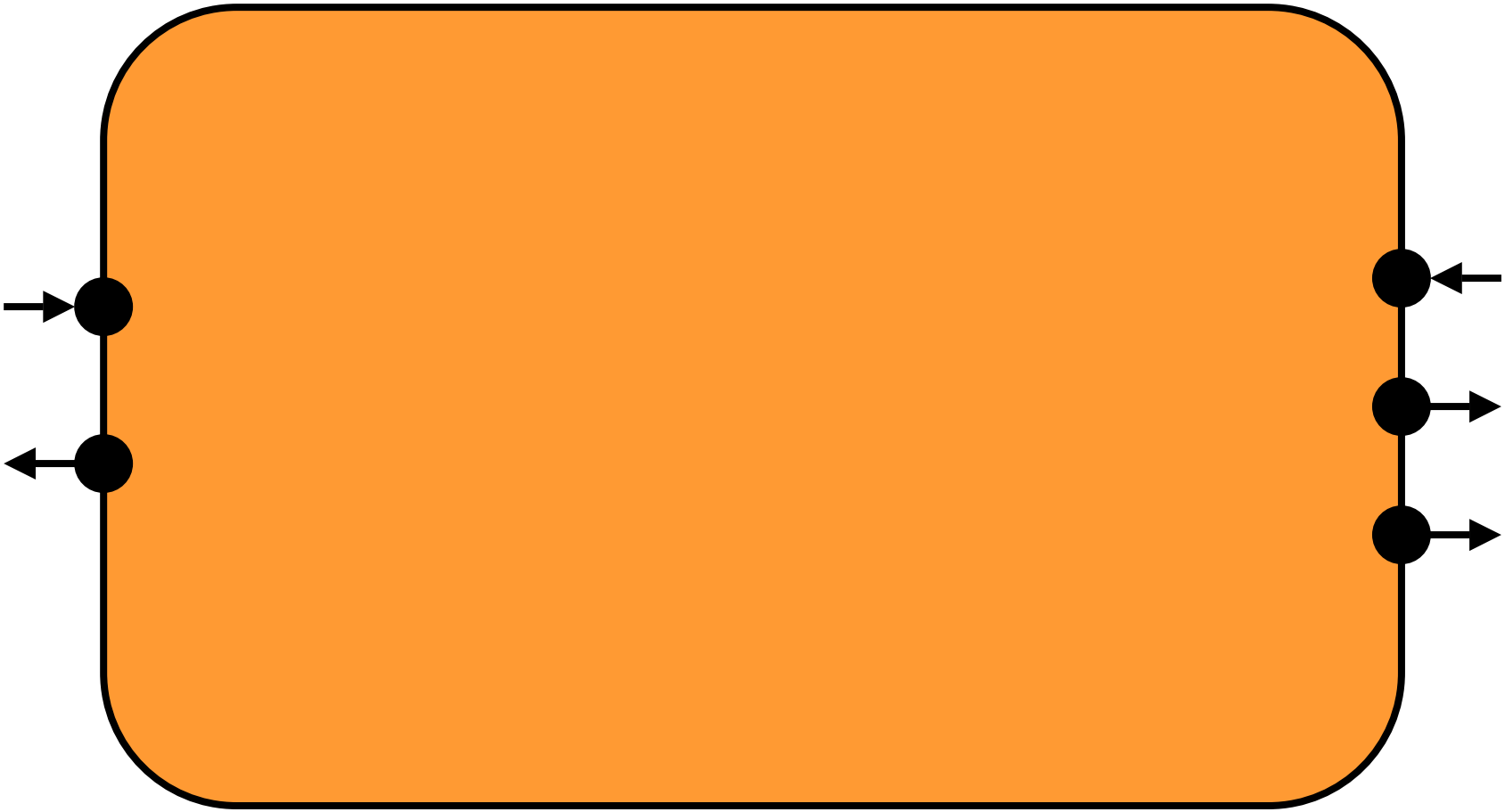
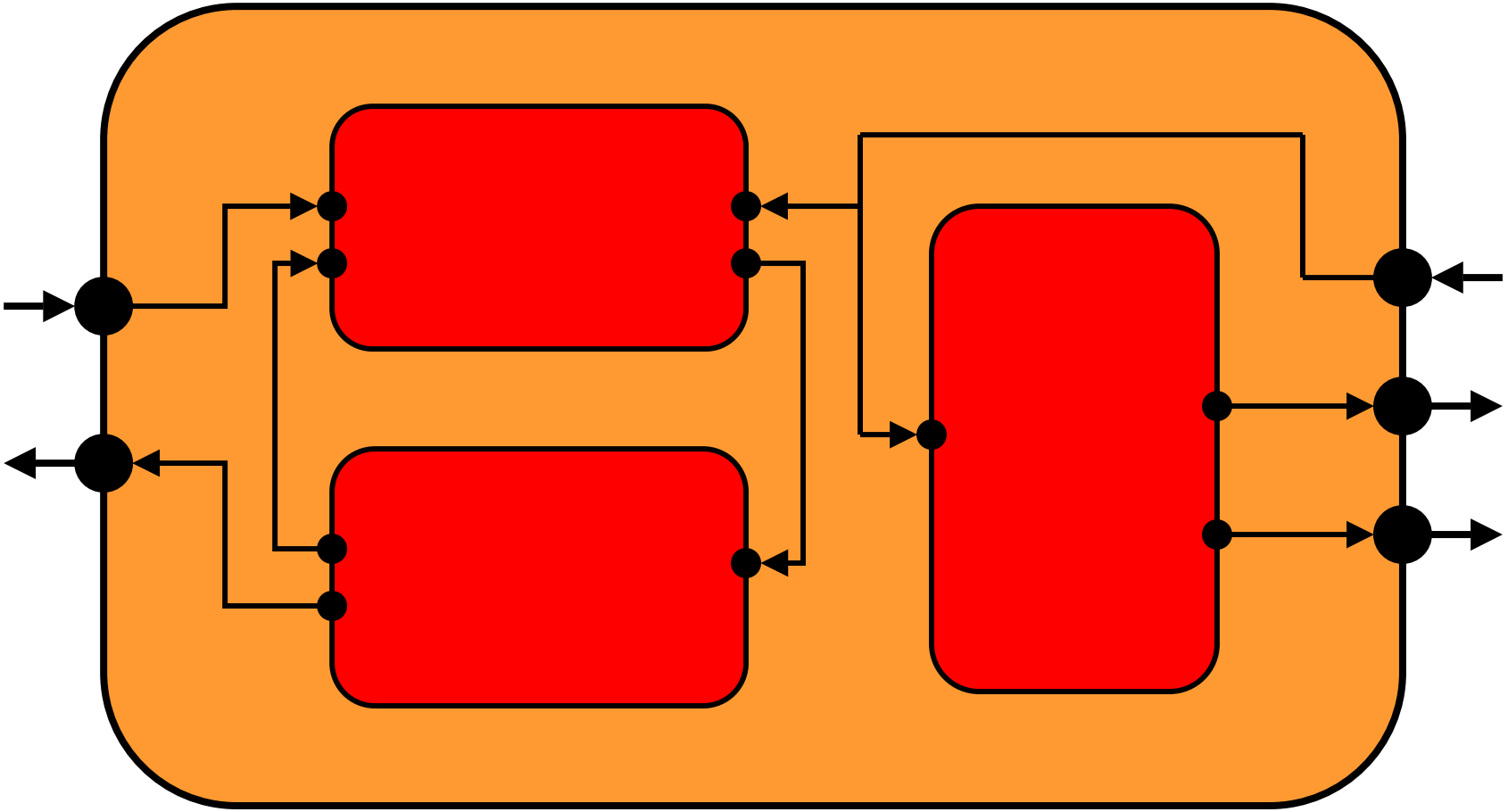
Interface-based Design of Embedded Systems

