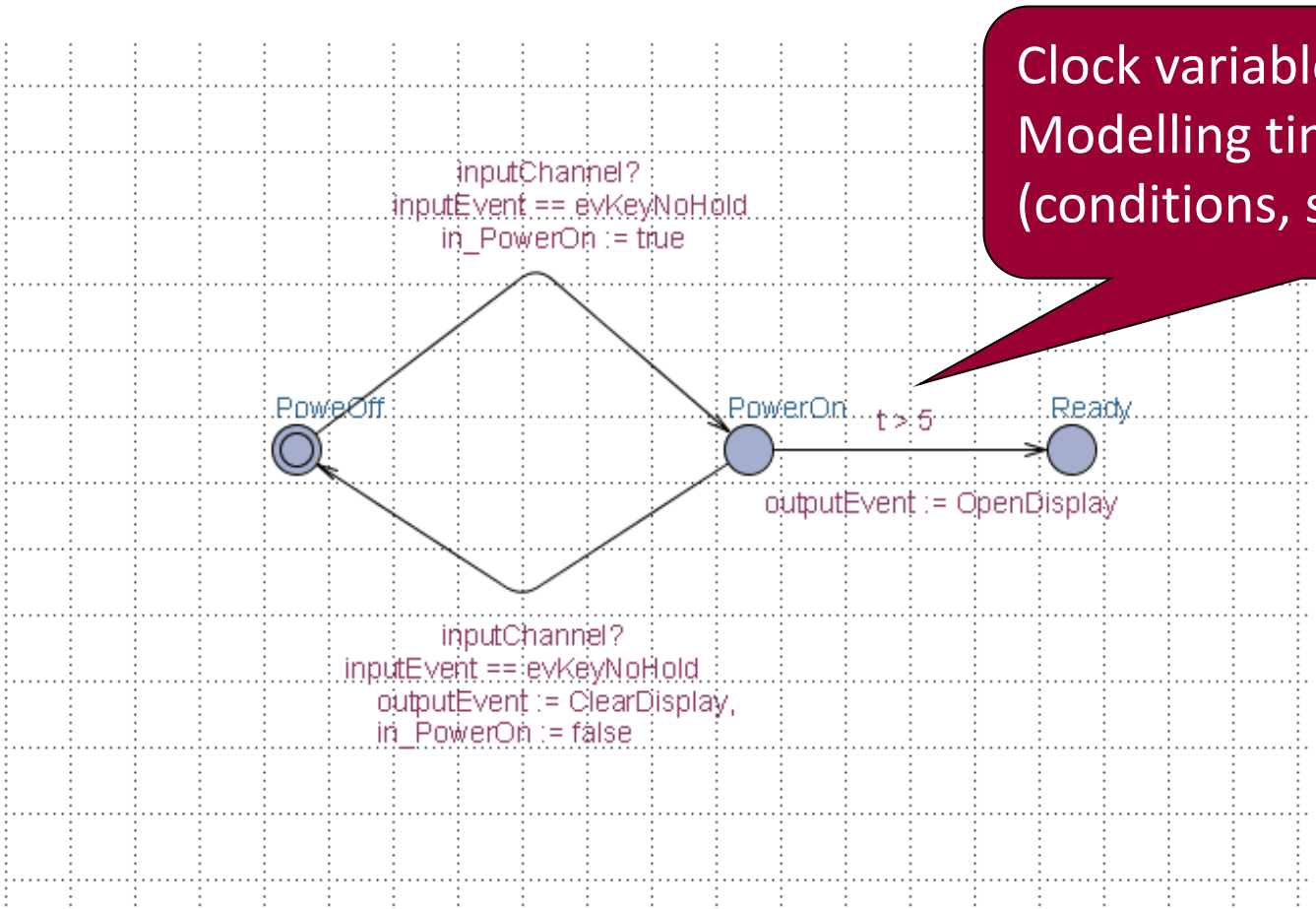
Options (compile time or run-time)	Time required for test generation	Length of all test sequences	Longest test sequence generated
-I	22m 32.46s	17	3
-dBFS	11m 48.83s	17	3
-i -m1000	4m 47.23s	17	3
-I	2m 48.78s	25	6
default	2m 04.86s	385	94
-I -m1000	1m 46.64s	22	4
-m1000	1m 25.48s	97	16
-m200 –w24	46.7s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



- 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

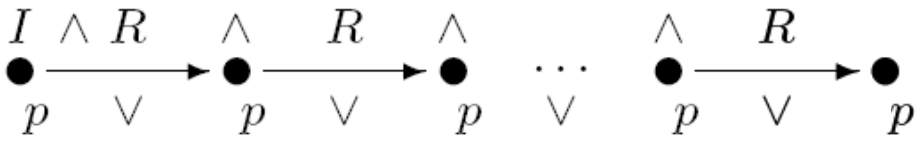
State:
(input.sending mobile.PowerOn mobile1.LineOK mobile2.CallWait)
t=6 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=5

Transitions:
input.sending->input.sendInput { 1, inputChannel!, 1 }
mobile2.CallWait->mobile2.VoiceMail { inputEvent == evKeyYes && t > 5 && in_PowerOn, inputChannel?, 1 }

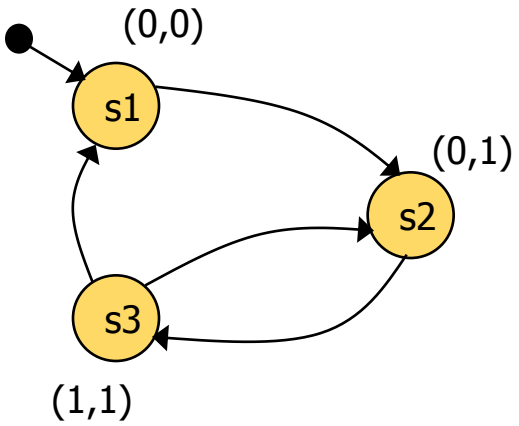
Test generation by bounded model checking

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:
  - Predicate for initial states: I(s)
  - Predicate for specified "bad" states: p(s)
  - State transition relation: R(s, s')
  - "Stepping forward" along the state transitions: $R(s_i, s_{i+1})$
- 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

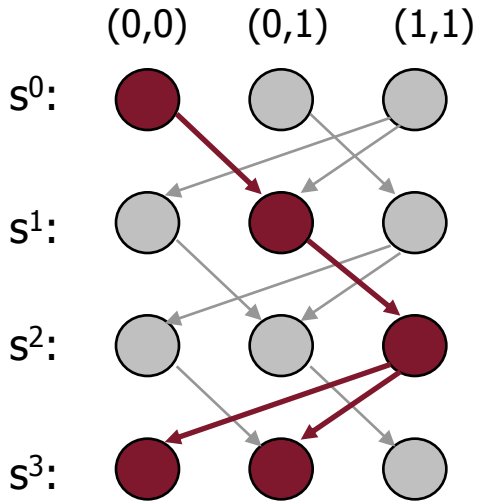


Initial state:

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

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:

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

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, …

$$\exists s_0, s_1, \ldots, s_k : I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge TG$$

State sequence | Model paths of k length | 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:	$R(s_i, s_{i+1})$	$R'(s'_i, s'_{i+1})$

Paths of length k from the initial state: $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$

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 k length in the original model
  - Traversing paths of k length in the mutant model
  - At least one output shall be different in the two models

$$\bigwedge_{i=0}^{k}(\mathrm{in}(s_i)=\mathrm{in}(s_i')) \wedge I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge I'(s_0') \wedge \bigwedge_{i=0}^{k-1} R'(s_i', s_{i+1}') \wedge \bigvee_{i=0}^{k}(\mathrm{out}(s_i)\neq \mathrm{out}(s_i'))$$

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: Belonging to the mutation
  - These are so-called negative tests (failed test: no mutation)
- How to define the “difference” between two models:
  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)

