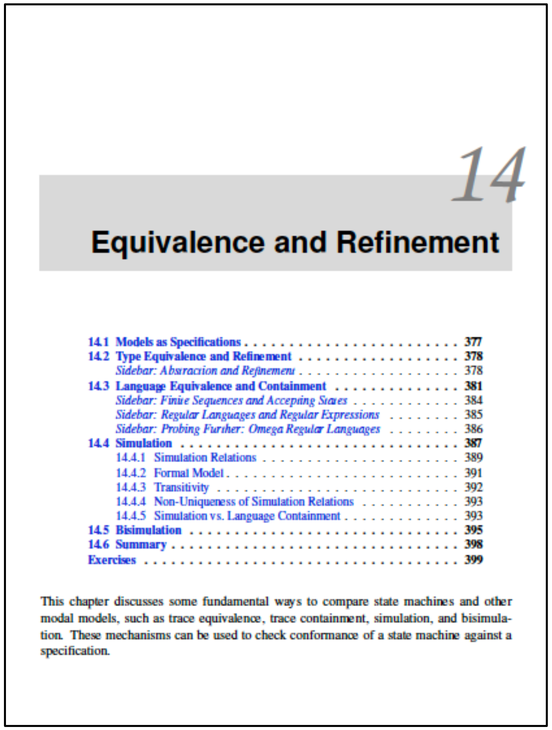
Lecture 17 :
Equivalence and Refinement

Slides were originally developed by Profs. Edward Lee and Sanjit Seshia, and subsequently updated by Profs. Gavin Buskes and Iman Shames.

Outline

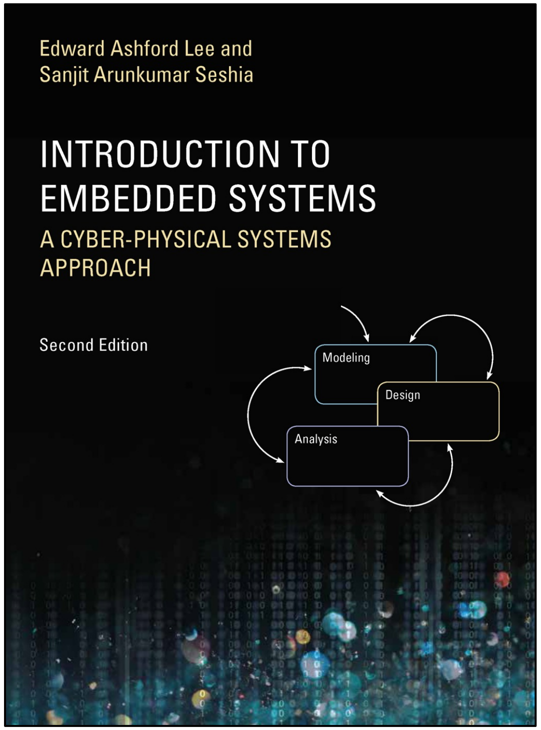
- Equivalence
  - Type
  - Language
  - Bisimulation
- Refinement:
  - Type
  - Language
  - Simulation



14

Equivalence and Refinement

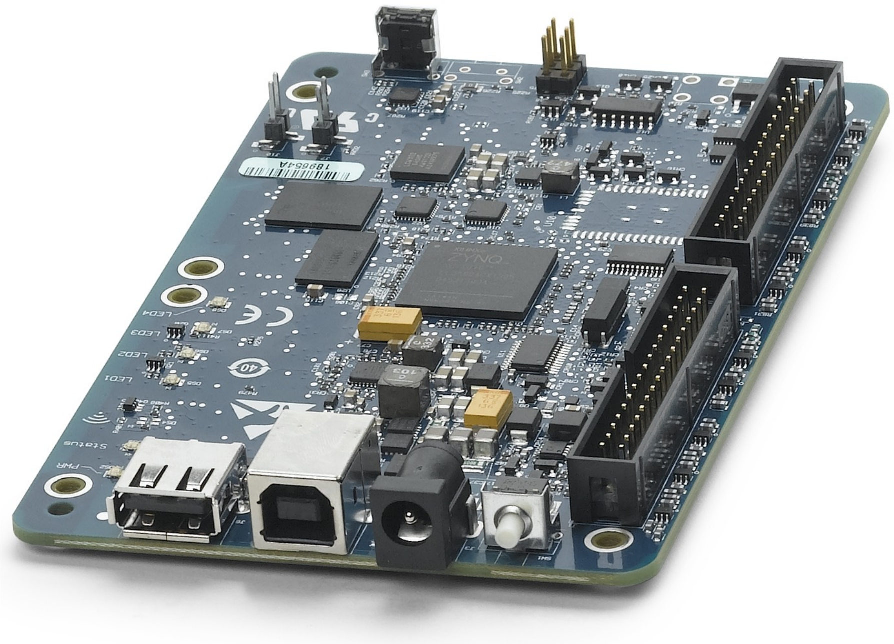
This chapter discusses some fundamental ways to compare state machines and other modal models, such as trace equivalence, trace containment, simulation, and bisimulation. These mechanisms can be used to check conformance of a state machine against a specification.



Component Substitution

Can we replace one component in a system by another and be assured that it will continue to work correctly?

What if we replace the Cortex-A9 core by a Cortex-A12?



myRIO 1950/1900

Comparing State Machines

Why compare state machines?

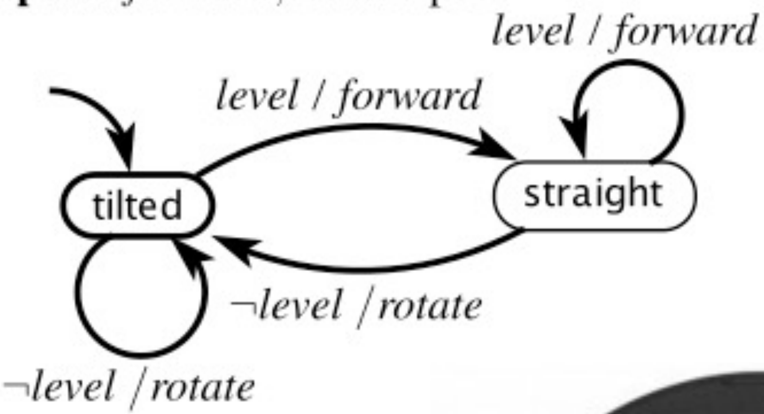
- Check conformance with a specification.
- Optimise a model by reducing complexity.
- Check if component substitution is OK.
- …

How can we compare two state machines?

- Equivalence: Do they 'do the same thing'?
- Refinement: Does one do 'more' than the other?
  - e.g. exhibit different behaviours? Produce different outputs?

FSM Controller for Kobuki

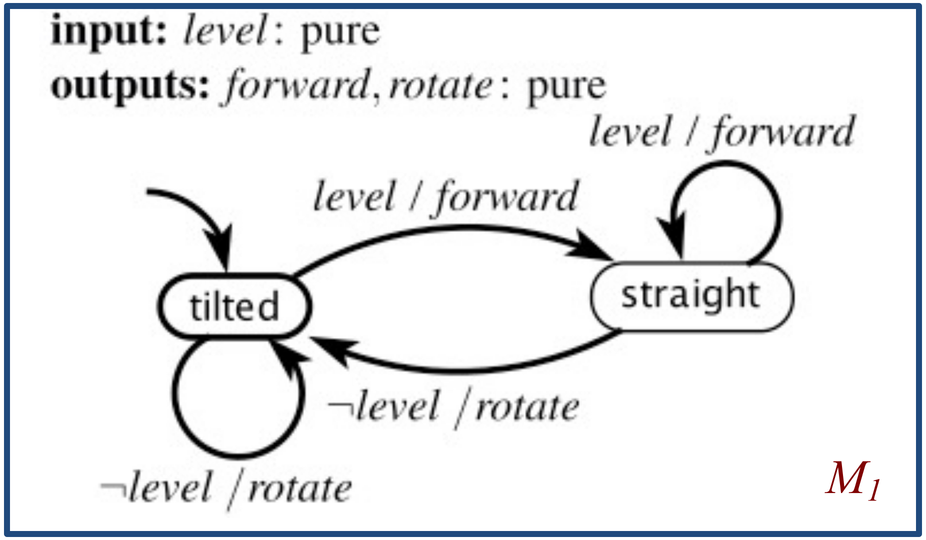
input: *level*: pure
outputs: *forward*, *rotate*: pure



Assume a time-triggered FSM.

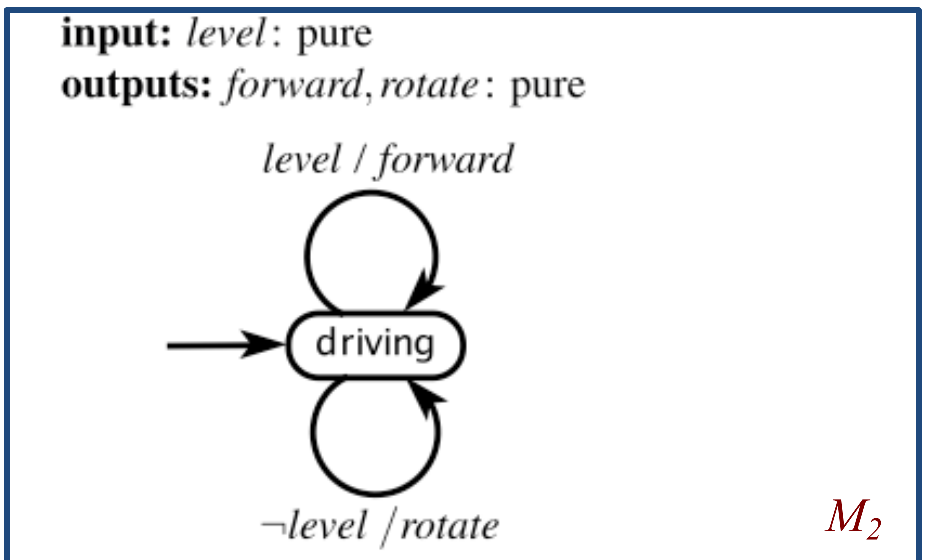
- If the level input is present, then it drives forward for a fixed amount of time by issuing a drive command.
- If the level input is absent, then it rotates for a fixed amount of time.

FSM Controller for Kobuki



Assume a time-triggered FSM.

