Model-driven Test Generation

Oleg Sokolsky

September 22, 2004

Outline and scope

- Classification of model-driven testing
- Conformance testing for communication protocols
- Coverage-based testing
  - Coverage criteria
  - Coverage-based test generation
- Can we do more? (open questions)

Testing classification

- By component level
  - Unit testing
  - Integration testing
  - System testing
- By abstraction level
  - Black box
  - White box
  - Grey box ???

Testing classification

- By purpose
  - Functional testing
  - Performance testing
  - Robustness testing
  - Stress testing
- Who performs testing?
  - Developers
  - In-house QA
  - Third-party

Functional testing

- An implementation can exhibit a variety of behaviors
- For each behavior, we can tell whether it is correct or not
- A *test* can be applied to the implementation and accept or reject one or more behaviors
  - The test fails if a behavior is rejected
- A *test suite* is a finite collection of tests
  - Testing fails if any test in the suite fails

Formal methods in testing

- "Testing can never demonstrate the absence of errors, only their presence."
  Edsger W. Dijkstra

- How can formal methods help?
- Add rigor!
  - Reliably identify what should to be tested
  - Provide basis for test generation
  - Provide basis for test execution

Model-driven testing

- Rely on a model of the system
  - Different interpretations of a model
- Model is a requirement
  - Black-box conformance testing
  - QA or third party
- Model is a design artifact
  - Grey-box unit/system testing
  - QA or developers

Conformance testing

- A specification prescribes legal behaviors
- Does the implementation conform to the specification?
  - Need the notion of conformance
- Not interested in:
  - How the system is implemented?
  - What went wrong if an error is found?
  - What else the system can do?

Test hypothesis

- How do we relate beasts of different species?
  - Implementation is a physical object
  - Specification is a formal object
- Assume there is a formal model that is faithful to implementation
  - We do not know it!
- Define conformance between the model and the specification
  - Generate tests to demonstrate conformance

Conformance testing with LTS

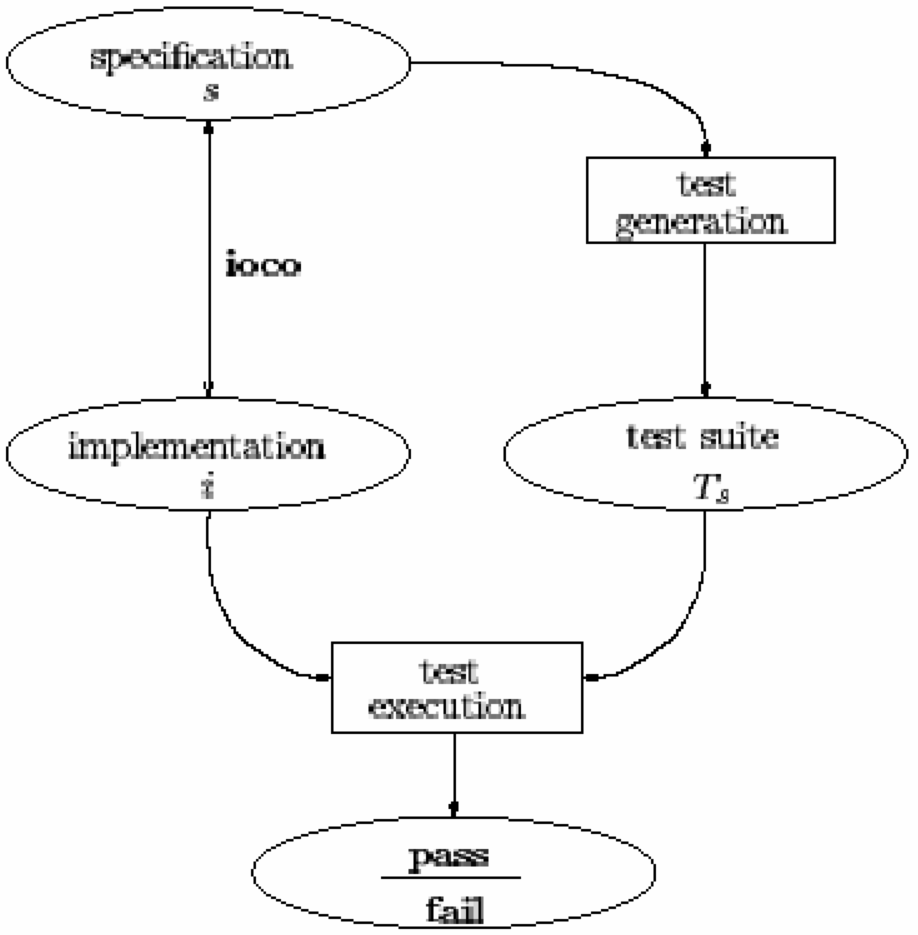
- Requirement is specified as a labeled transition system
- Implementation is modeled as an input-output transition system
- Conformance relation is given by **`ioco`**
  - [Tretmans96]
  - Built upon earlier work on testing preorders

Historical reference

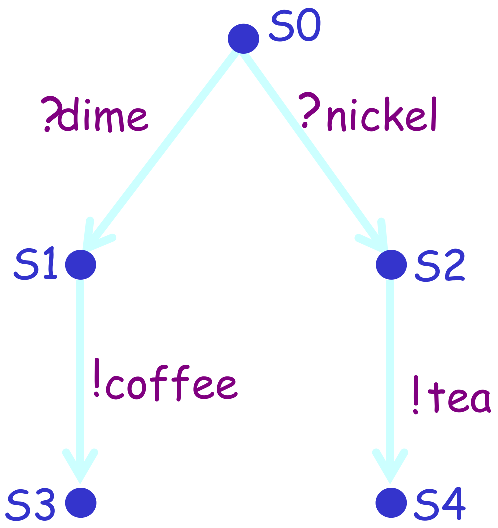
- Process equivalences:
  - Trace equivalence/preorder is too coarse
  - Bisimulation/simulation is too fine
- Middle ground:
  - Failures equivalence in CSP
  - may- and must-testing by Hennessy
  - Testing preorder by de Nicola
  - They are all the same!
- Right notion but hard to compute

Testing architecture

- Implementation relation
- Test generation algorithm
- Test execution engine



Input-Output Transition Systems

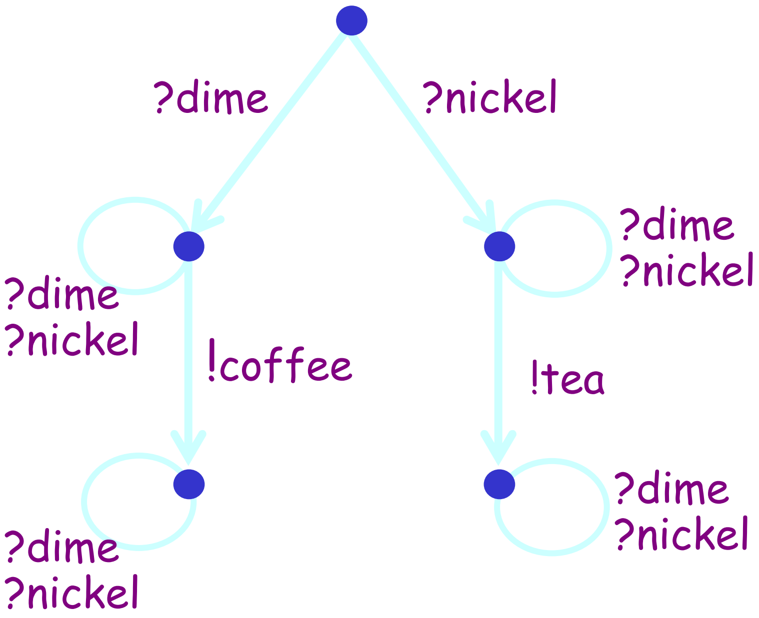


dime, nickel	coffee, tea
from user to machine initiative with user machine cannot refuse	from machine to user initiative with machine user cannot refuse
input L_I	output L_U
$L_I \cap L_U = \emptyset$	$L_I \cup L_U = L$

