Test Synthesis

Model-based Testing

- General idea: we have
  - A system under test (SUT)
  - A model (M) describing acceptable behaviour
  - Question: does SUT conform to M?
- A test suite (T) is a collection of test cases (TC)
  - TCs “capture” properties of M
  - We can run a TC on SUT and get a verdict (pass or fail or inconclusive)
  - Ideally, we want a test suite T such that

    SUT conforms to M $\Leftrightarrow$ SUT passes all cases in T

IOLTSs

- We will describe models, SUTs, and test cases via input/output LTSs
  – All actions are either outputs (!action) or inputs (?action), or the invisible action τ.
  – L1 || L2 (Parallel composition) = LTS product with synchronization on input/output pairs

- Test hypothesis: we can use the same formalism (namely IOLTS) for models (M) and implementations (SUT)

Input-Output Conformance (ioco)

- A trace of an LTS with initial state s_0 is a sequence of actions $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ such that

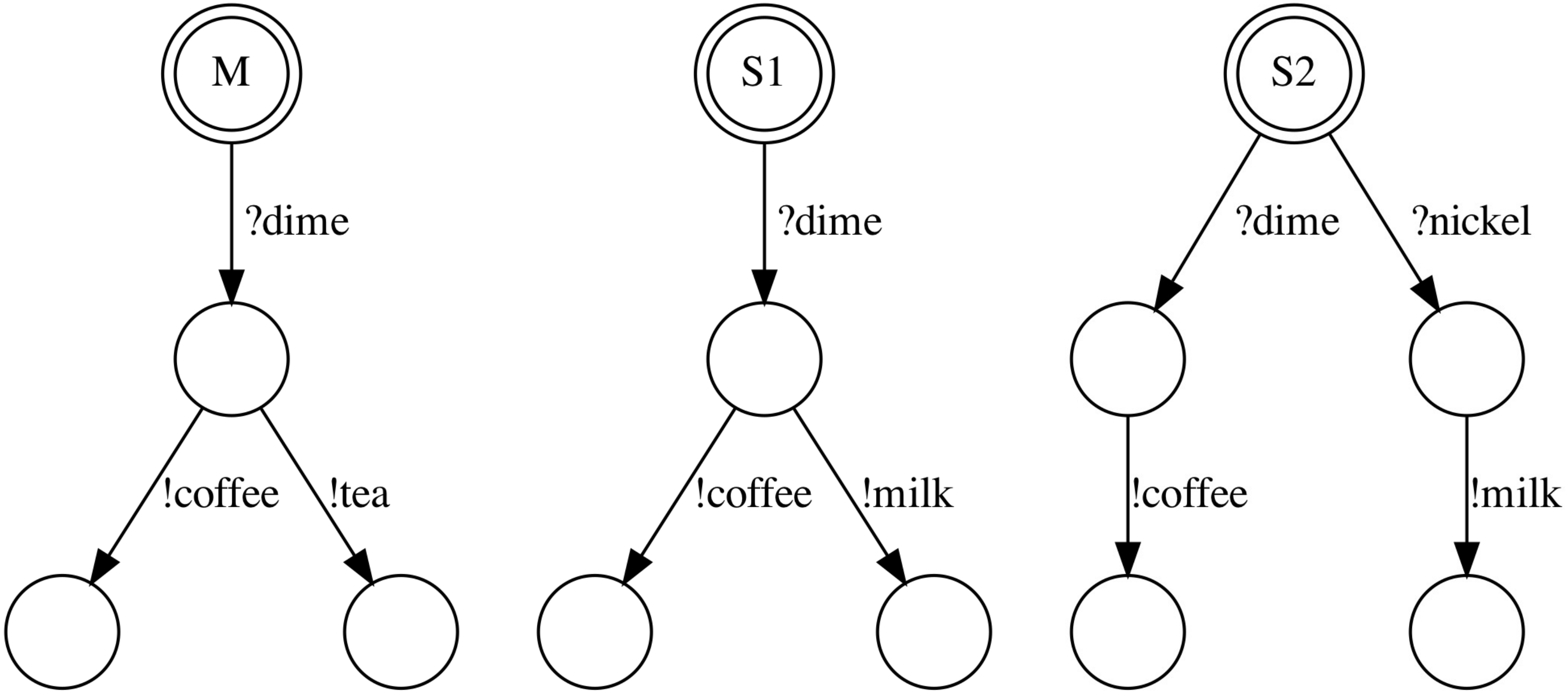
$$s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} \ldots \xrightarrow{\sigma_n} s_n$$

For some states $s_{1,2,\ldots,n}$ in the LTS.

