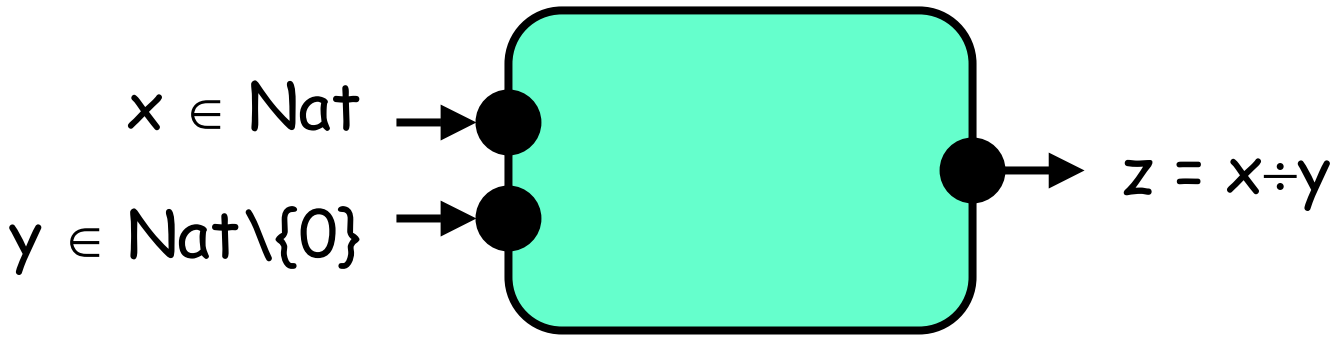
What is an Interface?

Thomas A. Henzinger

University of California, Berkeley

joint work with Luca de Alfaro

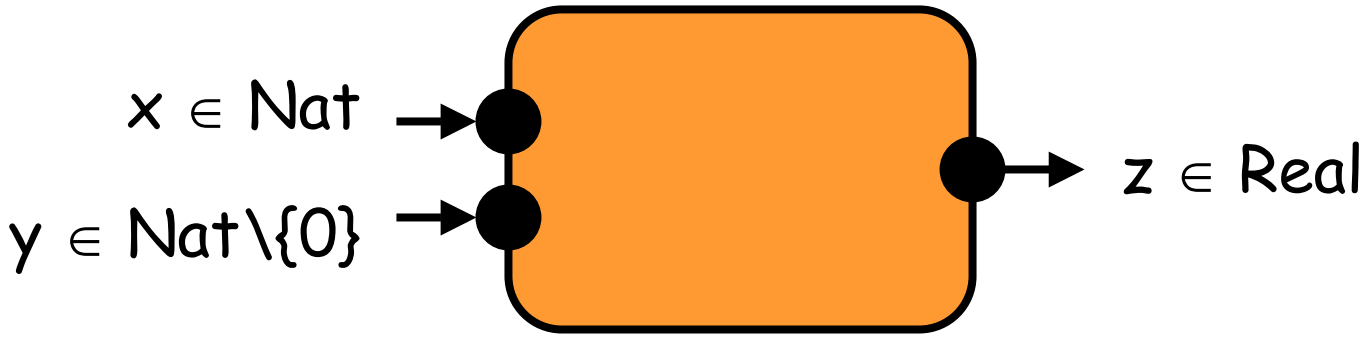
A Component Specification



Descriptive:

"What does the component do?"

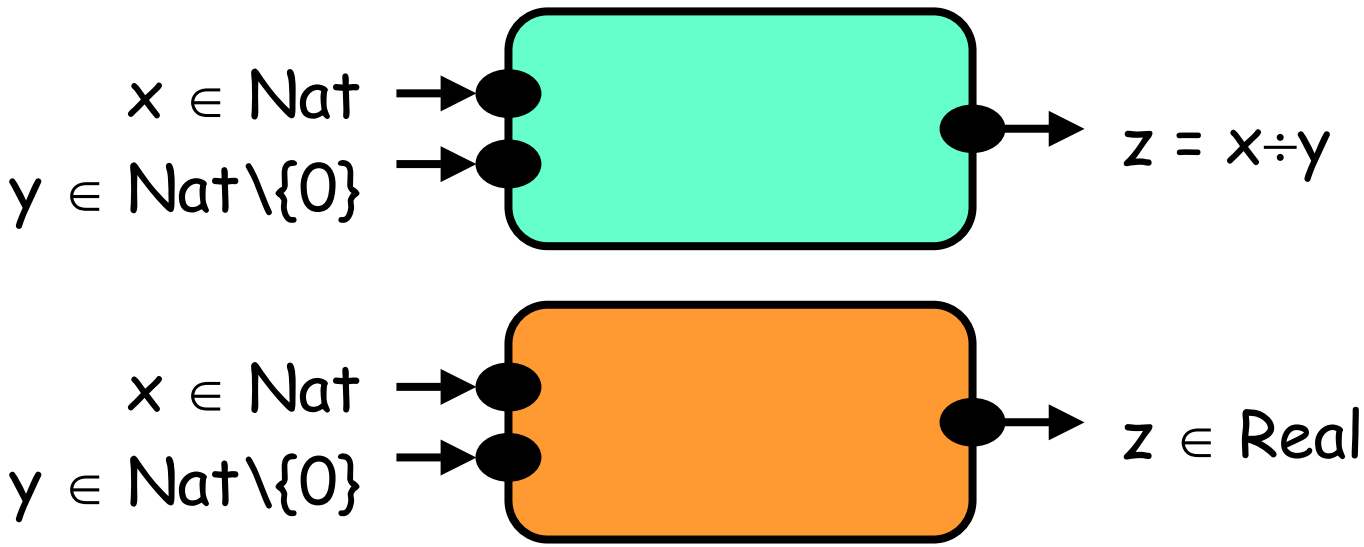
An Interface Specification



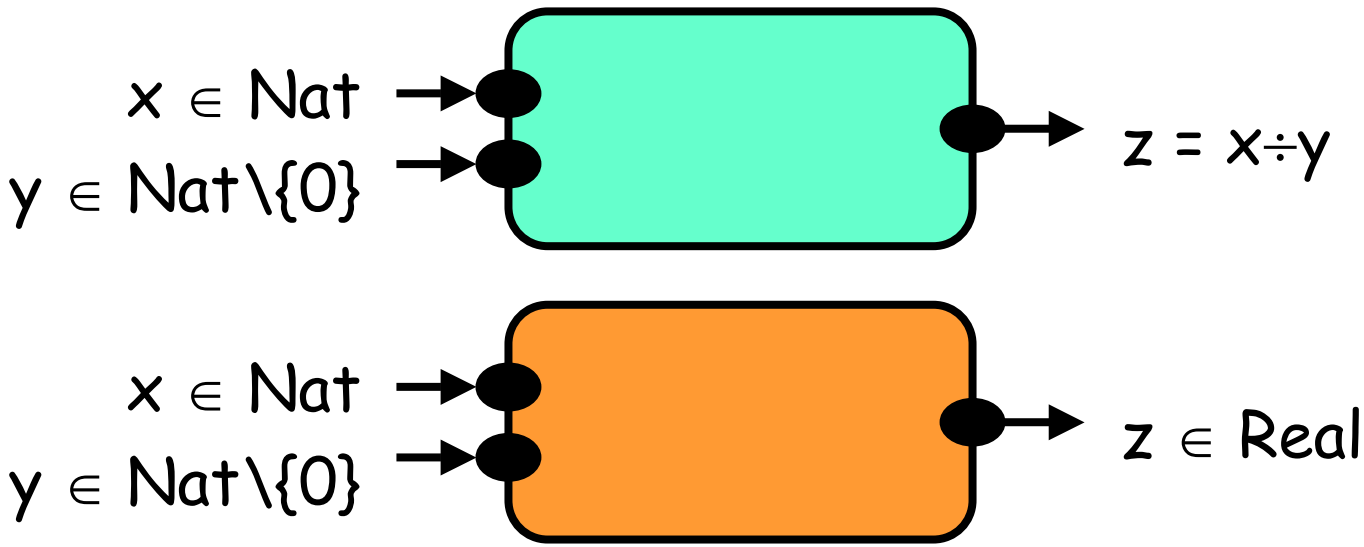
Prescriptive:

"How can the component be put together with other components?"

What's the Difference?

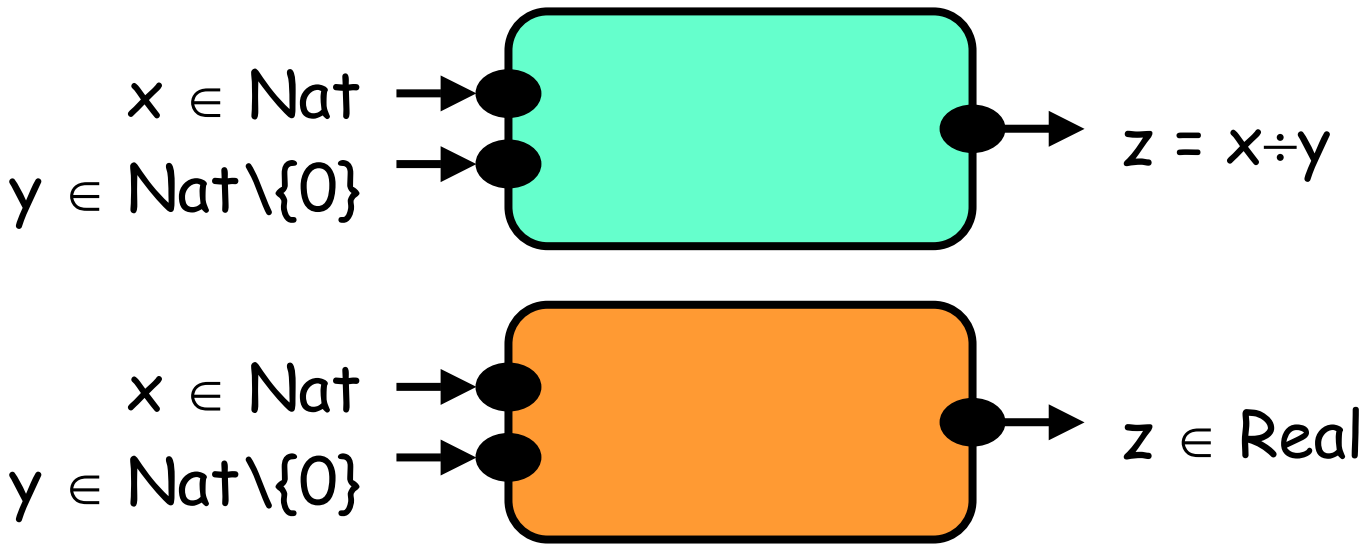


What's the Difference?