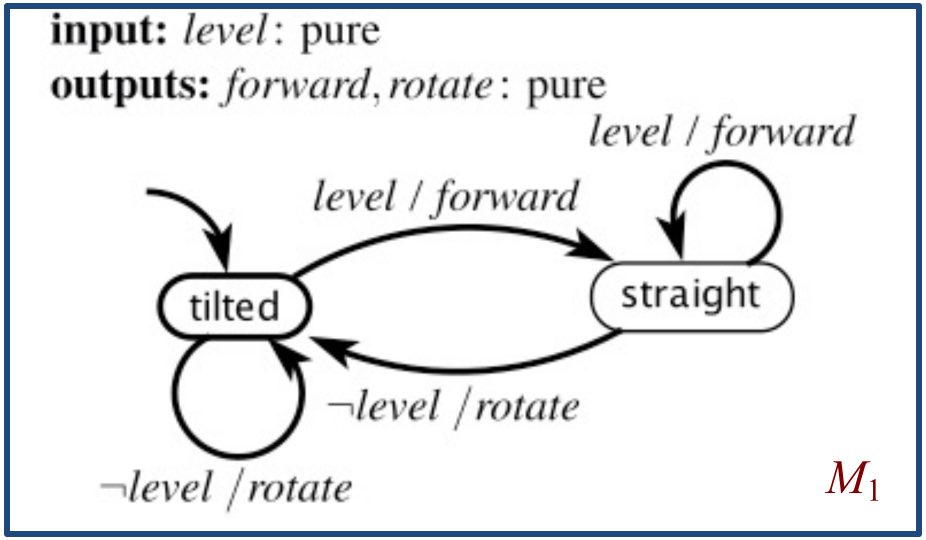
- If the level input is present, then it drives forward for a fixed amount of time by issuing a drive command.
- If the level input is absent, then it rotates for a fixed amount of time.

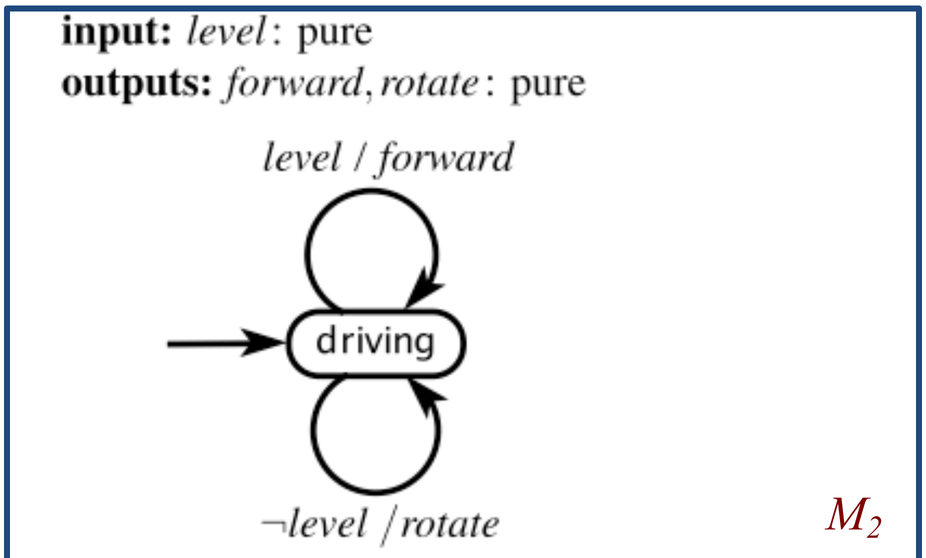


Alternative FSM.

Is machine M_2 equivalent to M_1?
In what sense?

Equivalence: Part 1: Type Equivalence



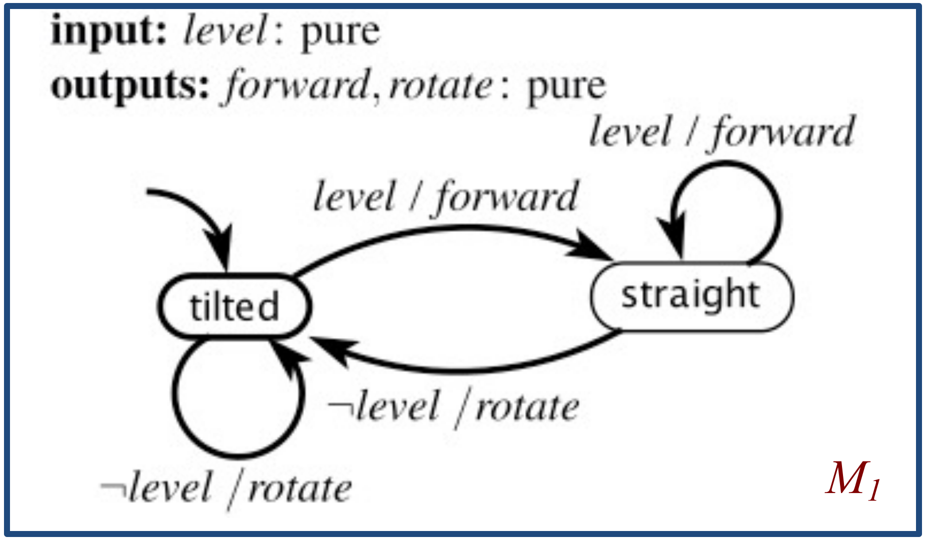


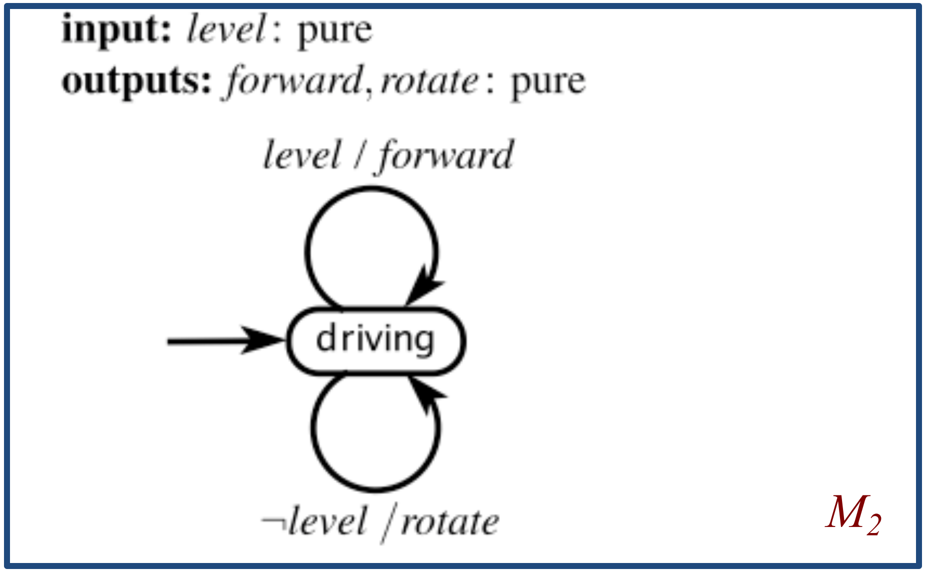
Notice that the actor models for these machines have the same input ports and the same output ports.

Moreover, the ports have the same types.

Therefore M_2 is **type equivalent** to M_1.

Equivalence: Part 2: Language Equivalence

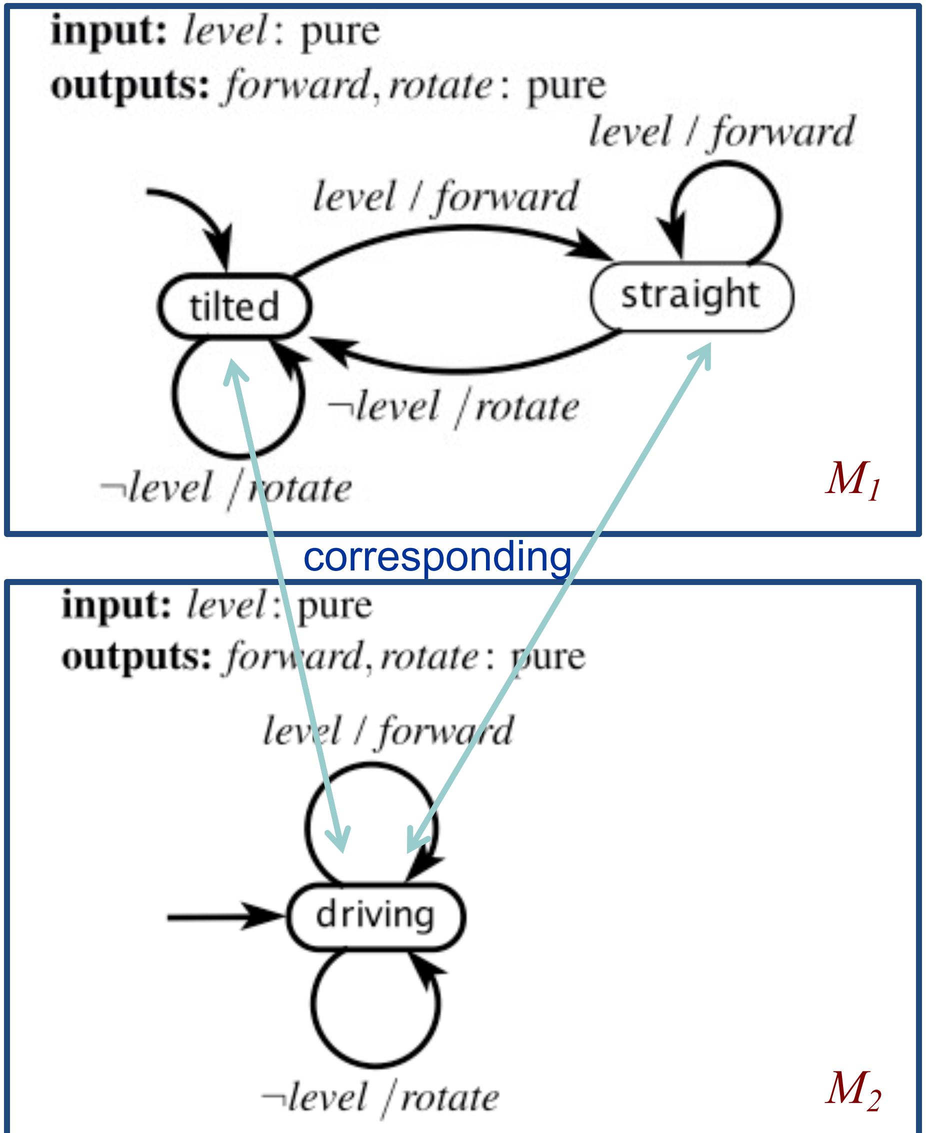




Notice that for every input sequence, the two machines produce the same output sequence.

Therefore M_2 is **language equivalent** to M_1.

Equivalence: Part 3: Bisimulation



This one is very subtle:
Notice that for every state of M_1 there is a corresponding state of M_2 that will react to inputs in exactly the same way and will then transition to another similarly corresponding state.

Therefore M_2 is **bisimilar** to M_1.

For deterministic machines, language equivalence and bisimilarity are the same. For nondeterministic machines they are not.

We will come back to this!
But first, refinement.

Equivalence vs. Refinement

Two state machines M_1 and M_2 that are not equivalent may nonetheless be related:

- M_2 may be type compatible with M_1 in that it can replace M_1 without causing a type conflict. (**type refinement**)
- M_2 may be a specialisation of M_1 in that it can produce only output sequences that M_1 can produce, given the same input sequences. (**language containment**)
- M_2 may be a specialisation of M_1 in that at every reaction M_2 can produce only output values that M_1 can produce. (M_1 simulates M_2) (**simulation**)

In all cases, if M_1 is "valid" in a system, then so is M_2, where only the meaning of "valid" varies.