$L_I = \{ ?dime, ?nickel \}$

$L_U = \{ !coffee, !tea \}$

Input-Output Transition Systems



L_I = { ?dime, ?nickel }

L_U = { !coffee, !tea }

Input-Output Transition Systems

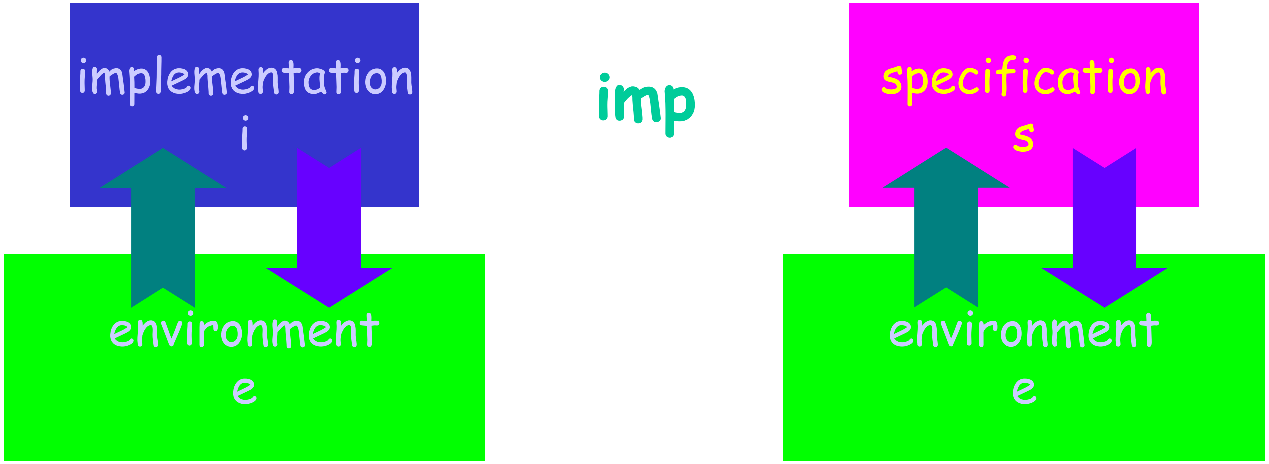
$\mathrm{IOTS}\,(L_I, L_U) \subseteq \mathrm{LTS}\,(L_I \cup L_U)$

IOTS is LTS with Input-Output and always enabled inputs:

for all states s,

for all inputs $?a \in L_I$: $s \xRightarrow{?a}$

Preorders on IOTS



implementation

i

environment

e

imp

specification

s

environment

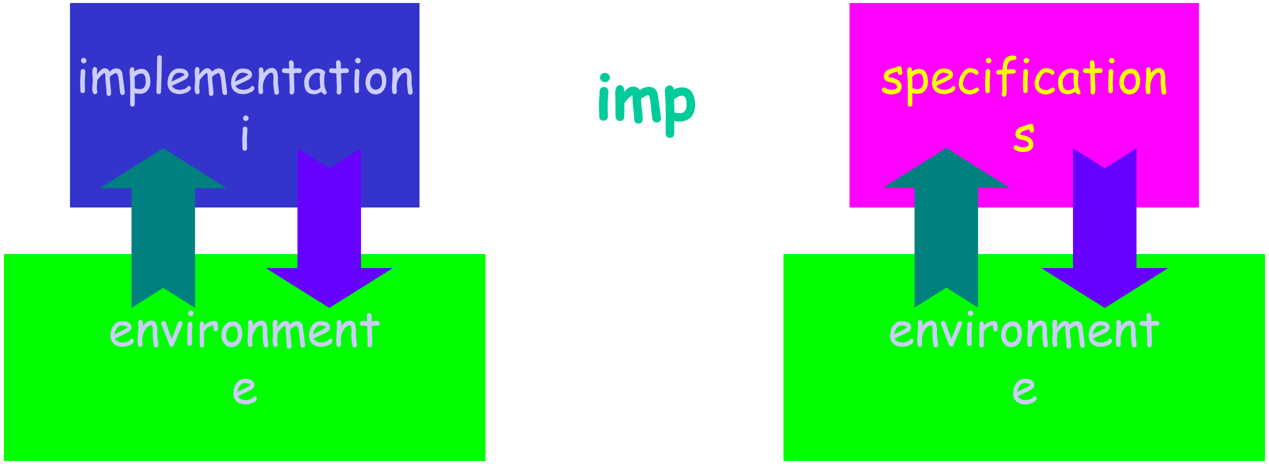
e

$i \in IOTS(L_I, L_U)$ $s \in LTS(L_I \cup L_U)$

$\mathbf{imp} \subseteq IOTS\,(L_I, L_U) \times LTS\,(L_I \cup L_U)$

Observing IOTS where system inputs
interact with environment outputs, and vice versa

Preorders on IOTS



implementation

i

imp

specification

s

environment

e

environment

e

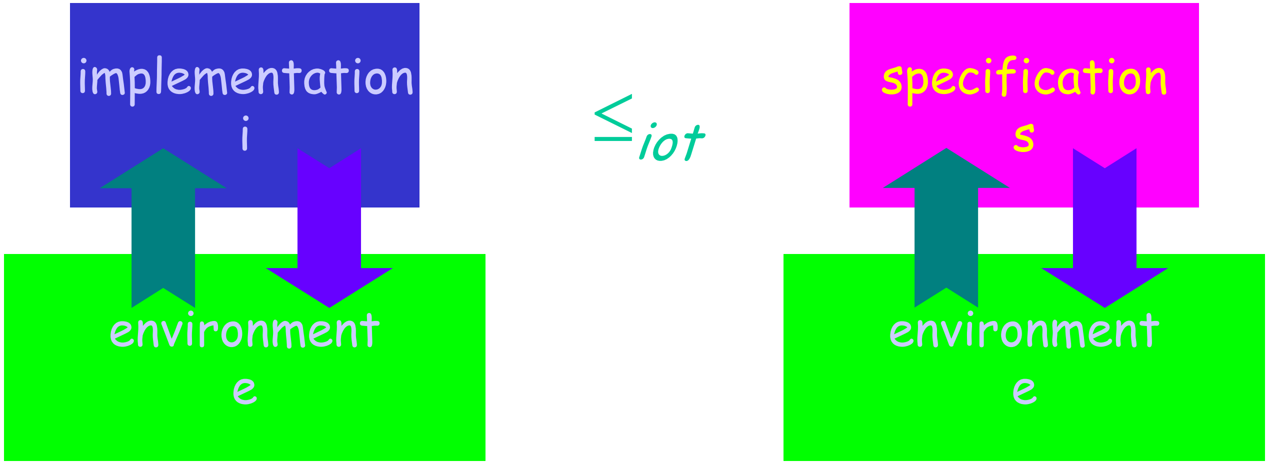
$i \in IOTS(L_I, L_U)$ $s \in LTS(L_I \cup L_U)$

$$i \ \mathbf{imp} \ s \iff \forall\, e \in E \ . \quad obs(e, i) \subseteq obs(e, s)$$

↓

$IOTS(L_U, L_I)$

Input-Output Testing Relation



$i \in \mathrm{IOTS}(L_I, L_U)$ $\quad$ $s \in \mathrm{LTS}(L_I \cup L_U)$

$$i \leq_{iot} s \Leftrightarrow \forall\ e \in \mathrm{IOTS}(L_U, L_I)\ .\mathrm{obs}(e, i) \subseteq \mathrm{obs}(e, s)$$