Thomas A. Henzinger
University of California, Berkeley

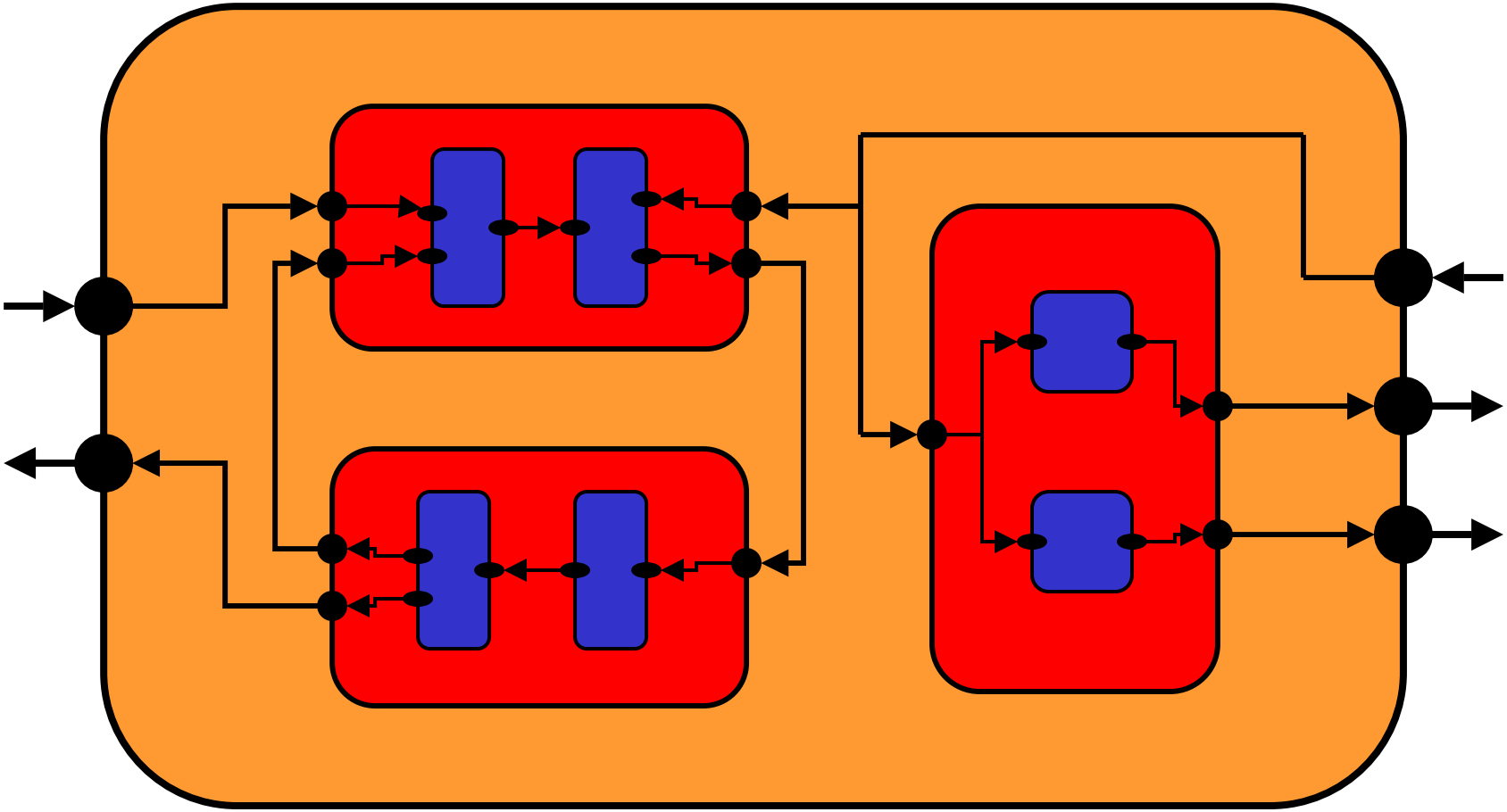
Interface-based Design



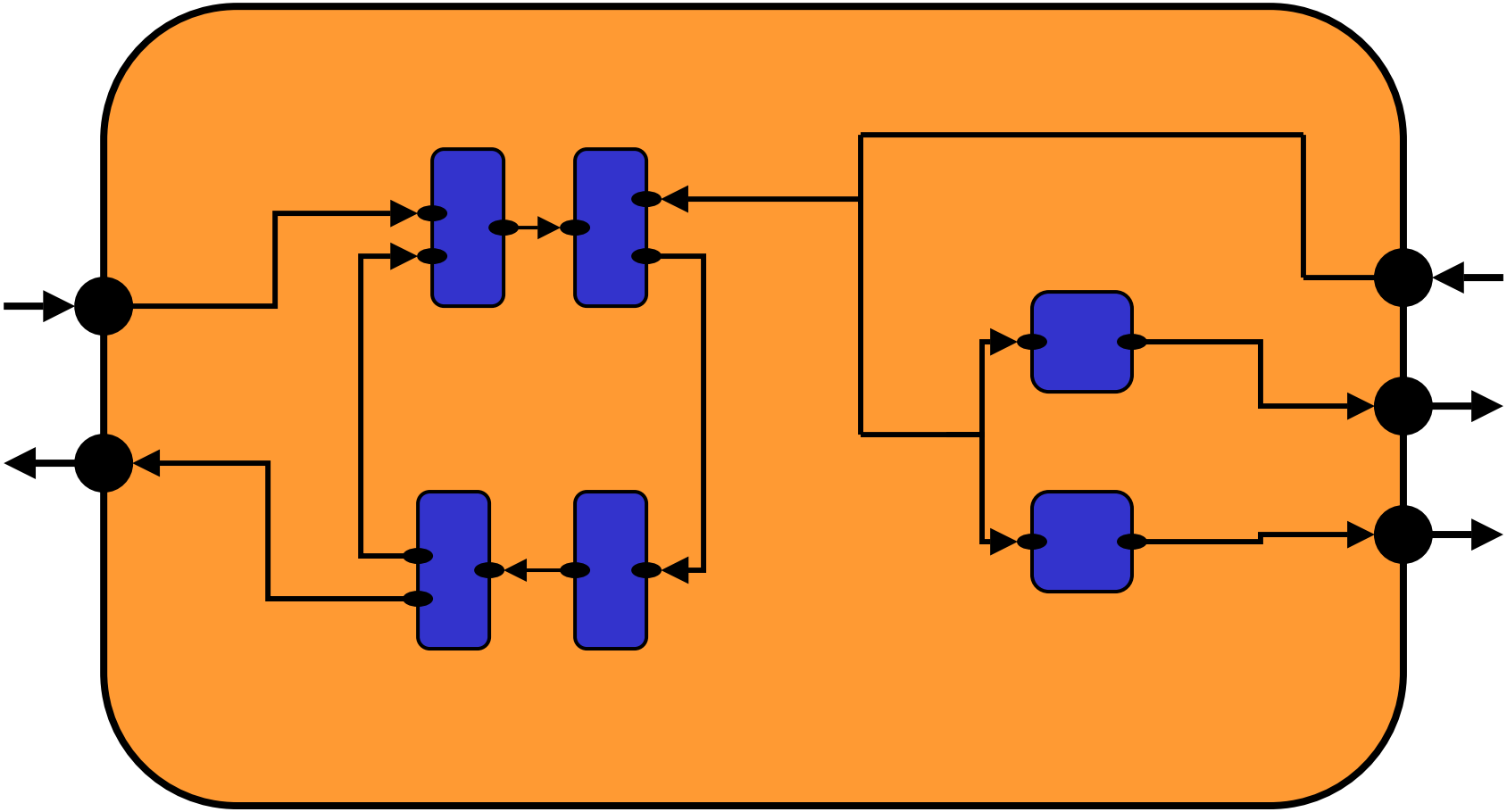
Interface-based Design



Interface-based Design



Interface-based Design



Compositional Component Models

If A||B is defined and A ≤ a and B ≤ b , then a||b is defined and A||B ≤ a||b.

enable independent component verification

Compositional Interface Models

If a||b is defined and A ≤ a and B ≤ b , then A||B is defined and A||B ≤ a||b.

enable independent interface implementation

A Component Model

$$x \in \text{Nat} \ \wedge \ y \in \text{Nat}\backslash\{0\} \ \Rightarrow \ z = x \div y$$

-(mis)behaves in every environment
-examples: circuit; executable code

An Interface Model

$$x \in \text{Nat} \ \wedge \ y \in \text{Nat}\backslash\{0\} \ \wedge \ z \in \text{Nat}$$

-constrains the environment
-example: type declaration

The Component Model

$$\forall x,y.\ \exists z.\ (\ x \in \mathrm{Nat}\ \wedge\ y \in \mathrm{Nat}\backslash\{0\}\ \Rightarrow\ z = x \div y\ )$$

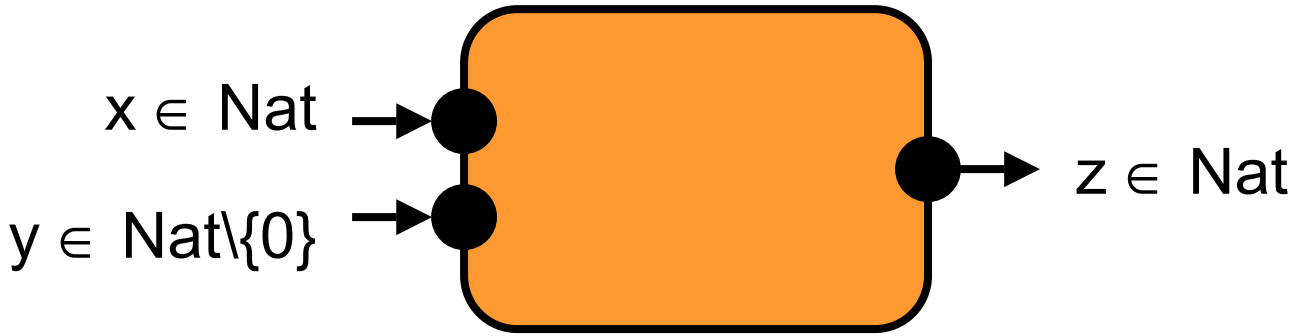
input-universal (adversarial environment)

The Interface Model

$$\exists x,y.\ \exists z.\ (\ x \in \mathrm{Nat}\ \wedge\ y \in \mathrm{Nat}\backslash\{0\}\ \wedge\ z \in \mathrm{Nat}\ )$$

input-existential (helpful environment)

The Interface Model



Input assumption

Output guarantee

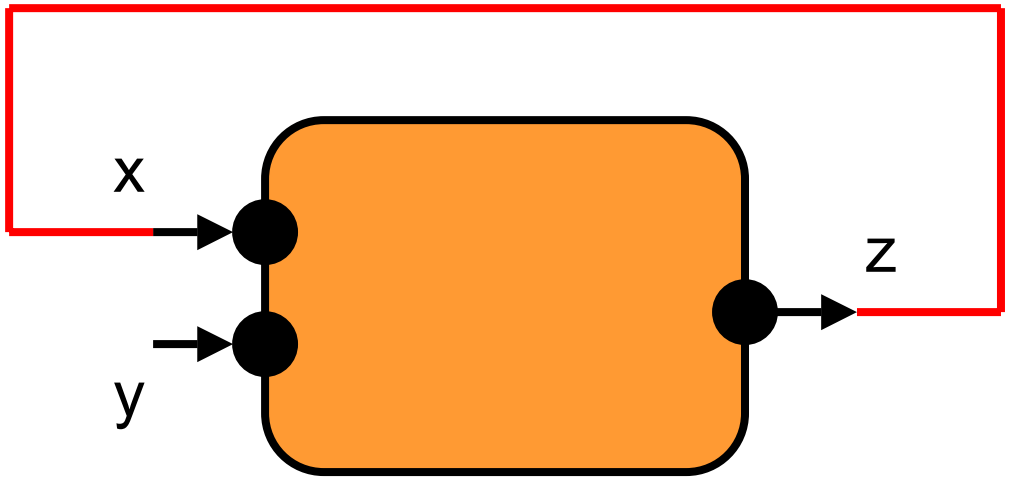
Prescriptive:

“How can the component be put together with other components?”