Refinement relations and testing

May preorder

Must preorder

Recap: Classification of relations

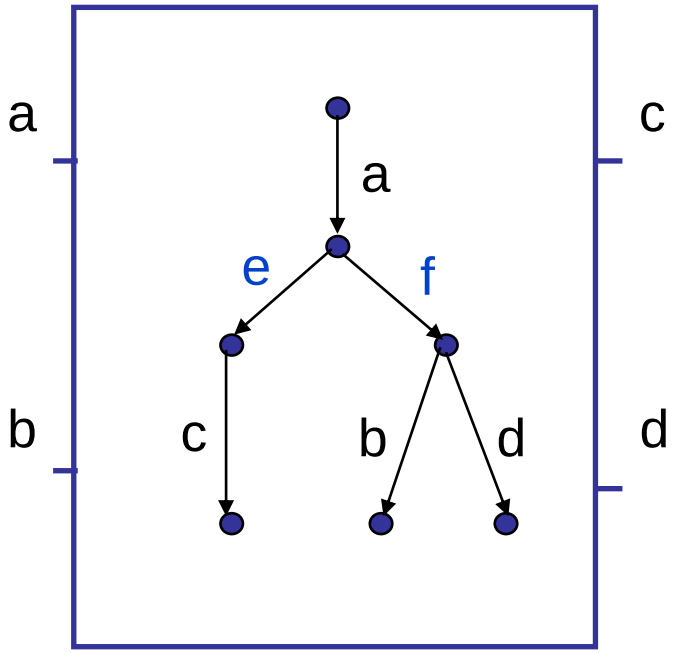
- Equivalence relations, denoted in general by =
  - Reflexive, transitive, symmetric

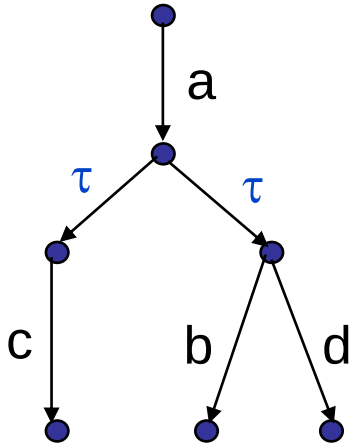
  Some equivalence relations are congruence:
  - If T1=T2, then for all C[ ] context C[T1]=C[T2]
  - The same context preserves the equivalence
  - Dependent on the formalism: how to embed in C[ ]

- Refinement relations, denoted by $\leq$
  - Reflexive, transitive, anti-symmetric ($\rightarrow$ partial order)

  Precongruence relation:
  - If $T1 \leq T2$, then for all C[ ] context $C[T1] \leq C[T2]$
  - The same context preserves the refinement

Recap: Modelling behavior and internal actions





Internal behavior
of the component:
e and f are internal actions

Observable behavior
of the component:
e and f are mapped to τ

Recap: The notion of “test” and “deadlock”

- “Test” in LTS based behavior checking:
  - Test: A sequence of actions that is expected (from initial state)
    - Analogy: interactions on ports during testing
    - Test steps are actions that may represent: sending or receiving messages, raising or processing events etc.
- “Deadlock” in LTS based behavior checking:
  - A given action cannot be provided by the system in an expected action sequence (test)
    - Analogy: no interaction is possible on a port
    - The deadlock is given by the action that is not possible; it may represent that is not possible to send or receive message, process an event etc.
  - Failure of a test: The action that cannot be provided (deadlock)
  - Successful test: The action sequence can be provided

May preorder: Definition

- Notation:

$\beta \in (Act - \tau)^*$ observable action sequence (τ deleted)

$s \overset{\beta}{\Rightarrow} s'$ if $\exists \alpha \in Act^*: s \overset{\alpha}{\rightarrow} s'$ and $\beta = \hat{\alpha}$

$\Delta(s)$ is the set of observable action sequences from s:

$$\Delta(s) = \left\{ \beta \mid \exists s' : s \overset{\beta}{\Rightarrow} s' \right\}$$

- Definition of the may preorder refinement relation:

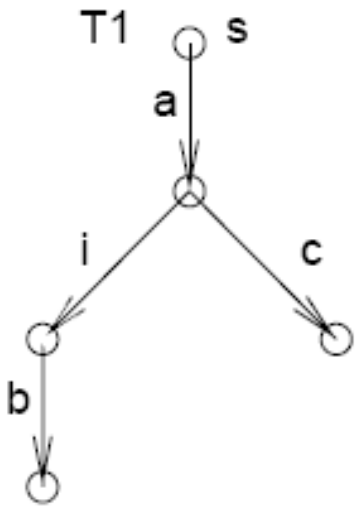
For T_1 and T_2 LTSs with initial states s_1 and s_2, *Act* actions:

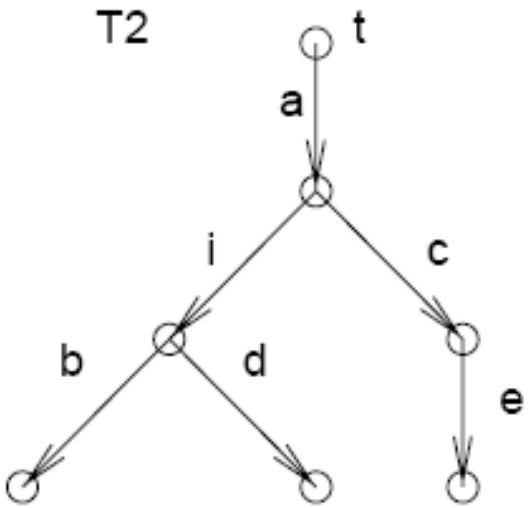
$$T_1 \leq_\Delta T_2 \text{ iff } \Delta(s_1) \subseteq \Delta(s_2)$$

Here T_2 offers more observable action sequences
(more possible behaviors that can be observed)

Example: May preorder

Two LTSs with observable action sequences: $T_1 \leq_\Delta T_2$





$\Delta(s) = \{a, ab, ac\}$ $\Delta(t) = \{a, ab, ac, ad, ace\}$

May preorder: Relationship with testing

- In case of $T_1 \leq_\Delta T_2$
  - Each test that may be successful in case of T_1, may also be successful in case of T_2
    - When a test "may be successful": due to nondeterministic behavior or internal actions it may also fail
  - The relation preserves the possibly successful tests: Possibly successful tests of T_1 are included in the possibly successful tests of T_2
- Refinement defined by may preorder:
  - Possible observable behavior is preserved (not lost)
- To be defined: Another refinement relation:
  - Mandatory observable behavior is preserved
  - Preserves the tests that are always successful (never fail)

Must preorder: Notation for failed tests

s refuses $E \subseteq Act - \{\tau\}$ actions, if $\forall e \in E$: there is no $s \overset{e}{\Rightarrow} s'$

s divergent ($s \Uparrow$),

if $\exists s_0 s_1 \ldots$ infinite sequence, where $s = s_0$ and $s_i \overset{\tau}{\rightarrow} s_{i+1}$

s divergent on β action sequence ($s \Uparrow \beta$),

if $\exists \beta'$ prefix of β, such that $s \overset{\beta'}{\Rightarrow} s'$ and $s' \Uparrow$

$\langle \beta, E \rangle$ is a failure of s, if

either $s \overset{\beta}{\Uparrow}$

or $\exists s'$: $s \overset{\beta}{\Rightarrow} s'$ and s' refuses E

(i.e., divergent on β, or after β it refuses E).

$F(s)$ is the set of all failures for s.

Must preorder: Definition

Refinement relation: Covering failures

$$T_1 \leq_F T_2 \quad \text{iff} \quad \mathrm{F}(\mathrm{s}_1) \supseteq F(\mathrm{s}_2)$$

i.e., there are less failures in T_2 than in T_1.

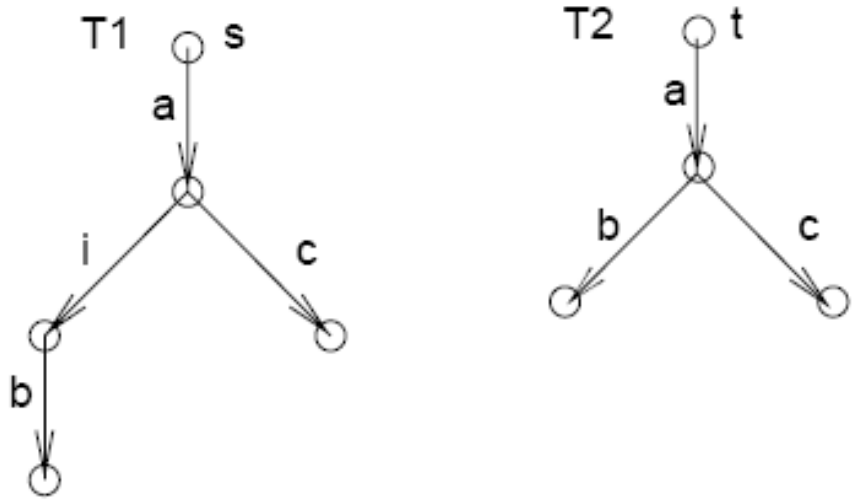
The role of failures:

- Failures: Refusing actions directly of because of divergence
- Less failures: Less possible refusals, more successful actions (action sequences)
- If the behavior is extended in the usual way then the number of failures will decrease (actions become possible)
- Reducing nondeterminism may also result in decrease of the number of failures

Must preorder: Relationship with testing

- In case of $T_1 \leq_F T_2$
  - Each test that is always successful in case of T_1, also always successful in case of T_2
    - T_2 has less failures, cannot refuse more actions (tests)
  - The refinement preserves the tests that are always successful:
    - Tests that are always successful in T_1 are included in the tests that are always successful in T_2
- Characteristics of must preorder:
  - The refined LTS preserves observable behaviors that were already included in the more abstract LTS
- Relation with deadlock possibility:
  - The refinement is sensitive to deadlocks

Example: Must preorder



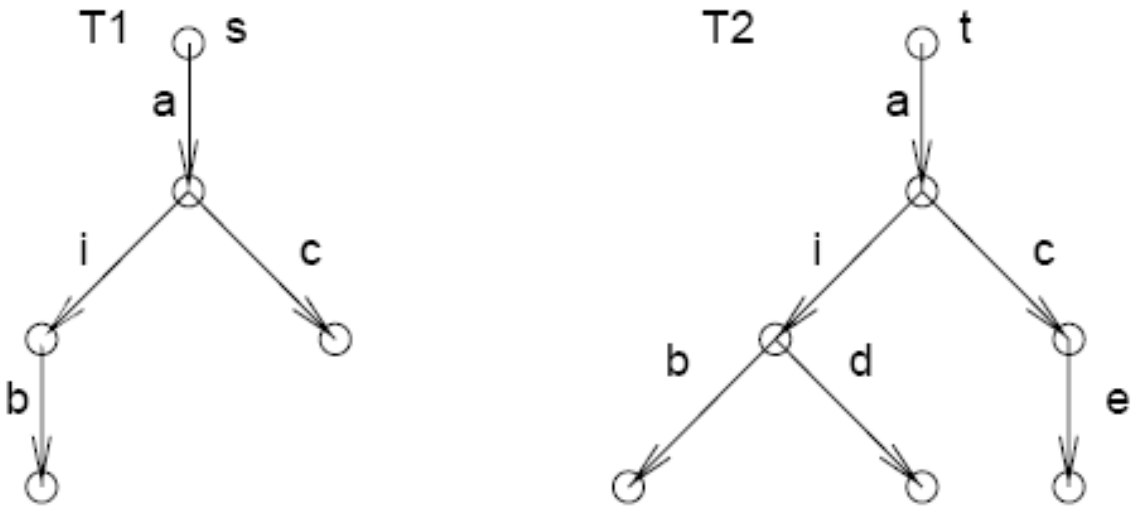
Tests of T1 that are always successful: {a,ab}

here <a,{c}> is a failure

Tests of T2 that are always successful: {a,ab,ac}

here <a,{c}> is not a failure

Example: Must preorder



Tests of T1 that are always successful: {a,ab}

Tests of T2 that are always successful: {a,ab,ad}

the set of failures is reduced

Conformance relation for testing: IOCO

Input Output Conformance

Desirable properties of a conformance relation

- Trace based relation (for test evaluation)
  - Goal is to compare the behavior observed during testing and the behavior described in the specification (to identify faulty behavior)
  - For black box testing: Distinguishing inputs, outputs, and internal (invisible) actions
  - Arbitrary interleaving of inputs and outputs (not fixed as input-output pairs)
  - The lack of output action is considered as a specific behavior (i.e., there is fault if the specification does not allow the lack of output)
  - Nondeterministic behavior shall be possible
- Model: More precise than LTS
  - Action types
    - Input actions: Provided by the test driver
    - Output actions: Observable by the test evaluator
    - Internal (invisible) action: Not controlled by the environment
  - Quiescent state:
    - There is no output transition labelled by output action or internal action
      → Output transition(s) labelled only by input action(s)

The IOLTS formalism

- Input-Output Labelled Transition System (IOLTS):

$$IOLTS = (S, Act, \rightarrow, s_0)$$

S is the set of states, s_0 initial state

Act is the set of actions: $Act = Act_{in} \cup Act_{out} \cup \{\tau\}$

$\rightarrow \subseteq S \times Act \times S$ is the state transition relation

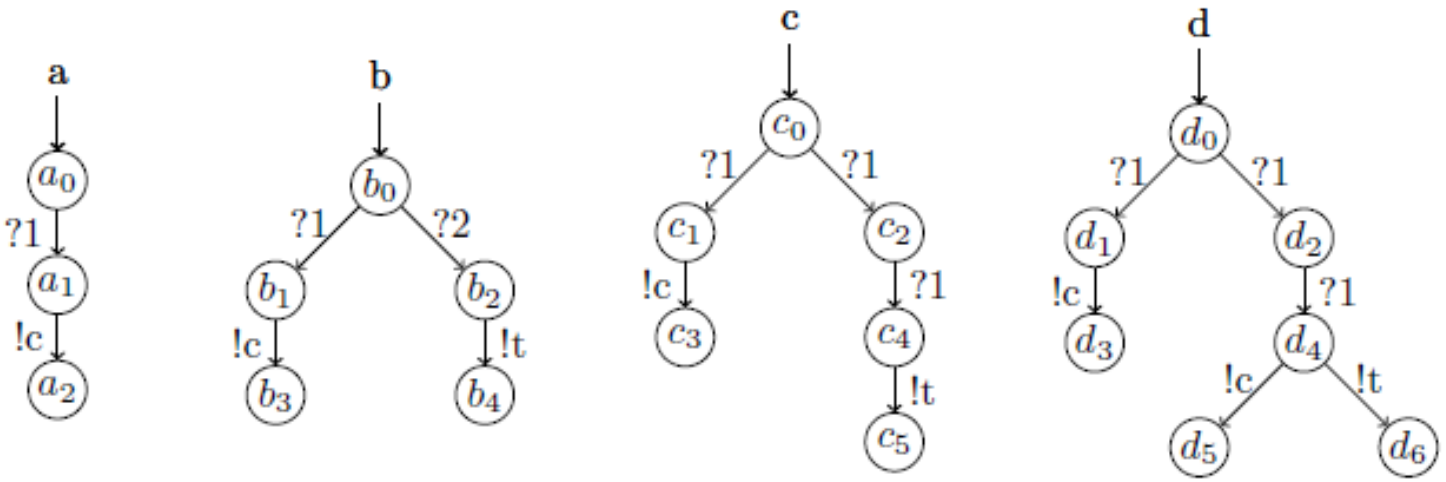
Act_{in} input, Act_{out} output actions, τ internal action

- Properties of an IOLTS:
  - Complete, if in each state there is transition defined for each action
  - Input complete (weakly input enabled), if in each state there is transition defined for each input action, possibly after τ^*
    - Property of implementation model: Each input is handled somehow
  - Deterministic, if there is only a single target state in case of each observable action sequence

IOLTS examples

Coffee automaton IOLTS:

- Act_{in}={1,2} inputs (coins)
  - Notation: with ? prefix: ?1, ?2
- Act_{out}={c,t} outputs (coffee or tee)
  - Notation: with ! prefix: !c, !t



Further notations and transformations

- Notations:
  - β observable action sequence
  - $\Delta(T)$ set of observable action sequences of IOLTS T
  - In(s) set of input actions on transitions from state s
  - Out(s) set of observable output actions from state s
  - Out(S) set of observable output actions from state set S
  - Reachable states: with an "after" operator

$$s \text{ after } \beta = \left\{ s' \mid s \stackrel{\beta}{\Rightarrow} s' \right\} \qquad S \text{ after } \beta = \bigcup_{s \in S} (s \text{ after } \beta)$$

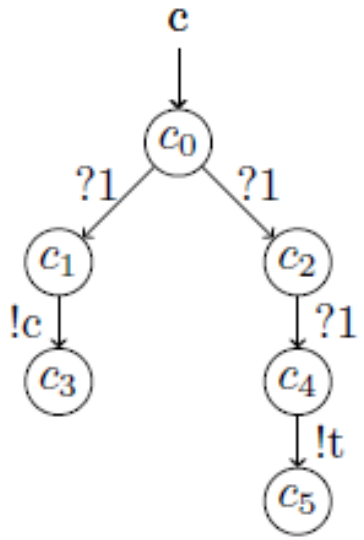
- Handling quiescent states in a uniform way:
  - The quiescent states (i.e., waiting for input) are denoted by a loop transition labelled with a specific δ output action
    - This way we get an extended IOLTS T_δ
  - In this IOLTS quiescence can be "observed" as output δ