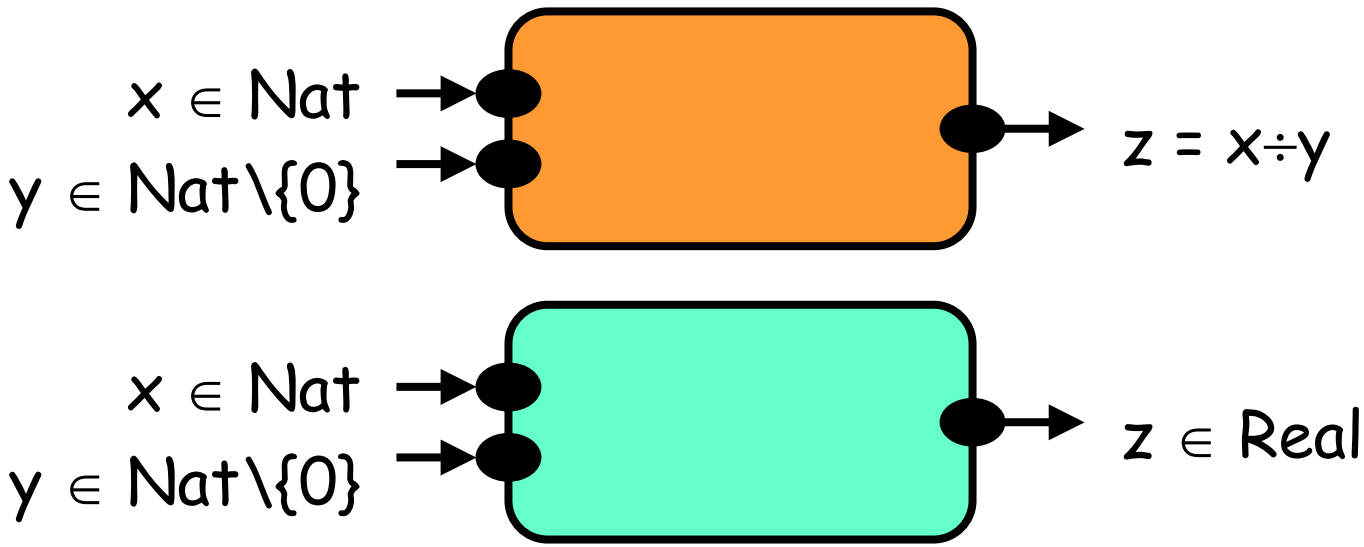
Language (syntax) No

What's the Difference?



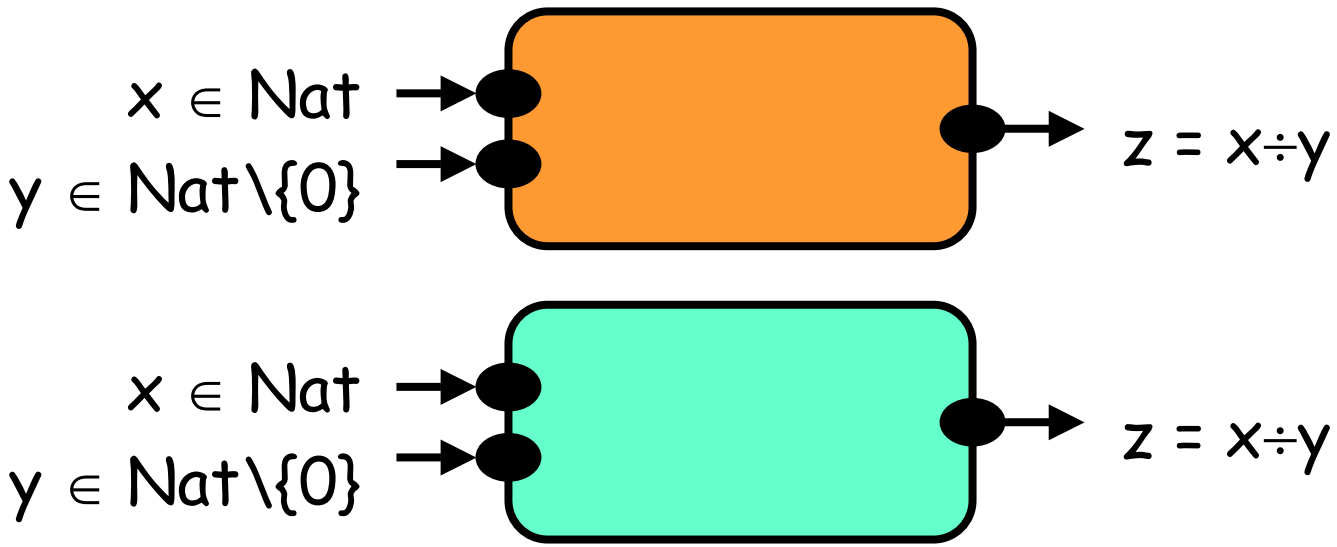
Language (syntax)	No
Level of detail	

What's the Difference?



Language (syntax)	No
Level of detail	No

What's the Difference?



Language (syntax)	No
Level of detail	No
Interpretation (semantics)	Yes

The Component Interpretation

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

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

The Component Interpretation

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

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

The Interface Interpretation

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

-constrains the environment;
other environments are rejected (by compiler)
-example: type declaration of procedure

Component Specification is Well-formed iff

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

-specification is implementable
-input-universal

Component Specification is Well-formed iff

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

-specification is implementable
-input-universal

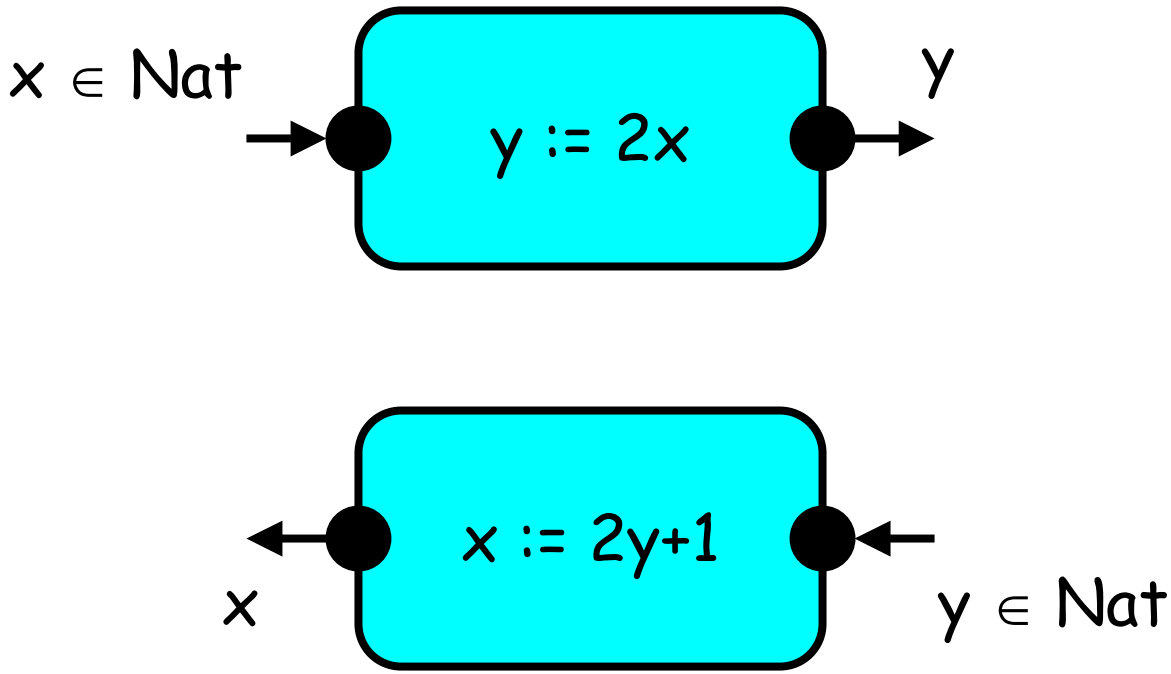
Interface Specification is Well-formed iff

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

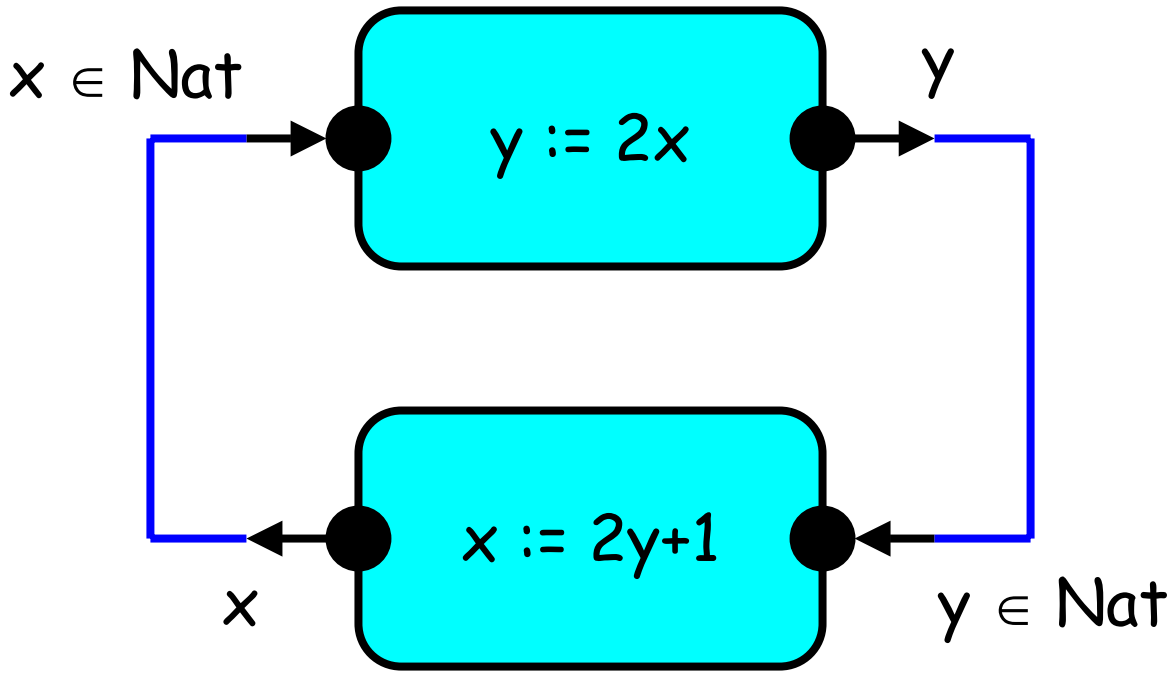
-specification is satisfiable
(i.e., usable in some environment)
-input-existential

