Model based testing (part 2)

Istvan Majzik
majzik@mit.bme.hu

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

Behavioral relations and testing

Influence of model refinement on testing

Conformance of specified and observed behavior

Recap: Classification of relations

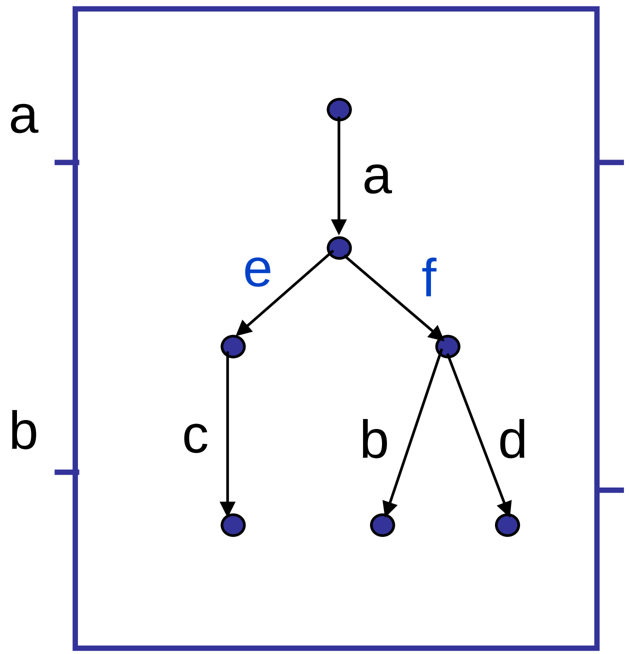
- Equivalence relations, denoted in general by =
  - Reflexive, transitive, symmetric

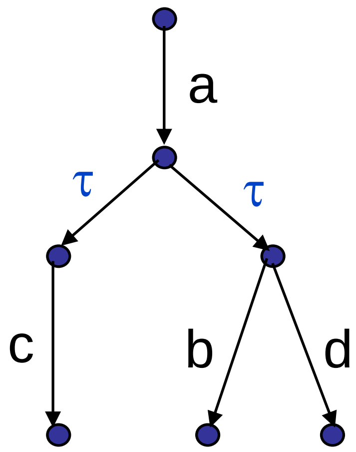
  Some equivalence relations are congruence:
  - If T1=T2, then for all C[ ] context C[T1]=C[T2]
  - The same context preserves the equivalence
  - Dependent on the formalism: how to embed in C[ ]

- Refinement relations, denoted by ≤
  - Reflexive, transitive, anti-symmetric (→ partial order)

  Precongruence relation:
  - If T1≤T2, then for all C[ ] context C[T1] ≤ 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:

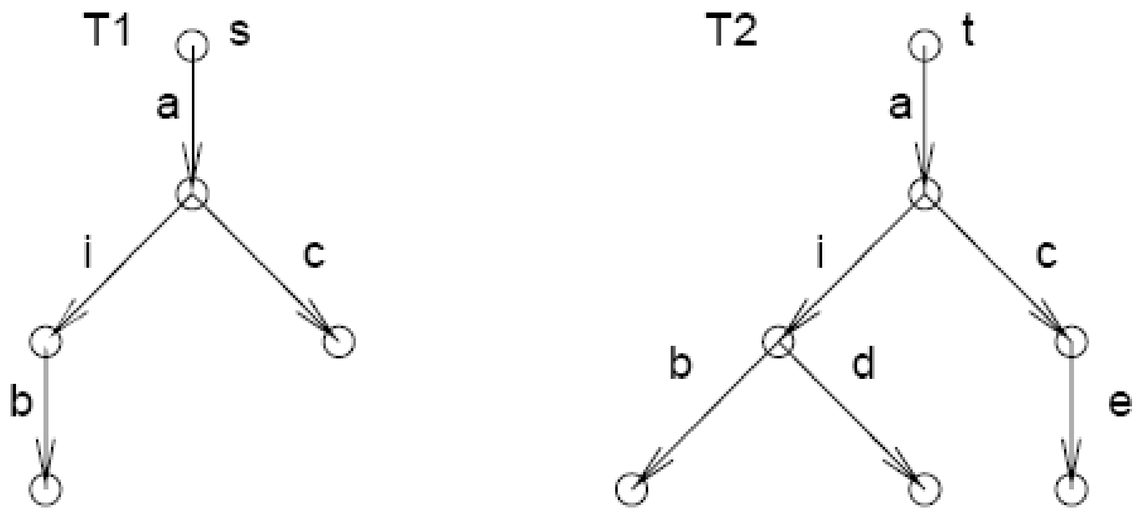
  For T_1 and T_2 LTSs with initial states s_1 and s_2, *Act* actions:

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

  T_2 refines T_1 as 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 $T_1 \leq_\Delta T_2$ (i.e., T_2 refines T_1):
  - 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
  - Idea: Collect failures $\rightarrow$ determine tests that never fail

Must preorder: Notation for failures

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

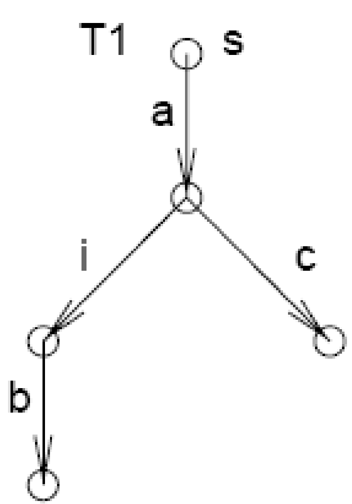
either $s \Uparrow \beta$

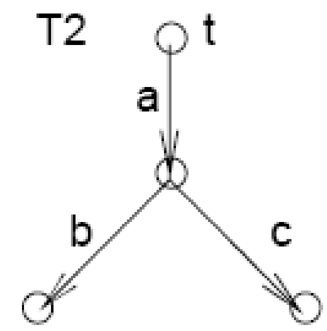
or $\exists s'$: $s \overset{\beta}{\Rightarrow} s'$ and s' refuses E

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

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

Example: Failure due to refused action





Here <a,{c}> is a failure

<a,{c}> $\in$ F(s)

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

However, <a,{c}> would be a failure if there is a τ self-loop in the second state (i.e., it is divergent)

Must preorder: Definition

Definition of must preorder: Covering failures

$$T_1 \leq_F T_2 \text{ iff } \mathrm{F}(s_1) \supseteq F(s_2)$$

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

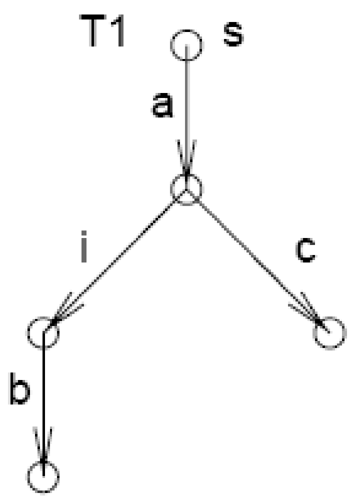
The role of failures

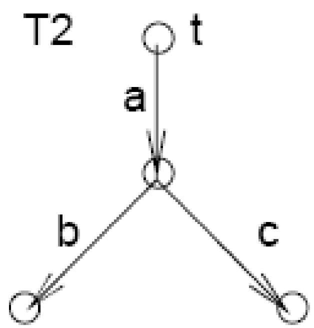
- Failures: Refusing actions directly of because of divergence
- Less failures: Less possible refusals, more successful actions (action sequences)
- If the behavior is extended by adding more actions then the number of failures will decrease (actions become possible)
- If nondeterminism is reduced then the number of failures may decrease (if failure is due to nondeterminism)

Must preorder: Relationship with testing

- In case of $T_1 \leq_F T_2$ (i.e., T_2 refines T_1):
  - T_2 has less failures, cannot refuse more actions (tests)
  - Tests that are always successful in T_1 are included in the tests that are always successful in T_2
    - The refinement preserves the tests that are always successful
    - T_2 has at least as many successful tests as T_1
- 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





Here <a,{c}> is a failure

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

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

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

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
      $\rightarrow$ 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