- M_2 is a type/language/simulation refinement of M_1.
- M_2 implements M_1 (here, M_1 is taken to be a specification).

Refinement: Part 1: Type Refinement

$P_1 = \{ x, w \}$ $\quad$ $Q_1 = \{ y \}$

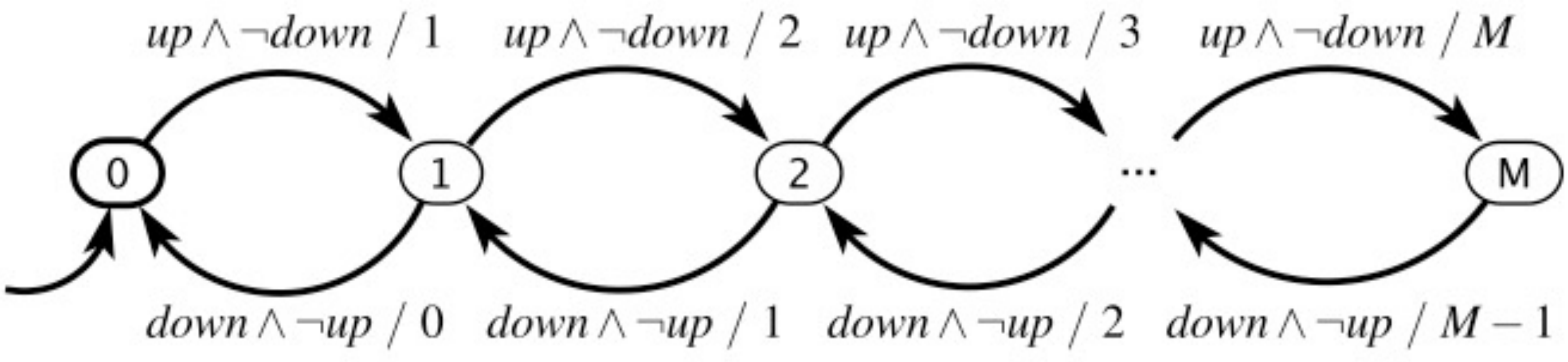
$P_2 = \{ x \}$ $\quad$ $Q_2 = \{ y, z \}$

M_2 is a **type refinement** of *M_1* if:

- $P_2 \subseteq P_1$
- $Q_1 \subseteq Q_2$
- $\forall\, p \in P_2, \quad V_p \subseteq V'_p$
- $\forall\, p \in Q_1, \quad V'_p \subseteq V_p$

M_2 can replace *M_1* without causing a type conflict.

Recall the Garage Counter

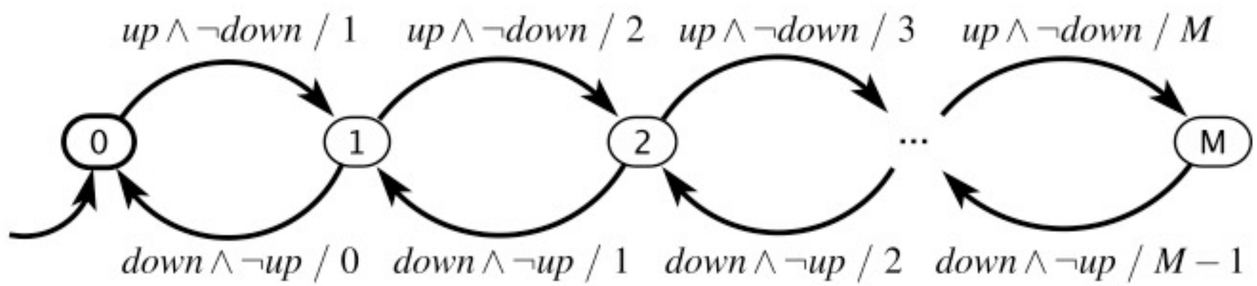


Input ports: $P = \{up, down\}$, with types $V_{up} = V_{down} = \{present\}$.
Output port: $Q = \{count\}$ with type $V_{count} = \{0, \cdots, M\}$.

A behavior:

$$\begin{aligned} s_{up} &= (present, absent, present, absent, present, \cdots) \\ s_{down} &= (present, absent, absent, present, absent, \cdots) \\ s_{count} &= (absent, absent, 1, 0, 1, \cdots) \ . \end{aligned}$$

Example of Type Refinement



Consider a garage counter M_1 with $M = 99$ spaces.

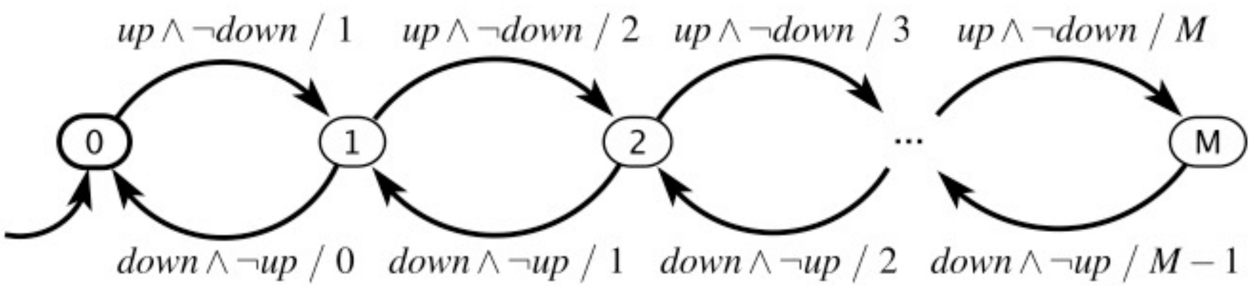
Suppose another garage counter M_2 has $M = 90$ spaces.

M_2 is a type refinement of M_1 .

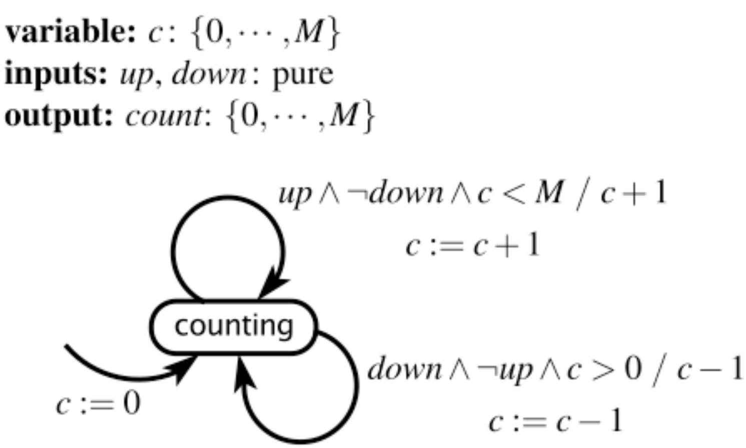
Why might this matter?

Is it always OK to replace M_1 with M_2?

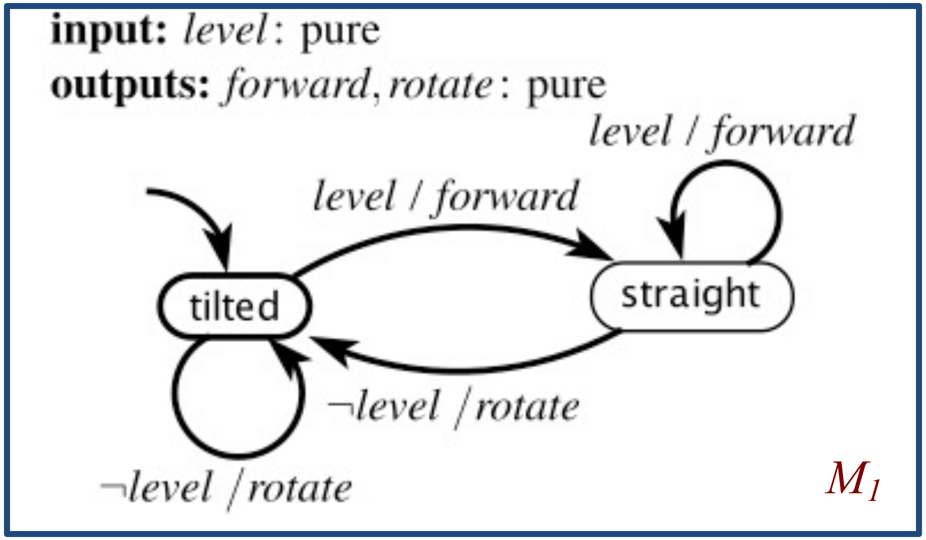
When is Replacement OK?

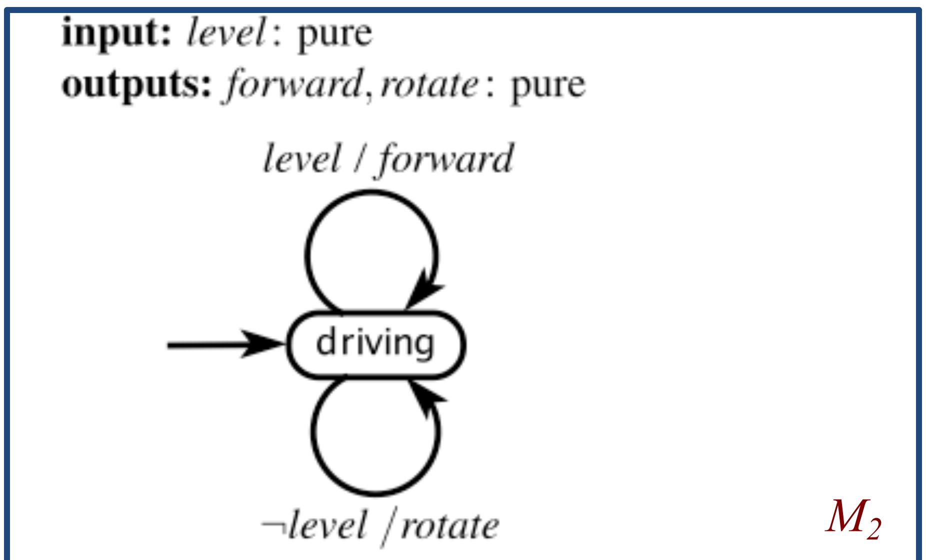


The counter machine above can be replaced by the "equivalent" machine below:



When is Replacement OK?





The two machines are again "equivalent." How to define equivalence? Is equivalence always required?

For *deterministic* machines:

language refinement.

For *nondeterministic* machines:

simulation

Behaviour (Execution Trace) of a State Machine

An **execution trace** is a sequence of the form

$$q_0,\ q_1,\ q_2,\ q_3,\ \ldots,$$

where $q_j = (x_j, s_j, y_j)$ where s_j is the state at step j, x_j is the input valuation at step j, and y_j is the output valuation at step j. Can also write as

$$s_0 \xrightarrow{x_0/y_0} s_1 \xrightarrow{x_1/y_1} s_2 \xrightarrow{x_2/y_2} \cdots$$

For language refinement, traces will comprise only of inputs and outputs, not of states.