- We say that (a system SUT) ioco (a model M) if, for all traces σ of M:
  - When SUT can perform an output !x after a trace σ, M can also perform !x after σ
  - When SUT cannot perform any output after a trace σ, the same must be true of M

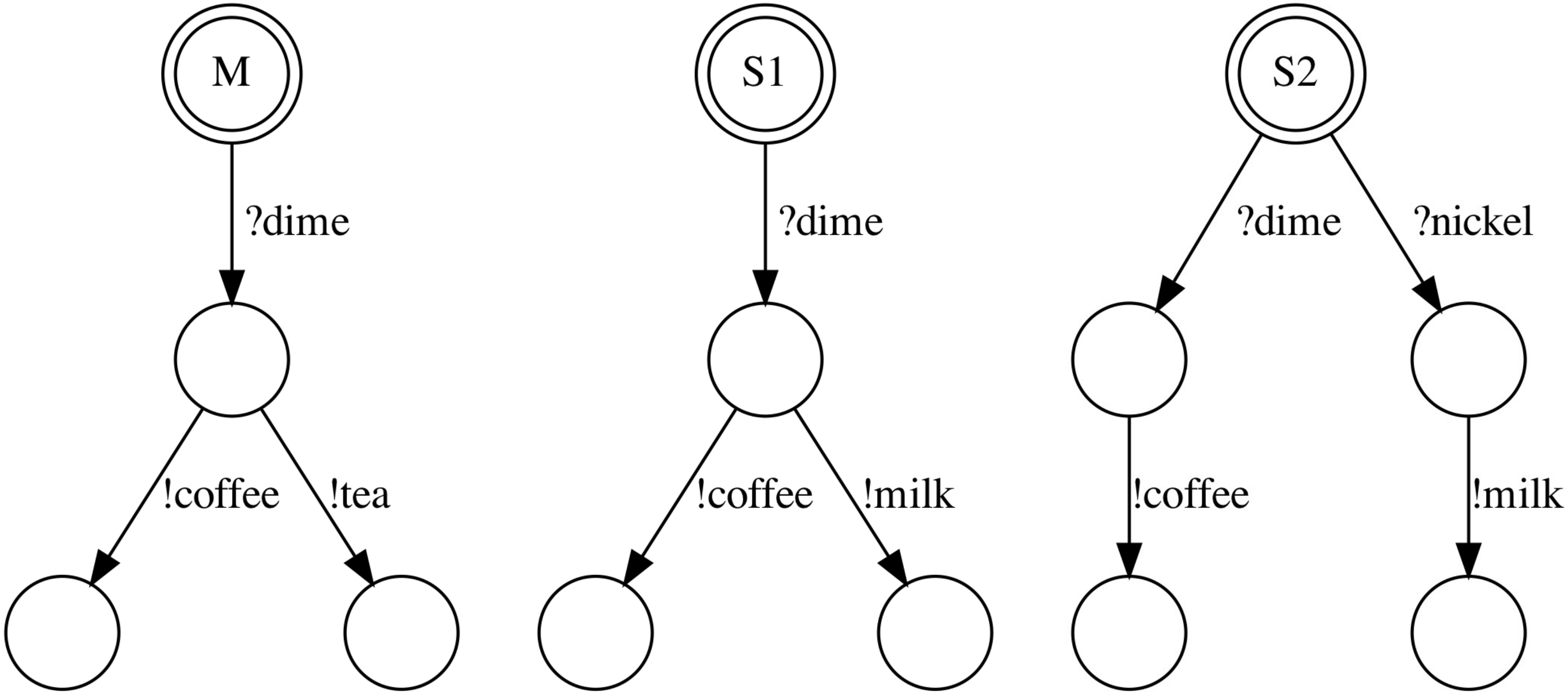
Exercises

- S1 ioco M ?
- S2 ioco M ?



Solution

- S1 not ioco M (M cannot do *!milk* after *?dime*)
- S2 ioco M (ioco “does not care” about ?nickel)
  - but M not ioco S2 (ioco is not symmetrical)