Example: IOLTS extended with quiescence

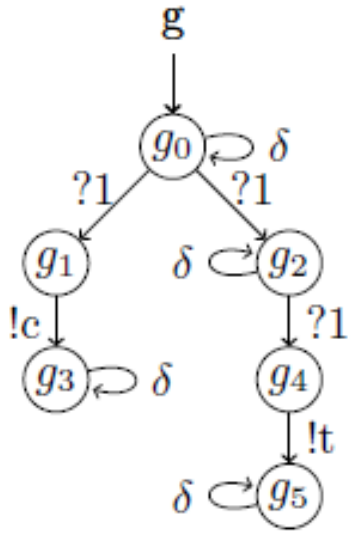
Coffee automaton IOLTS:

- Act_{in}={1,2} inputs (coins), ? prefix
- Act_{out}={c,t} outputs (coffee or tee), ! prefix

If there is no output action from a state then a δ loop transition is added



Extended with quiescence:

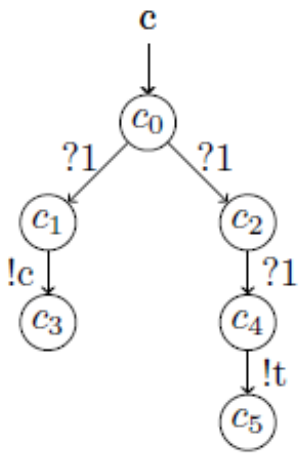


Example: IOLTS made complete

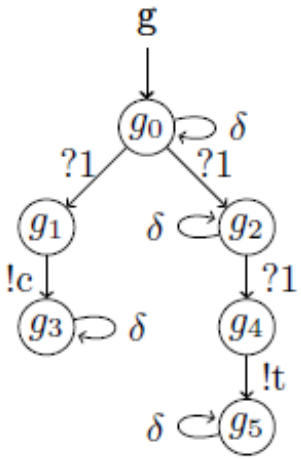
Coffee automaton IOLTS:

- Act_{in}={1,2} inputs (coins), ? prefix
- Act_{out}={c,t} outputs (coffee or tee), ! prefix

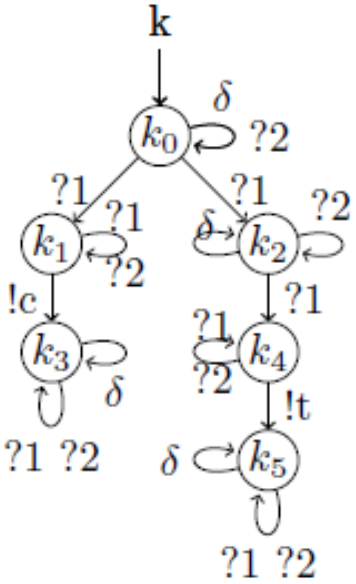
Loop transitions for actions that were missing:



Extended with quiescence:



Then made input complete:



k-equivalence for IOLTS

- Elements of the definition:
  - T_δ IOLTS as "specification" (expected behavior)
  - M_δ IOLTS as "implementation" (provided behavior)
  - Outputs follow inputs (reactive behavior)
- Definition:
  - In the "specification" T_δ and "implementation" M_δ, the same input sequence results in the same output sequence for the first k steps
- Properties
  - Simple relation
  - Strict for testing (in k steps): Restrictions, extensions of the behavior are not allowed

IOCO relation for IOLTS

- Elements of the definition:
  - T_δ IOLTS as "specification" (expected behavior)
  - M_δ IOLTS as "implementation", that is made input complete
  - The set of potential actions is the same
- Definition: M ioco T ("M is ioco conform to specification T")

  For all observable action sequence in the specification: In each state that is reachable by such action sequence, the output actions provided by implementation M form a subset of the output actions given in specification T

$$\forall \beta \in \Delta(T_\delta) : \mathrm{Out}(s_{0,M_\delta} \text{ after } \beta) \subseteq \mathrm{Out}(s_{0,T_\delta} \text{ after } \beta)$$

For all observable action sequence β in the specification

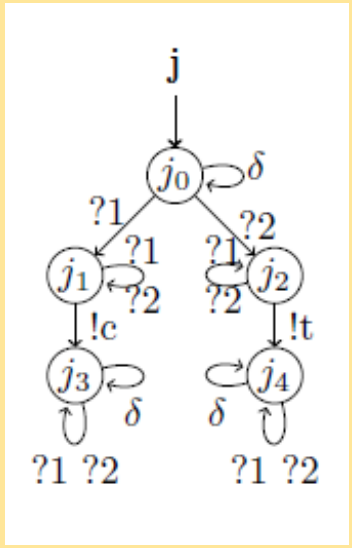
Observable output actions in the implementation after β

Observable output actions in the specification after β

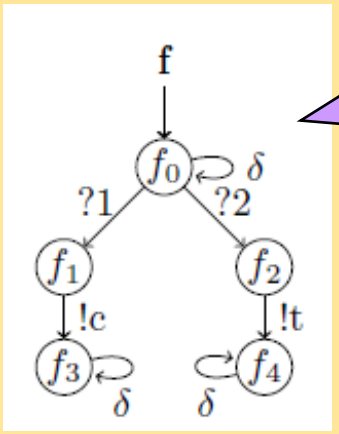
Properties of IOCO

- Explaining the definition:
  - Def.: For all observable action sequence in the specification: In each state that is reachable by such action sequence, the output actions provided by implementation M form a subset of the output actions given in specification T
  - The specification provides limits for the allowed behavior
  - The implementation fits the frame given by the specification
- What are allowed?
  - Restricted behavior: The implementation may contain less potential output action than in the specification
    - E.g., in case of a partial implementation, or partial resolution of nondeterminism
  - Extended behavior: The implementation may contain outputs after action sequences that are not included in the specification
    - E.g., the specification is not complete (some action sequences are not included)
- What is not allowed?
  - Implementation (its outputs) cannot be extended in case of action sequences that are included in the specification, i.e., it is not allowed to “provide more”

Examples for satisfying IOCO



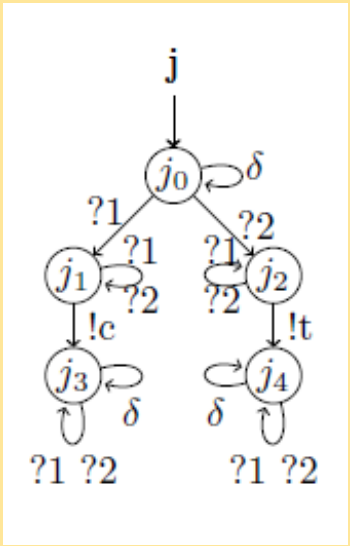
ioco



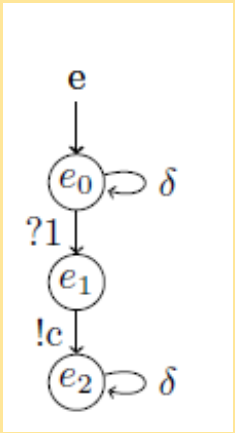
specification

The observable output actions shall be checked after each observable action sequence

The implementation may contain additional action sequences, but keeps the behavior for action sequences given in the specification



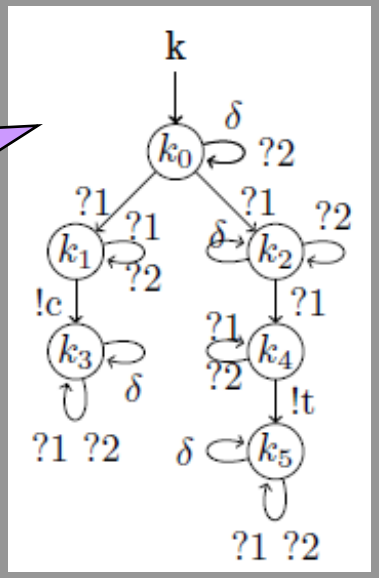
ioco



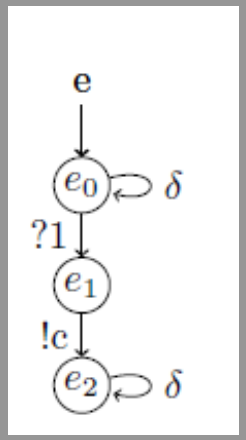
specification

Examples for violating IOCO

The implementation extends the behavior in case of action sequences given in the specification