Behaviour of a State Machine

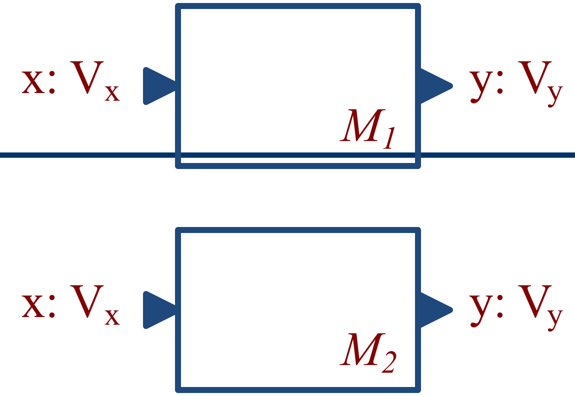
Consider a port p of a state machine with type V_p. This port will have a sequence of values from the set $V_p \cup \{absent\}$, one value at each reaction. We can represent this sequence as a function of the form

$$s_p : \mathbb{N} \to V_p \cup \{absent\} \ .$$

This is the signal received on that port (if it is an input) or produced on that port (if it is an output).

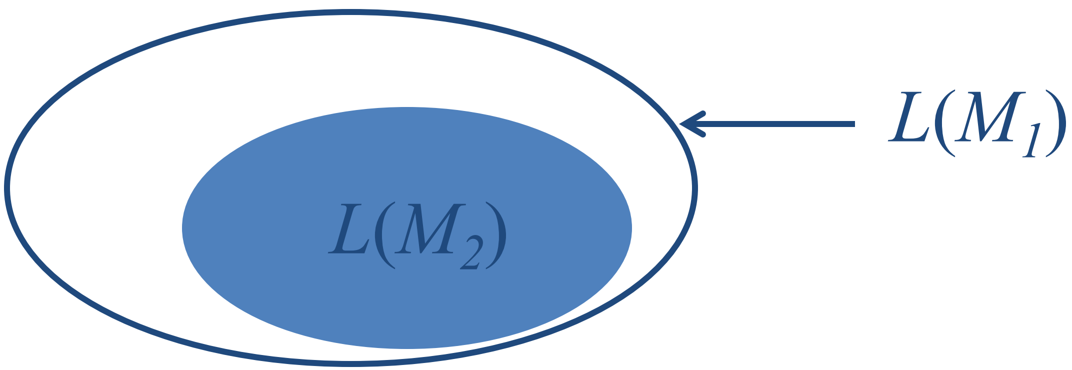
A **behavior** of a state machine is an assignment of such a signal to each port such that the signal on any output port is the output sequence produced for the given input signals.

Language Refinement



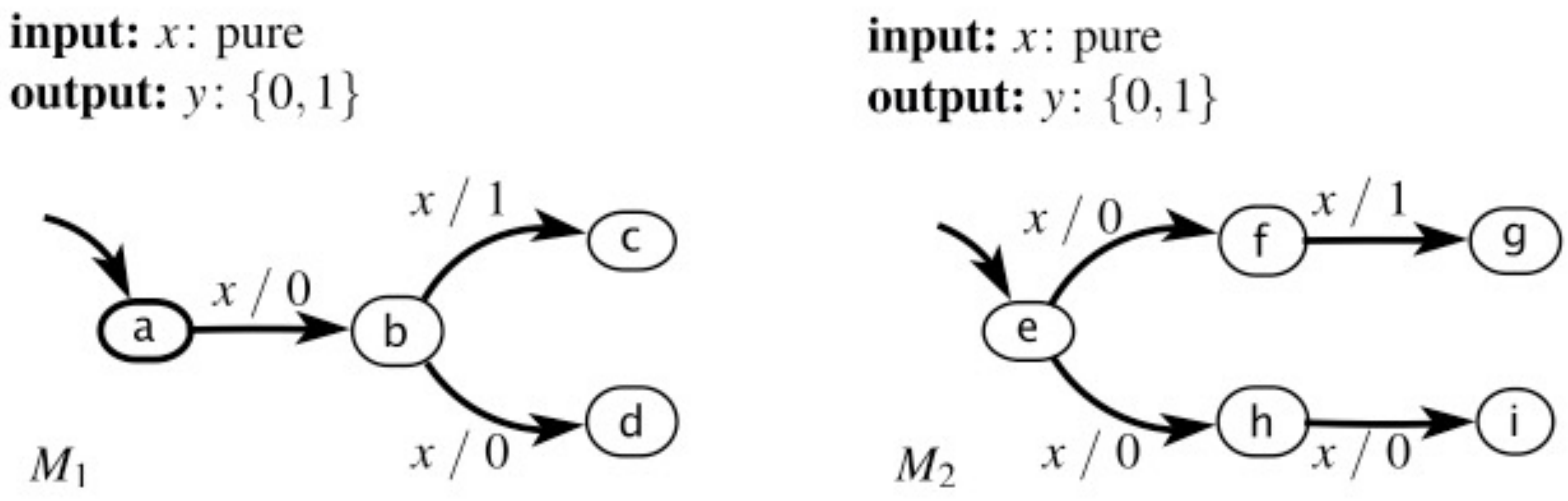
The language $L(M)$ of a state machine M is the set of all behaviors.

For type equivalent state machines M_1 and M_2, M_2 is a **language refinement** of M_1 if $L(M_2) \subseteq L(M_1)$.



M_2 can replace M_1 if its observable (I/O) behaviour is a subset of that of M_1.

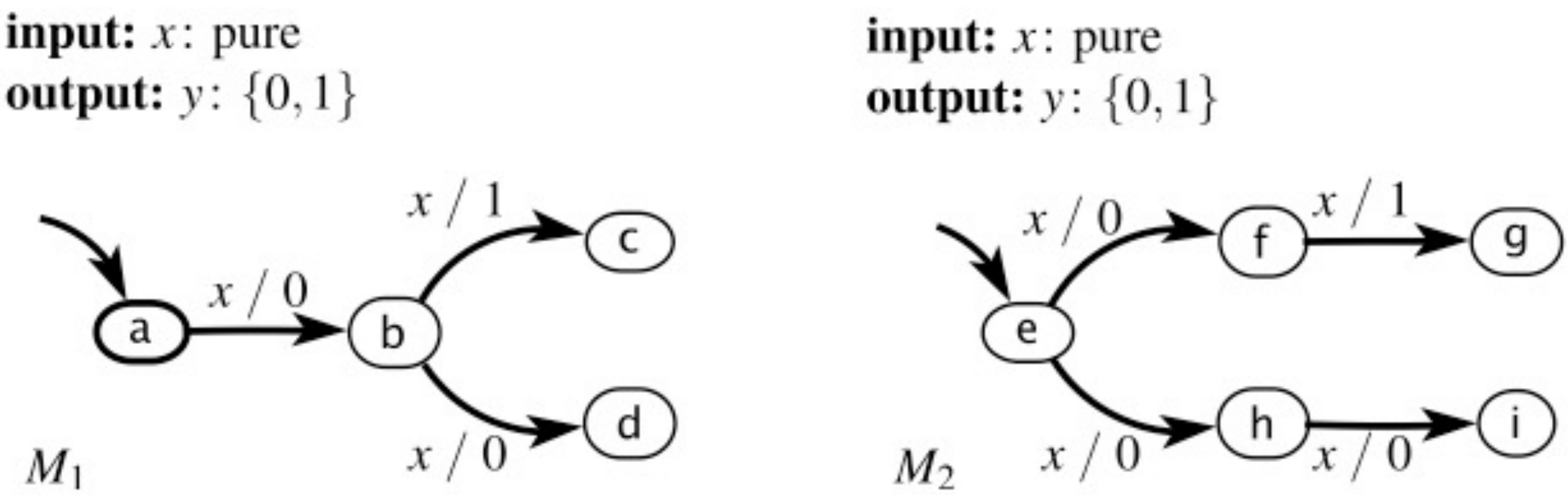
Language Equivalence is not Enough in General



Note that these two machines are language equivalent.

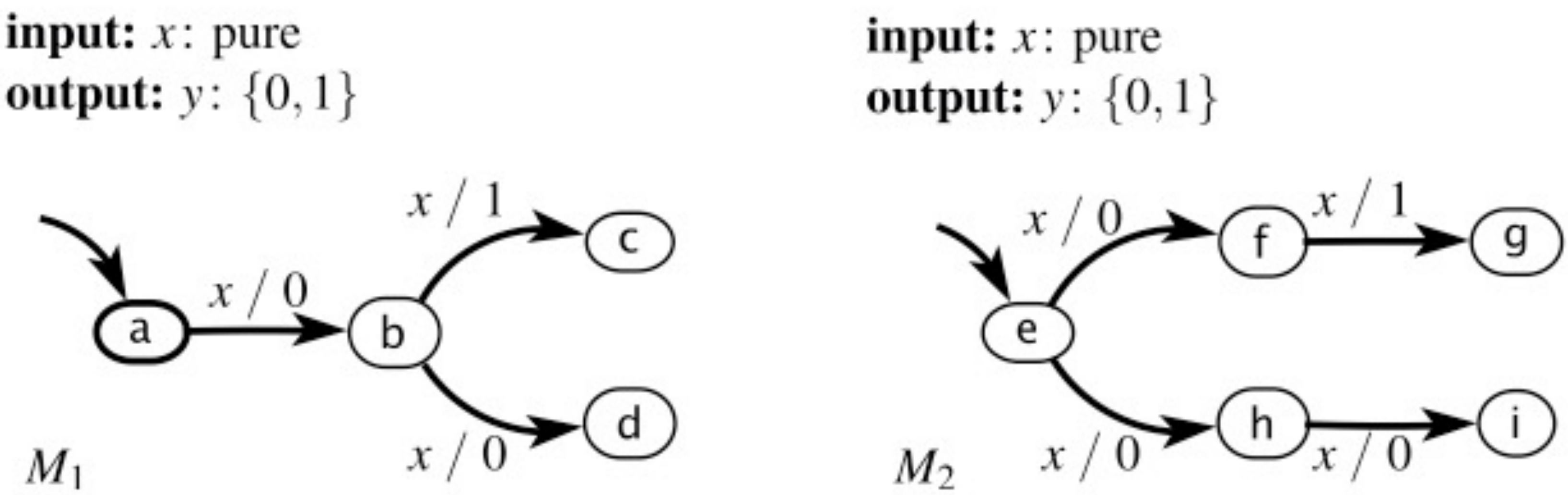
Yet….

Language Equivalence is not Enough in General



Even though these machines have exactly the same input/output behaviours, there is a context in which M_1 is not a valid replacement for M_2.