Propagation of Environment Constraints

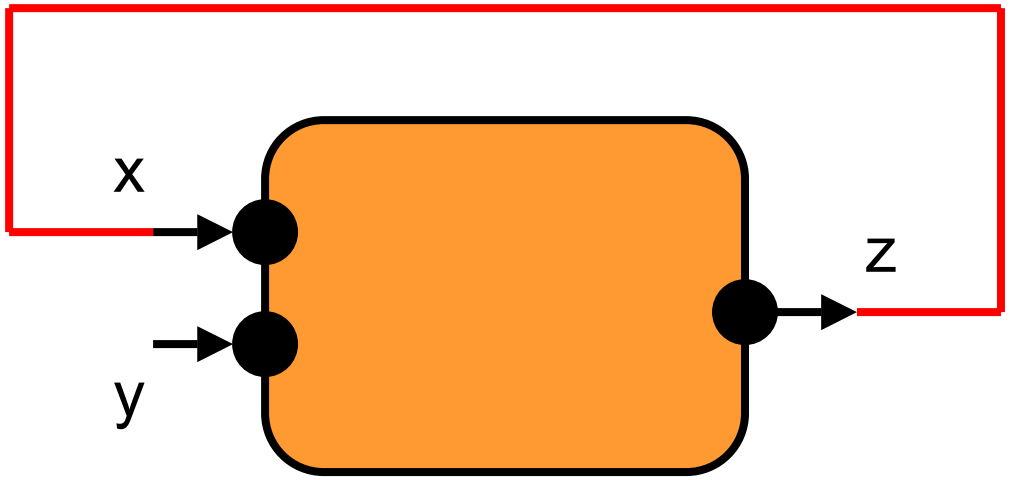
$x=0 \Rightarrow y=0$ true

Propagation of Environment Constraints



$x=0 \Rightarrow y=0$ true

Propagation of Environment Constraints



$x=0 \Rightarrow y=0$ true

$y = 0$

$\forall x,z.\ (\ \text{true} \wedge x=z \Rightarrow (\ x=0 \Rightarrow y=0\ ))$

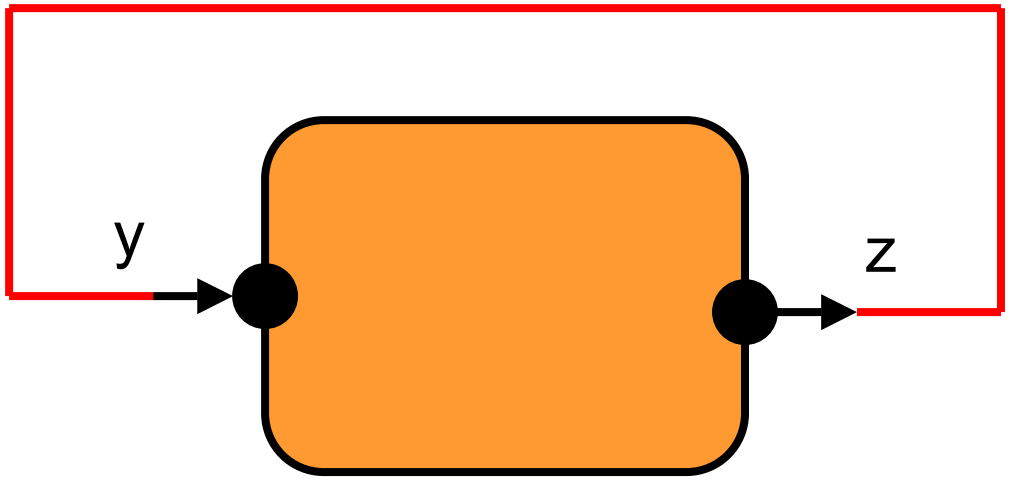
Propagation of Environment Constraints

y = 0

true

The resulting interface.