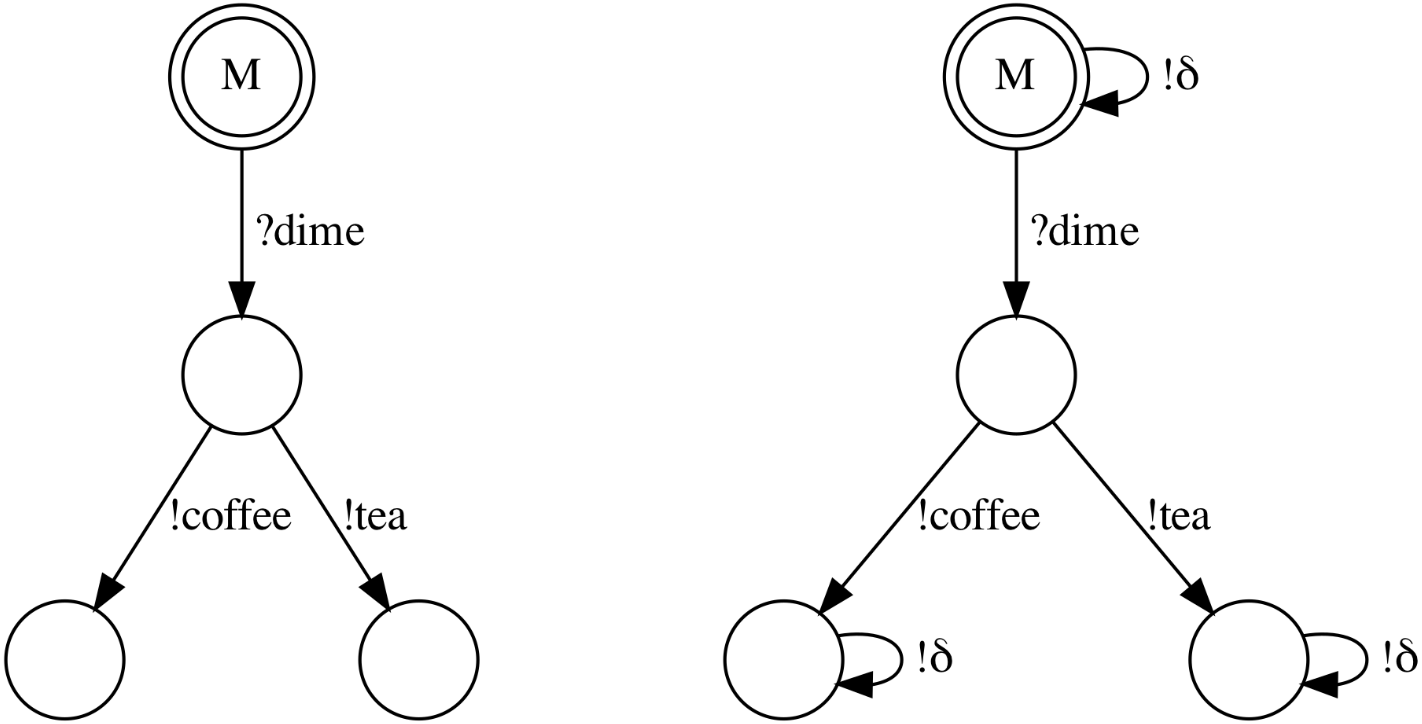


Suspension automata (1/2)

- A state is quiescent if:
  - Has no outgoing actions at all (deadlock)
  - Can only wait for some input (outputlock)
  - Is part of a cycle of internal actions (livelock)
- For ioco to “work”, we must make quiescence explicit
- To do so, we find all quiescent states s and add a special self-transition $s \xrightarrow{!\delta} s$ to them
- The result is called a suspension automaton

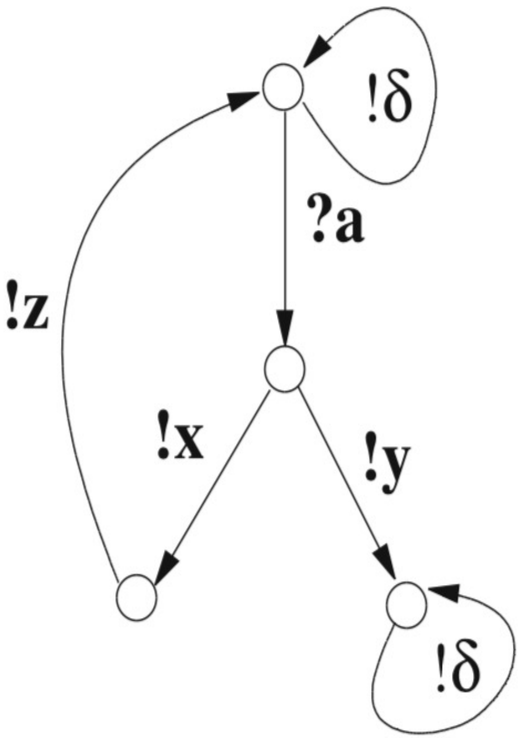
Example (1/2)

- Example: suspension automata for M
  - Initial state is outputlocked (must wait for *!dime*)
  - States at the bottom are deadlocked