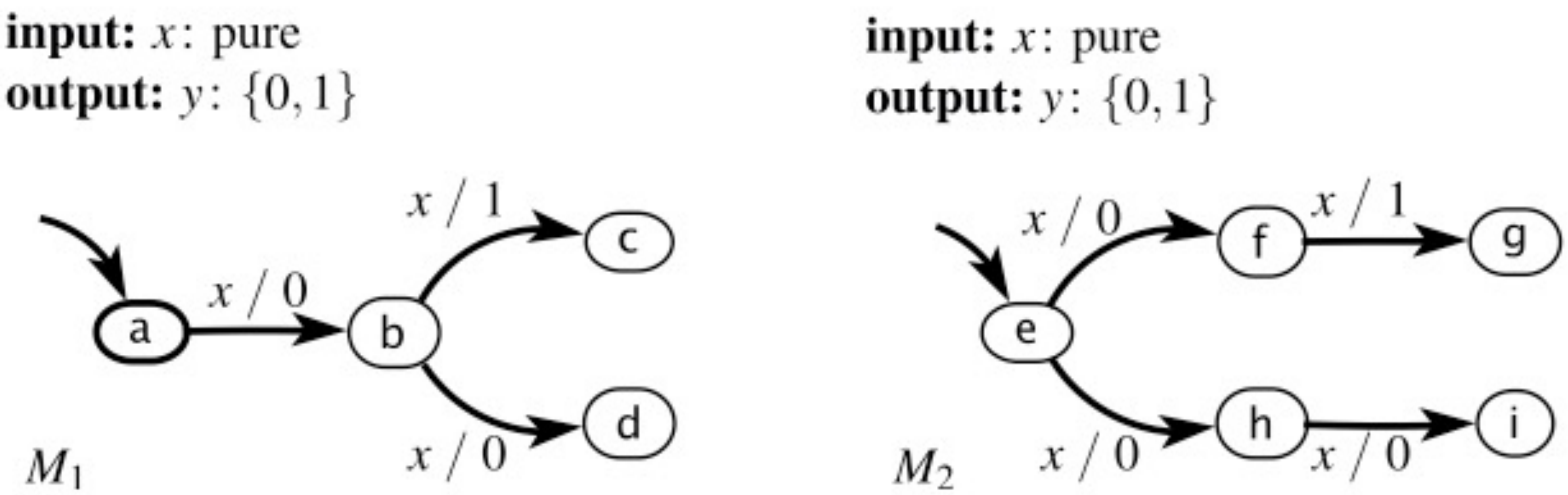
Language Equivalence is not Enough in General



Suppose M_1 is the specification (everything it does is OK).

It is fine to replace it with M_2 because at each step, any move M_2 can make is OK (because any move M_1 can make is OK).

Language Equivalence is not Enough in General

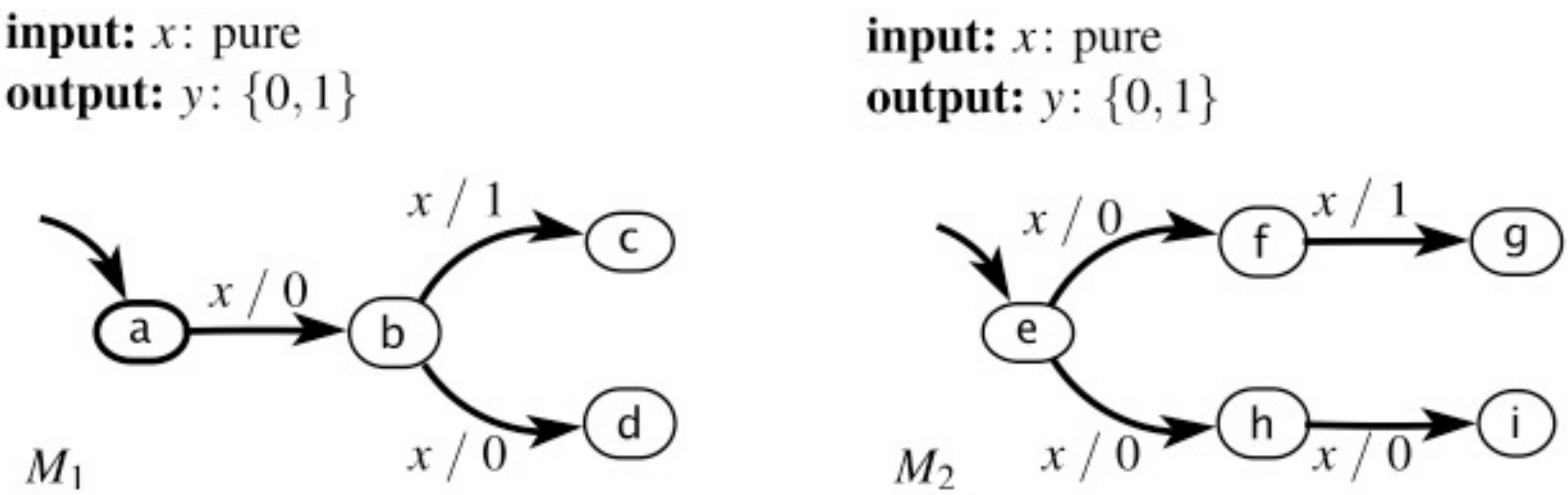


Conversely,

Suppose M_2 is the specification (everything it does is OK).

It is not OK to replace it with M_1 because in state b, M_1 is always capable of making a move that M_2 cannot make (think of a malicious M_1 that watches M_2).

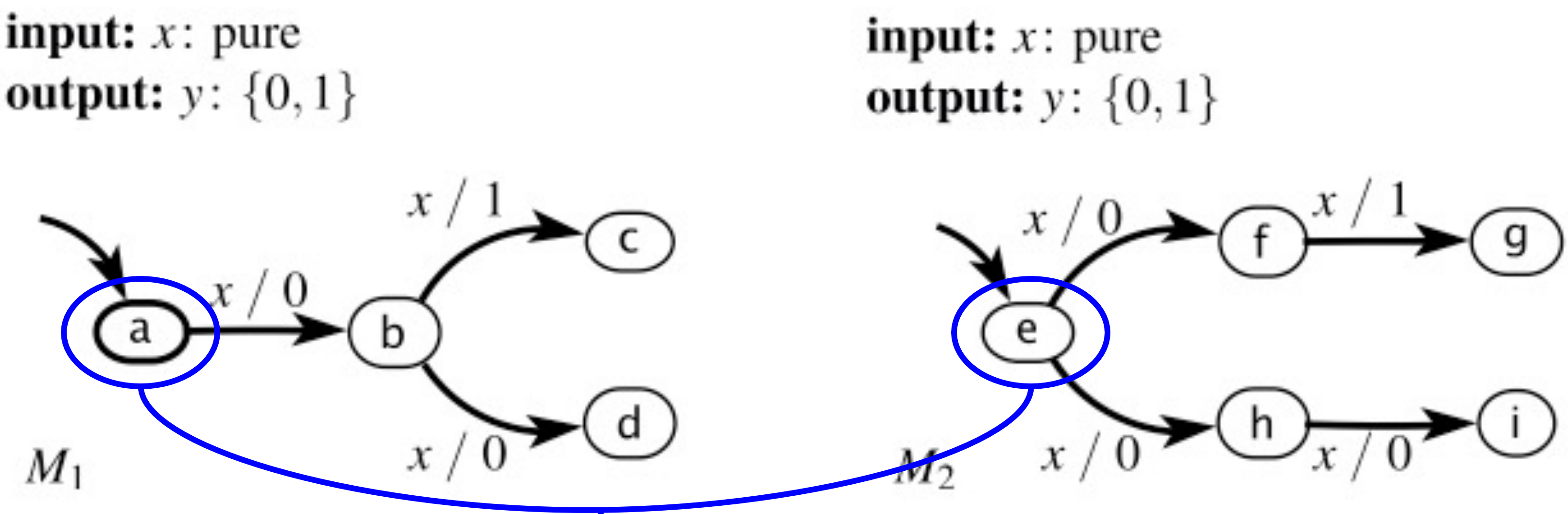
Simulation Relation: The Matching Game



M_1 simulates M_2.

$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$
$S \subseteq S_2 \times S_1$ is a **simulation relation**

Simulation Relation: The Matching Game



M_1 simulates M_2.

Game: each machine starts in its initial state.

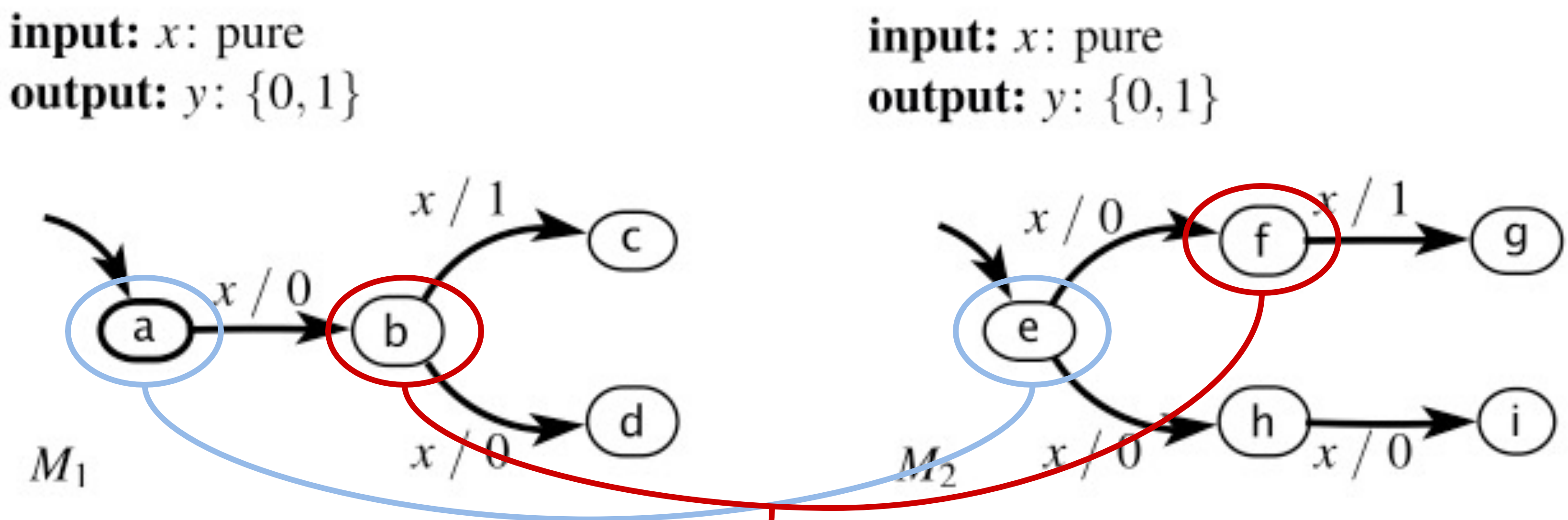
$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

M_2 moves first

$S \subseteq S_2 \times S_1$ is a **simulation relation**

$S = \{(e, a), \cdots\}$

Simulation Relation: The Matching Game



M_1 simulates M_2.

Game: M_2 moves first, and then M_1 matches the move.