$$\mathrm{obs}(e, p) = (\ \mathrm{traces}(e||p),\ \mathrm{qtraces}(e||p)\ )$$

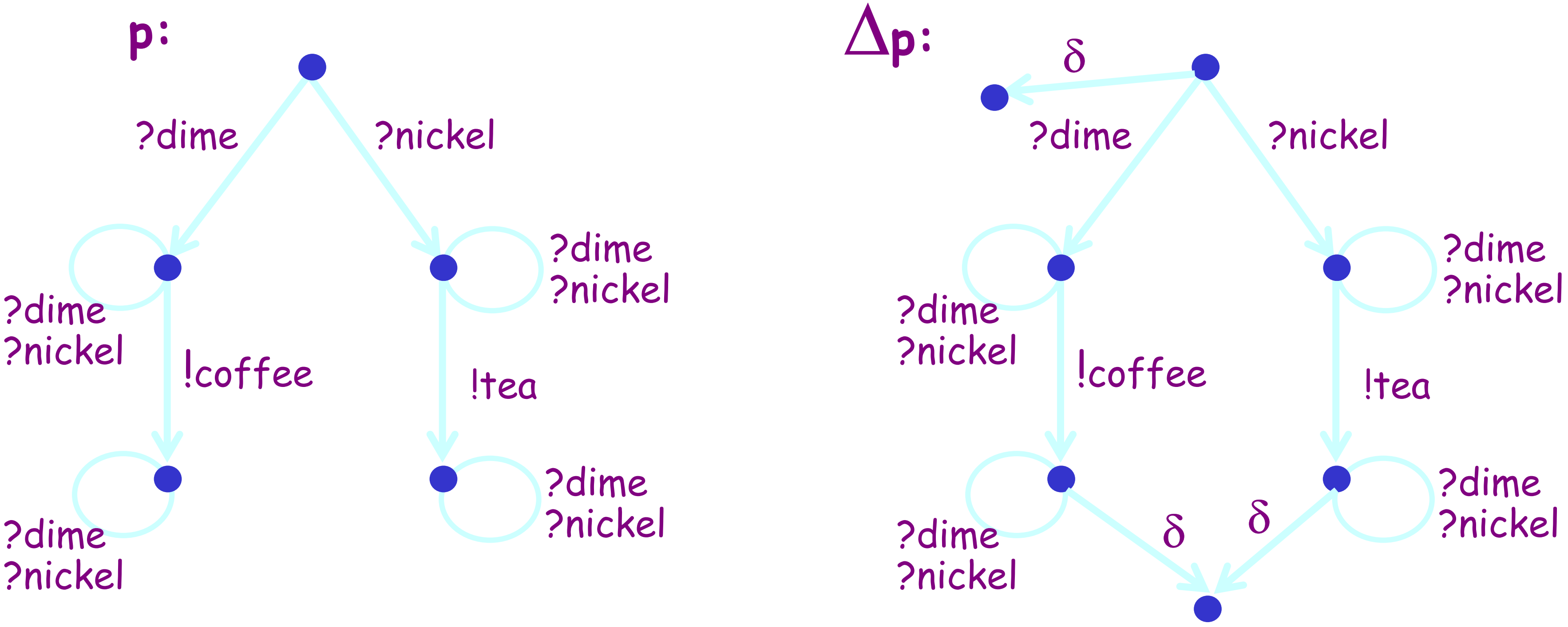
$$\mathrm{qtraces}(p) = \sigma \in L^*.\ p\ \mathrm{after}\ \sigma\ \mathrm{refuses}\ L_U$$

Testing preorders – a side note

- One of the reasons for using IOTS over LTS is that $\leq_{iot}$ is computationally simpler than conventional testing preorder
  - Testing preorder requires us to compare sets of pairs (trace, refusal set)
  - At the same time $\leq_{iot}$ allows us to use inclusion of weakly quiescent traces:
    - inputs can never be refused by i, and outputs can never be refused by e
    - i after σ refuses $A \quad \Rightarrow \quad A = \emptyset$ or $A = L_U$

Representing quiescence

- Extend IOTS with quiescent transitions
  - deterministic δ-trace automata



Conformance relation **ioconf**

- Finally…

  $i \leq_{iot} s \Leftrightarrow \forall\ \sigma \in L^*.out(\ \Delta i \text{ after } \sigma\ ) \subseteq out(\ \Delta s \text{ after } \sigma\ )$
- Allow underspecification
  - restrict to traces of *s*

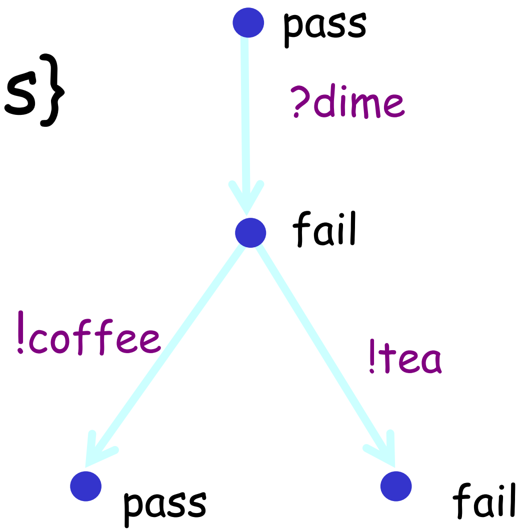
  i **ioconf** $s =_{def}$

  $\forall \sigma \in traces(\Delta s) \cap L^*.out(\Delta i \text{ after } \sigma) \subseteq out(\Delta s \text{ after } \sigma)$
- **ioconf**$_F$: use arbitrary *F* instead of traces of *s*
- Conformance relation **ioco** accounts for *repetitive* quiescence

Test cases

- A test case is a deterministic IOTS(L_U,L_I) with finite behaviors
  - Note reversed inputs and outputs
  - Do not allow choice between outputs or between input and output
- Verdict function ν: S → {fail,pass}
- Test run: i **passes** t $=_{def}$

  (i||t) after σ deadlocks $\Rightarrow$
  ν(t after σ)=pass



Test generation

- Test suite T_s for a specification s is complete:
  i **ioconf** s iff $\forall t \in Ts$. i **passes** t
- Test suite T_s is sound if
  i **ioconf** s $\Rightarrow$ $\forall t \in Ts$. i **passes** t
- Complete test suites are usually infinite
  - Aim at generating sound test suites

Test generation algorithm

- $\mathrm{Gen}(\Delta_s, \boldsymbol{F})$
  - Choose non-deterministically:

1. $t = \mathrm{stop}$ and $\nu(t) = \mathrm{pass}$
2. $t = a\,.\,\mathrm{Gen}(\Delta'_S, \boldsymbol{F} \text{ after } a)$, with $\overset{a}{\Delta}_S \rightarrow \Delta'_S$

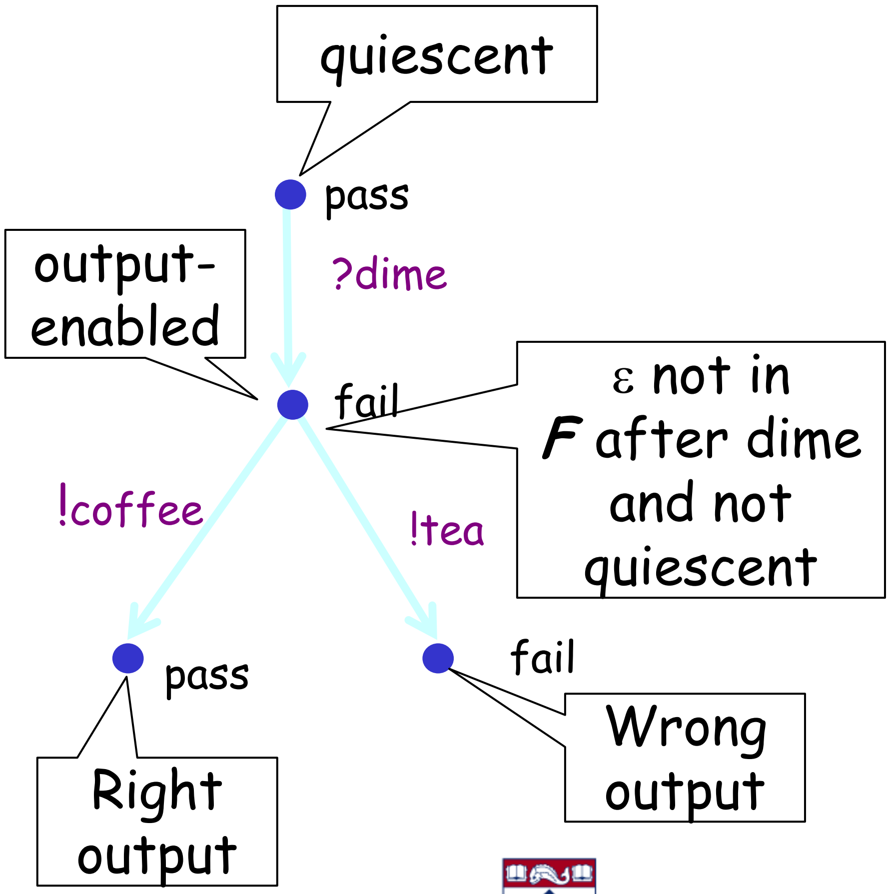
   $\nu(t) = \mathrm{pass}$
3. $t = \sum\{x.\mathrm{stop} \mid x \in L_U, x \notin out(\Delta_S)\} + \sum\{x.t_x \mid x \in out(\Delta_S)\}$