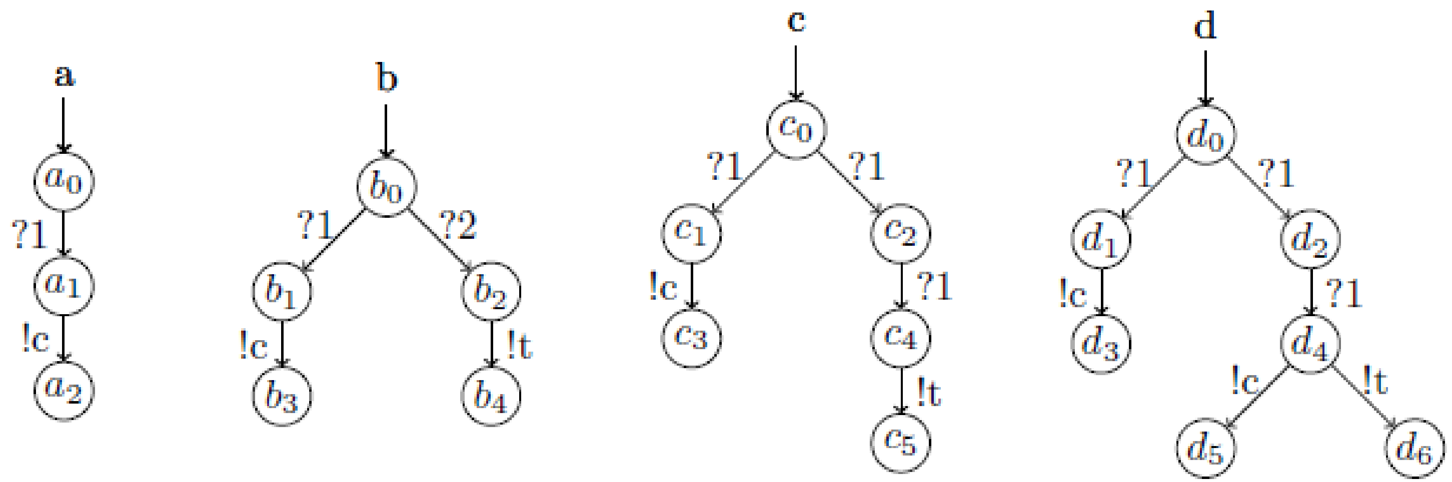
Act_{in} input, Act_{out} output actions, τ internal action

- 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 processed 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

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

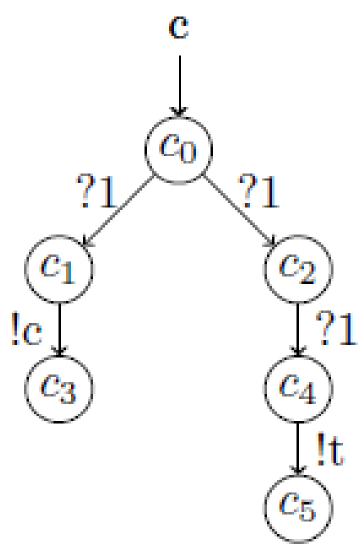
- Handling quiescent states in a uniform way:
  - The quiescent states (i.e., waiting for input) are denoted by a loop transition labelled with a specific δ output action
    - This way we get an extended IOLTS T_{δ}
  - In this IOLTS quiescence is considered as output δ

Example: IOLTS extended with quiescence

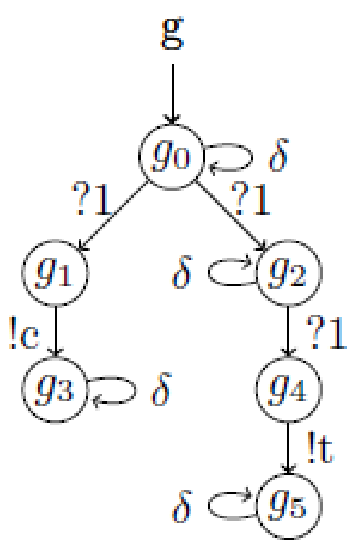
Coffee automaton IOLTS:

- 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:

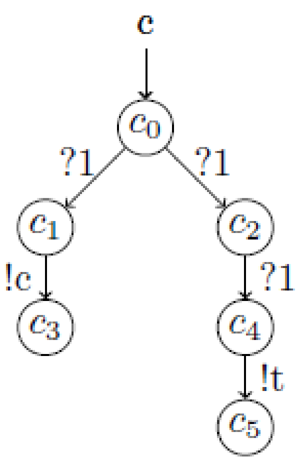


Example: IOLTS made complete

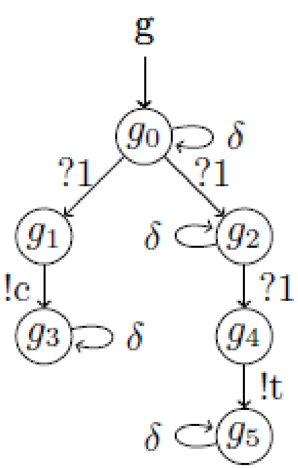
Coffee automaton IOLTS:

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

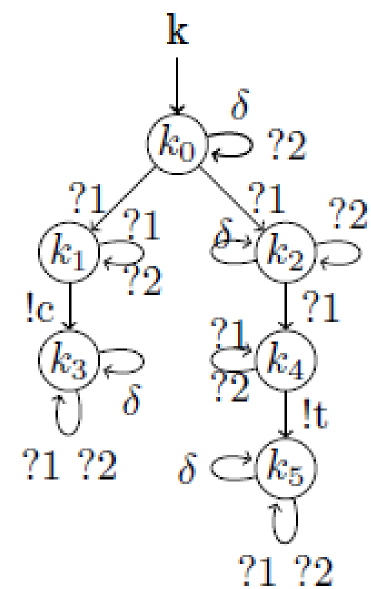
Loop transitions for actions that were missing:



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

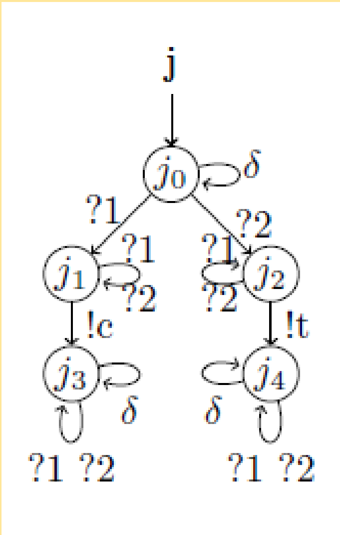
Observable output actions in the implementation after β

Observable output actions in the specification after β

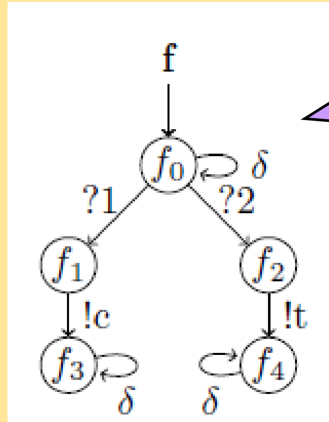
Properties of IOCO

- Explaining the definition:
  - Def.: For all observable action sequences in the specification: In each state that is reachable by the action sequence, the output actions provided by implementation M form a subset of the output actions given in specification T
  - This way the specification shall "cover" the implementation
  - The implementation shall "fit" into the frame given by the specification
- What are allowed?
  - Restricted behavior: The implementation may contain less potential output actions 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