1. The logic view ✓
2. The block-diagram view
3. The state-transition view

Components: Total Functions

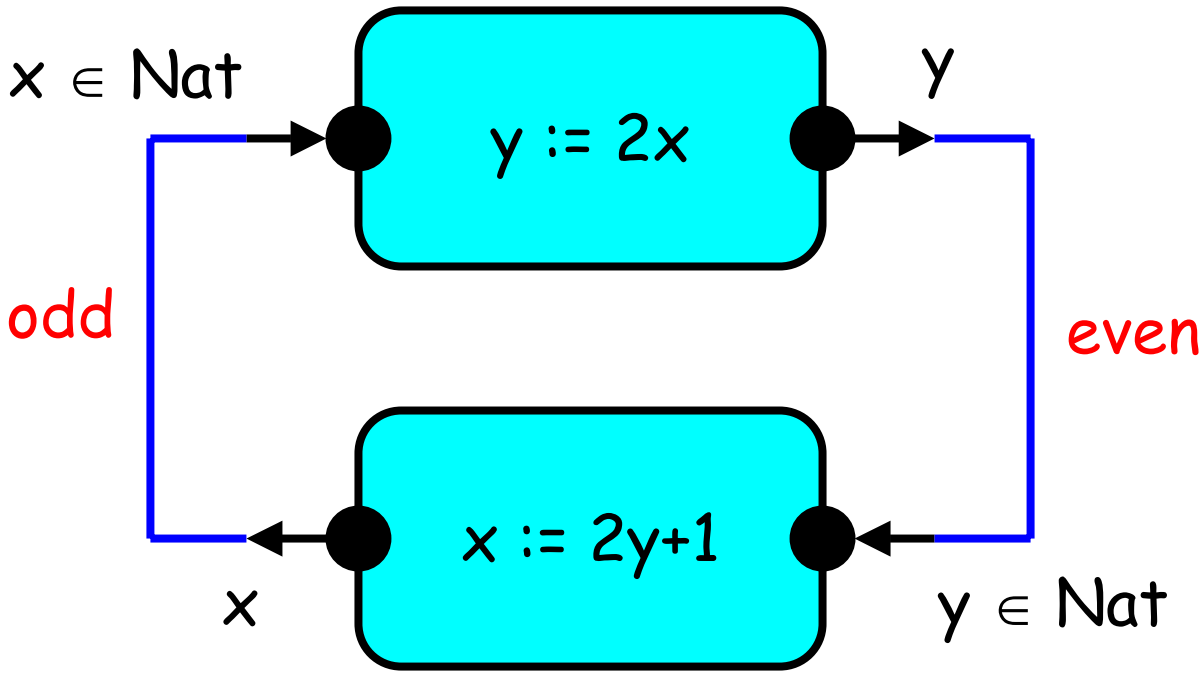


Components: Total Functions

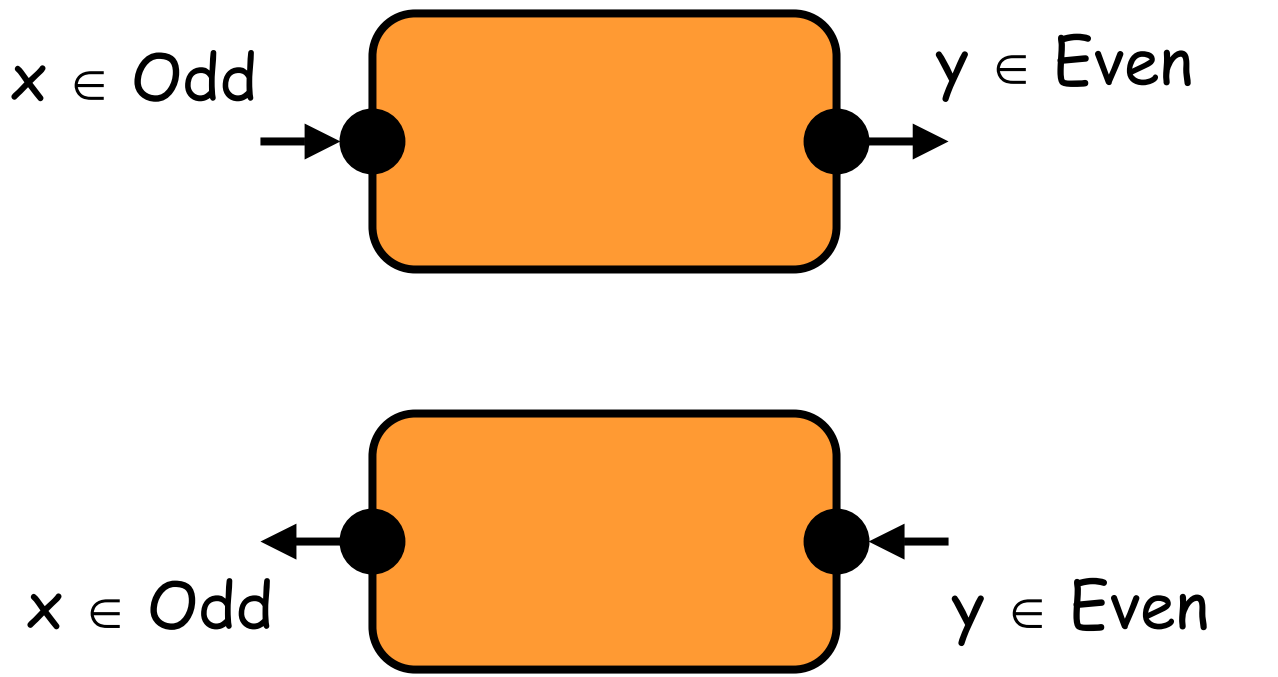


Composition always defined.