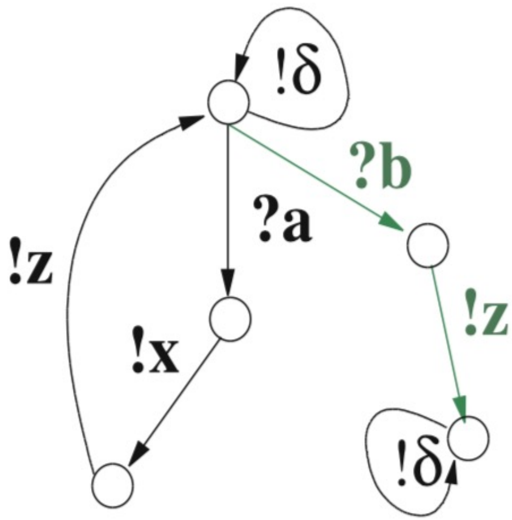


Example (2/2)

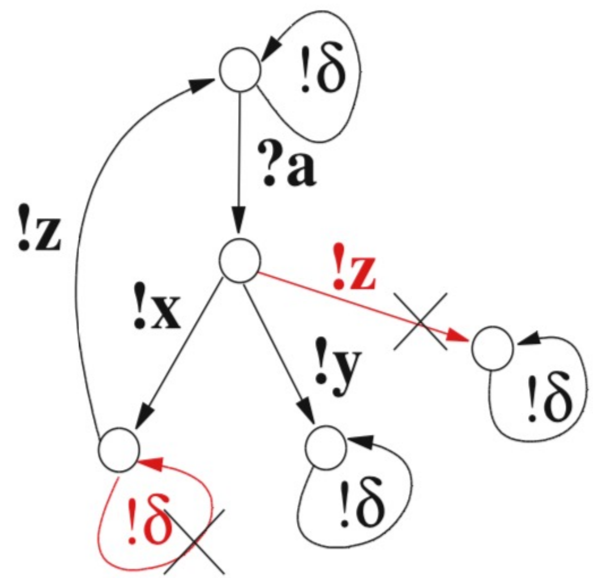
Model



IUT1
implementation choice
partial specification



IUT2
forbidden output
forbidden quiescence



Test case

- Informally, a test case TC is an IOLTS where:
  - The I/Os of the TC correspond to O/Is of a SUT
  - some states are marked pass, fail, or inconclusive
  - Special action ?θ that complements !δ
- "Running" a TC on a SUT = compute traces of TC || SUT and check which marked states are reached
  - Verdict = pass/fail/inconc, depending on state reached

Test purposes

- TCs are somewhat too “low-level” to be practical
  - Idea: select/generate TCs based on a more abstract description called test purpose
- A TP is an IOLTS (again) which describes some desired behaviours of the SUT
- Some states in a TP are marked accept or refuse
  - An accept state is reached = the desired behaviour has been observed (corresponds to a pass in the test case)
  - When a refuse state is reached, it means that this execution is not relevant to the test purpose. It does not correspond to a failure!

Test synthesis

- From a model M and a test purpose, generate a complete test graph (CTG)
- CTG
  - Describes one or more test cases
  - Obtained by "combining" the TP with the model, marking states as pass/fail/inconclusive based on the TP, etc.
- On-line testing
  - Generate CTG
  - Compute traces of CTG || SUT
  - All at the same time

Example of an LNT test purpose

- Use loops to mark accept/refuse states
  - Desired behaviour: y followed by z
  - If you observe z: do not care about what's next

```
process TP
[TESTOR_ACCEPT, TESTOR_REFUSE, y, z: none] is
   select
      y; z; loop TESTOR_ACCEPT end loop
   []
      z; loop TESTOR_REFUSE end loop
   end select
end process
```

Soundness, exhaustiveness, completeness (1/2)