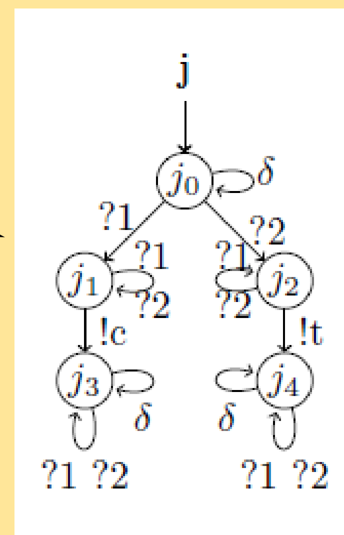
ioco



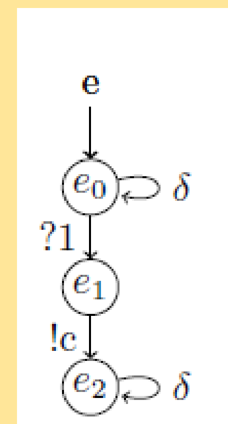
specification

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



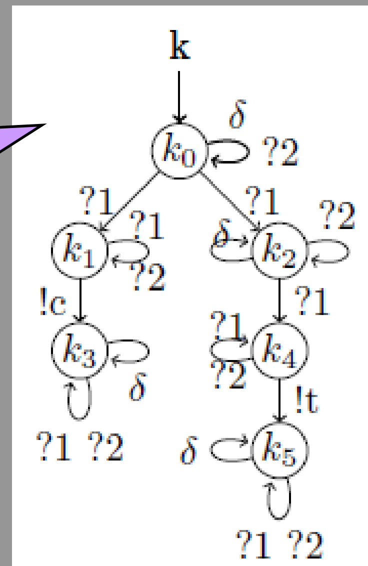
ioco



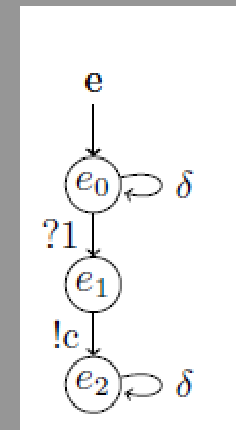
specification

Examples for violating IOCO

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



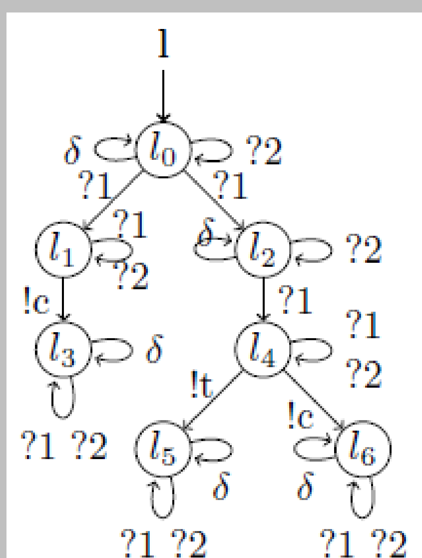
not ioco



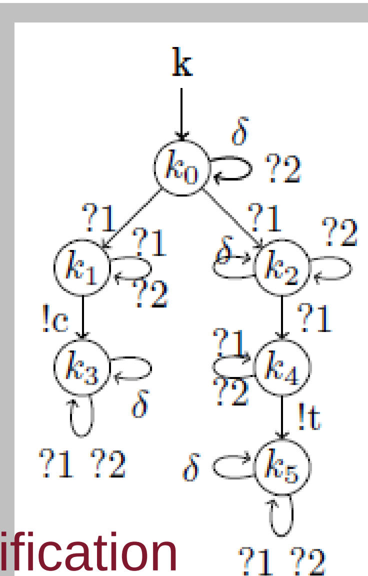
specification

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: $T \approx_{\Lambda} T' \text{ iff } \Lambda(s)=\Lambda(s')$
  - Strong bisimulation: $T \sim T' \text{ iff } \exists B : (s, s') \in B$
  - Observation equivalence: $T \approx T' \text{ iff } \exists WB : (s, s') \in WB$
- Preorders: For model refinement and testing
  - May preorder: $T \leq_{\Delta} T' \text{ iff } \Delta(s) \subseteq \Delta(s')$
  - Must preorder: $T \leq_{F} T' \text{ iff } F(s) \supseteq F(s')$
- 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 the model
  - Goal state: State to be reached (covered)
  - Actions: Activities 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

Using evolutionary algorithms for test generation

- Evolutionary algorithms (e.g., genetic algorithms)
  - Having an initial test suite generated by random walk
  - Modifications:
    - Mutating a test input sequence (removing, adding, reordering elements)
    - Merging parts of test input 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 suite length, execution time, …
- Example tool:
  - DOTgEAr

Generating tests for abstract data types

- Abstract data types: Define operations and axioms
- Abstract test inputs to test operations are generated on the basis of the axioms
  - Equivalence partitions, boundary values can be used

```
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 (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)

- 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

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
  - Test for (abstract) data types: On the basis of operators' axioms