Propagation of Environment Constraints



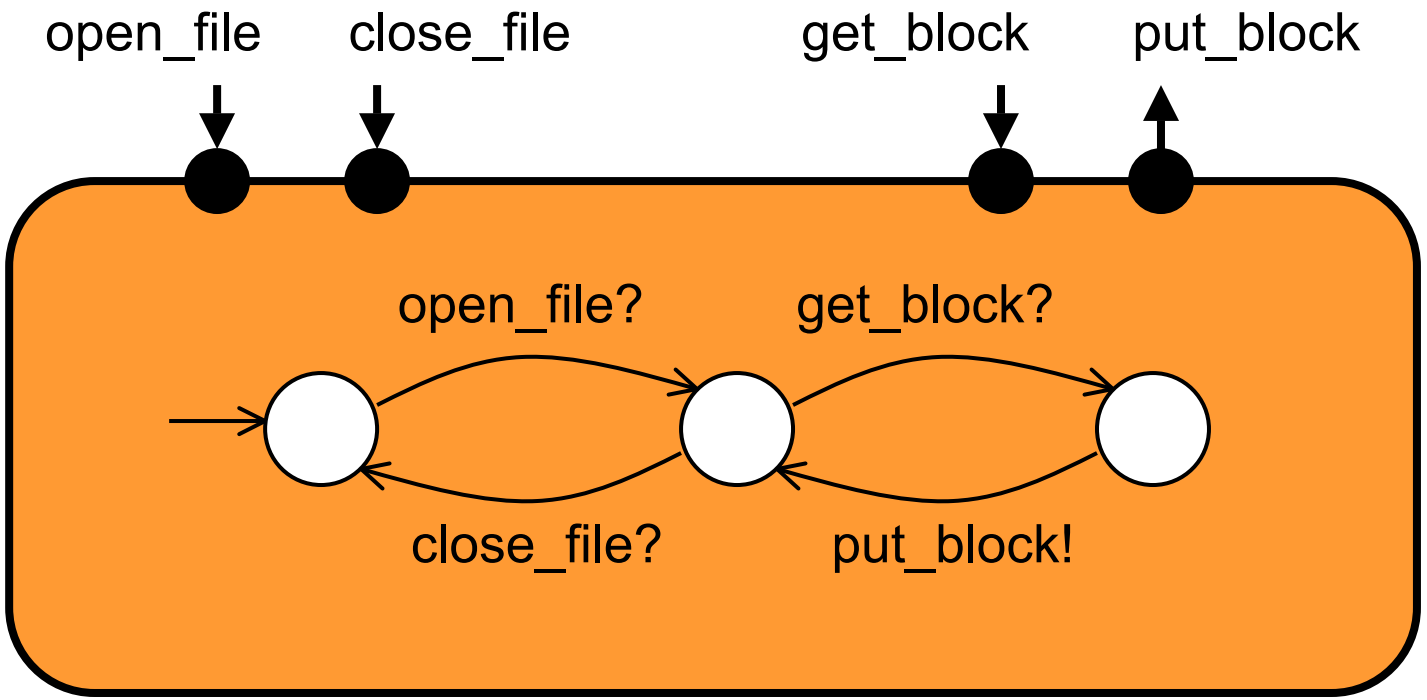
y = 0

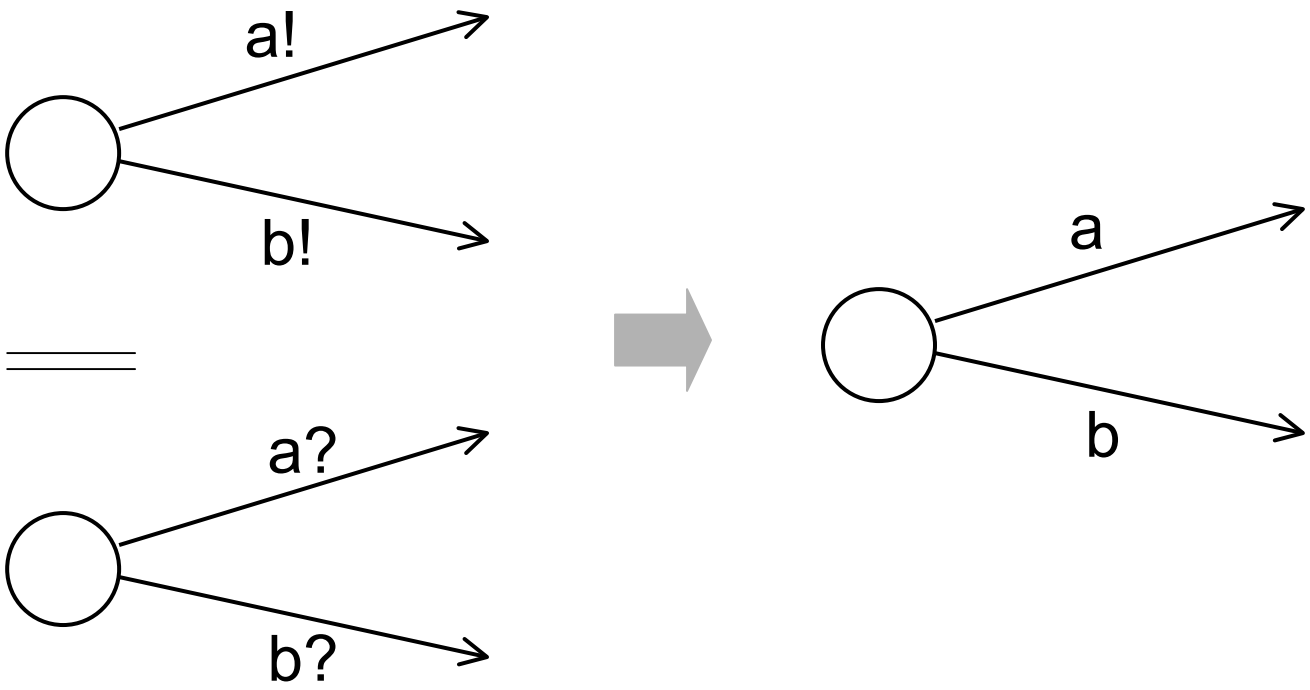
true

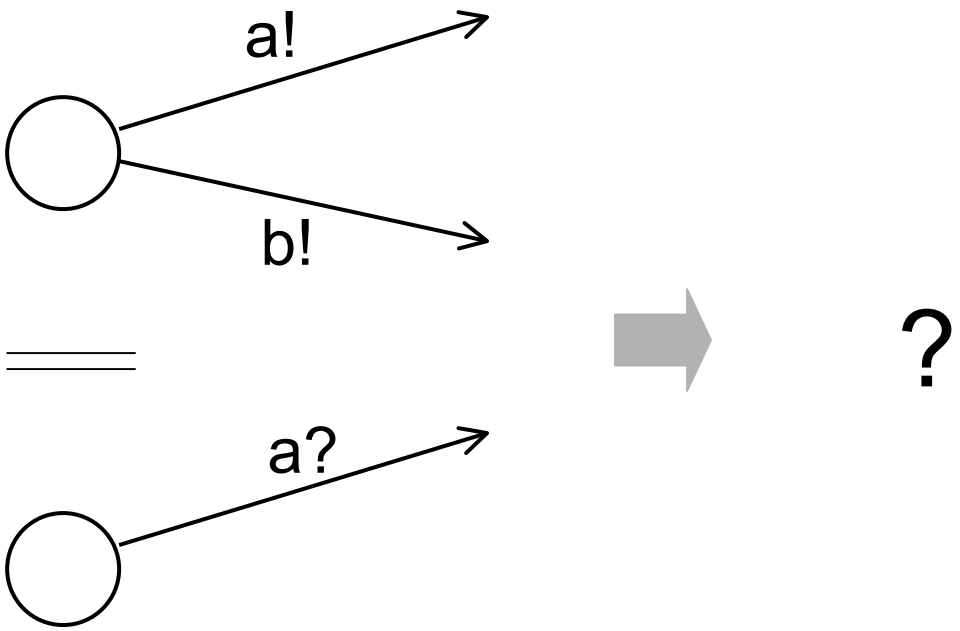
Illegal connection.

Stateless interface models (traditional “types”):
value constraints

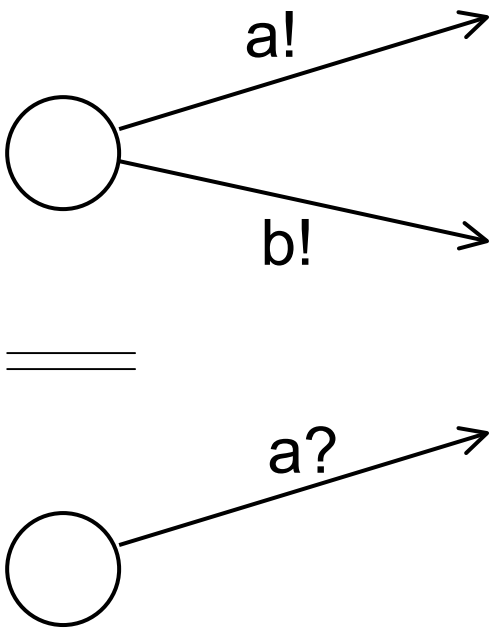
Stateful interface models (“behavioral types”):
temporal ordering constraints, real-time constraints, etc.




a!
b!
a?
b?
a
b


a!
b!
a?
?

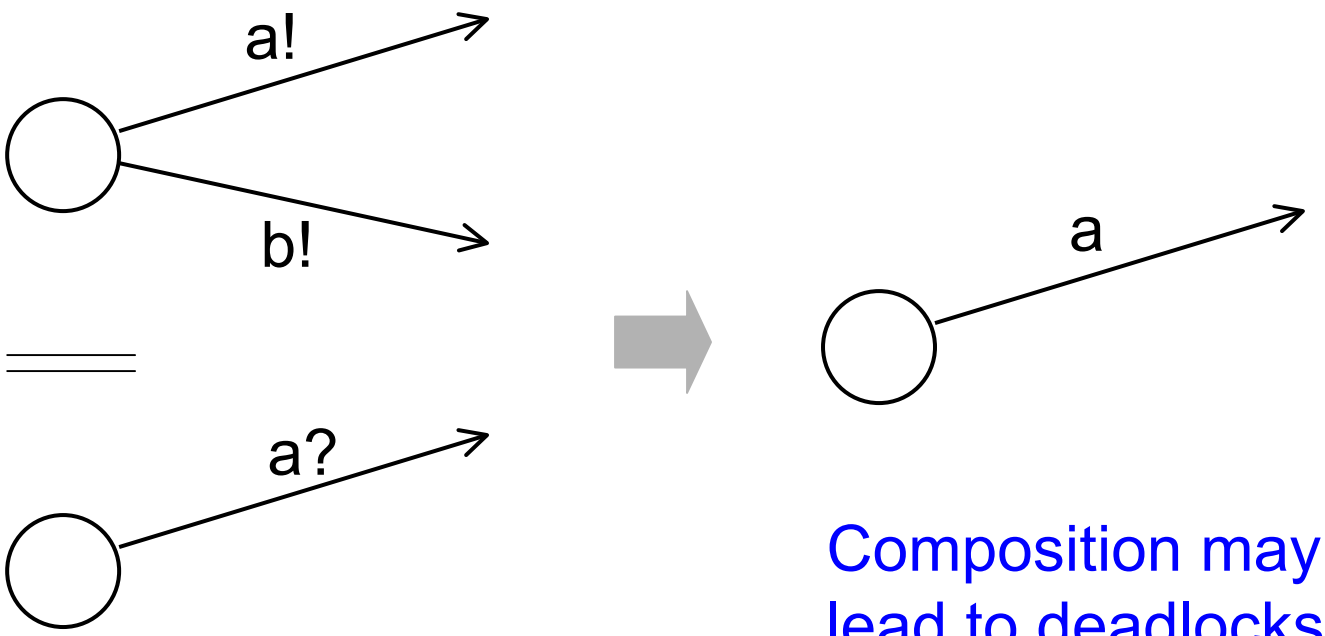
A Component Model: I/O Automata



This is an illegal component, because it is not prepared to accept input b.

[Lynch, also Lamport, Alur/H]

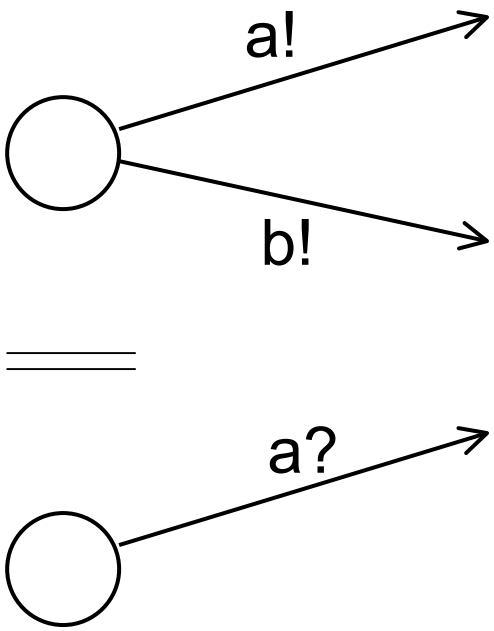
Another Component Model: CSP



Composition may lead to deadlocks, and requires verification if this is undesirable.

[Hoare, also Milner, Harel]

An Interface Model: Interface Automata



These interfaces are incompatible, because the receiver expects the environment to provide input b.

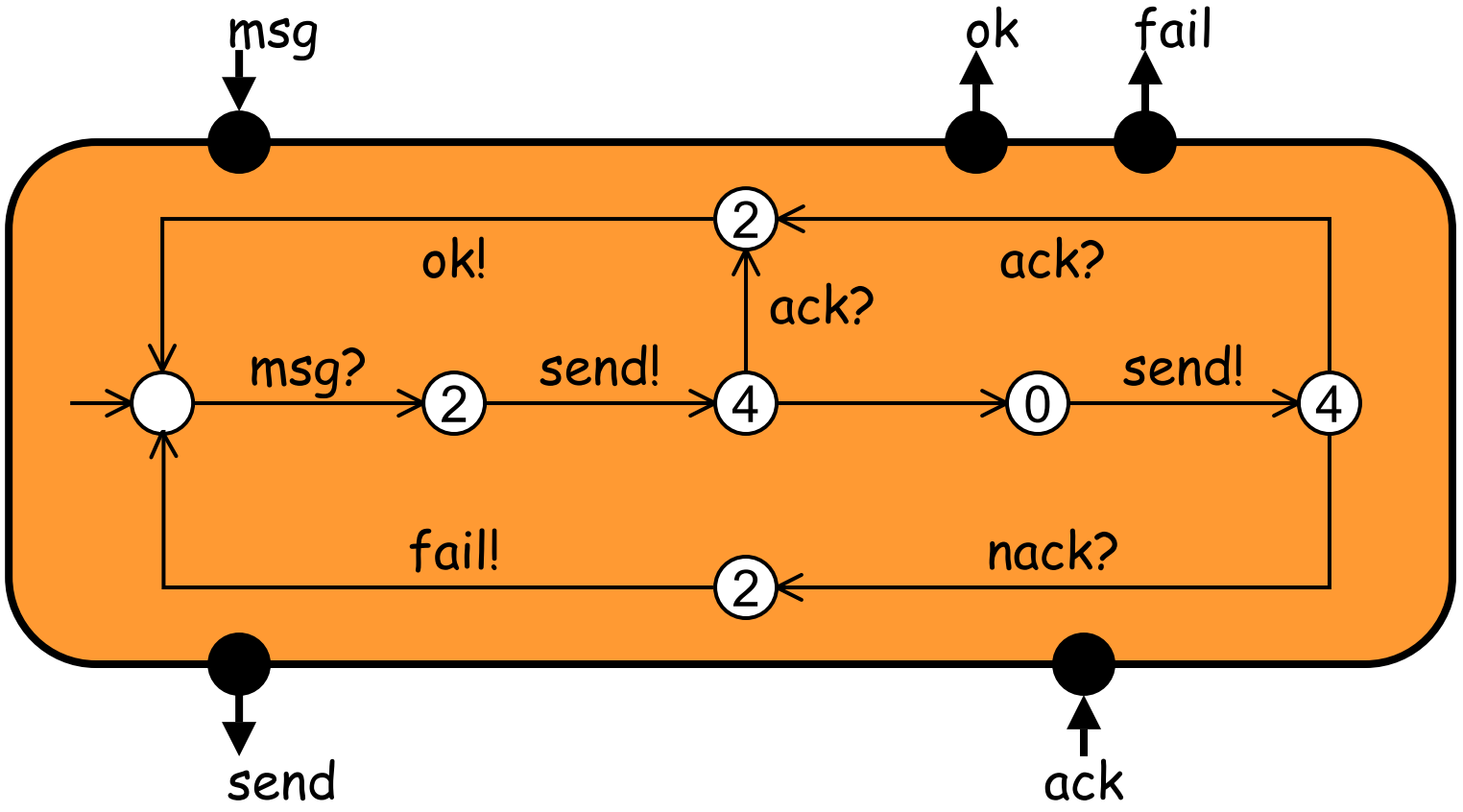
[de Alfaro/H, also Dill]