- A suite T is sound (with respect to a model M) if

  SUT conforms to M $\Rightarrow$ SUT passes T

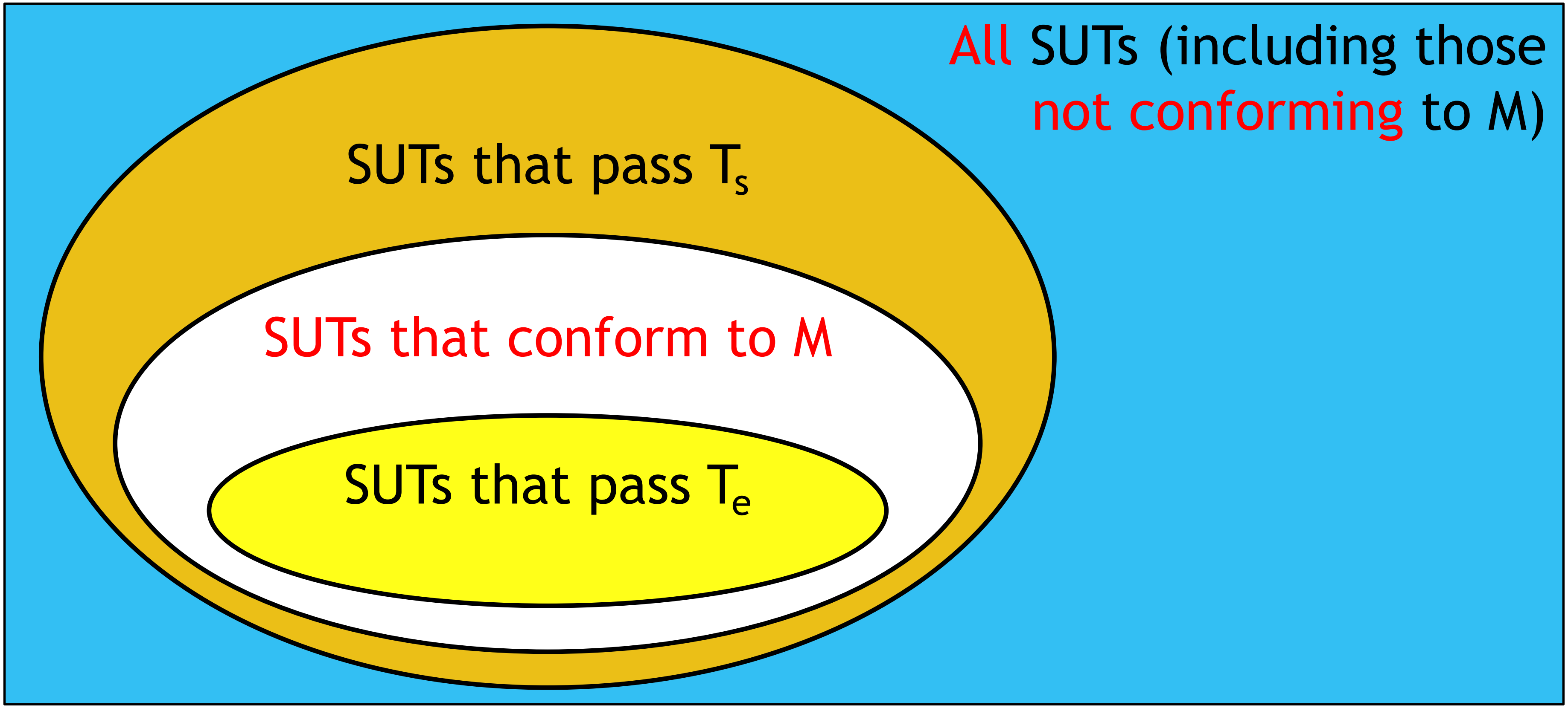
  (non-conforming SUTs might pass!)
- A suite T is exhaustive (wrt. M) if

  SUT passes T $\Rightarrow$ SUT conforms to M

  (conforming SUTs might fail!)
- It is complete if it is sound and exhaustive
- Unfortunately, exhaustiveness is difficult to achieve (it may need infinitely many test cases)
  - So we focus on generation of sound test suites

Soundness, exhaustiveness, completeness (2/2)

Assume that T_e is exhaustive and T_s is sound with respect to some model M. Then you have the following sets of SUTs:



Testor

- Example invocation:

`lnt.open model.lnt testor -io actions.io purpose.bcg testcase.bcg`

  - `model.lnt`: LNT description of the model
  - `actions.io`: specifies which actions are inputs/outputs
  - `purpose.bcg`: IOLTS of the test purpose
  - `testcase.bcg`: filename of generated LTS (test case)
- You can use generator to create a `purpose.bcg` from a `purpose.lnt`

bcg_execute and Testor (1/2)

- Goal: we want to execute CTG || SUT
- `bcg_execute`: utility to execute a SUT, described in BCG format
  - Output actions will be printed
  - User provides input actions from command line
  - E.g., the SUT waits on `?x` until the user types `x<Enter>`
- We can also generate/execute the CTG on-line, with `testor -interactive`
  - Can we send CTG outputs to SUT and vice versa?
  - On Linux/macOS, we can, by using named pipes

bcg_execute and Testor (2/2)

- Example:

```
mkfifo sut.input
mkfifo sut.output
bcg_execute sut.bcg -io sut.io > sut.output < sut.input &
testor -interactive -io sut.io tp.bcg < sut.output 2> sut.input
```

- If SUT is nondeterministic, multiple runs can lead to different results