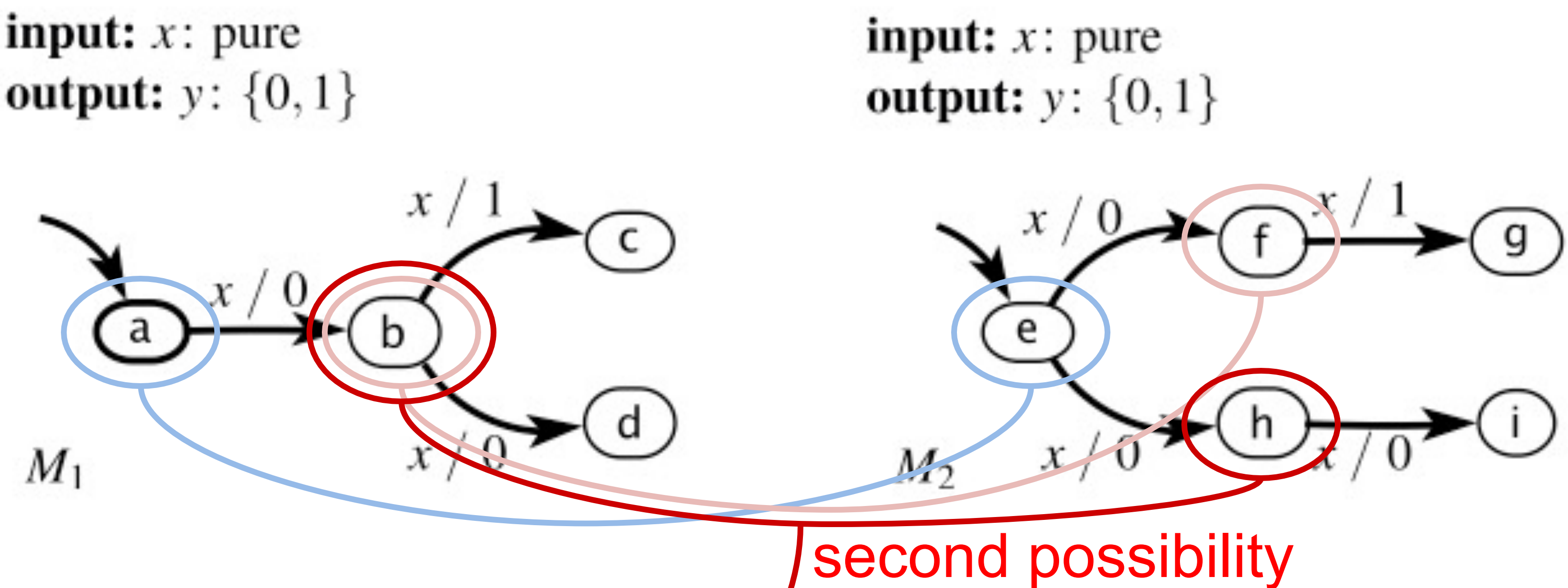
$S_1 = \{a, b, c, d\}$, $\quad S_2 = \{e, f, g, h, i\}$

M_2 moves first

$S \subseteq S_2 \times S_1$ is a **simulation relation**

$S = \{(e, a), (f, b), \cdots\}$

Simulation Relation: The Matching Game



M_1 simulates M_2.

Game: “matching” the move: same input, same output.

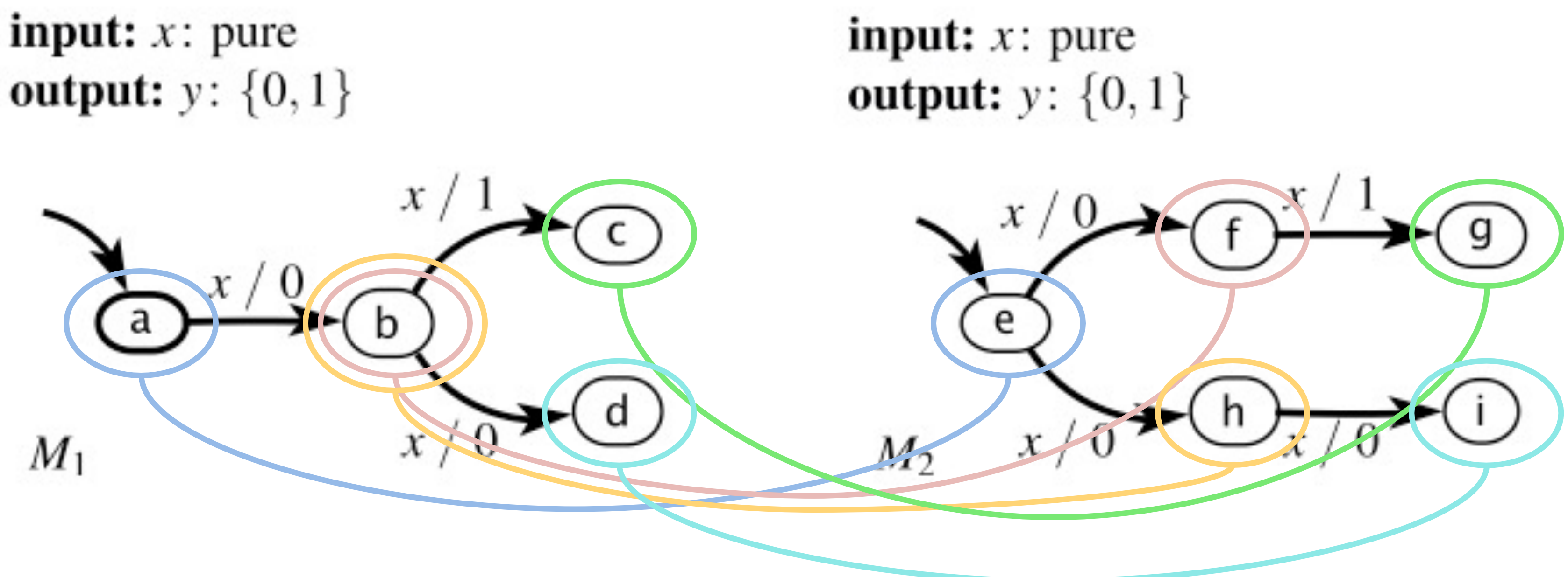
$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

M_2 moves first

$S \subseteq S_2 \times S_1$ is a **simulation relation**

$S = \{(e, a), (f, b), (h, b), \cdots\}$

Simulation Relation: The Matching Game



M_1 simulates M_2.

Game: Get to all reachable states of M_2.

$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

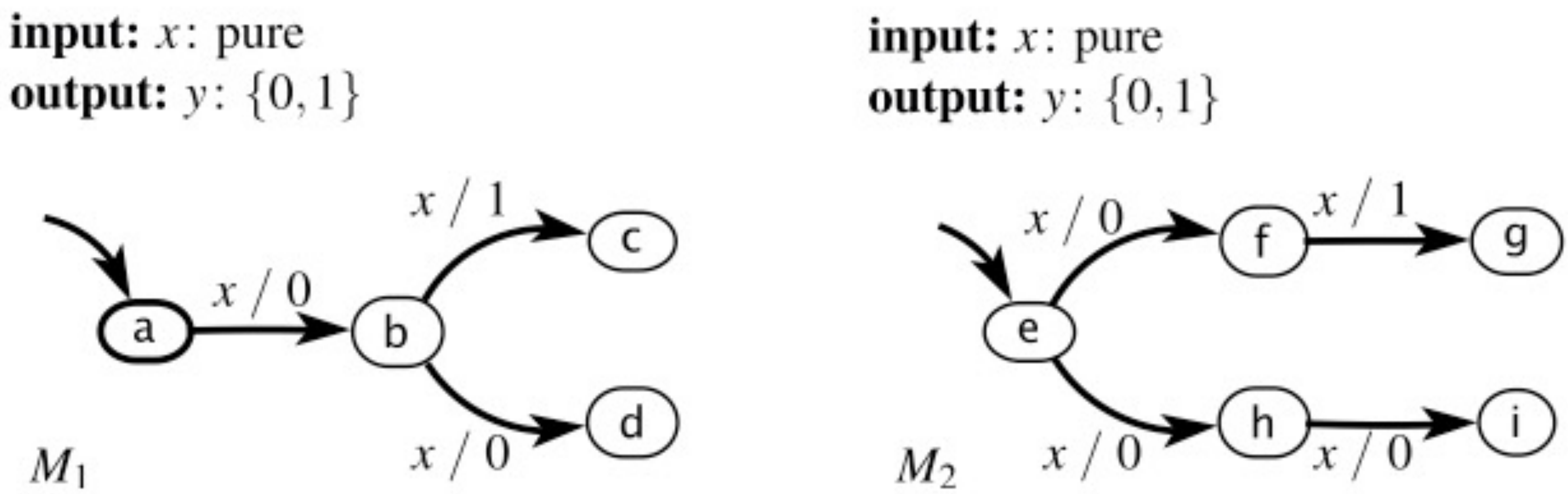
M_2 moves first

$S \subseteq S_2 \times S_1$ is a **simulation relation**

the simulation relation

$S = \{(e, a), (f, b), (h, b), (g, c), (i, d)\}$

Simulation Relation: The Matching Game



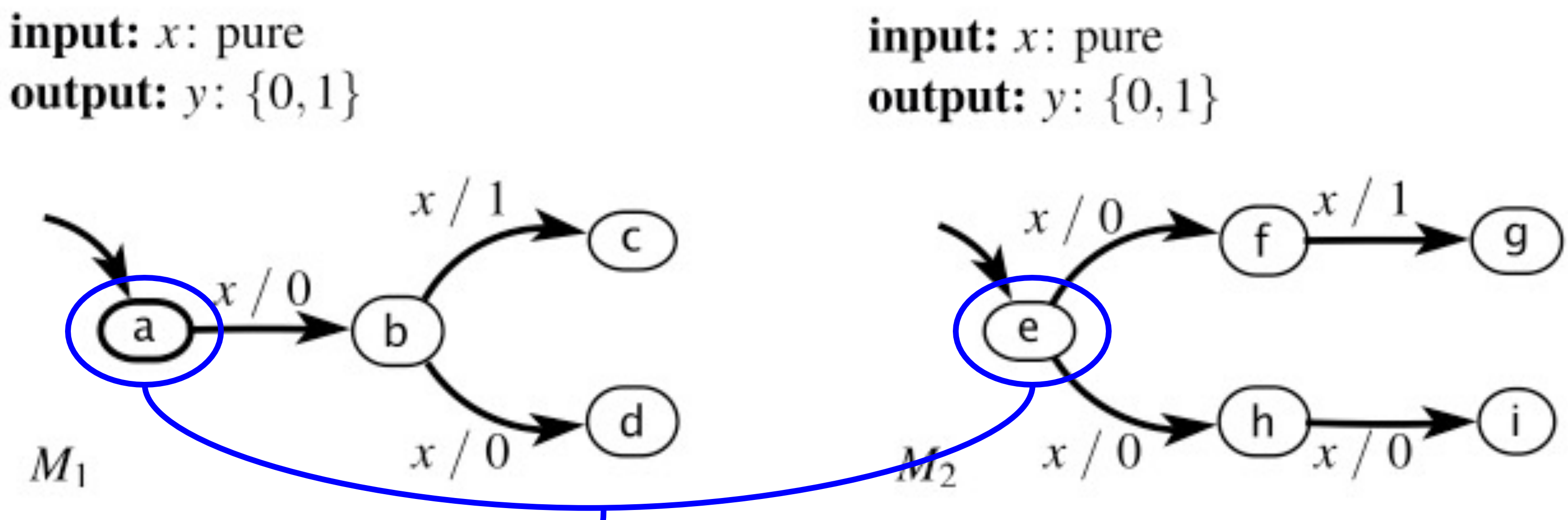
Since M_1 simulates M_2, M_2 refines M_1, M_2 can replace M_1, everywhere M_1 is OK, so is M_2.