   $\nu(t) = \mathrm{pass}$ if $\delta \in out(\Delta_S) \vee \varepsilon \in F$; otherwise fail

   $\nu(\mathrm{stop}) = \mathrm{fail}$ if $\varepsilon \notin F$; otherwise pass

Example

- ***F*** = dime.coffee



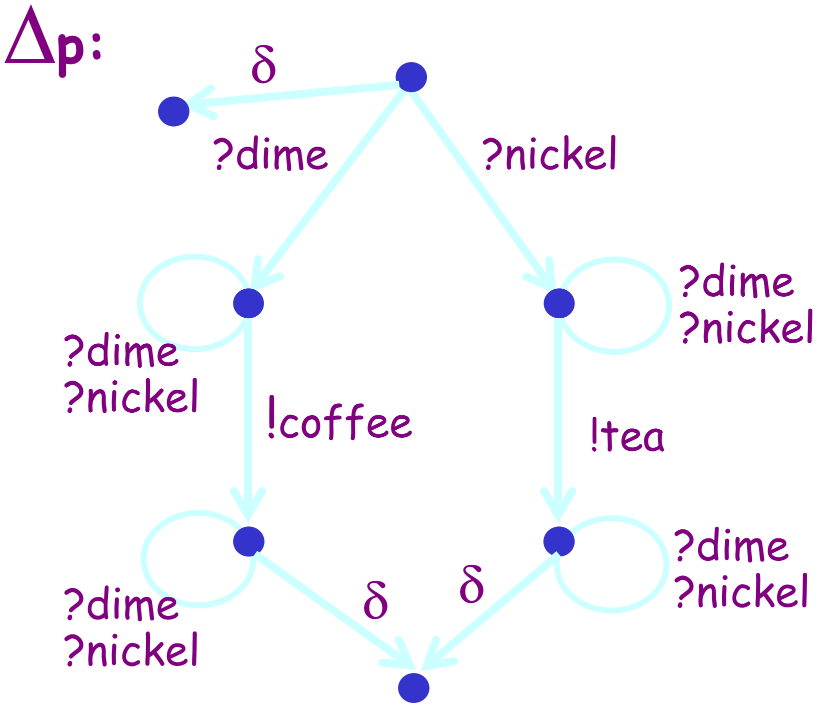
quiescent

output-enabled

ε not in ***F*** after dime and not quiescent

Right output

Wrong output



Test purposes

- Where does ***F*** come from?
- Test purposes:
  - Requirements, use cases
  - Automata, message sequence charts
- Test purposes represent “interesting” or “significant” behaviors
  - Define “interesting” or “significant”…
- Can we come up with test purposes automatically?

Summary: conformance testing

- Advantages:
  - Very rigorous formal foundation
  - Size of the test suite is controlled by use cases
- Disadvantages:
  - How much have we learned about the system that passed the test suite?
  - Does not guarantee coverage

Coverage-based testing

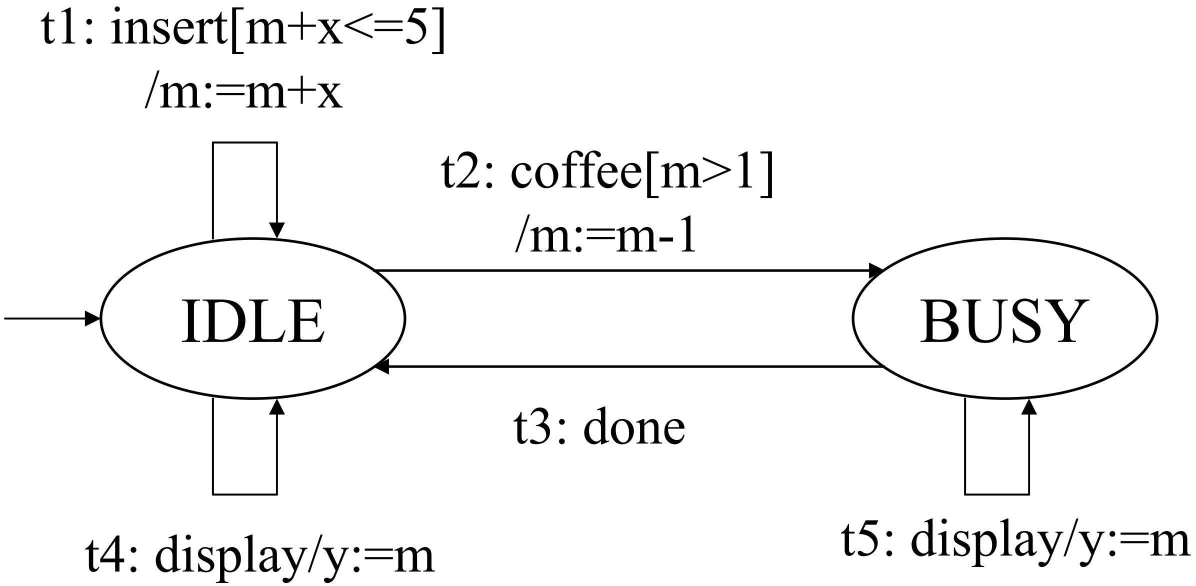
- Traditional:
  - Tests are derived from the implementation structure (code)
- Model-driven:
  - Cover the model instead of code
  - Model should be much closer to the implementation in structure
- Relies on coverage criteria

Coverage criteria and tests

- [HongLeeSokolskyUral02]
- Control flow:
  - all-states
  - all-transitions
- Data flow:
  - all-defs
  - all-uses
  - all-inputs
  - all-outputs
- Test is a linear sequence of inputs and outputs

Specifications: EFSM

- Transition systems equipped with variables
- Transitions have guards and update blocks

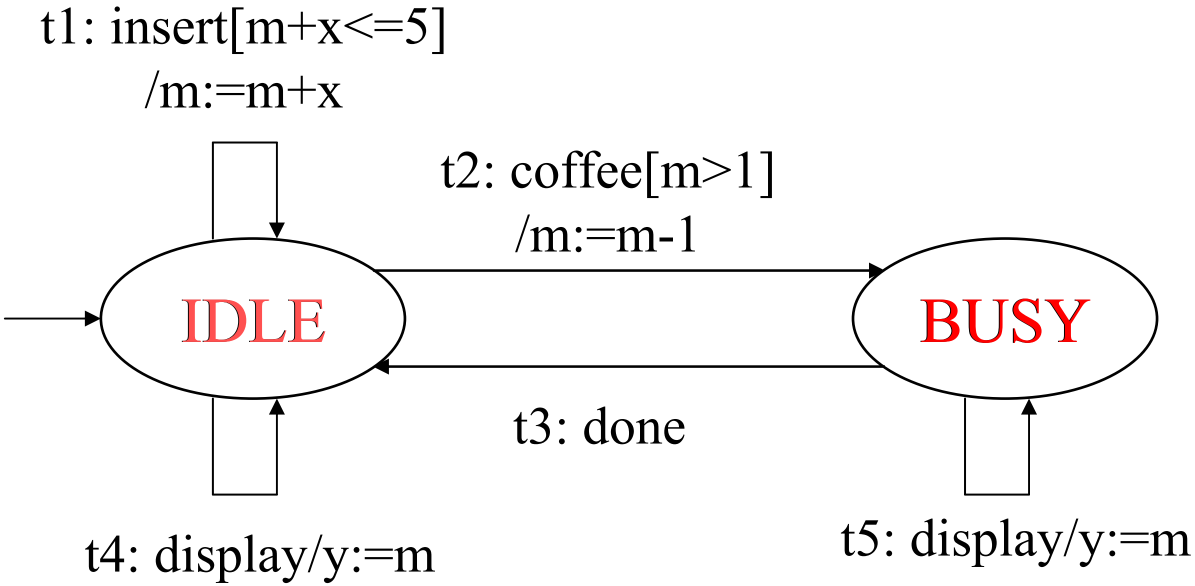


Coverage criteria

- Each coverage criterion is represented by a set of temporal logic formulas
  - WCTL: a subset of CTL
    - Atomic propositions $p_1, \ldots, p_n$
    - Temporal operators EX, EU, EF
    - Conjunctions: at most one non-atomic conjunct
    - Negations is applied only to atomic propositions
    - Unrestricted disjunctions
    - E.g.: EF(p_1 & EFp_2)
  - WCTL formulas have linear witnesses