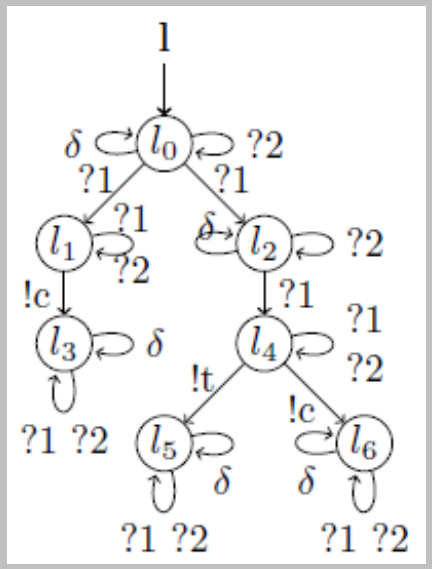
not ioco



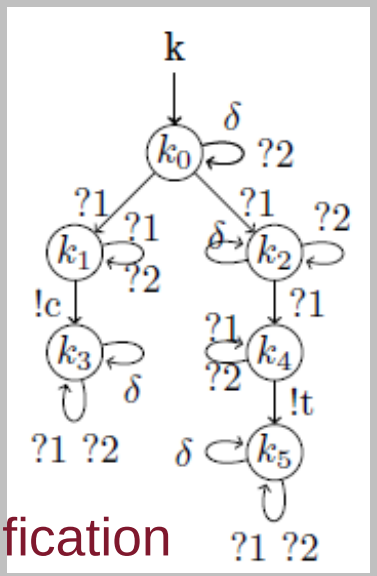
specification

k_0 after ?1 = {!c,δ}

e_0 after ?1 = {!c}



not ioco



specification

k_0 after <?1, δ, ?1>

l_0 after <?1, δ, ?1>

Summary of IOCO features

Input-output conformance relation (IOCO) by Tretmans, 1996:

- This relation is designed for functional black box testing of systems with inputs and outputs
- Inputs are under control of the environment, i.e. the tester, while outputs are under control of the implementation under test
- IOCO allows one to use incomplete specifications
- The specification and the implementation can be non-deterministic
- The models used for IOCO allow arbitrary interleaving of inputs and outputs
- IOCO considers the absence of outputs as error if this behavior is not allowed by the specification

These properties make input-output conformance testing applicable to practical applications

Summary of the studied behavioral equivalences

- Equivalences: For verification
  - Trace equivalence: $T \approx_{\Lambda} T'$ iff $\Lambda(s)=\Lambda(s')$
  - Strong bisimulation: $T \sim T'$ iff $\exists B : (s, s') \in B$
  - Observation equivalence: $T \approx T'$ iff $\exists WB : (s, s') \in WB$

- Preorders: For model refinement
  - May preorder: $T \leq_{\Delta} T'$ iff $\Delta(s) \subseteq \Delta(s')$
  - Must preorder: $T \leq_{F} T'$ iff $F(s) \supseteq F(s')$

- Conformance relation: For testing
  - k-equivalence
  - Input-output conformance (IOCO)

Other techniques and tools for model based test generation

Using a planner for test generation

- Planning problem in AI
  - Construction of an action sequence to reach a goal state from an initial state (using a set of actions with conditions and effects)
- Elements of the planning problem for test generation:
  - Initial state: Initial state of application (model)
  - Goal state: State to be reached (covered)
  - Actions: Activities (functions) executed on the basis of inputs in the application
- Test: Determined by the generated action sequence
  - Instances of actions: Determine required inputs for triggering
  - Partial ordering of actions (as given by mapping the conditions and effects) $\rightarrow$ partial ordering of inputs
  - Test input sequence results from linearization of the input sequence

Application for GUI testing (1): GUI model

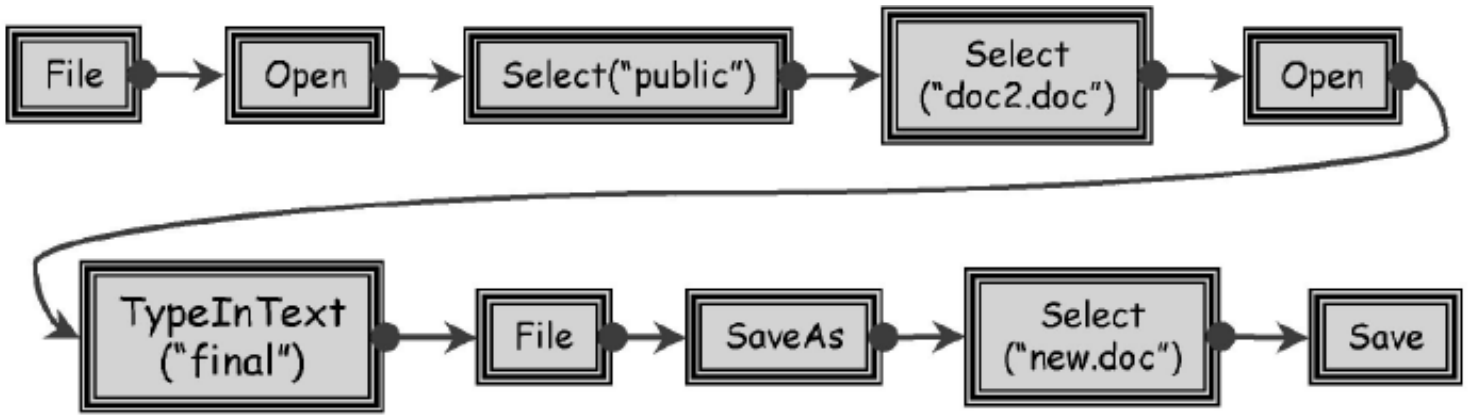
Operator model of the GUI: Basis of model based test generation

- System objects
  - Elements of the background application: text, data, files, …
- Events on the GUI
  - Menu events (ME): Expanding operations (e.g. File/Save)
  - Focus extension events (FEE): Opening new windows, palettes, etc.
  - System related events (SRE): Changing the state of system objects
- Operators
  - GUI operators: Combining ME and FEE
    - Using GUI widgets (e.g., opening a new file selection window by File/Open)
  - System related operators: (ME,FEE)*SRE
    - Initiating the change of system objects (e.g., Edit/Cut and its effect on the text)
  - Composite (abstract) operators: Sequences of basic operators
    - E.g., saving a file to a selected folder

Application for GUI testing (2): Testing approach

Scenario based testing approach:

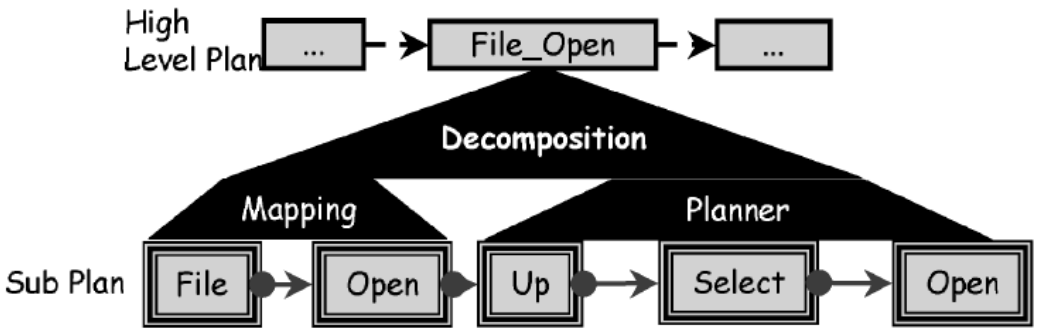
1. Specifying test elements and goals
   - Identification of operators and system objects
   - Identification of use case scenarios (with initial and goal state)
2. Construction of operator sequences
   - Implementation of the use cases with composite operators
3. Mapping each operator sequence to a concrete event sequence
   - Test sequence is defined by the GUI event sequence
   - Composite operators can be resolved by a planner



Application for GUI testing (3): Using the planner

Solving a planning problem for test generation:

- Elements of the planning problem for GUI testing:
  - Initial state: Initial state of the GUI and system objects
  - Goal state: GUI and system state to be reached (covered)
  - Actions (with conditions and effects): GUI operators and events, variables of operators are system objects
- Solution by the planner: Plan to reach goal state from the initial state
  - Actions are identified (with partial ordering) → GUI operators and events
  - Conditions and effects of actions → Conditions and changes in system objects
  - Variables of actions → Instantiated with system objects
- Test sequence is derived by ordering (linearization)