Components: Total Functions



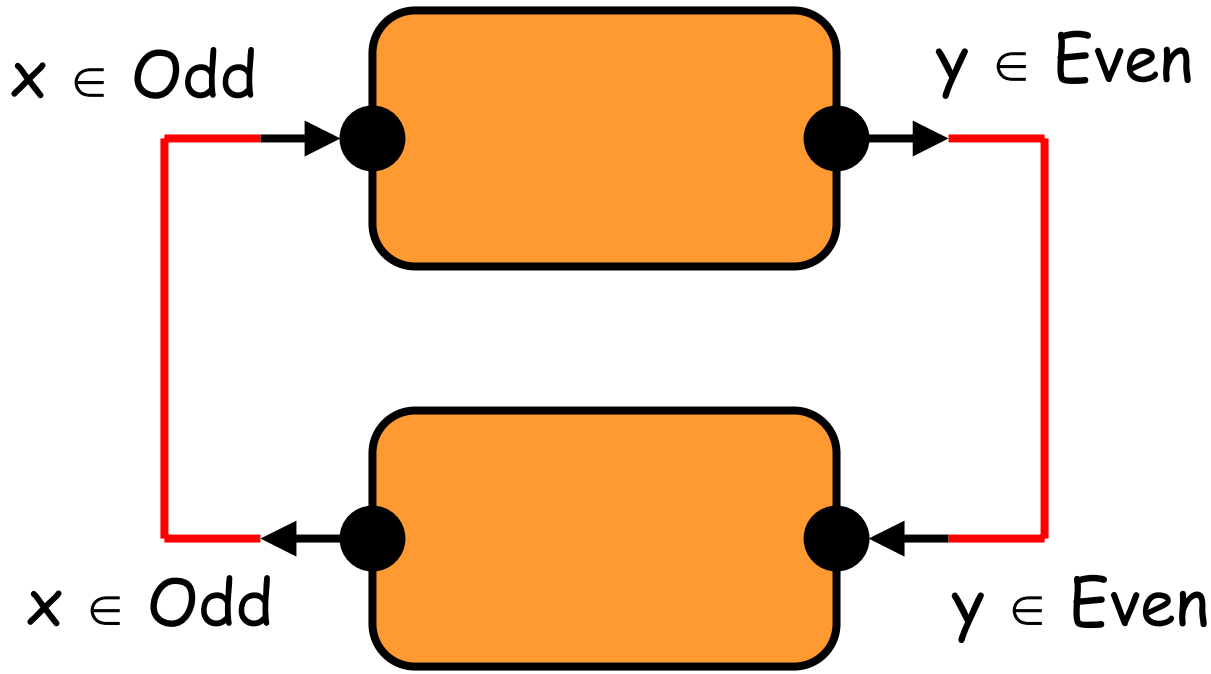
Interfaces: Partial Functions



Output guarantee

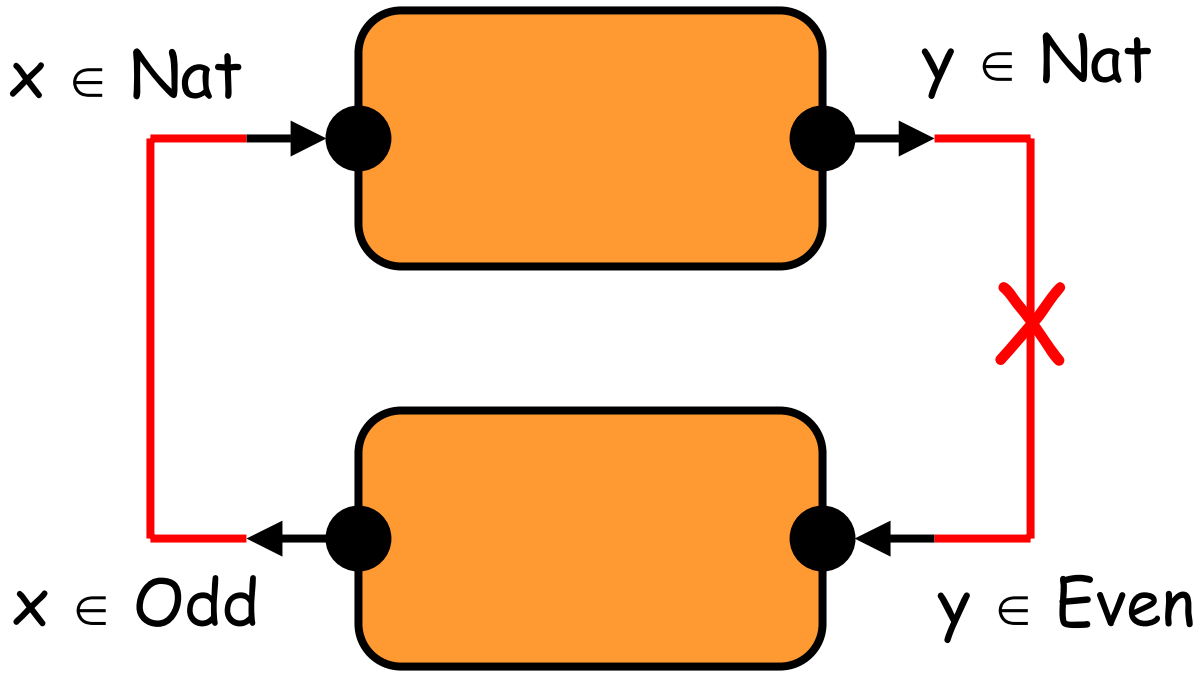
Input assumption

Interfaces: Partial Functions



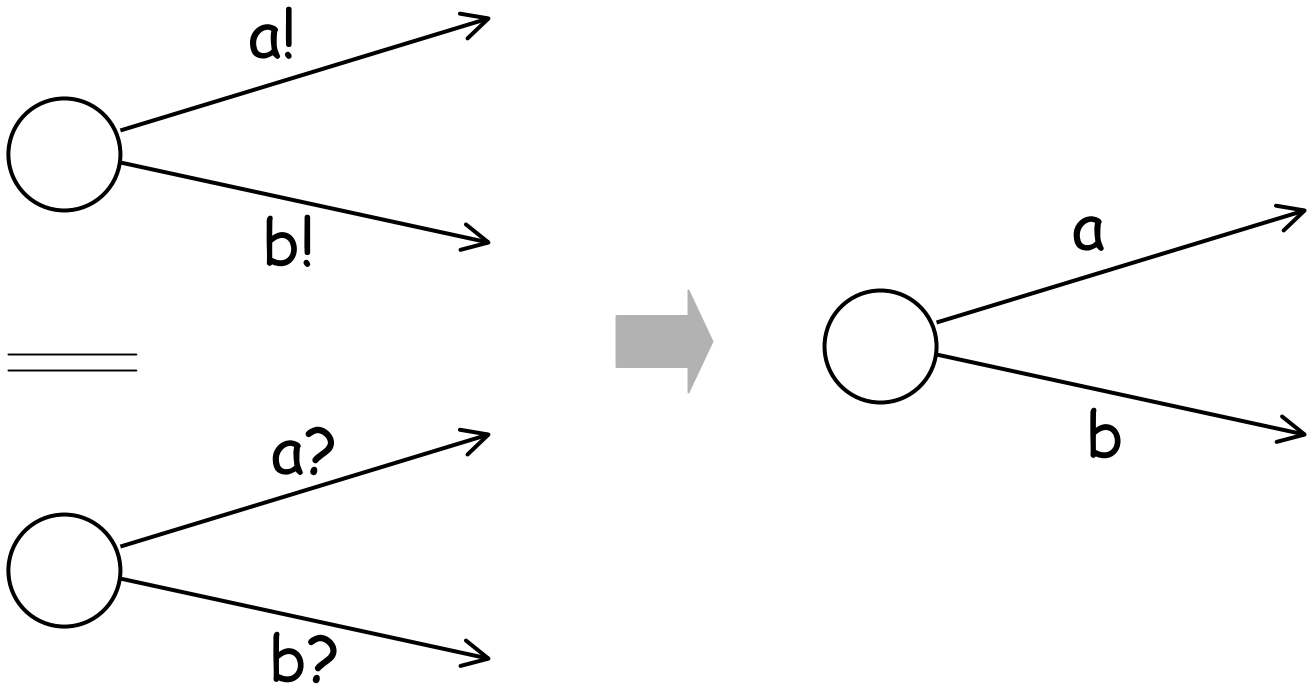
Composition defined: Compatible interfaces.

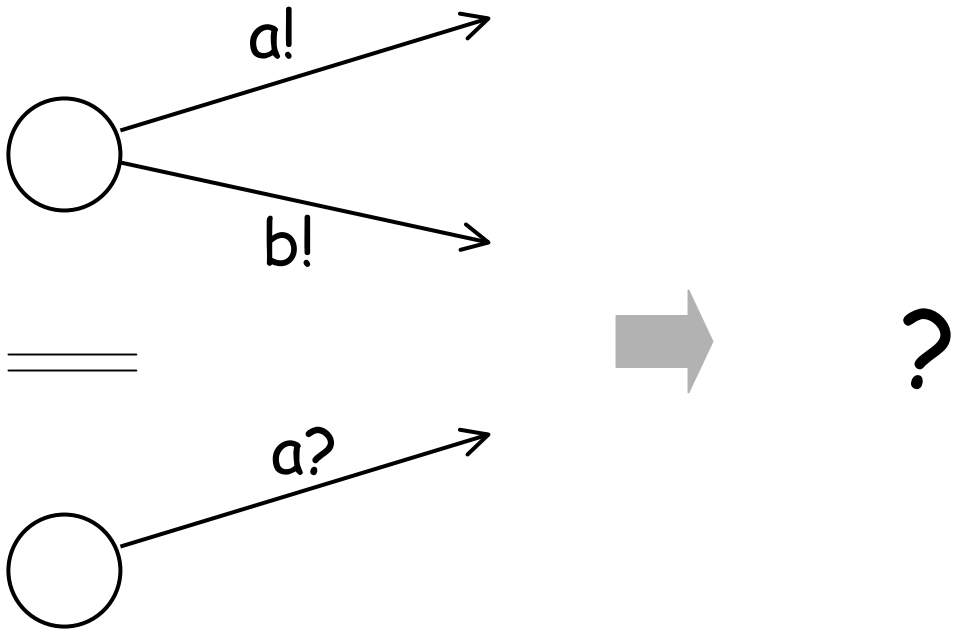
Interfaces: Partial Functions



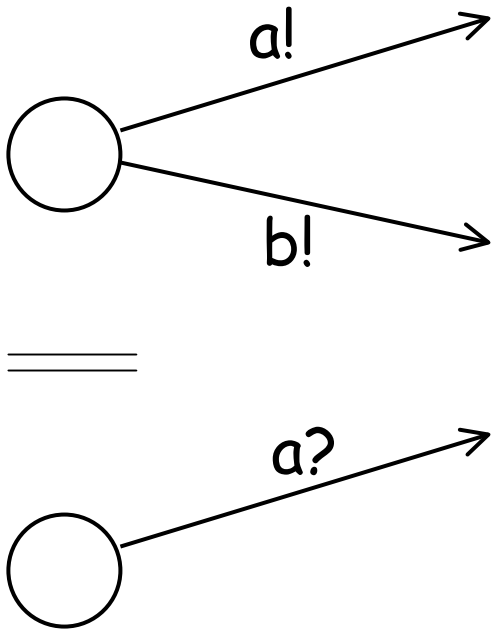
Composition undefined: Incompatible interfaces.

1. The logic view ✓
2. The block-diagram view ✓
3. The state-transition view


a!
b!
a?
b?
a
b


a!
b!
a?
?

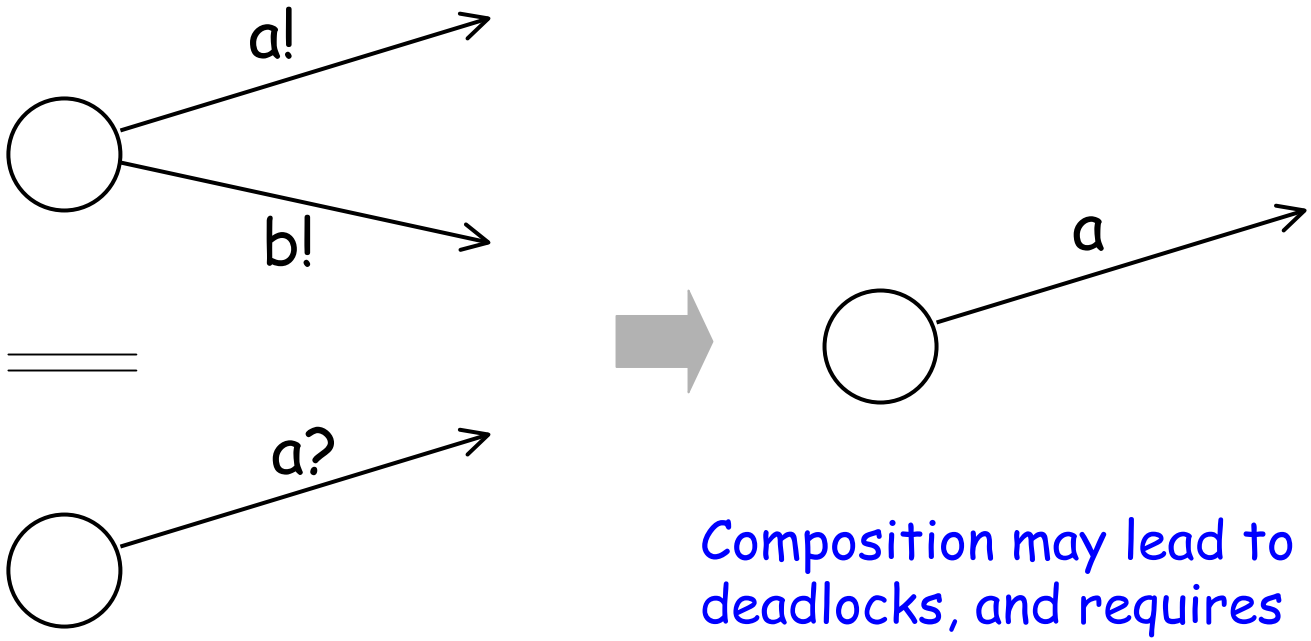
Components, Alternative 1: Be Prepared for Every Input



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

[I/O Automata, TLA, Reactive Modules]

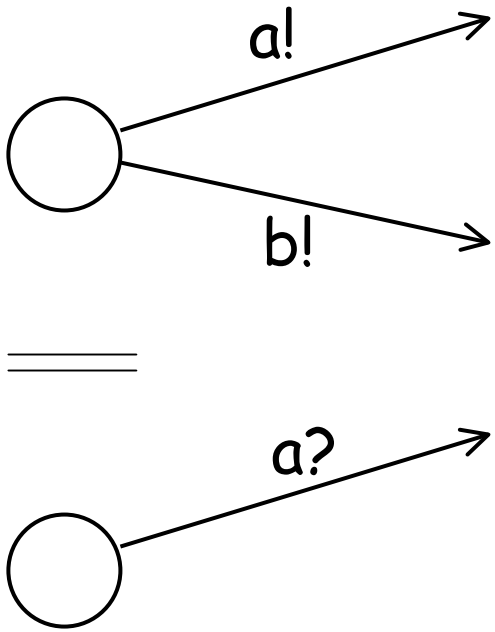
Components, Alternative 2: Compose, then Check



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

[CSP/CCS, Statecharts, Esterel]