All-states coverage criterion

- Requires every state be covered at least once
- With every state *s*, associate EF(*s* & EF*exit*)

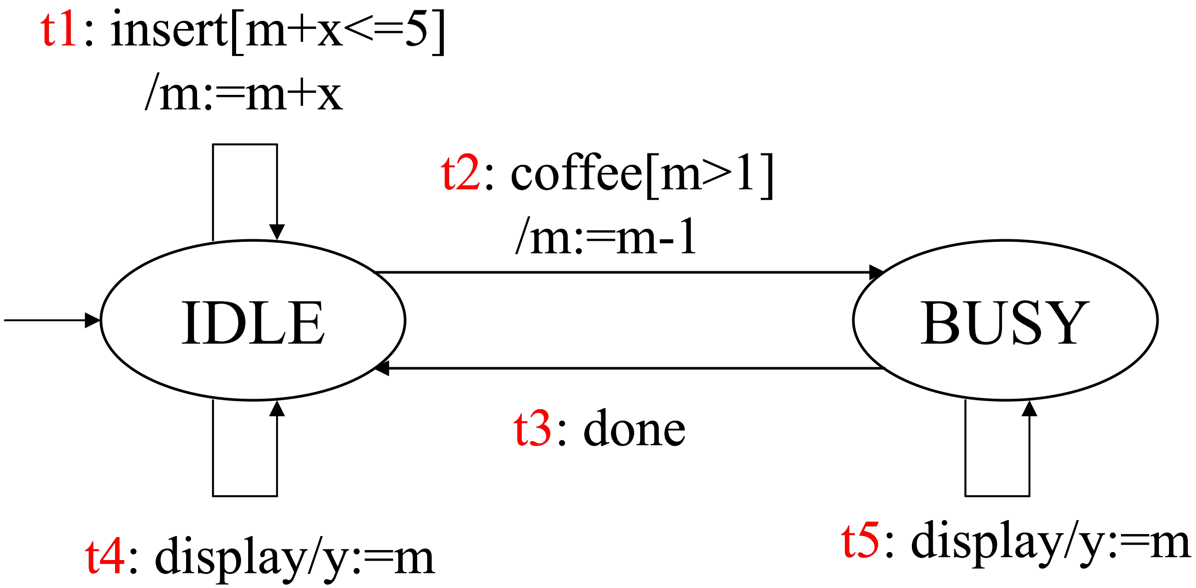


EF(*idle* & EF*exit*)

EF(*busy* & EF*exit*)

All-transitions coverage criterion

- Requires every transition be covered at least once
- With every transition *t*, associate EF(*t* & EF*exit*)



EF(*t1* & EF*exit*)

EF(*t2* & EF*exit*)

EF(*t3* & EF*exit*)

EF(*t4* & EF*exit*)

EF(*t5* & EF*exit*)

Data flow: definitions and uses

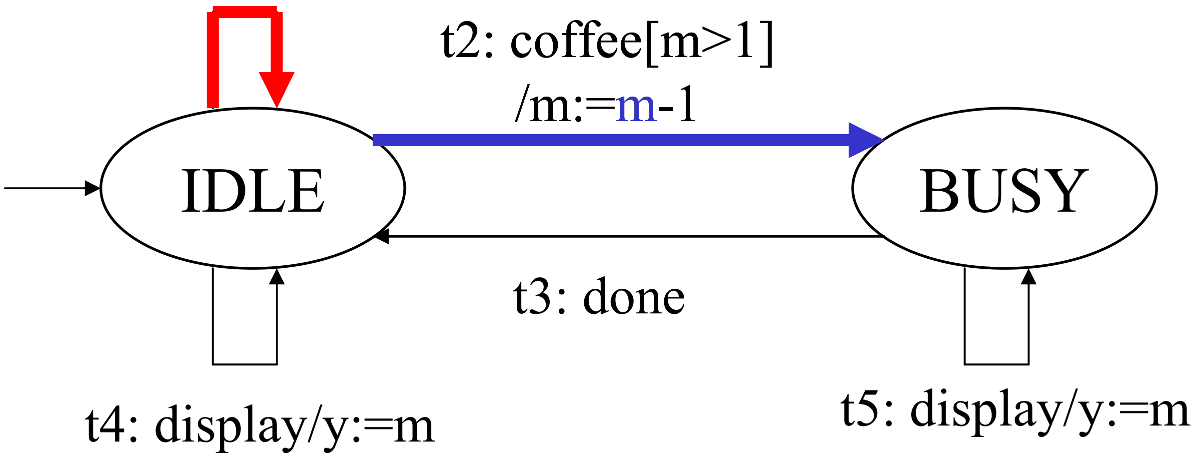
- Definition: a value is assigned to a variable
- Use: a value of a variable is used in an expression
- Variables are defined and used in transitions
- Definition-use pair: (v,t,t')
  - v is defined by t
  - v is used by t'
  - There is a path from t to t' free from other definitions of v

Covering a du-pair

- With a du-pair (*v*, *t*, *t'*), *associate*
  - EF(*t* & EXE[!def(*v*) U (*t'* & EF*exit*)])
  - def(*v*) : disjunction of all transitions that define *v*

EF(*t1* & EXE[!(*t1* | *t2*) U (*t2* & EF*exit)]*)

t1: insert[m+x<=5]
/m:=m+x



t2: coffee[m>1]
/m:=m-1

IDLE

BUSY

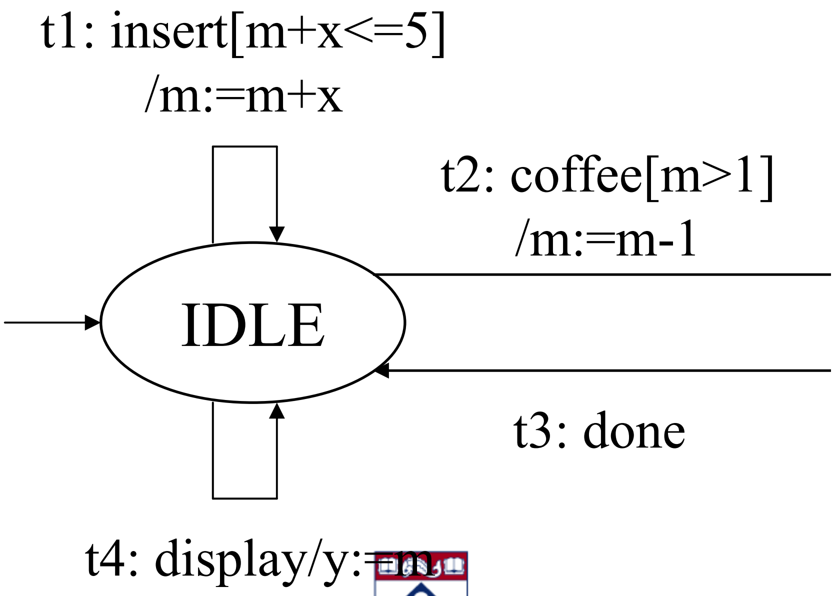
t3: done

t4: display/y:=m

t5: display/y:=m

Data-flow coverage criteria

- All-defs coverage criterion: a definition-clear path
  - from ***every*** definition to ***some*** use
- All-uses coverage criterion: a definition-clear path
  - from ***every*** definition to ***every*** use