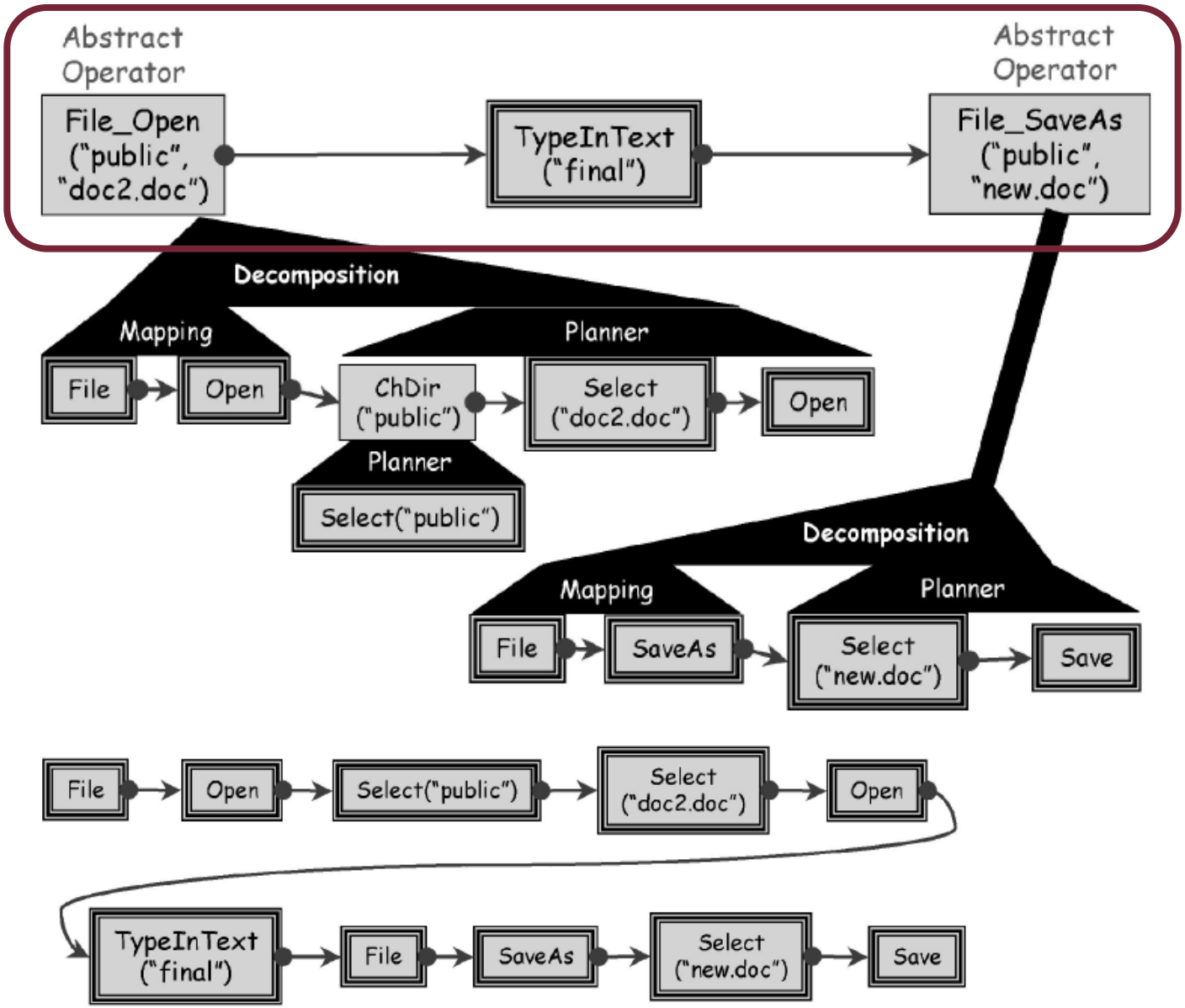


Application for GUI testing (4): Example

High level plan:

Mapping with planner:

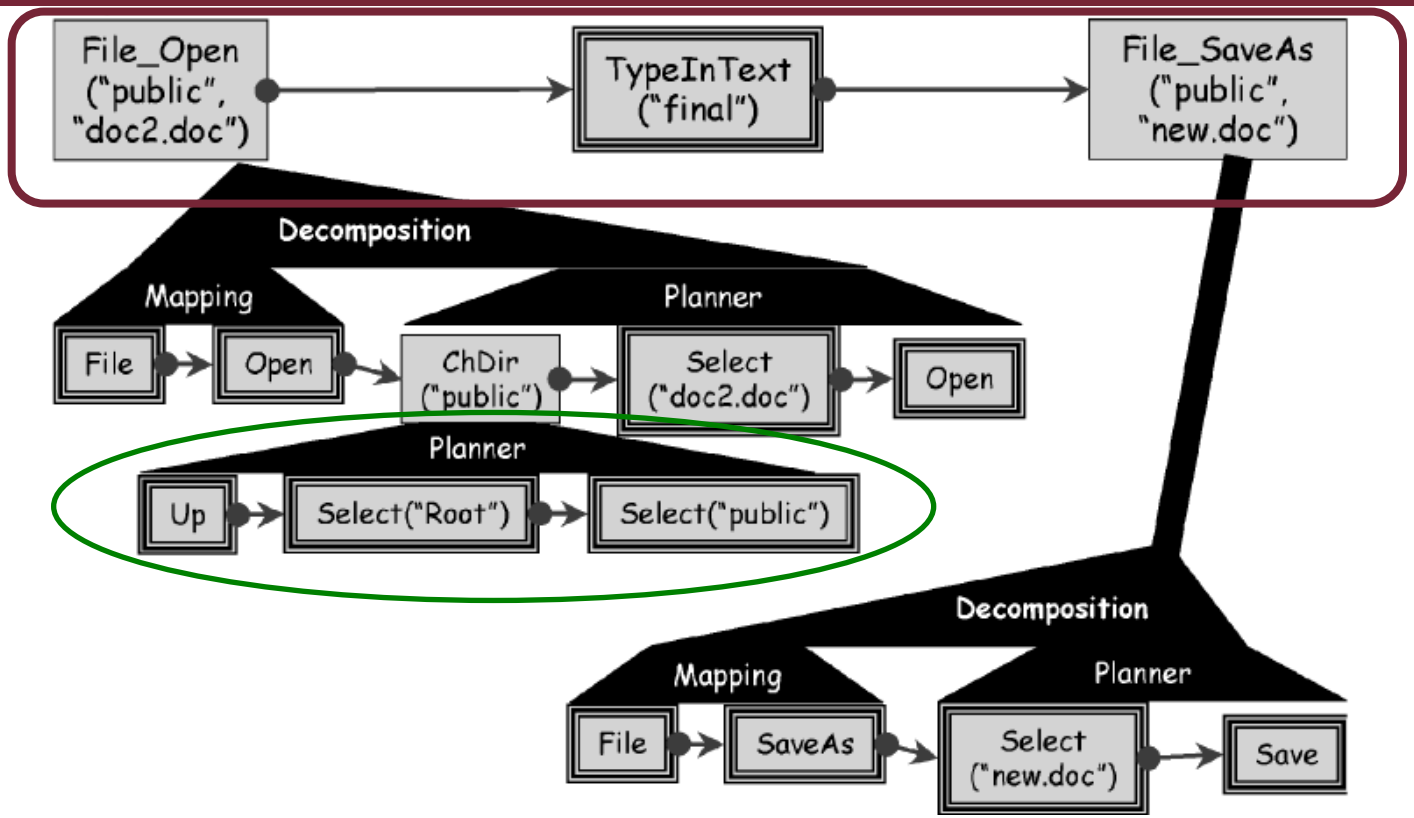
Resulting sequence:



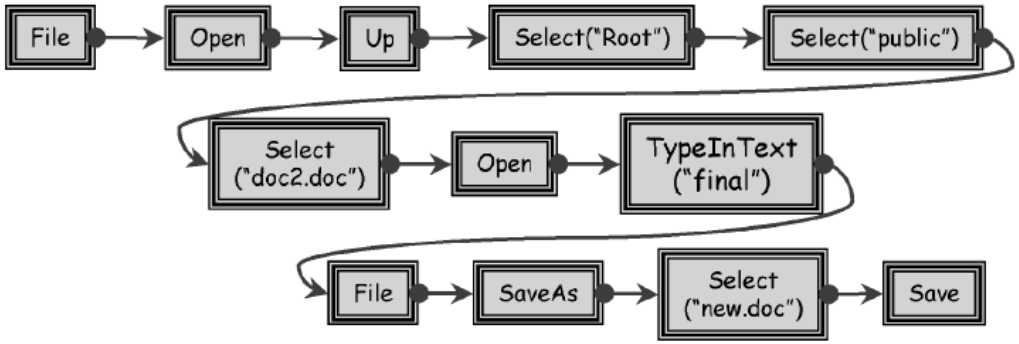
Application for GUI testing (5): Example (mod.)