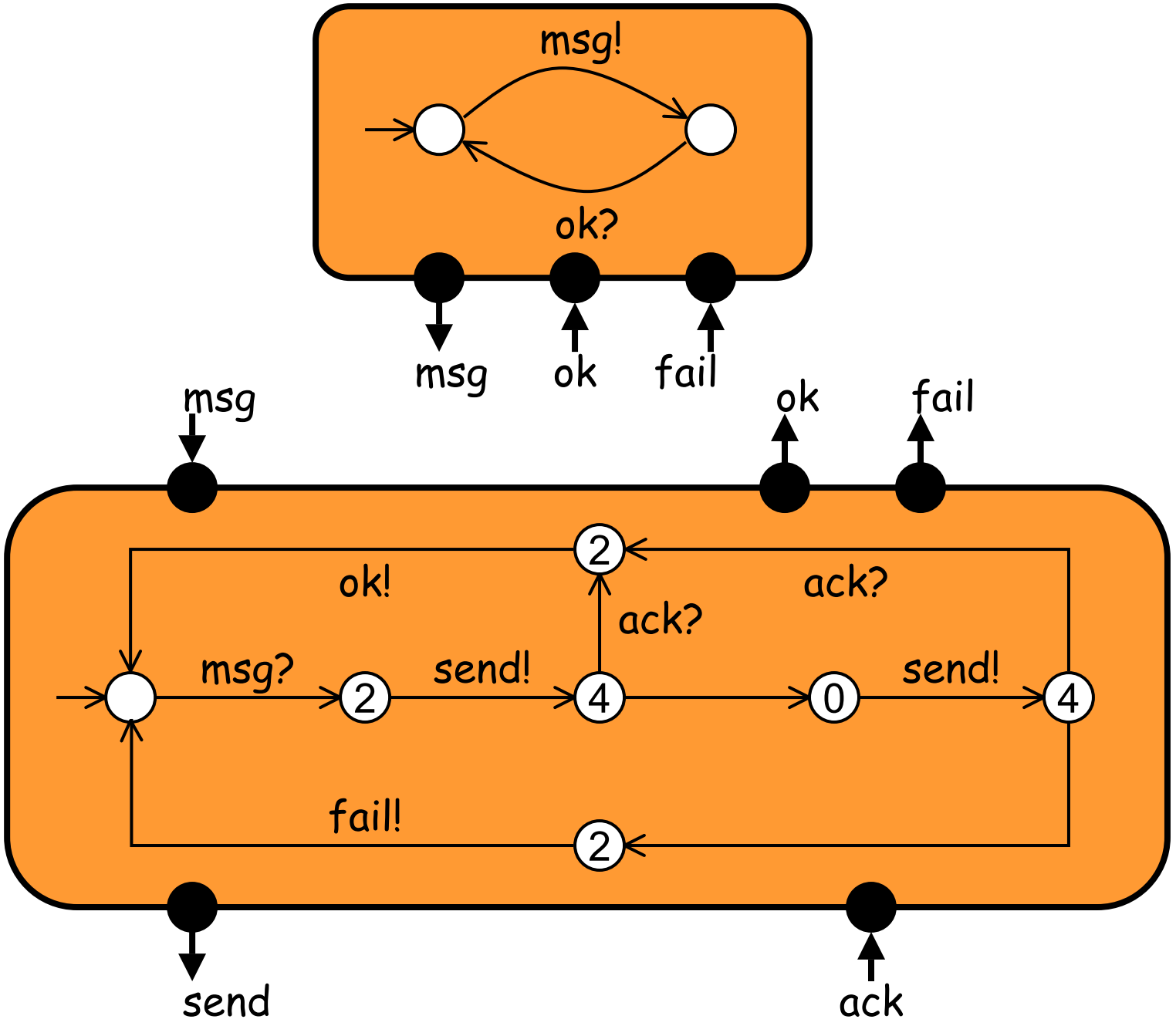
Component Models

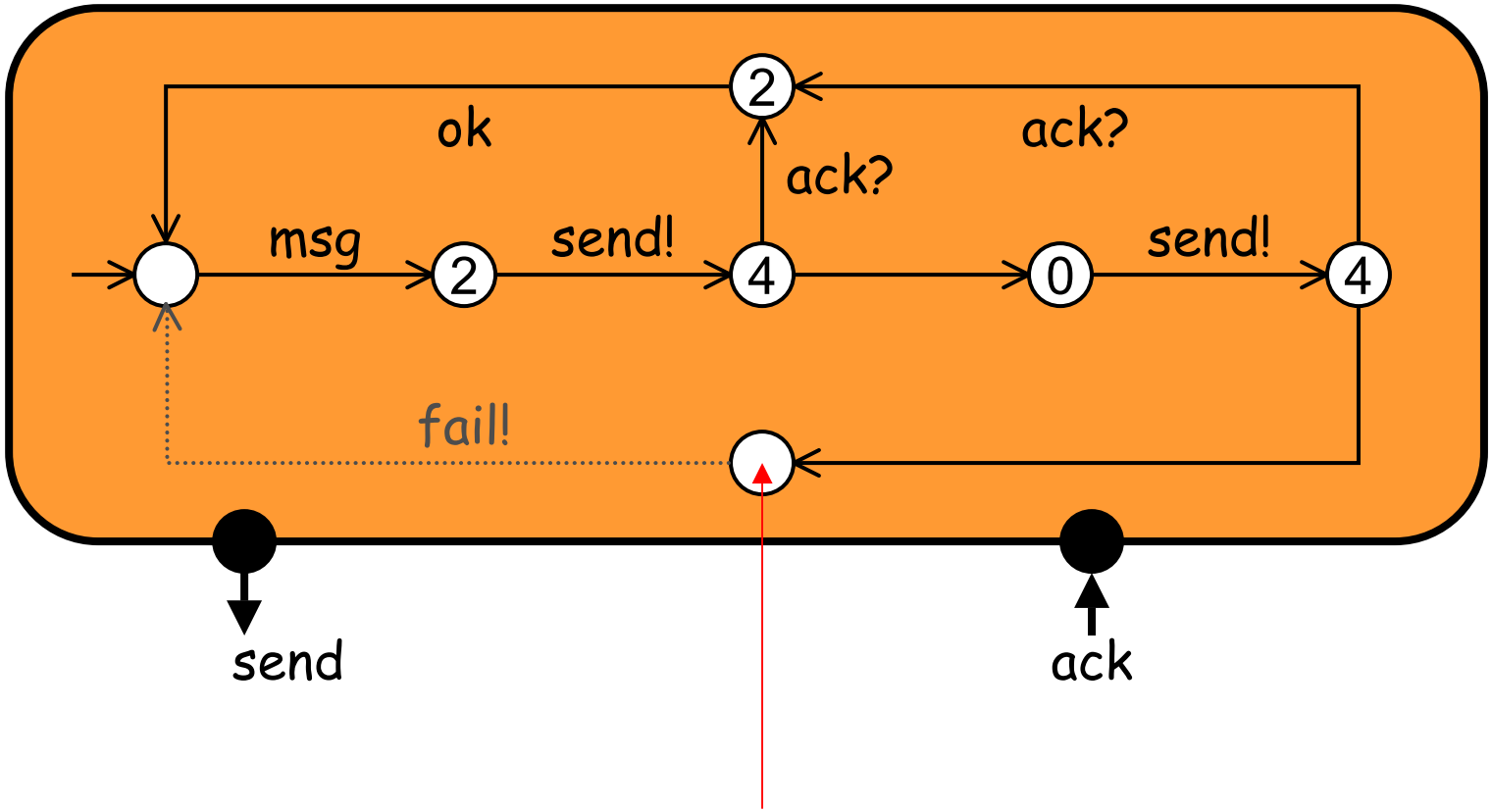
-composition || is conjunction/product
-abstraction $\leq$ is covariant