$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

$S \subseteq S_2 \times S_1$ is a **simulation relation**

$S = \{(e,a),(f,b),(h,b),(g,c),(i,d)\}$

M_2 does not simulate M_1



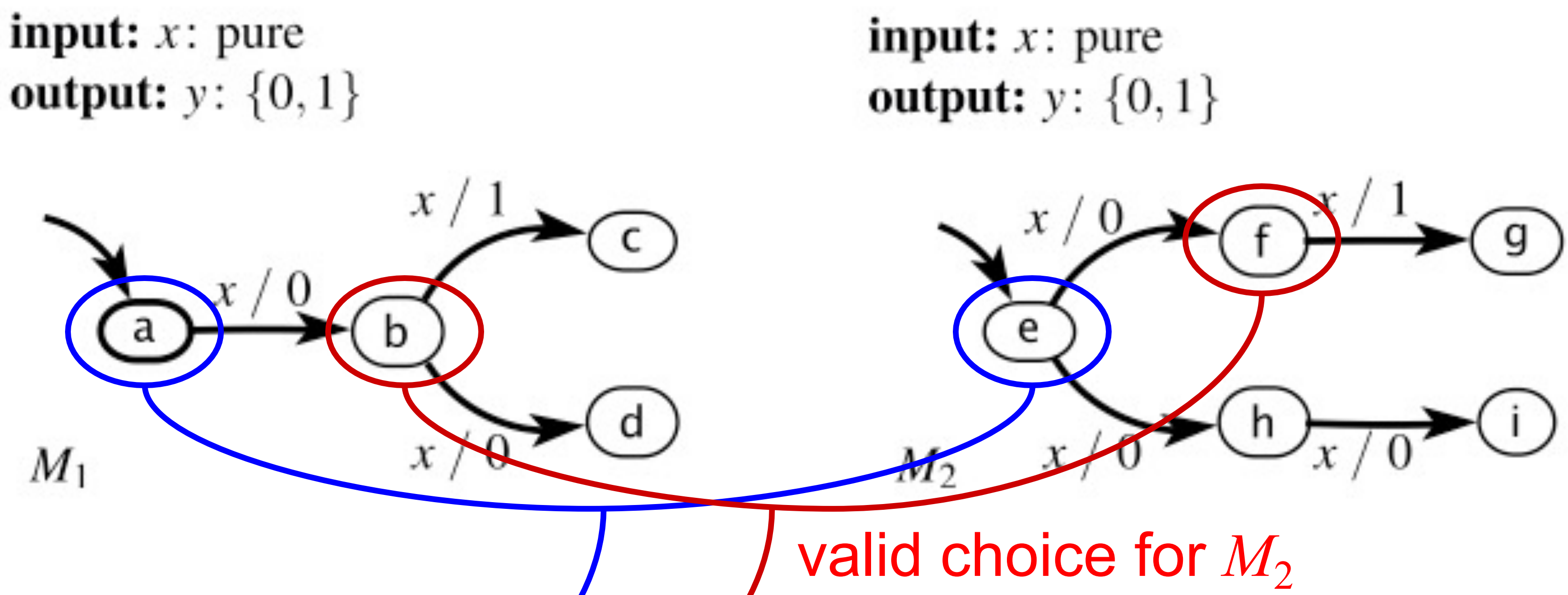
$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

We seek a **simulation relation** $S \subseteq S_1 \times S_2$.

$S = \{(a, e), \cdots\}$

M_1 moves first

M_2 does not simulate M_1



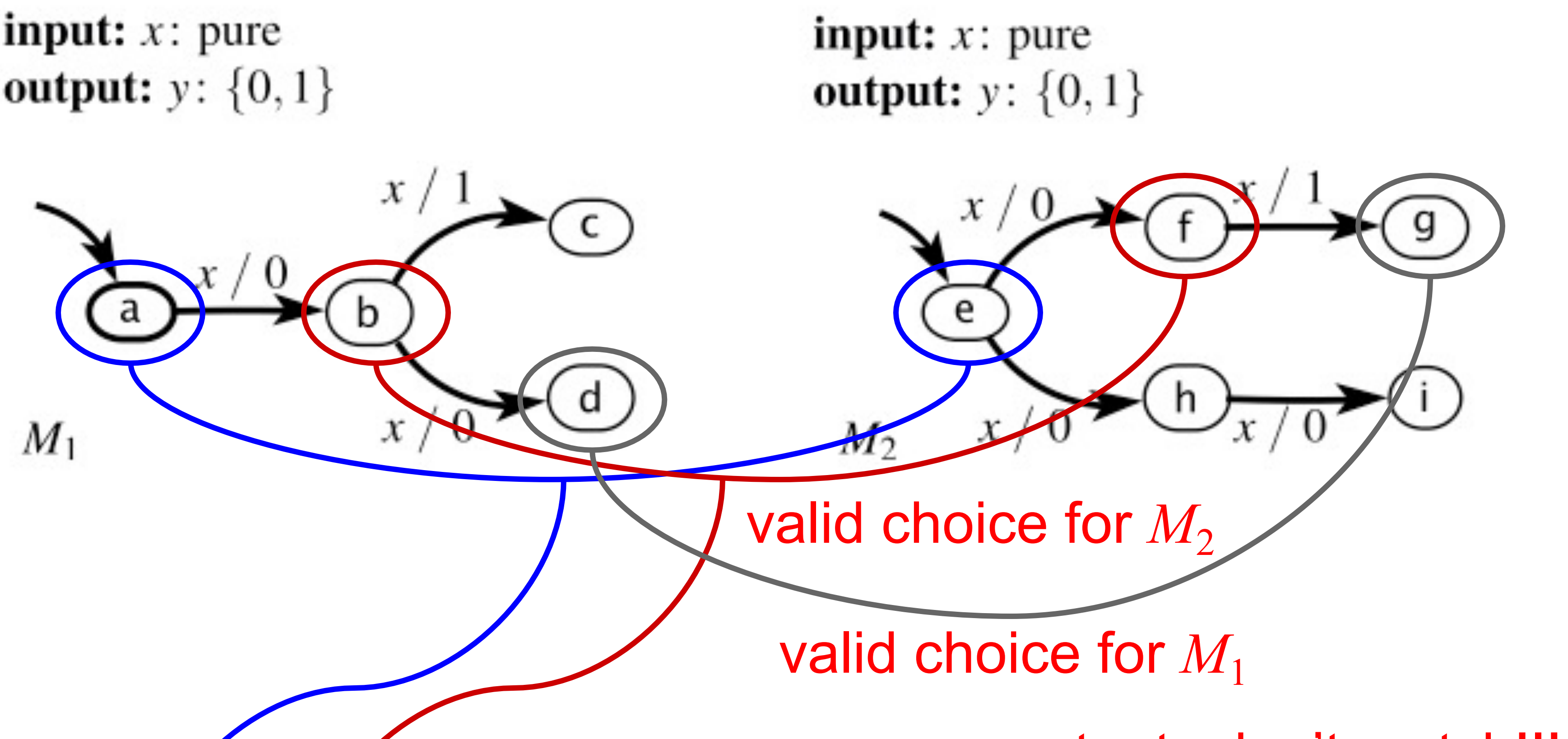
$S_1 = \{a, b, c, d\}$, $\quad S_2 = \{e, f, g, h, i\}$

We seek a **simulation relation** $S \subseteq S_1 \times S_2$.

$S = \{(a, e), (b, f), \cdots\}$

M_1 moves first

M_2 does not simulate M_1

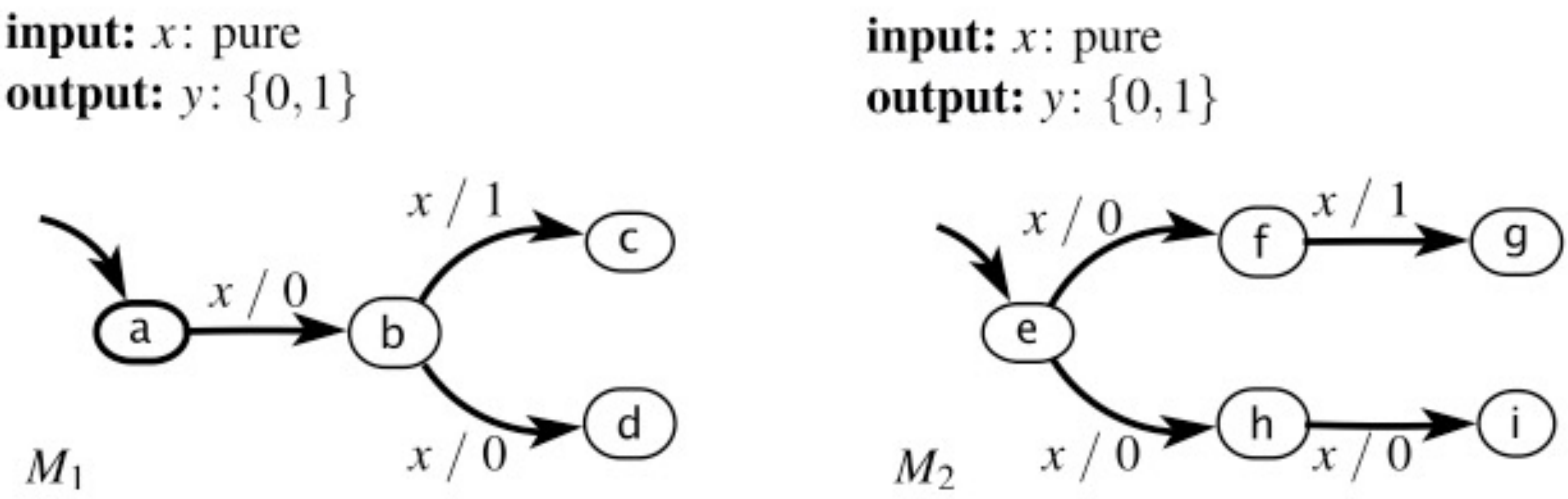


$S_1 = \{a, b, c, d\}, \quad S_2 = \{e, f, g, h, i\}$

We seek a **simulation relation** $S \subseteq S_1 \times S_2$.

$S = \{(a, e), (b, f), \}$

Simulation Relation: The Matching Game



Postponing decisions is more powerful than not.

- *Free will* is more powerful than *preordination*.
- M_1 can do anything M_2 can do, but not vice versa.
- M simulates M_2, but not vice versa.