Interfaces: Check, then Compose

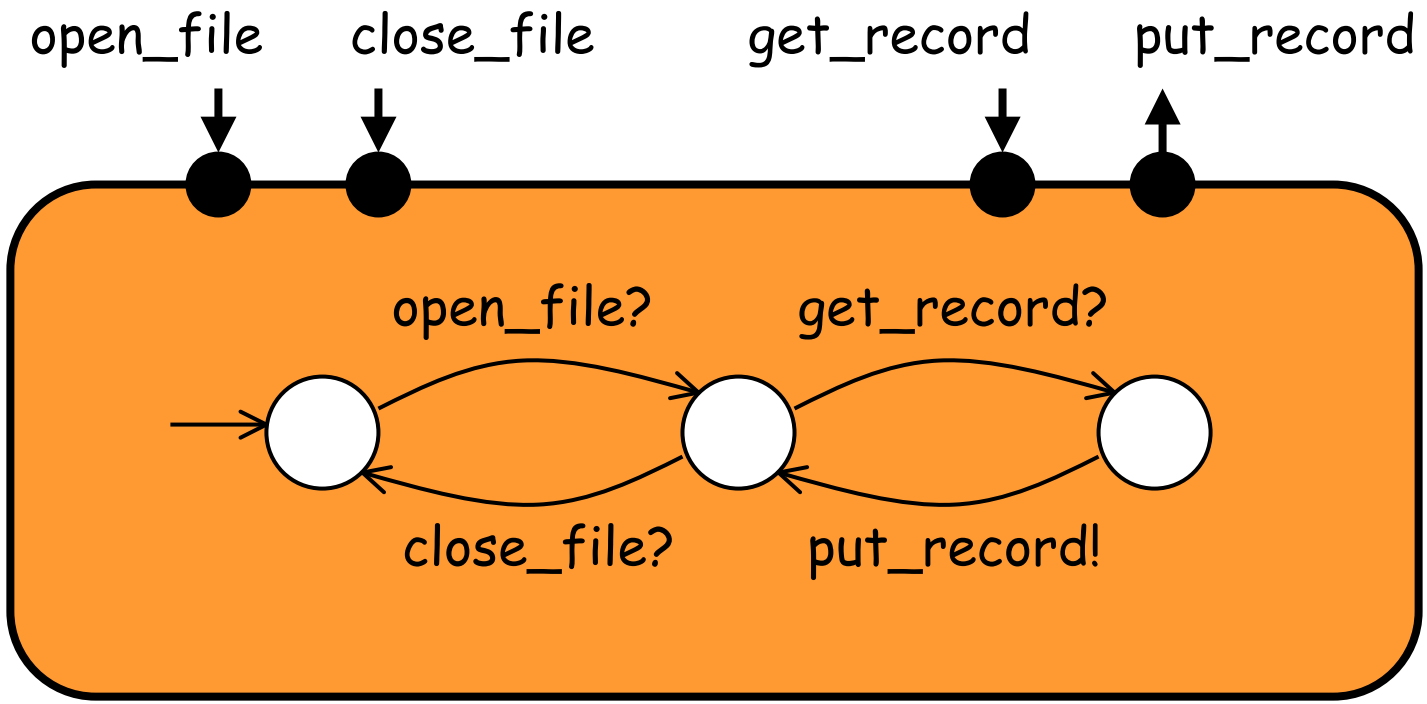


These interfaces are incompatible, because the receiver expects the environment to provide input b.

[Trace Theory, Interface Automata]

Simple interfaces (traditional "types"):
value constraints

Rich interfaces ("types for component interaction"):
temporal ordering constraints, real-time constraints, etc.



Interface specifications are a programmer's task, in burden between type declarations and invariant annotations.

Interface compatibility checking is a program analysis opportunity, in difficulty between type checking and behavioral verification.

(Note: unlike software contracts, interfaces are checked statically, at compile time.)

Components

- no environment constraints

Interfaces

- environment constraints

  environment constraints propagate by composition

Components

- no environment constraints
- composition is total

Interfaces

- environment constraints
  environment constraints propagate by composition
- composition is partial
  interface compatibility checking is game solving

Components

- no environment constraints
- composition is total
- refinement is covariant

Interfaces

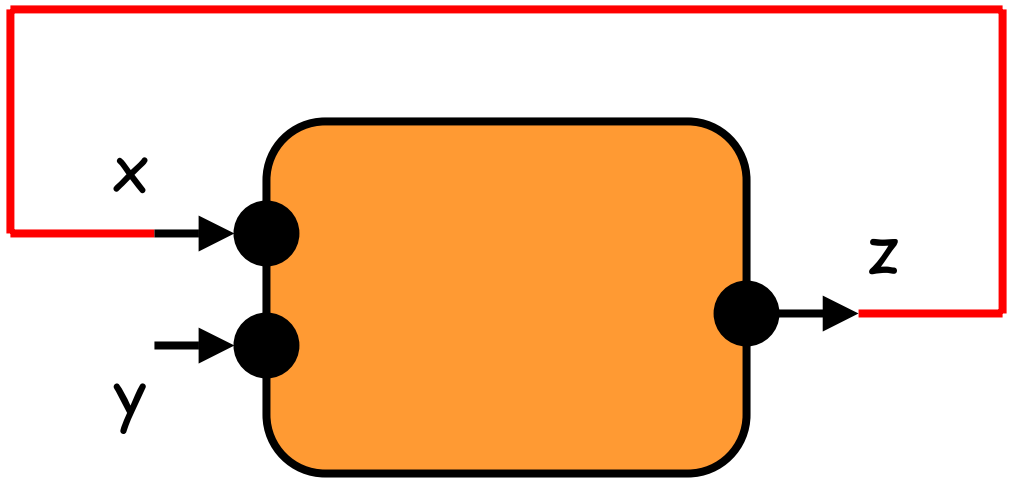
- environment constraints
  environment constraints propagate by composition
- composition is partial
  interface compatibility checking is game solving
- refinement is contravariant
  I/O alternating simulation relations

Components	Interfaces
•no environment constraints	•environment constraints environment constraints propagate by composition
•composition is total	•composition is partial interface compatibility checking is game solving
•refinement is covariant	•refinement is contravariant I/O alternating simulation relations
•for compositional analysis	•for compositional design

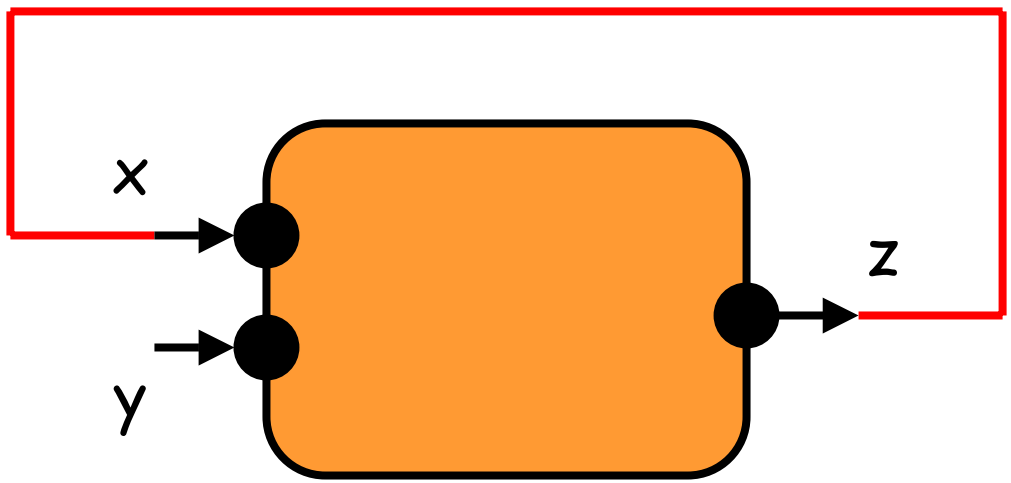
Interface Composition and Propagation of Environment Constraints:

The Block-Diagram View

$x=0 \Rightarrow y=0$ true



$x=0 \Rightarrow y=0$ true



$x=0 \Rightarrow y=0$ true

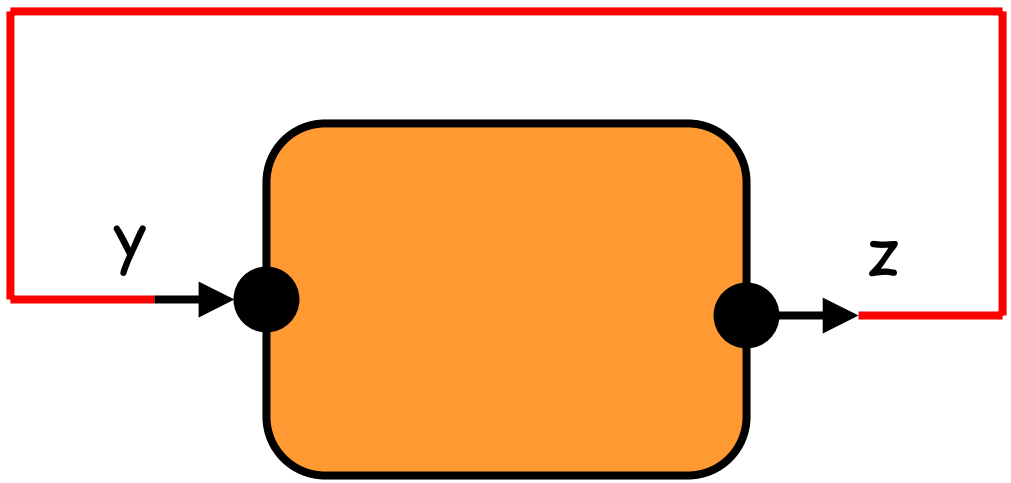
$y = 0$

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

y = 0

true

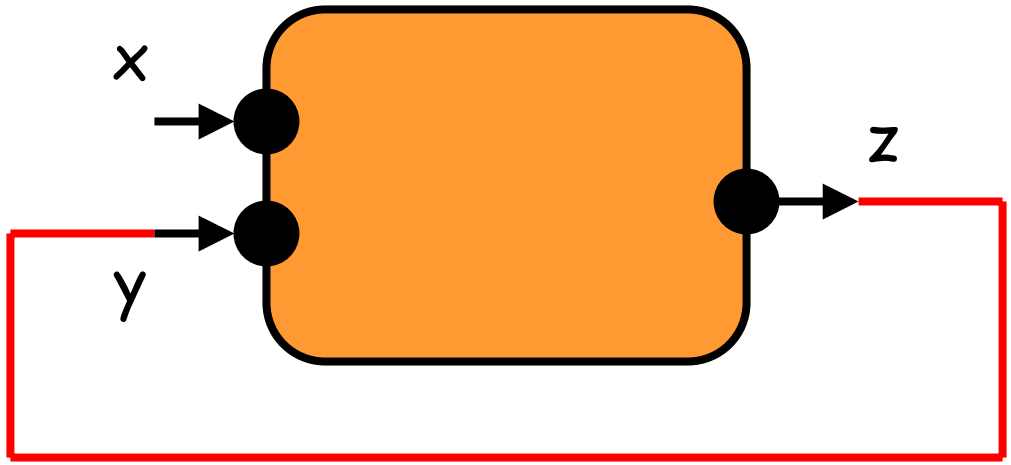
The resulting interface.