High level plan:

Mapping with planner:



Resulting sequence:



Using evolutionary algorithms for test generation

- Evolutionary algorithms (e.g., genetic algorithms)
  - Modifying an initial test suite generated by random walk
  - Modifications: mutation of a test input sequence, merging test input sequences (parts of sequences)
  - Test suite that has better properties w.r.t. given test criteria is kept for further modifications
- Test criteria:
  - Control flow based criteria: Coverage of states, branches, conditions, paths, …
  - Data flow based criteria: All-defs, all-uses coverage
  - Test length, execution time, …
- Example tool:
  - Java: DOTgEAr

Recap: Using symbolic execution

- Static program analysis technique: Here applied for models (e.g., CFA)
  - Following computation of paths with symbolic variables
  - Deriving reachability conditions as constraints
  - Constraint solving (e.g., SMT solver): Yields inputs to execute a given path
- Test input generation: Inputs for covering the paths of the program
  - Generation of expected outputs by path simulation
  - Checking generic correctness criteria (lack of exceptions, no crash)
- Challenges
  - Loops (with a large number of paths)
  - Complex types and operations (to be supported by the SMT solver)
  - Handling dependencies and libraries
- Example tools
  - Java: Symbolic PathFinder (based on Java PathFinder)
  - .Net: PEX, IntelliTest
  - C: KLEE, CUTE, EXE

Examples for automated test generation tools (1)

- Test generation with model checkers
  - FSHELL: For C programs
    - CBMC (bounded model checker) generates a counterexample to be used as test sequence for a specified test goal
  - BLAST:
    - Counterexample generated: Abstract test case for a test goal
    - Includes symbolic execution (for CEGAR): Generated test data
  - UPPAAL CoVer, TRON:
    - Modeling time-dependent behavior by timed automata
    - Counterexamples for non-coverage are generated by the UPPAAL model checker
    - Conformance relation for testing:
      Relativized timed input-output conformance (RTIOCO) – consistent with IOCO in untimed models

Examples for automated test generation tools (2)

- Testing based on abstract data types
  - Test inputs are generated on the basis of the axioms given in the abstract data type
    - Equivalence partitions, boundary values can be used
- Tools s
  - C
  - AC
    -
  - Aut
  - STG
  - TDE
  - T-V

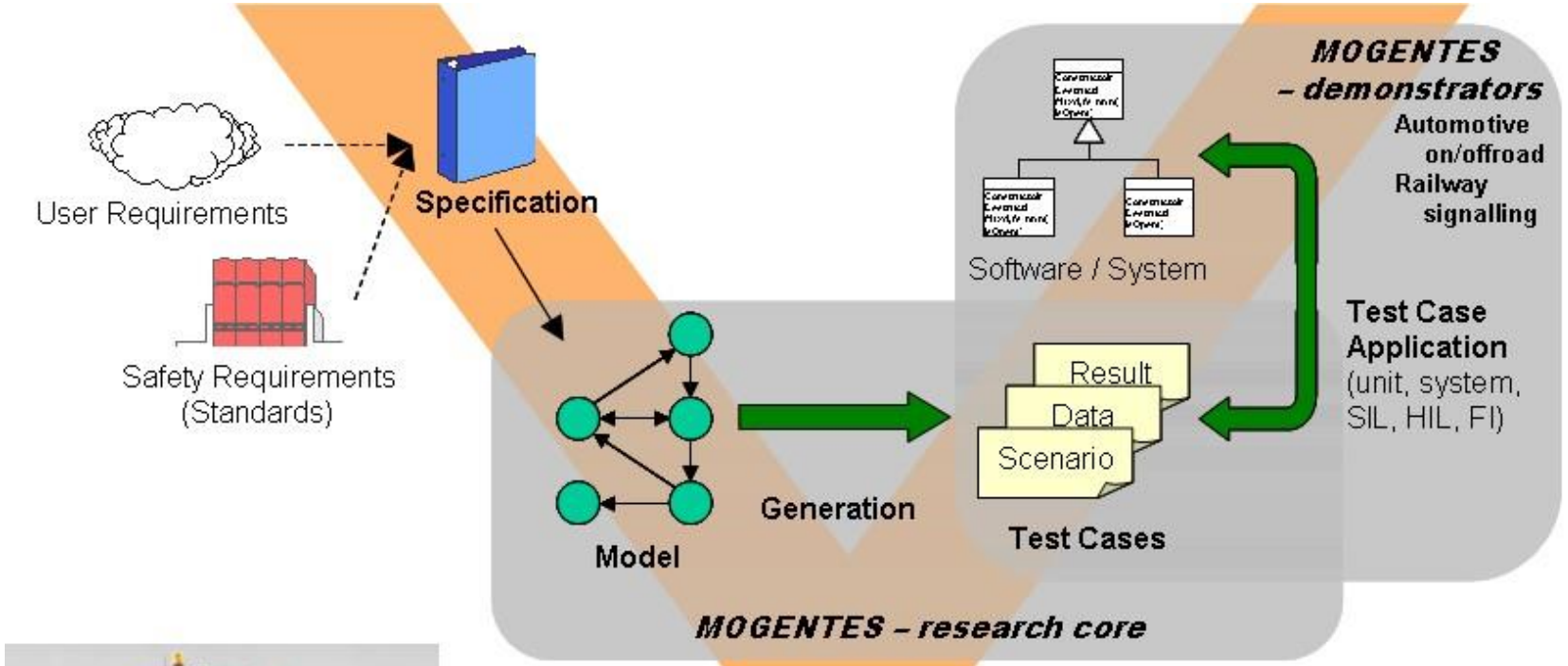
Abstract data types: Operations with axioms

```
Type Boolean is
    sorts Bool
    opns
        false, true : -> Bool
        not : Bool -> Bool
        and : Bool, Bool -> Bool
    eqns
        forall x, y: Bool
        ofsort Bool
            not(true) = false;
            not(false) = true;
            x and true = x;
```

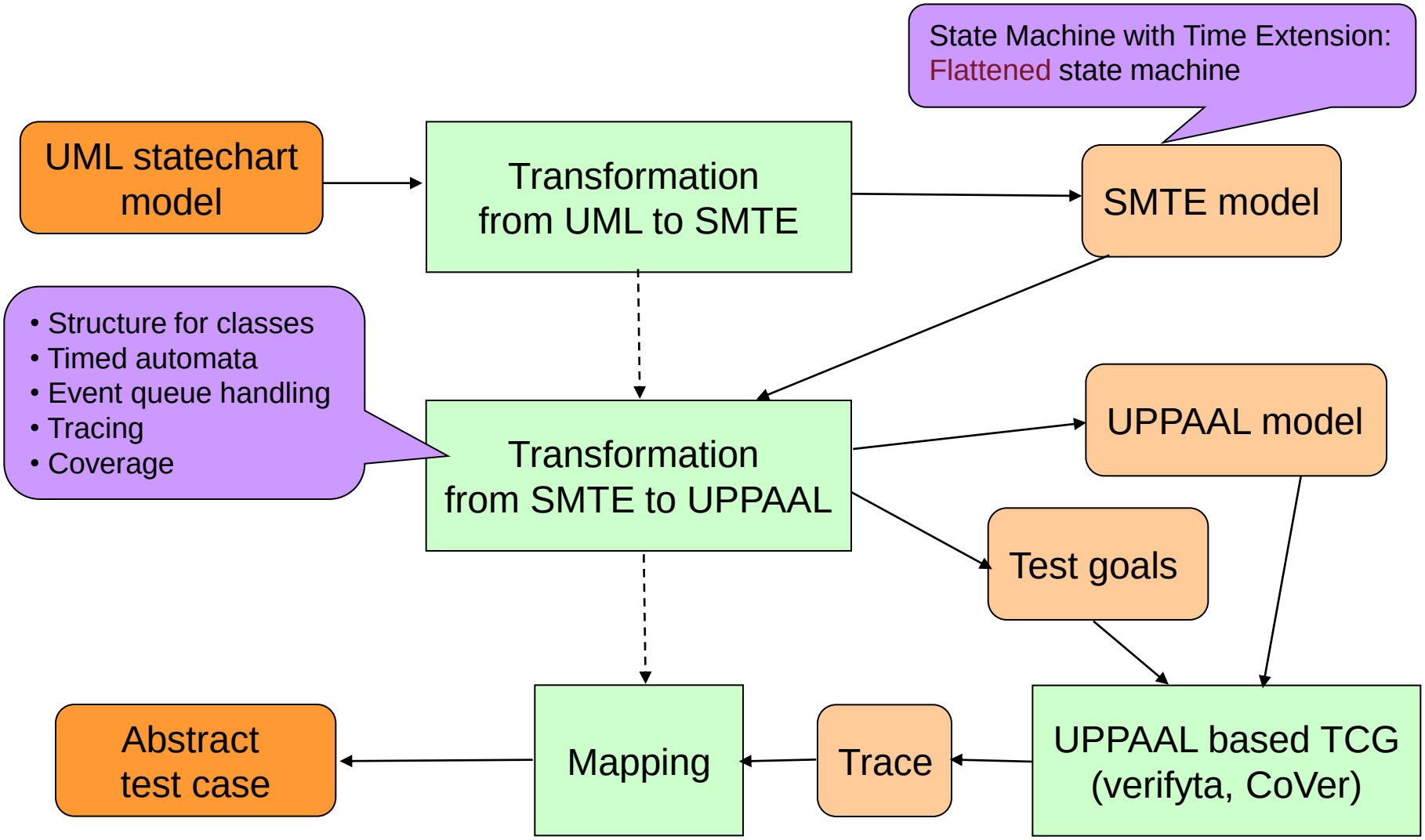
Examples for automated test generation tools (2)