Formal definition of Simulation

Given $M_1 = (S_1, I_1, O_1, U_1, s_{10})$ and $M_2 = (S_2, I_2, O_2, U_2, s_{20})$ where M_2 is a type refinement of M_1, M_1 simulates M_2 if there is a relation $S \subseteq S_2 \times S_1$ where:

1. $(s_{20}, s_{10}) \in S$

2. for all $(s_2, s_1) \in S$, the following condition holds:
   For all $i \in I_2$ and $(s_2', o_2) \in U_2(s_2, i)$
   there exists an $(s_1', o_1) \in U_1(s_1, i)$ such that
   $(s_2', s_1') \in S$ and $o_2 \subseteq o_1$

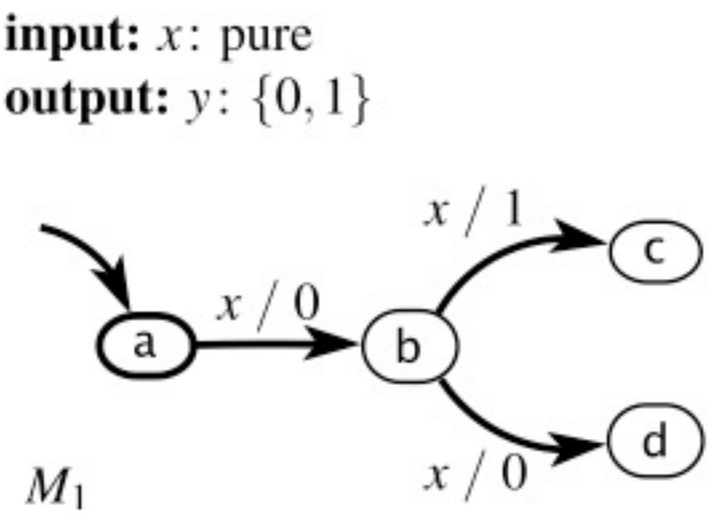
Bisimulation

A still stronger form of equivalence is called *bisimulation*.

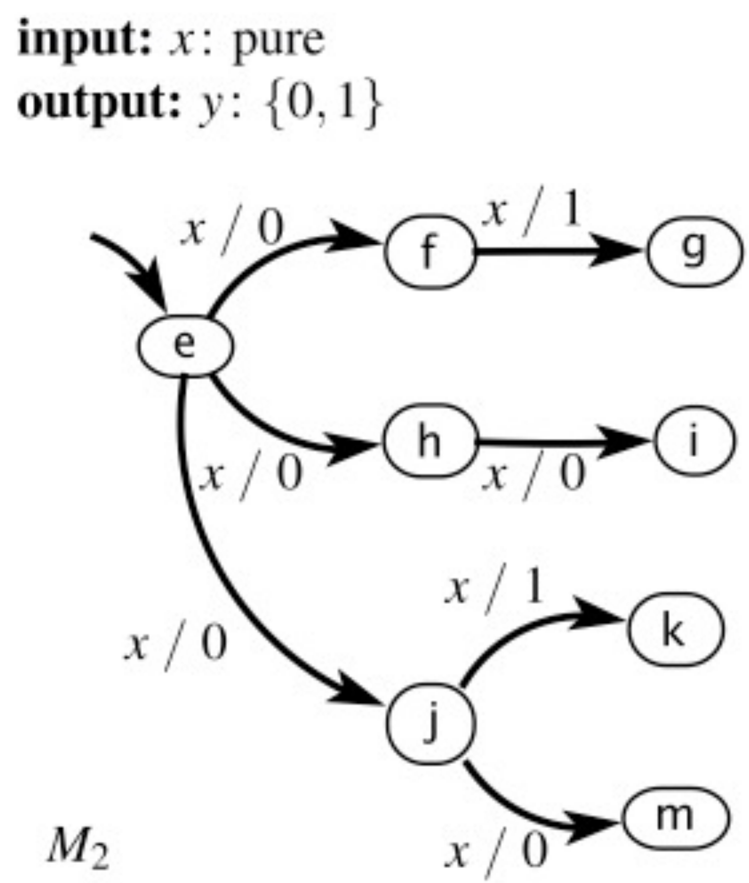
M_1 is *bisimilar* to M_2 if they are type equivalent and, when playing the game, on each move, either machine can move first, and the other machine can match its move.

Bisimulation

It is possible to have two machines that simulate each other that are not bisimilar.

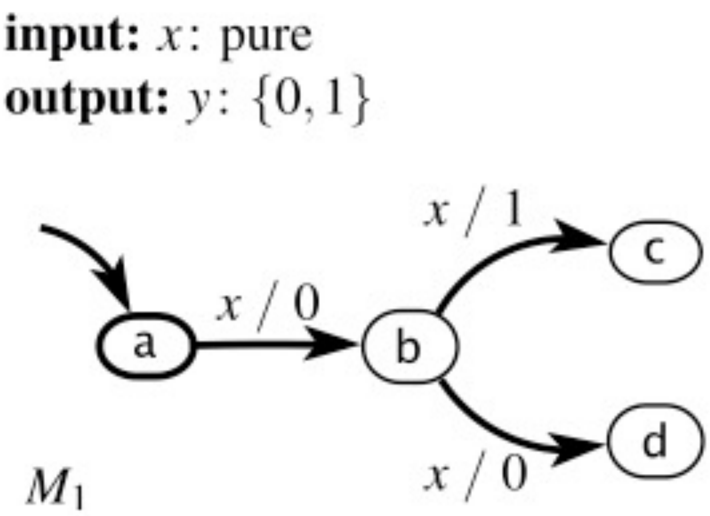


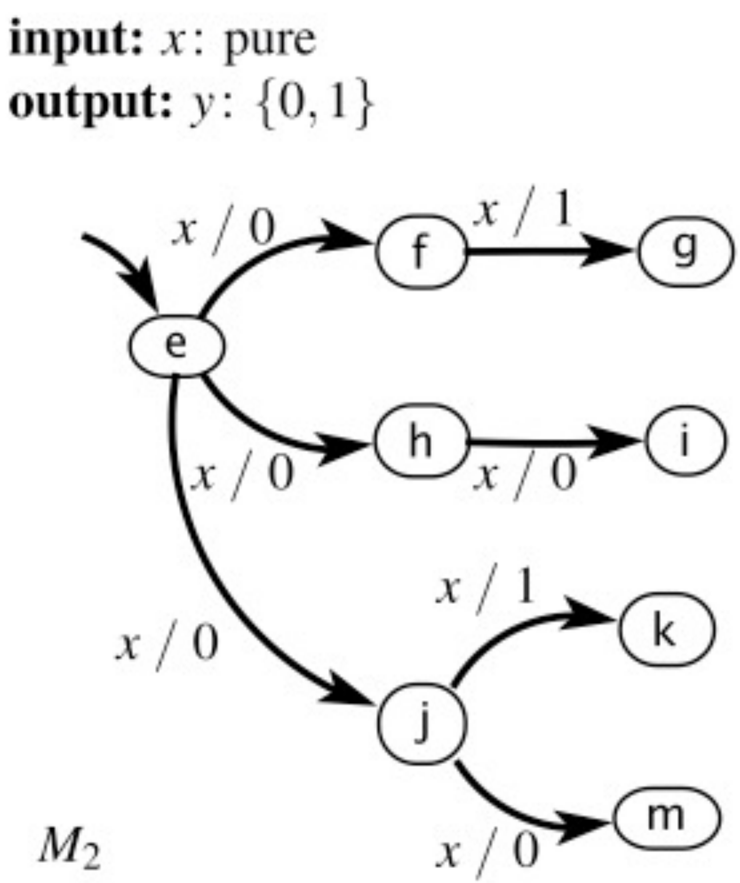
M_1 simulates M_2 and vice versa, but they are not bisimilar.



Bisimulation

It is possible to have two machines that simulate each other that are not bisimilar.





Having the ability to make decisions early does not hurt you unless you exercise it.

Bisimulation, Formally

Given $M_1 = (S_1, I, O, U_1, s_{10})$ and $M_2 = (S_2, I, O, U_2, s_{20})$, M_1 is **bisimilar** to M_2 if there is a relation $S \subseteq S_2 \times S_1$ where:

1. $(s_{20}, s_{10}) \in S$
2. for all $(s_2, s_1) \in S$, the following condition holds:
   For all $i \in I$ and $(s_2', o_2) \in U_2(s_2, i)$
   there exists an $(s_1', o_1) \in U_1(s_1, i)$ such that
   $(s_2', s_1') \in S$ and $o_2 = o_1$
   and
   For all $i \in I$ and $(s_1', o_1) \in U_1(s_1, i)$
   there exists an $(s_2', o_2) \in U_2(s_2, i)$ such that
   $(s_2', s_1') \in S$ and $o_2 = o_1$.

Simulation and Trace Containment

Theorem: If M_1 simulates M_2, then $L(M_2) \subseteq L(M_1)$.

Note: If $L(M_2) \subseteq L(M_1)$, it is not necessarily the case that M_1 simulates M_2.

Summary

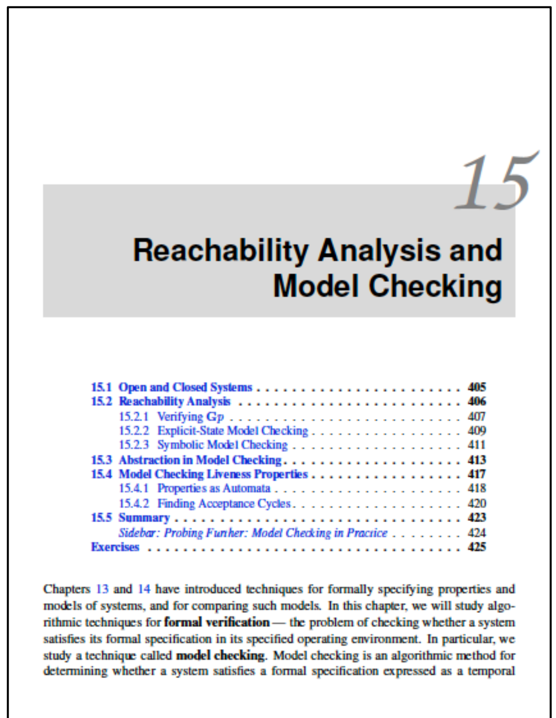
- M_2 is a type refinement of M_1:
  M_2 can replace M_1 without causing a type conflict.
- M_2 is a language refinement of M_1:
  M_2 can produce only output sequences that M_1 can produce, given the same input sequences.
- M_2 is a simulation refinement of M_1:
  (equivalently, M_1 simulates M_2)
  At every reaction, M_2 can produce only outputs that M_1 can produce.
- M_2 is bisimilar to M_1:
  At every state, either machine can produce only outputs that the other can produce.

In all cases, if M_1 is "valid" in a system, then so is M_2, where only the meaning of "valid" varies. Alternative terminology:

- M_2 *implements* M_1 (here, M_1 is taken to be a *specification*).

Things to do …

- Next lecture: Reachability Analysis and Model Checking
- Read Chapter 15



15

Reachability Analysis and Model Checking

Chapters 13 and 14 have introduced techniques for formally specifying properties and models of systems, and for comparing such models. In this chapter, we will study algorithmic techniques for **formal verification** — the problem of checking whether a system satisfies its formal specification in its specified operating environment. In particular, we study a technique called **model checking**. Model checking is an algorithmic method for determining whether a system satisfies a formal specification expressed as a temporal