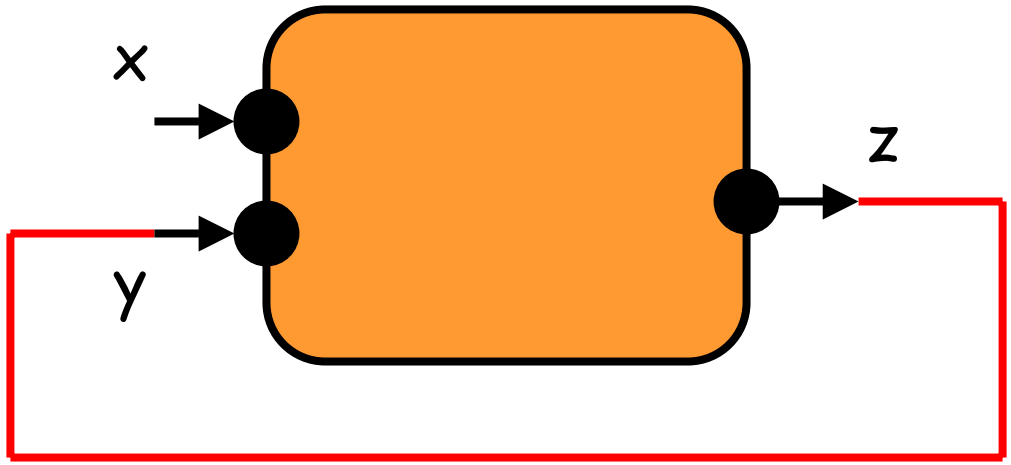
y = 0

true

Illegal connection.



$x=0 \Rightarrow y=0$ true



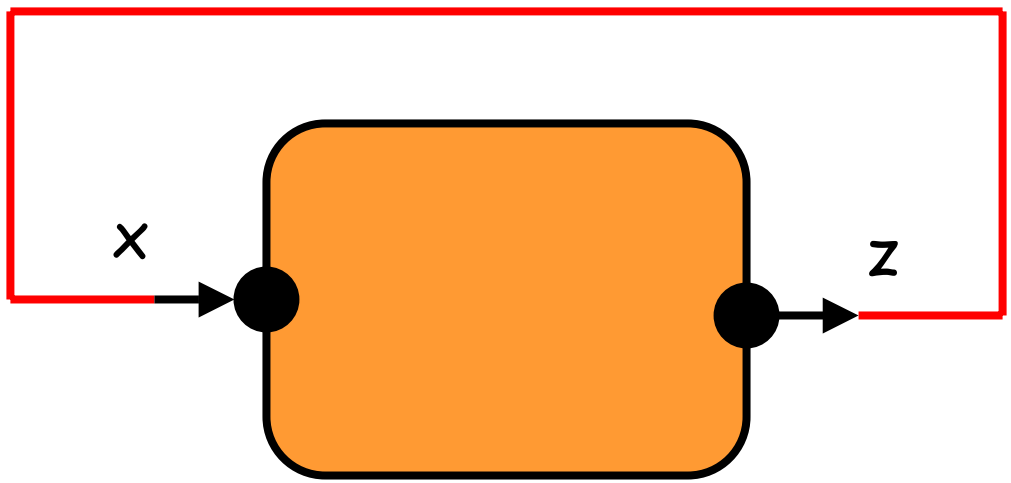
$x=0 \Rightarrow y=0$ true

$x \neq 0$

$x \neq 0$

true

The resulting interface.

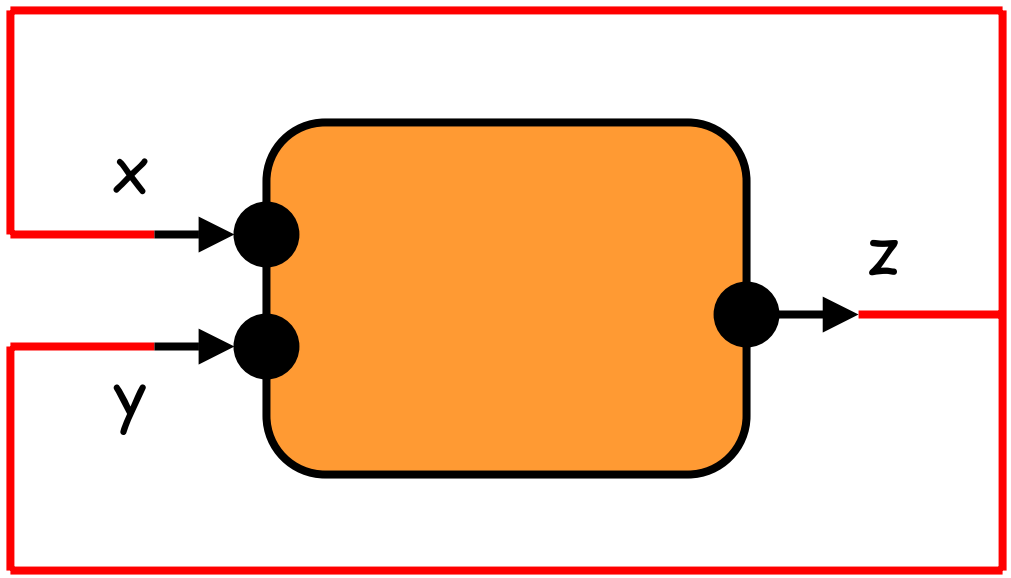


$x \neq 0$

true

Illegal connection.

Interface Composition is Not Associative.