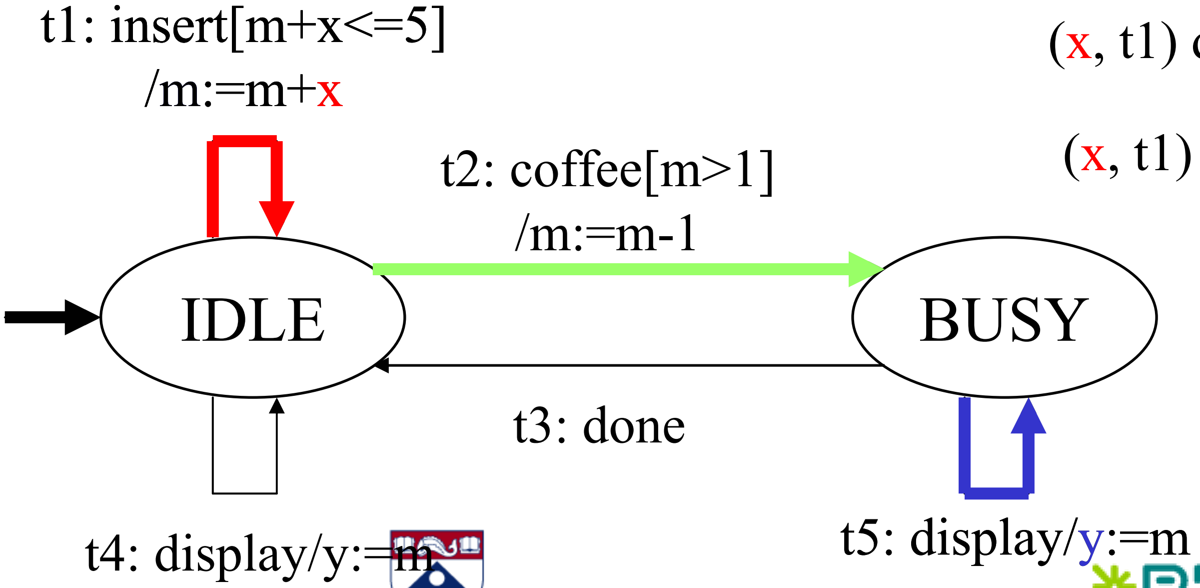


All-uses coverage criterion

EF(*t1* & EXE[!(*t1* | *t2*) U (*t1* & EF*exit*)])
EF(*t1* & EXE[!(*t1* | *t2*) U (*t2* & EF*exit*)])
EF(*t1* & EXE[!(*t1* | *t2*) U (*t4* & EF*exit*)])
EF(*t1* & EXE[!(*t1* | *t2*) U (*t5* & EF*exit*)])
EF(*t2* & EXE[!(*t1* | *t2*) U (*t1* & EF*exit*)])
EF(*t2* & EXE[!(*t1* | *t2*) U (*t2* & EF*exit*)])
EF(*t2* & EXE[!(*t1* | *t2*) U (*t4* & EF*exit*)])
EF(*t2* & EXE[!(*t1* | *t2*) U (*t5* & EF*exit*)])

Data flow chains

- Affect pair (v, t, v', t'): the value of v used by t affects the value of v' defined at t'
  - Either $t=t'$ ((v,t) directly affects (v',t')) or
  - there is a du-pair (v'',t,t'') s.t. (v,t) directly affects (v'',t) and (v'',t'') affects (v',t')



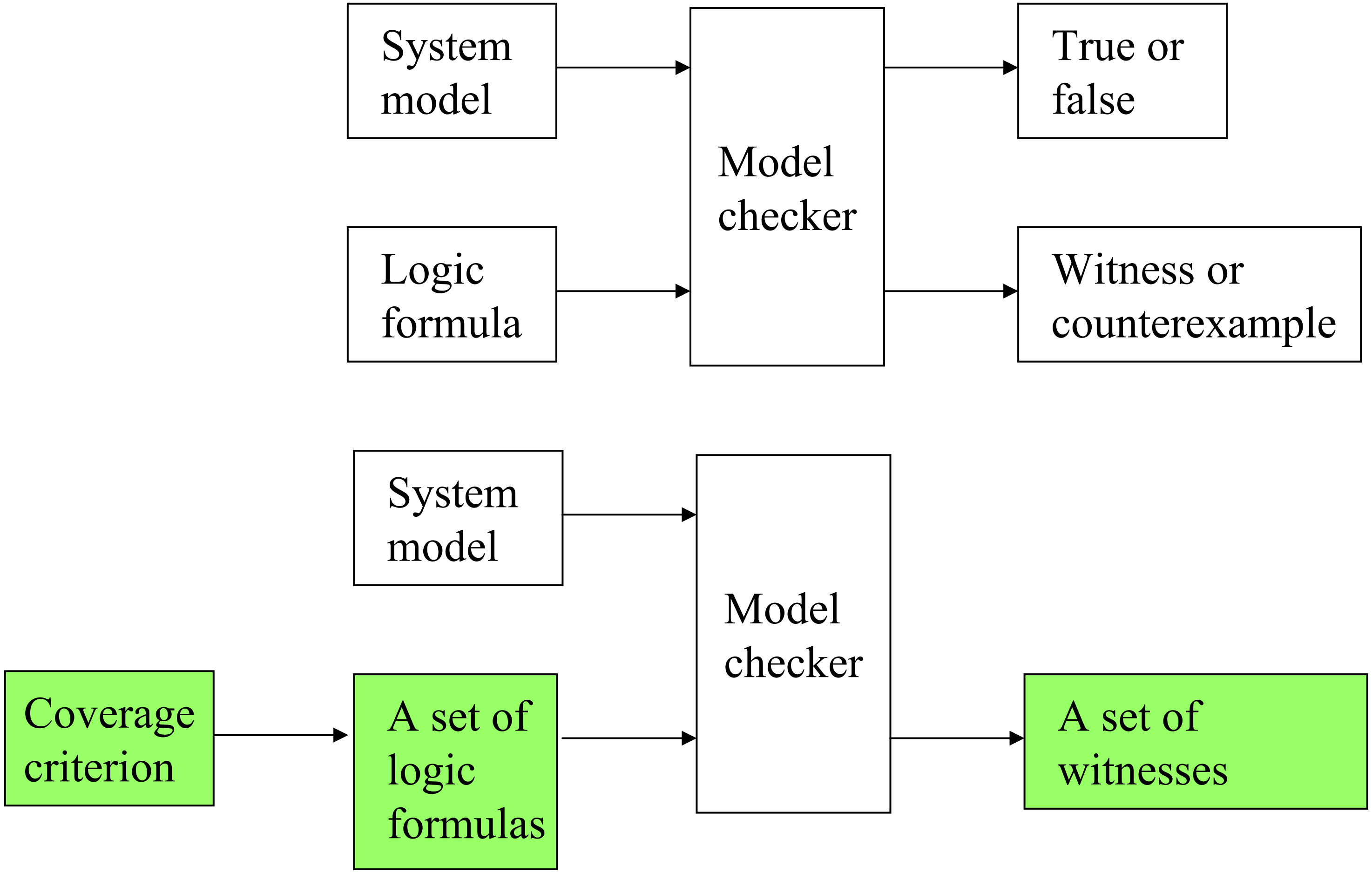
(x, t1) directly affects (m, t1)

(x, t1) affects (y, t5)

Data flow chain coverage

- Affect pair (v, t, v', t')
  - May consist of an arbitrary number of definition-use pairs
  - We extend CTL with least fixpoint operators
    - Alternatively, we can use (alternation-free) mu-calculus
- All-inputs coverage criterion
  - Requires a path from ***every*** input to ***some*** output be covered at least once
- All-outputs coverage criterion
  - Requires a path from ***every*** input to ***every***

Test Generation



System model

Logic formula

Model checker

True or false

Witness or counterexample

System model

Coverage criterion

A set of logic formulas

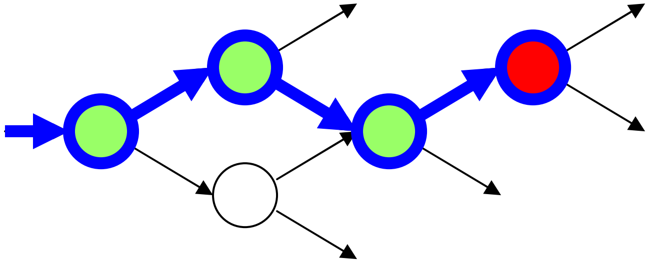
Model checker

A set of witnesses

Test Generation

- Generating a witness for a formula
  - Cost: the length of a witness
  - A minimal-cost witness for a formula
    - Existing model checkers generate a minimal-cost witness by breadth-first search of state space