- Testing based on abstract data types
  - Test inputs are generated on the basis of the axioms given in the abstract data type
    - Equivalence partitions, boundary values can be used
- Tools supporting specific modeling languages
  - Conformiq: For UML (statechart) models
  - AGATHA: UML, SDL, STATEMATE models
    - Generating path conditions and constraint solving to get test inputs
  - Autolink: SDL and MSC models are supported
  - STG: For LOTOS language
  - TDE/UML: Coverage criteria and constraints can be specified
  - T-Vec, DesignVerifier, Reactis, AutoFocus: For Simulink models

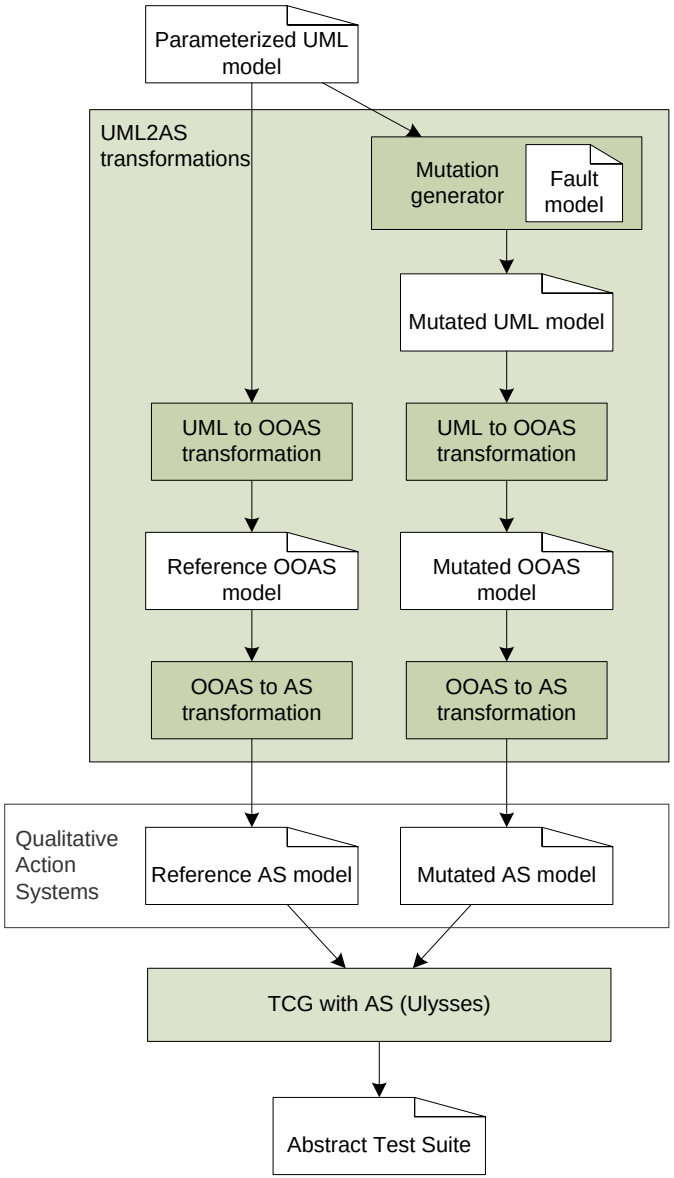
Example: MOGENTES project for model based test generation



Example: Model checking based TCG in MOGENTES



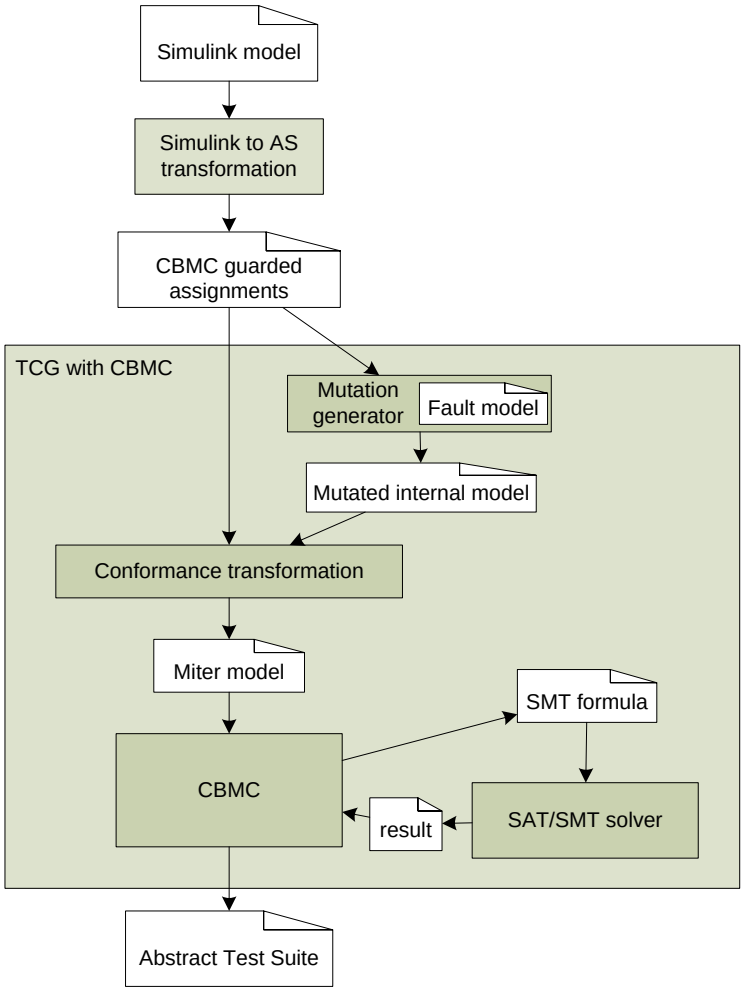
Example: Mutation based TCG in MOGENTES (1)



UML based TCG:

- Generation of mutations:
  - Applied on UML models
  - Based on a fault library defined on models
- Internal formalism:
  - OO Action System
  - Guarded action language
- Test generation:
  - Detect mutation based on IOCO relation
  - SMT solver is used
  - Bounded length of the test cases can be specified

Example: Mutation based TCG in MOGENTES (2)



Simulink based TCG:

- Generation of mutations:
  - Applied on C programs generated from Simulink models (with automated code generator)
  - Based on a fault library defined on source code
- Bounded model checking:
  - CBMC for C programs
- Test generation:
  - Detect mutation based on IOCO relation

Summary

- Model based test case generation
  - On the basis of coverage criteria
    - Control flow oriented: states, transitions coverage
    - Data flow oriented: def-use coverage
  - On the basis of mutations
    - Using conformance relations (k-equivalence, IOCO) for distinguishing original and mutated behavior
- Algorithms and tools
  - Direct (graph-based) algorithms
  - Model checkers: Counterexample as test case
  - Planner algorithms: Goal-oriented action sequence
  - Evolutionary algorithms: Optimizing (random) test suite
  - Symbolic execution: Path coverage
  - Test for (abstract) data types: On the basis of operators' axioms