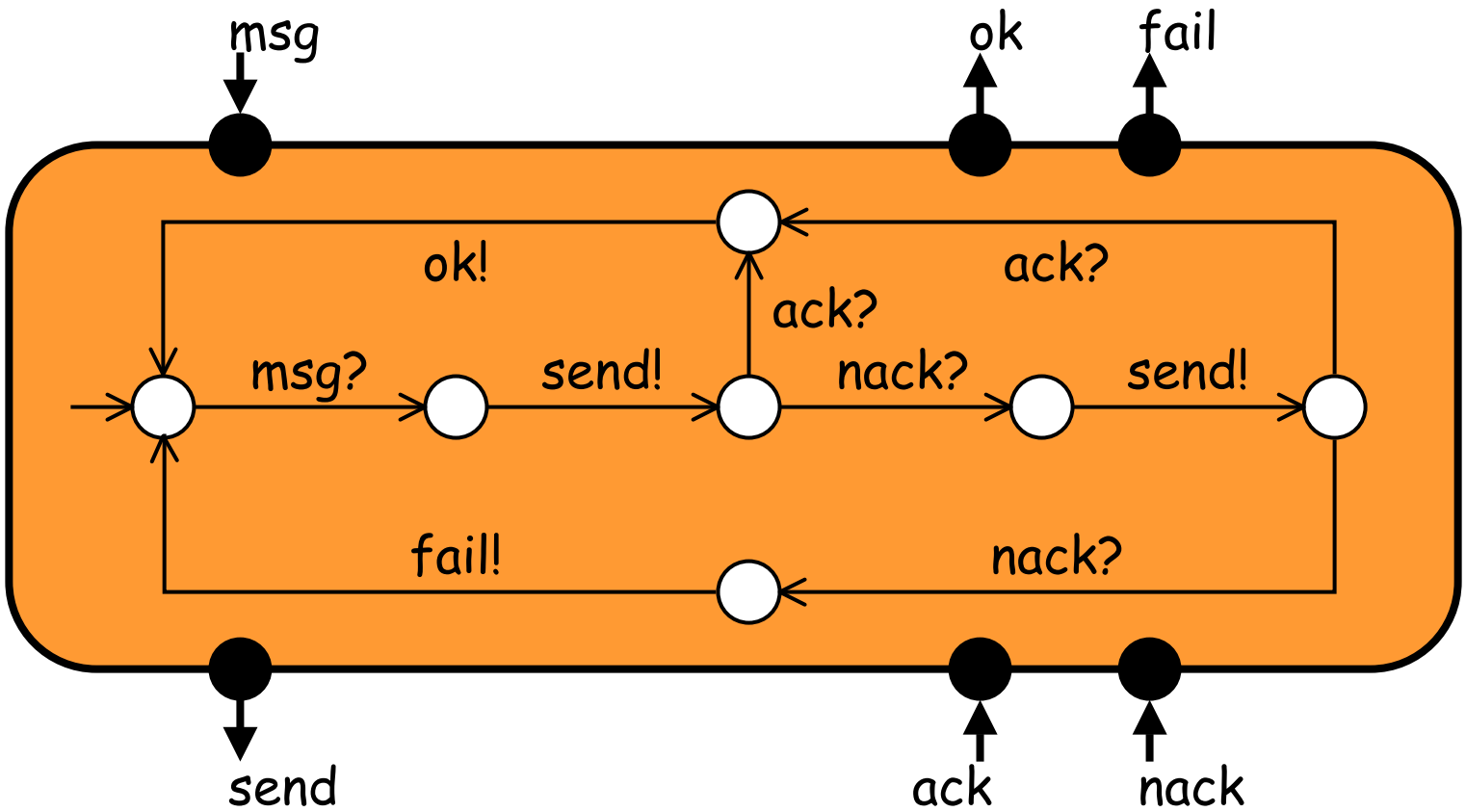


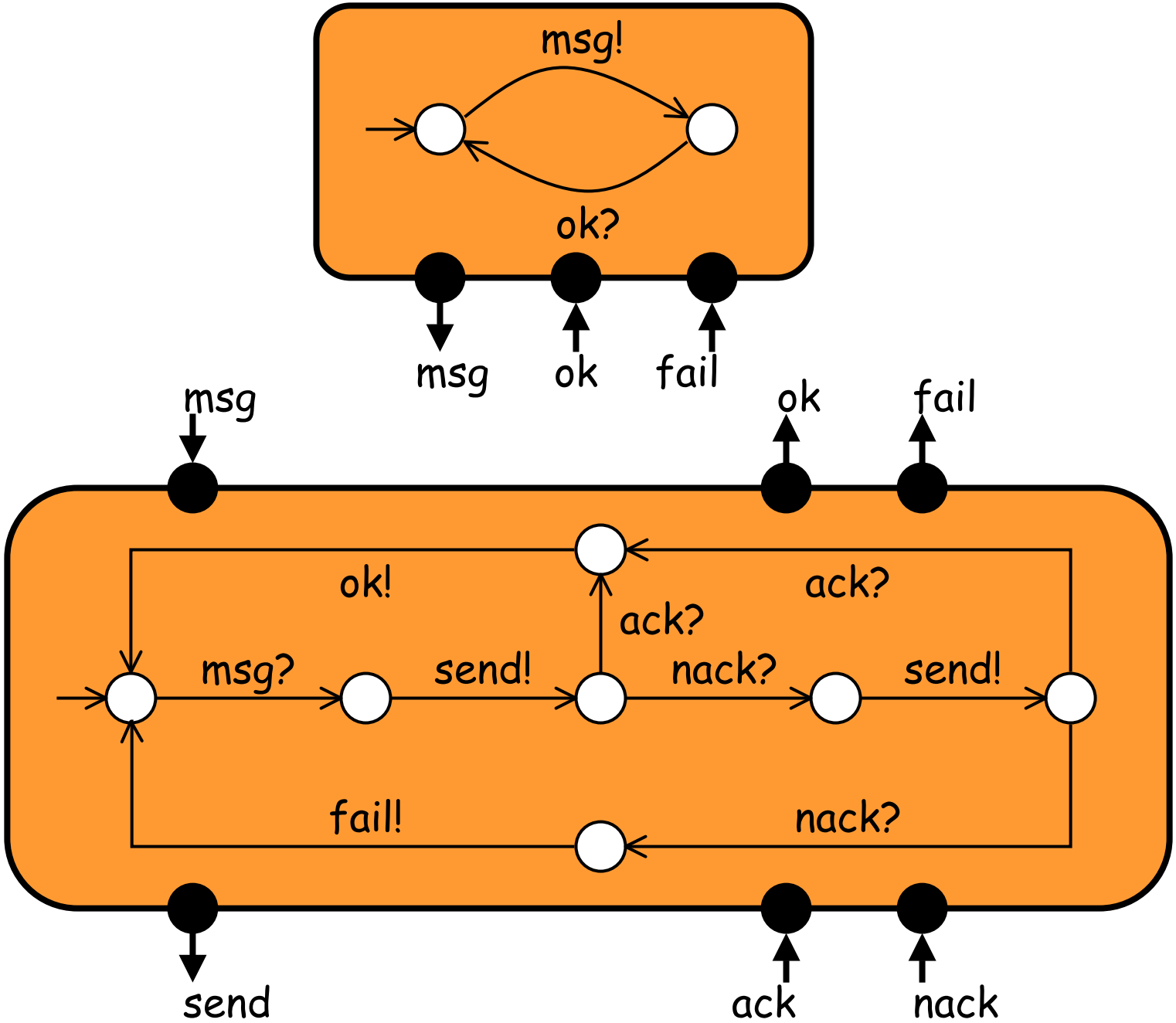
$x=0 \Rightarrow y=0$ true

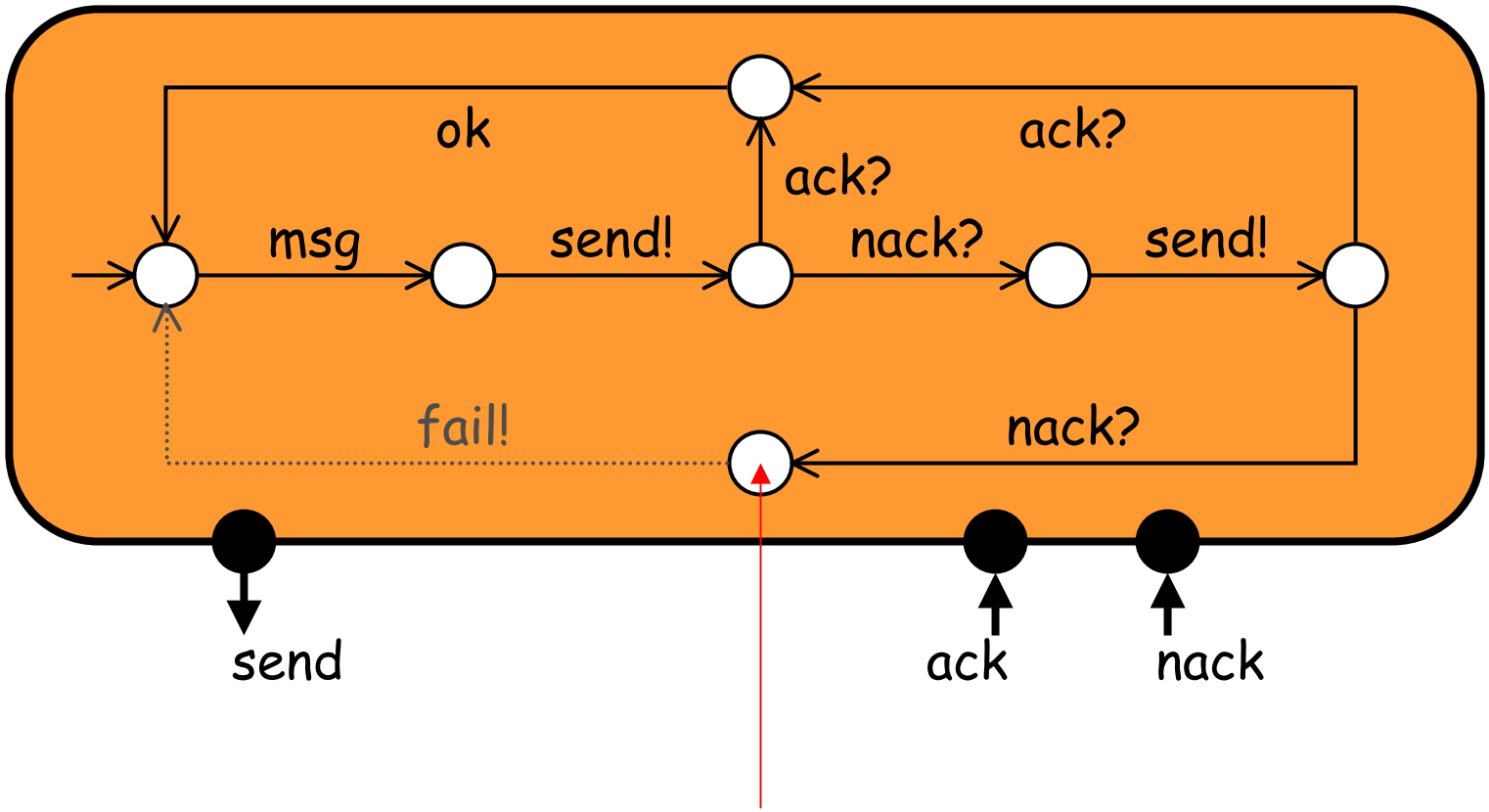
true

Interface Composition and Propagation of Environment Constraints:

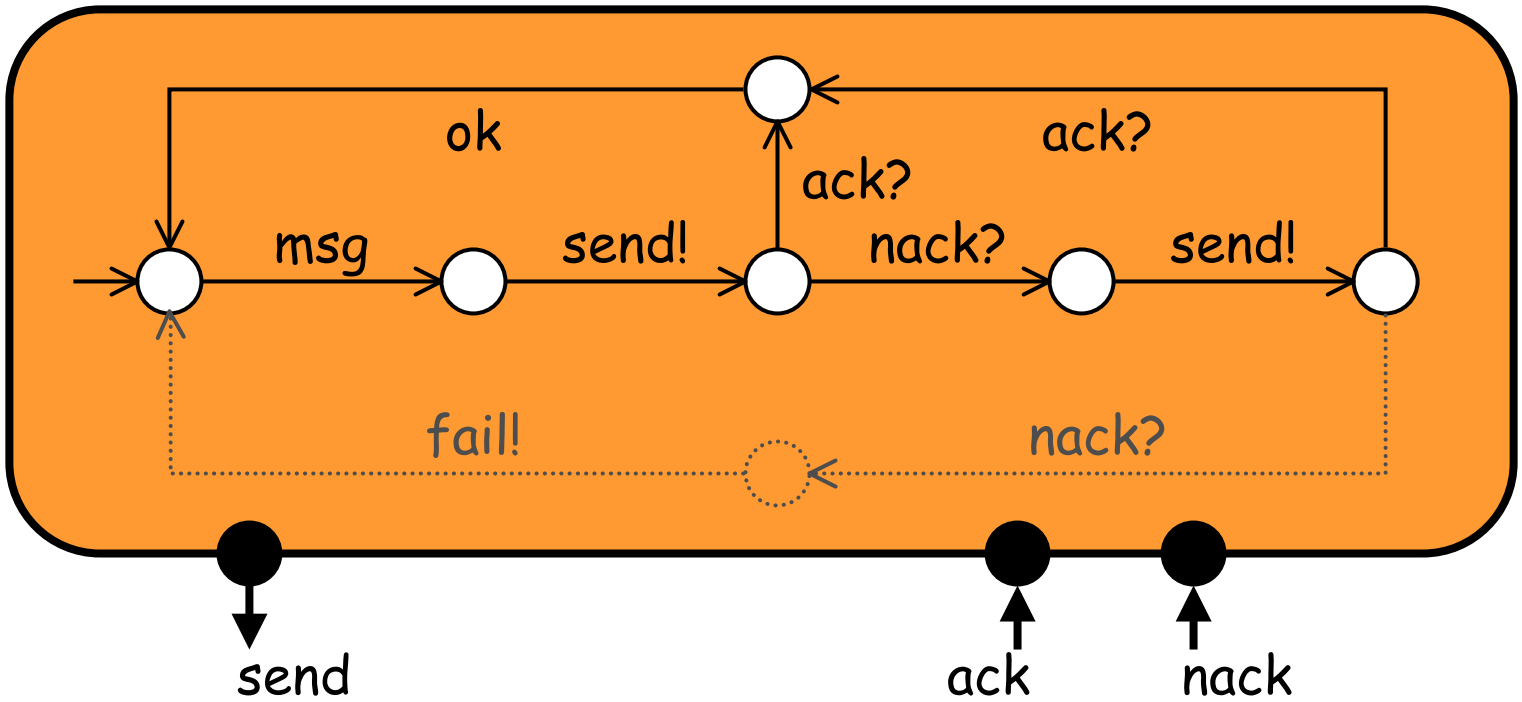
The State-Transition View


msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack


msg!
ok?
msg
ok
fail
msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack



Incompatible product state, but environment can prevent this state.



The Composite Interface.

Computing the Composite Interface

1. Construct product automaton.

Computing the Composite Interface

1. Construct product automaton.
2. Mark deadlock states as incompatible.

Computing the Composite Interface

1. Construct product automaton.
2. Mark deadlock states as incompatible.
3. 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.