E[🟢 U 🔴]

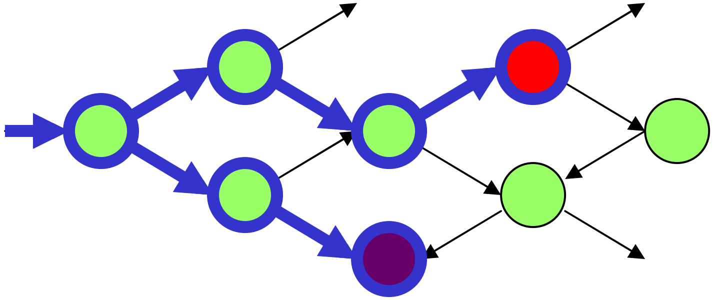


Test Generation

- Costs
  - The total length of witnesses or
  - The number of witnesses
- Both optimization problems are NP-hard

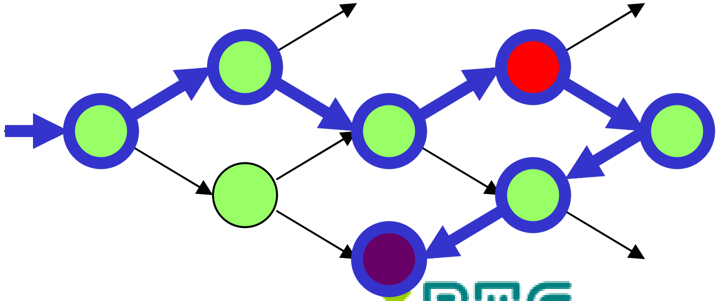
E[🟢 U 🔴]

E[🟢 U 🟣]



E[🟢 U 🔴]

E[🟢 U 🟣]

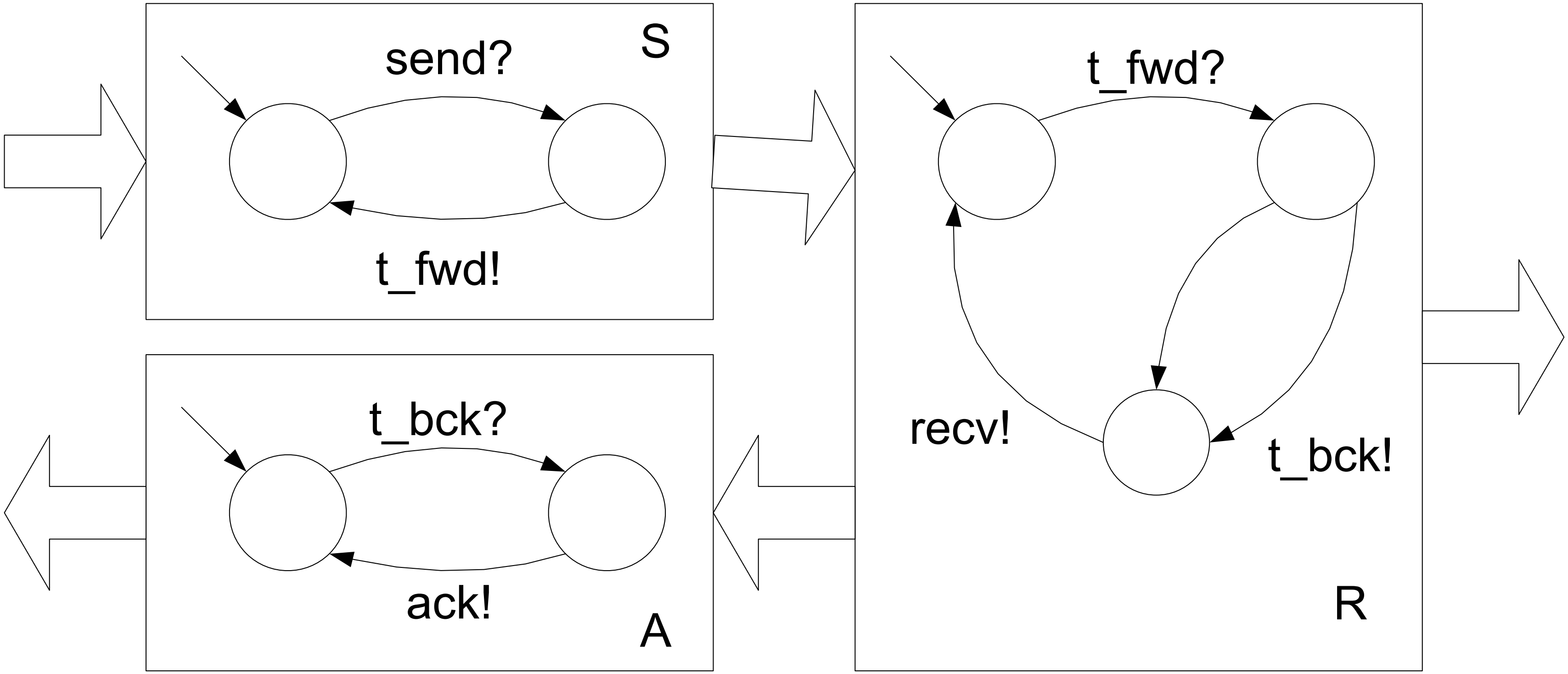


Coverage for distributed systems

- What if our system is a collection of components?
- Possible solutions:
  - Generate tests for each components
    - Clearly unsatisfactory; does not test integration
  - Generate tests from the product of component models
    - Too many redundant tests
- Non-determinism is another problem

Example

- Producer-consumer with acknowledgements



Covering product transition system

- Linear tests bring trouble:

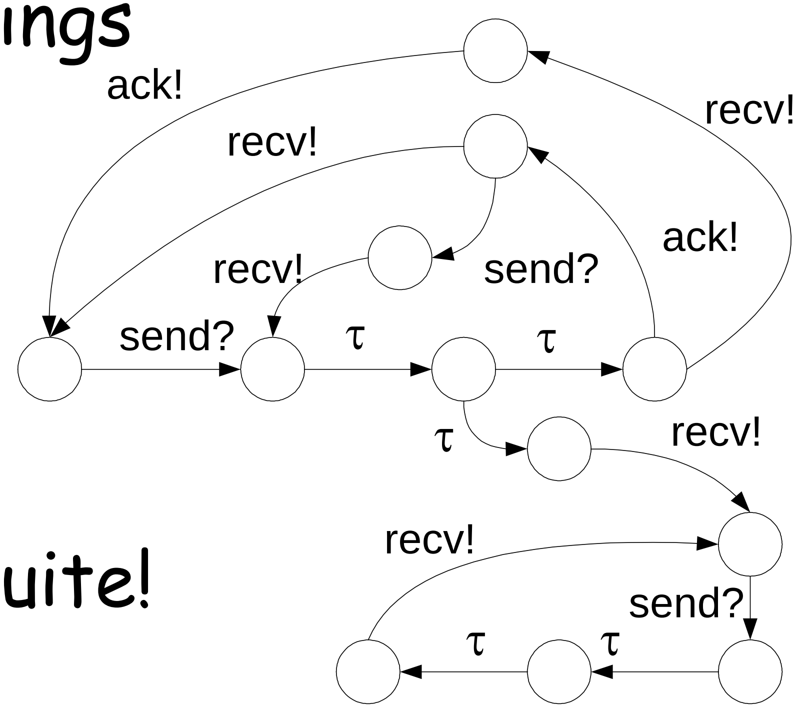
  send?.ack!.recv!

  - May fail if the system chooses a different path
- Tests differ in interleavings of independent events
  - No need to test

    send?.ack!.recv!

    send?.recv!.ack!

    separately
- State explosion in test suite!