Interface Models

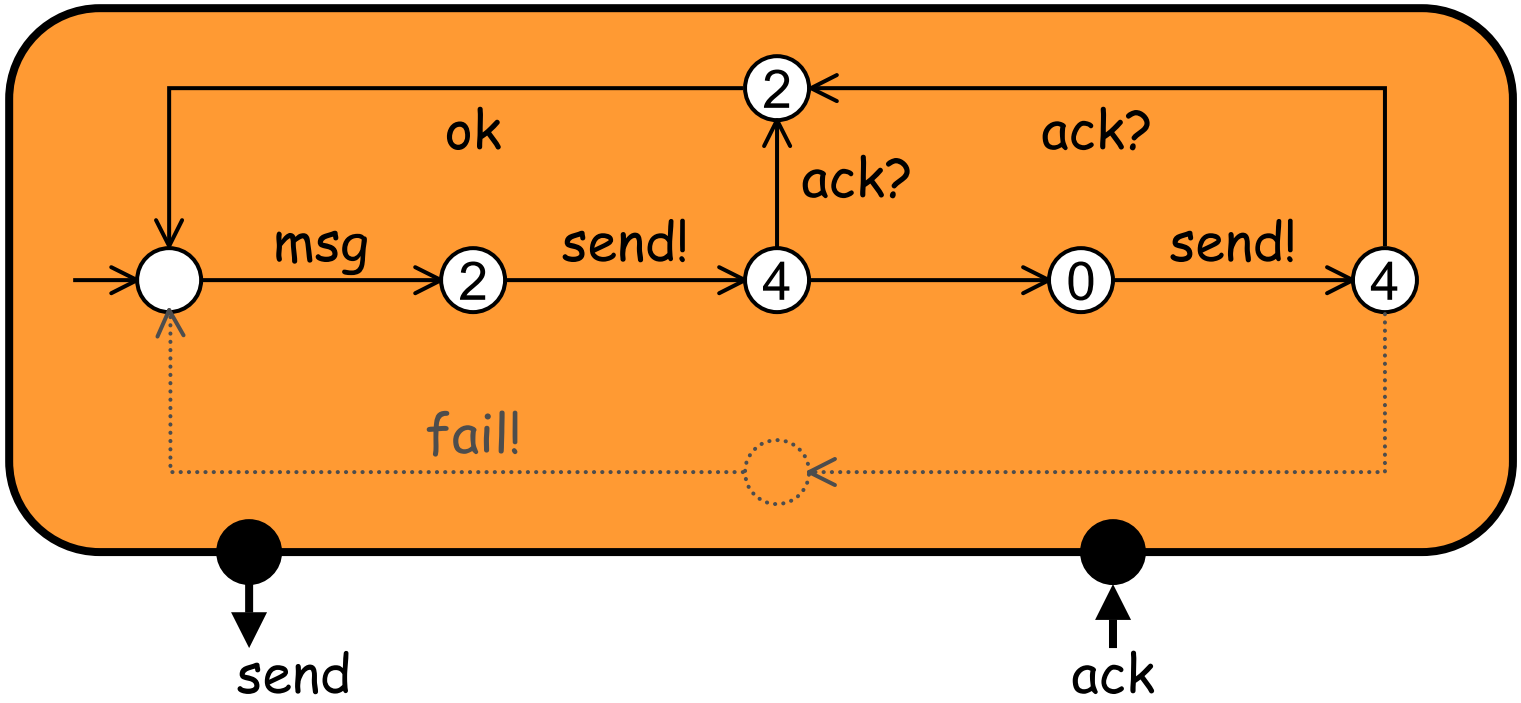
-composition || is game-theoretic
-implementation $\leq$ is contravariant


msg
ok
fail
2
ok!
ack?
ack?
msg?
2
send!
4
0
send!
4
fail!
nack?
2
send
ack

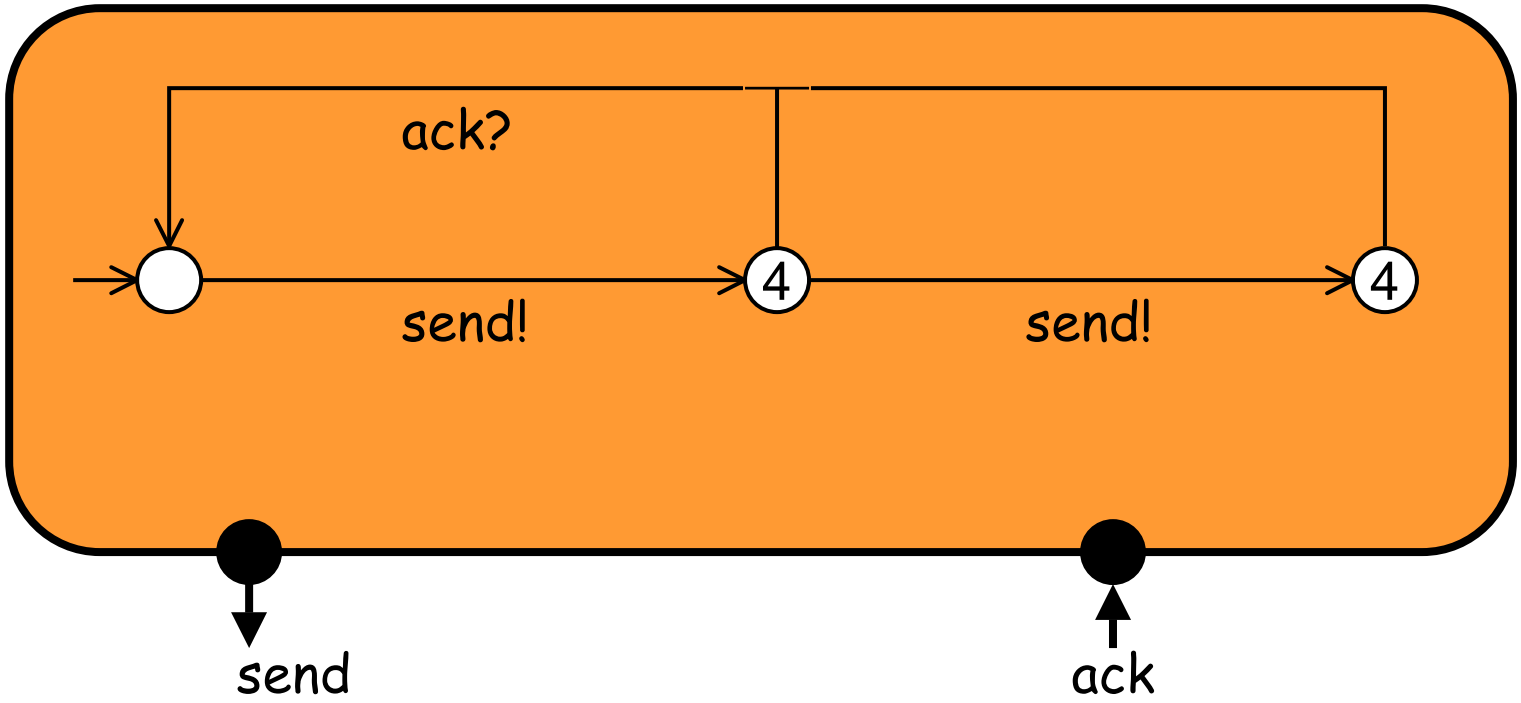

msg!
ok?
msg
ok
fail
msg
ok
fail
2
ok!
ack?
ack?
msg?
2
send!
4
0
send!
4
fail!
2
send
ack



Incompatible product state, but environment can prevent this state.



The Composite Interface.



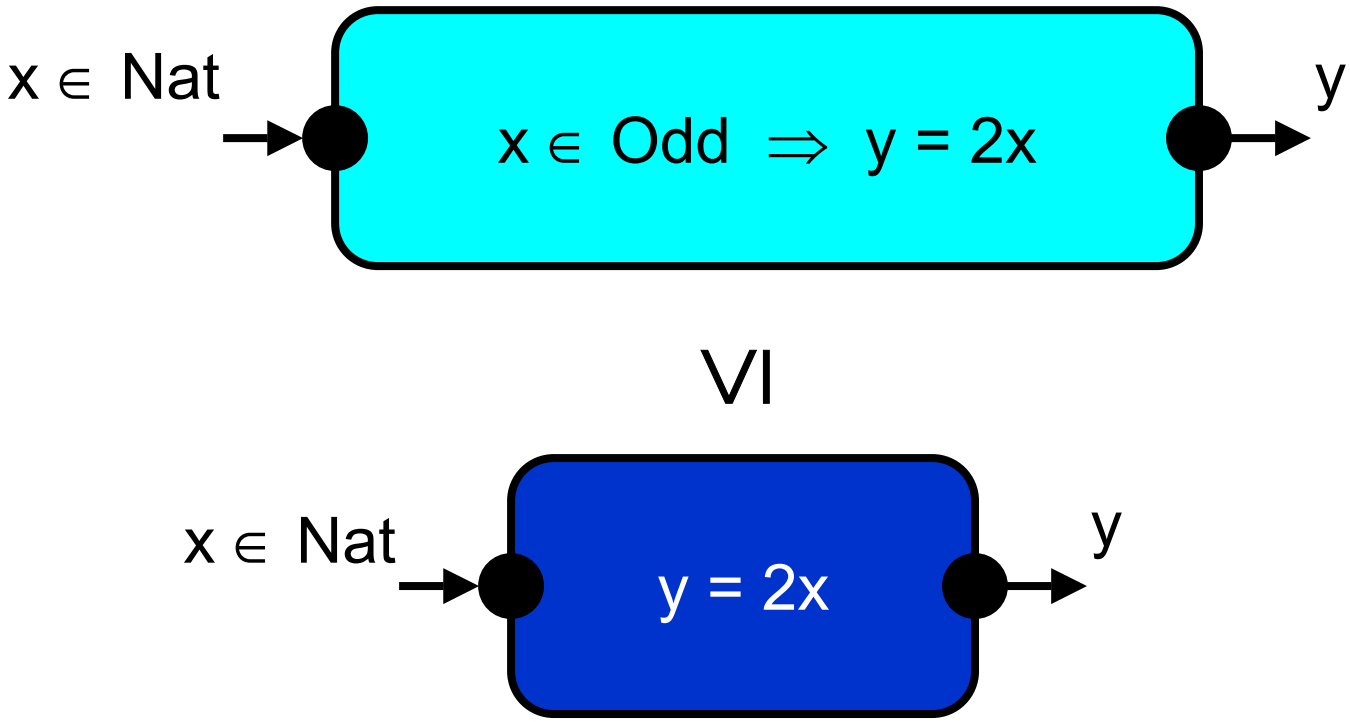
The Composite Interface.

Computing the Composite Interface

- Construct product automaton.
- Mark deadlock states as incompatible.
- Until no more incompatible states can be added: mark state q as incompatible if the environment cannot prevent an incompatible state to be entered from q.
- If the initial state is incompatible, then the two interfaces are incompatible. Otherwise, the composite interface is the product automaton without the incompatible states.

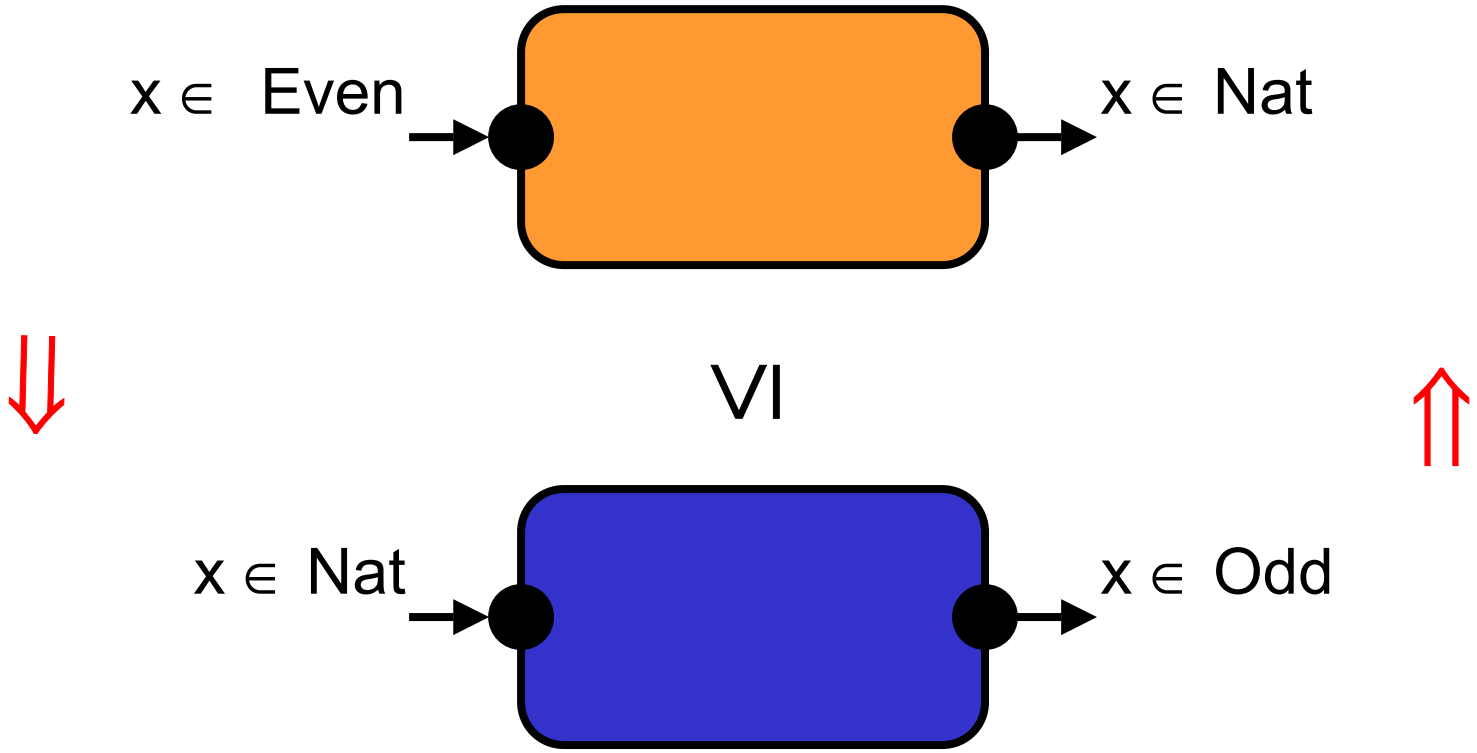
This computes the states from which the environment has a strategy to avoid deadlock. The propagated environment constraint is that it will apply such a strategy.

Component Abstraction



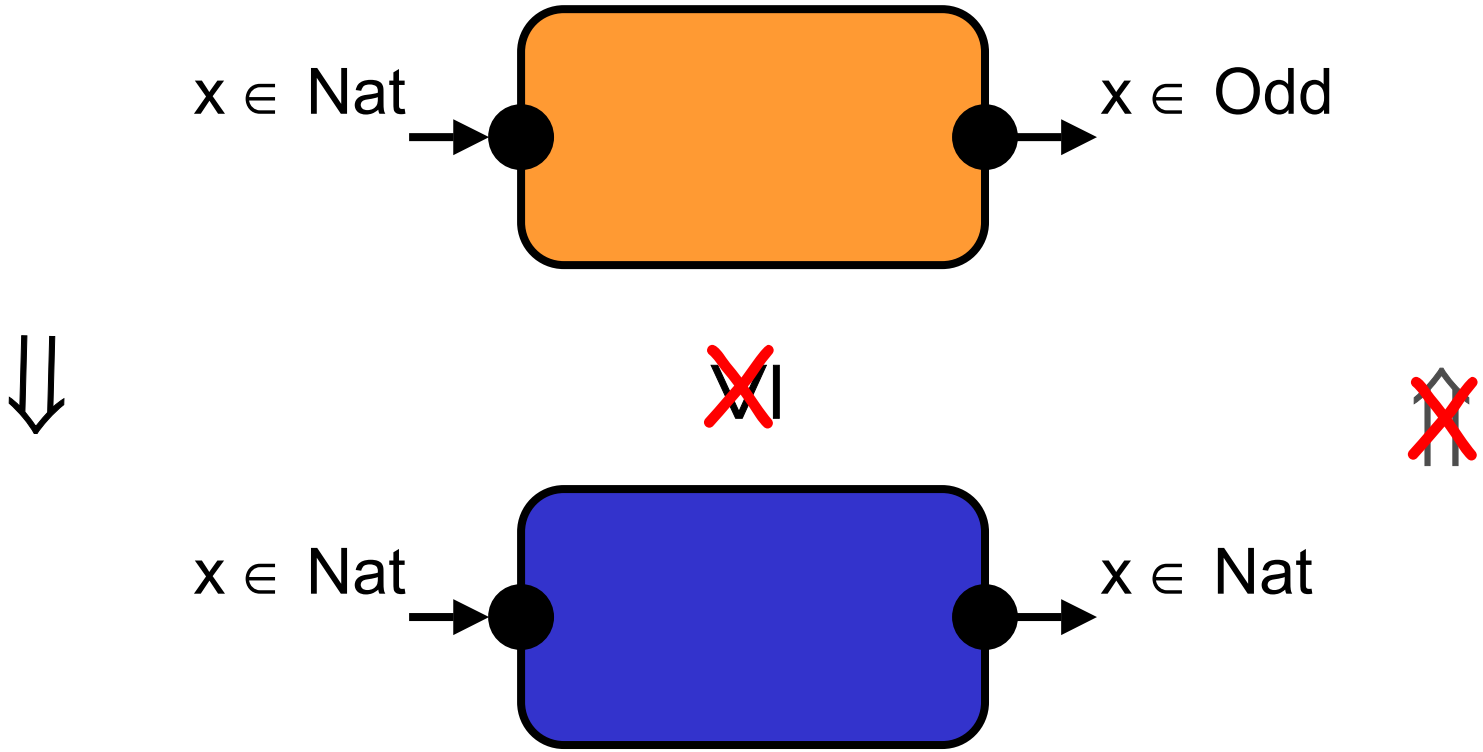
Abstraction is implication (simulation; trace containment).

Interface Implementation



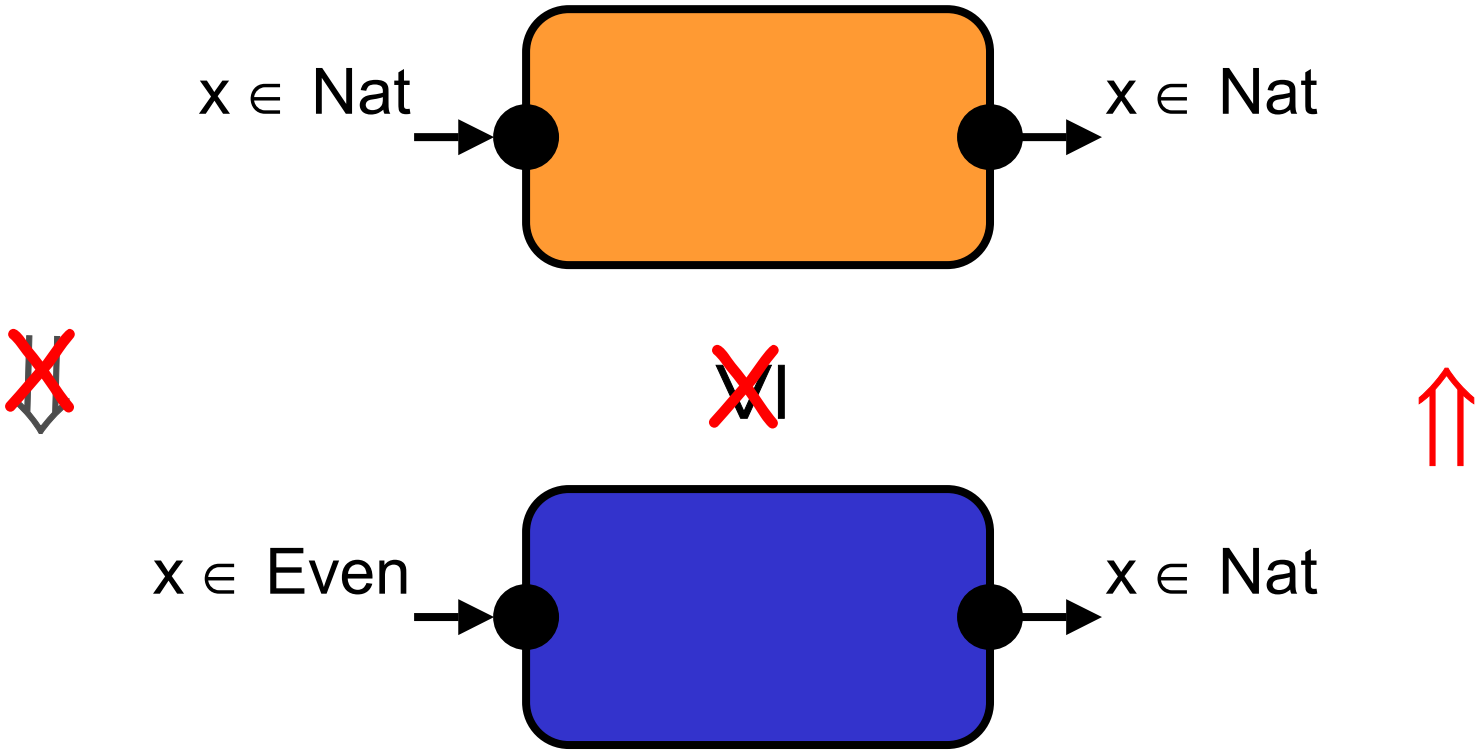
Implementation is I/O contravariant.