Partial orders for test generation

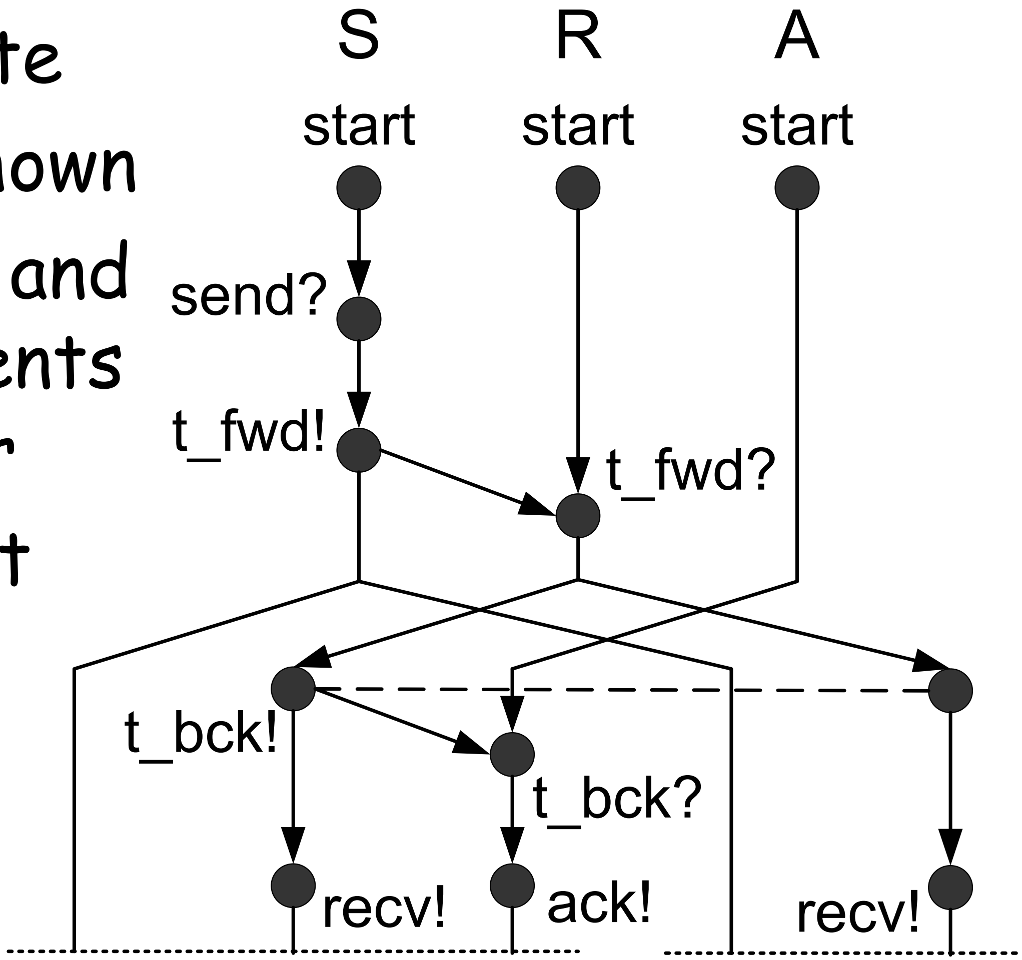
- Use event structures instead of transition systems [Heninger97]
- Test generation covers the event structure
- Allows natural generation of distributed testers

Prime event structures (PES)

- Set of events E
  - Events are occurrences of actions
- Causality relation $\prec\subseteq E\times E$
  - Partial order
- Conflict relation $\#\subseteq E\times E$
  - irreflexive and symmetric
- Labeling function $l : E\rightarrow A$
- Finite causality $\{e' \mid e'\prec e\}$ is finite
- Conflict inheritance $e\#e'\wedge e'\prec e''\Rightarrow e\#e''$

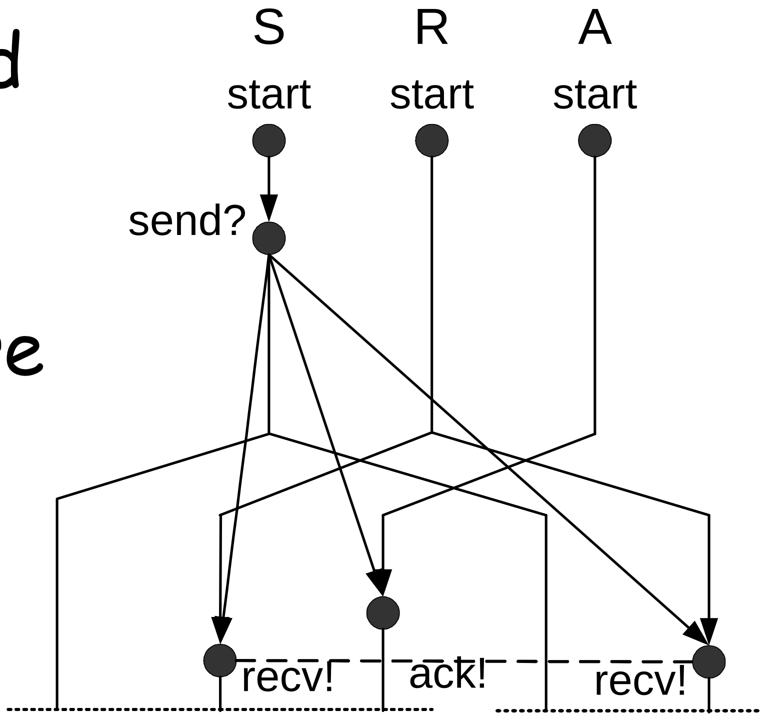
Producer-consumer PES

- Structure is infinite
  - Initial part is shown
- Causally unrelated and non-conflicting events can occur together
- Behaviors will start repeating
  - Can stop with finite structure



Test generation with PES

- Project PES on observable actions, propagating conflicts
- Every path in the PES should be covered
- Tests consist of distributed testers with coordination messages between tests
  - Coordination messages are inserted when there is a causal edge between locations



Summary: coverage-based testing

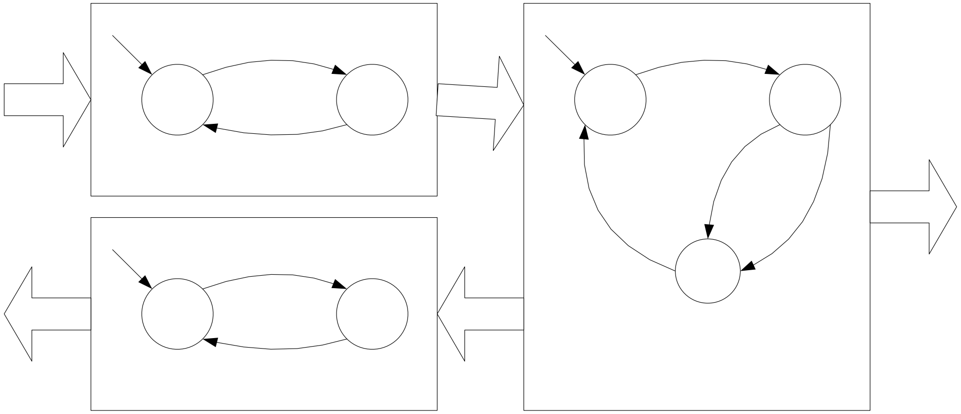
- Advantages:
  - Exercise the specification to the desired degree
  - Does not rely on test purpose selection
- Disadvantages:
  - Large and unstructured test suites
  - If the specification is an overapproximation, tests may be infeasible

Generation of test purposes

- Recent work: [HenningerLuUral-03]
- Construct PES
- Generate MSC (message sequence charts) to cover PES
- Use MSC as test purposes in **ioco**-based test generation

Controllability of testing

- Conformance testing may not provide enough guarantees
  - With branching tests, test purpose behavior may be avoided
  - What if I never see ack?



- Problem: inherent uncertainty in the system

How to contain uncertainty?

- Avoidance (no need to increase control)
  - During testing, compute confidence measure
    - E. g., accumulate coverage
  - Stop at the desired confidence level
- Prevention (add more control)
  - Use instrumentation to resolve uncertainty
  - What to instrument?
    - Use model for guidance
- Anyone needs a project to work on?

Test generation tools

- TorX
  - Based on **ioco**
  - On-the-fly test generation and execution
  - Symbolic testing (data parameterization)
  - LOTOS, Promela, …
  - http://fmt.cs.utwente.nl/tools/torx/
- TGV
  - Based on symbolic **ioconf**
  - LOTOS, SDL, UML
  - http://www.irisa.fr/pampa/VALIDATION/TGV/TGV.html