Interface Implementation

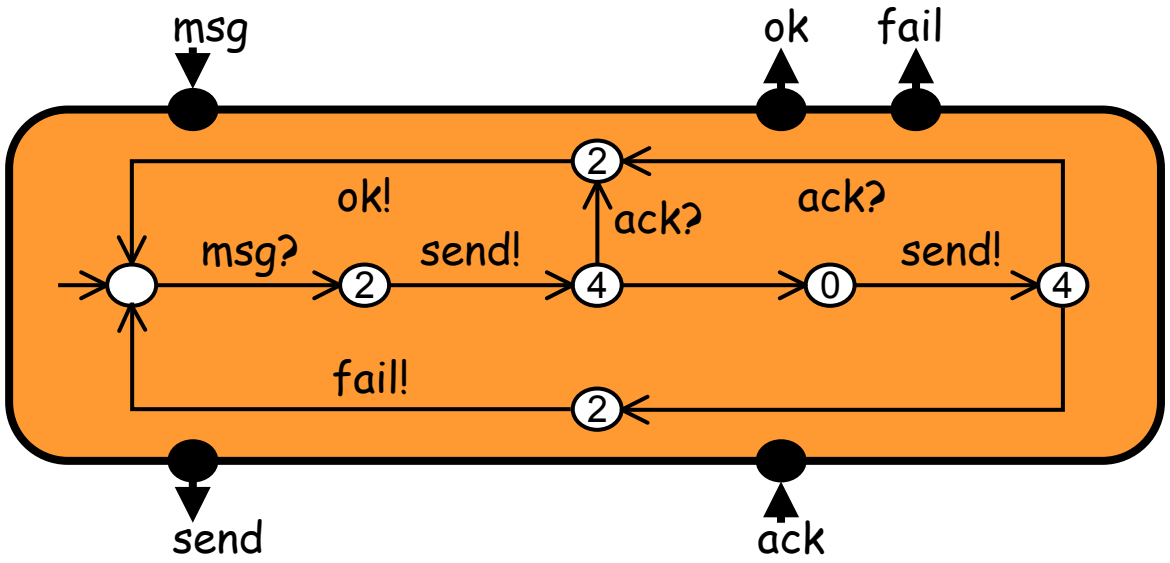


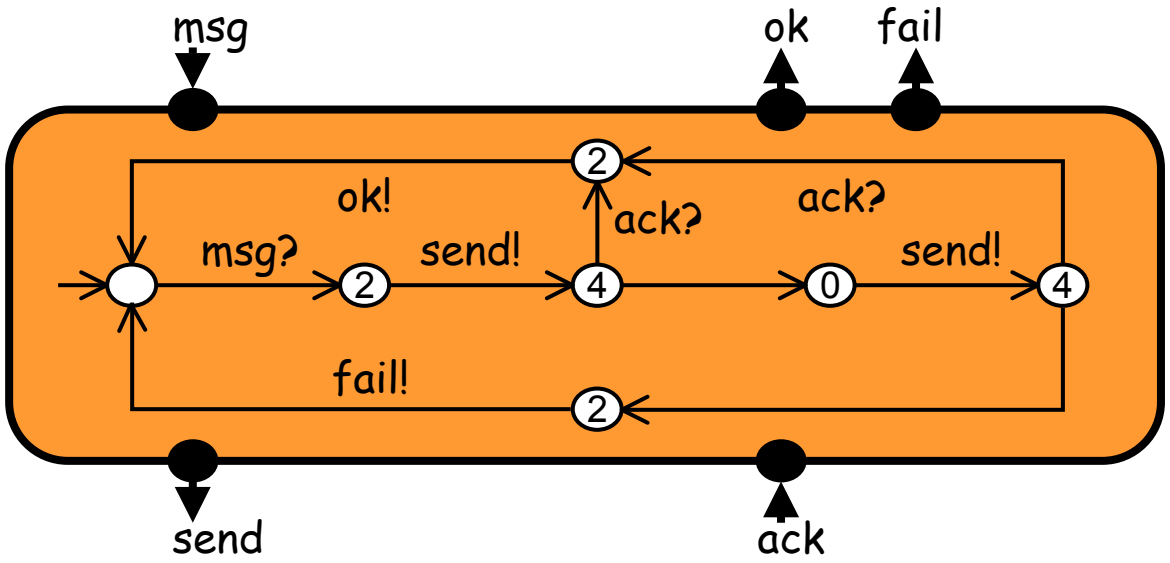
Implementation must obey output guarantee.

Interface Implementation

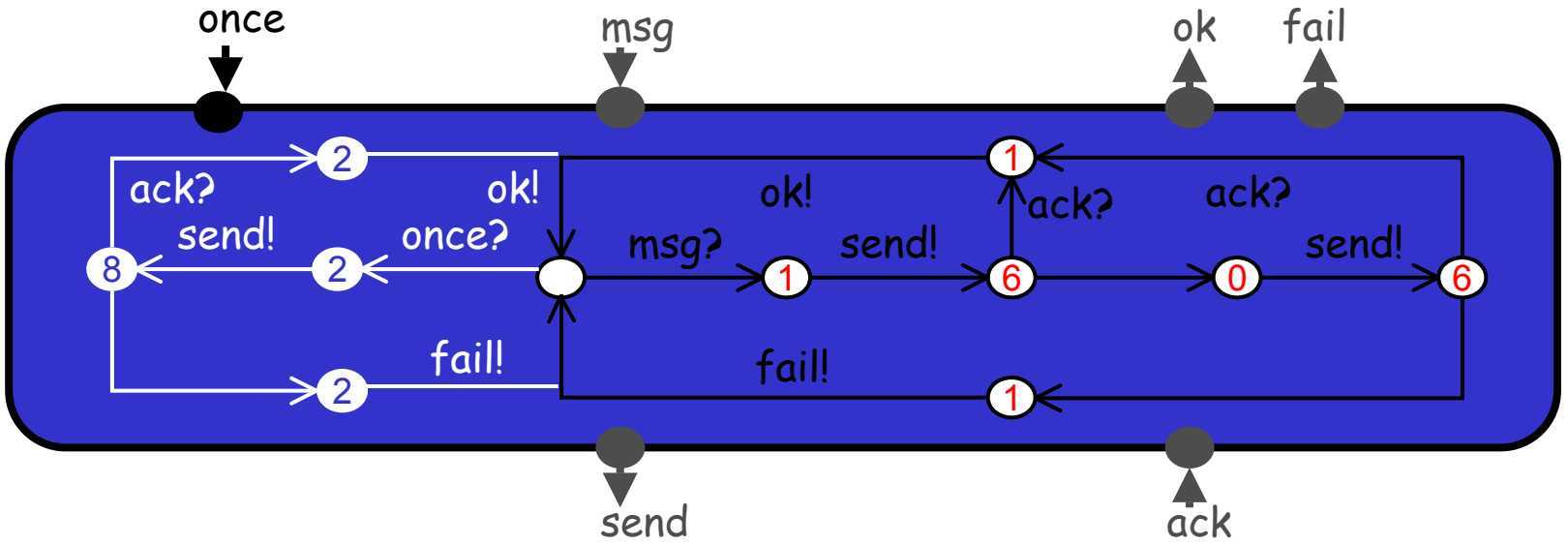


Implementation must accept all permissible inputs.


msg
ok
fail
2
ok!
ack?
ack?
msg?
send!
send!
2
4
0
4
fail!
2
send
ack


msg
ok
fail
2
ok!
ack?
ack?
msg?
send!
send!
2
4
0
4
fail!
2
send
ack

VI


once
msg
ok
fail
2
ack?
ok!
send!
once?
8
2
ok!
msg?
send!
1
6
ack?
ack?
send!
0
6
fail!
fail!
2
1
send
ack

Alternating Simulation

$Q \leq q$

iff

1. for all inputs i, if $q \text{ –i?-> } q'$, then there exists Q' such that $Q \text{ –i?-> } Q'$ and $Q' \leq q'$,

and

2. for all outputs o, if $Q \text{ –o!-> } Q'$, then there exists q' such that $q \text{ –o!-> } q'$ and $Q' \leq q'$.

If there is a helpful environment at q, then there is a helpful environment at Q [Alur/H/Kupferman/Vardi].

Algorithms & Tools

-interface compatibility (reachability game) can be checked in linear time

-interface implementation (alternating simulation) can be checked in quadratic time

We are currently implementing this in JBuilder [Chakrabarti/de Alfaro/H/Jurdzinski/Mang].