Computing the Composite Interface

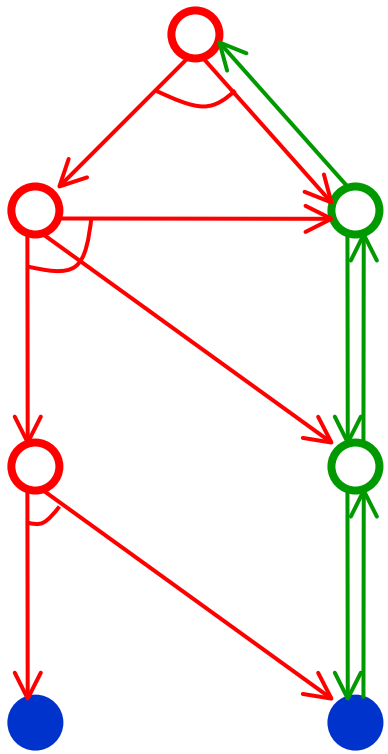
1. Construct product automaton.
2. Mark deadlock states as incompatible.
3. Until no more incompatible states can be added: mark state q as incompatible if the environment cannot prevent an incompatible state to be entered from q.
4. If the initial state is incompatible, then the two interfaces are incompatible.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

Computing the Composite Interface

1. Construct product automaton.
2. Mark deadlock states as incompatible.
3. 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.
4. 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.

Interface Compatibility Checking as Game Solving



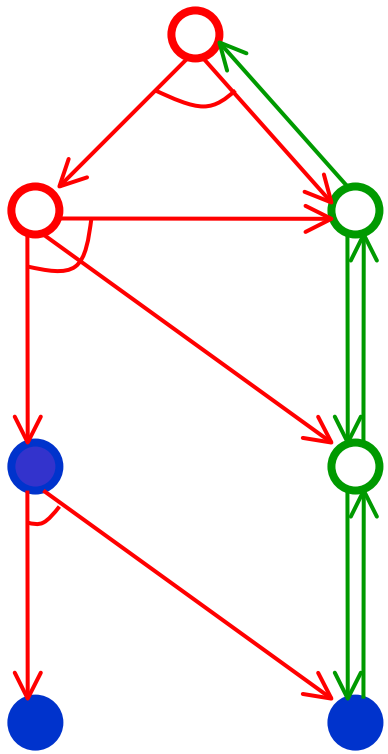
AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

Interface Compatibility Checking as Game Solving



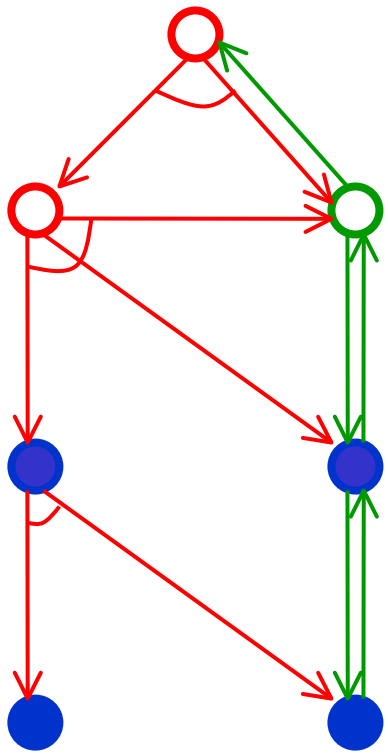
AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

Interface Compatibility Checking as Game Solving



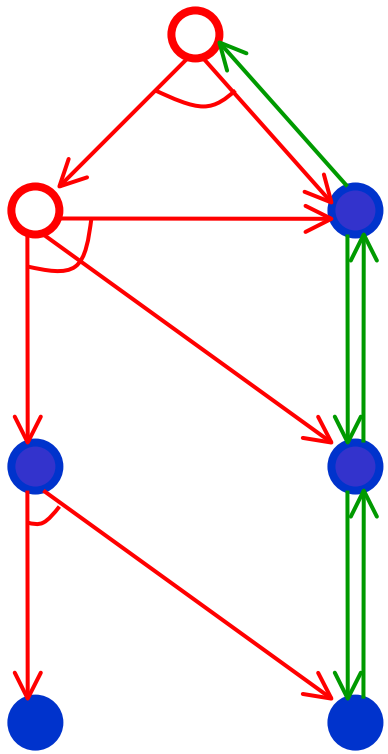
AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

Interface Compatibility Checking as Game Solving



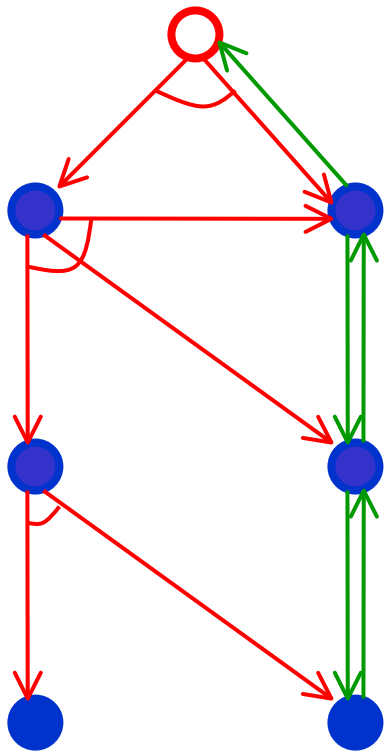
AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

Interface Compatibility Checking as Game Solving



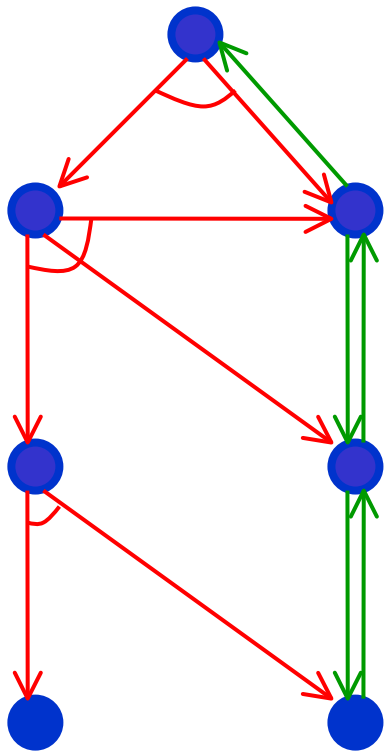
AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

Interface Compatibility Checking as Game Solving



AND-OR game graph:

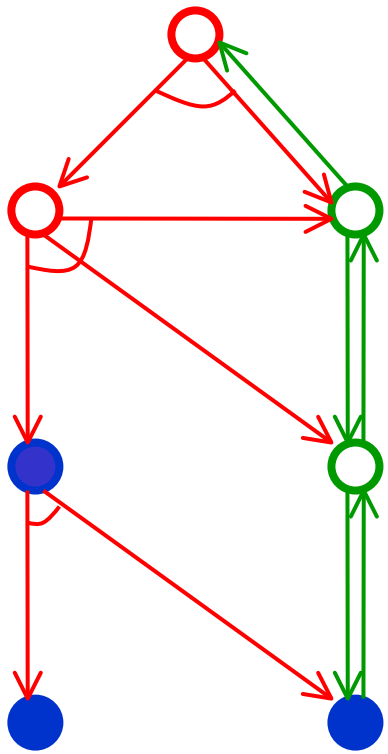
AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

NO

Interface Compatibility Checking as Game Solving



AND-OR game graph:

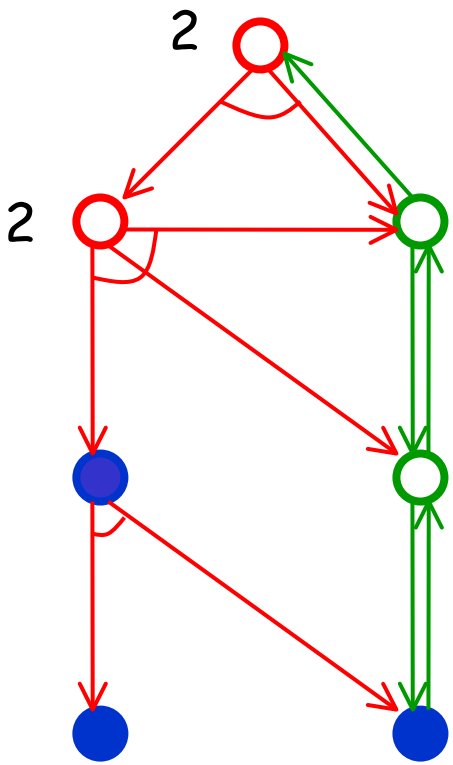
AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

COMPLEXITY?

Interface Compatibility Checking as Game Solving



AND-OR game graph:

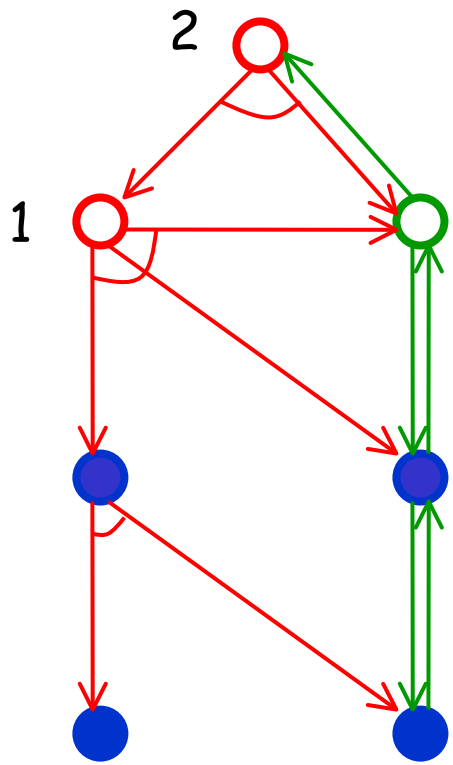
AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

LINEAR TIME (P-COMPLETE)

Interface Compatibility Checking as Game Solving



AND-OR game graph:

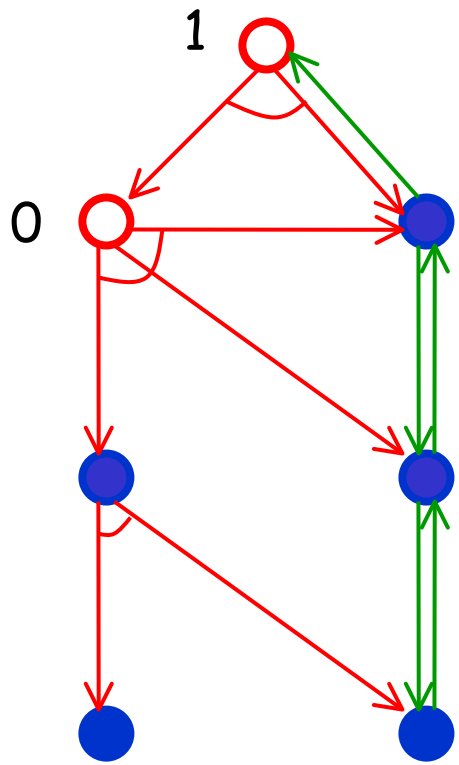
AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

LINEAR TIME (P-COMPLETE)

Interface Compatibility Checking as Game Solving



AND-OR game graph:

AND player -interface product
OR player -environment

Deadlock states:

Does the OR player have a strategy to avoid them?

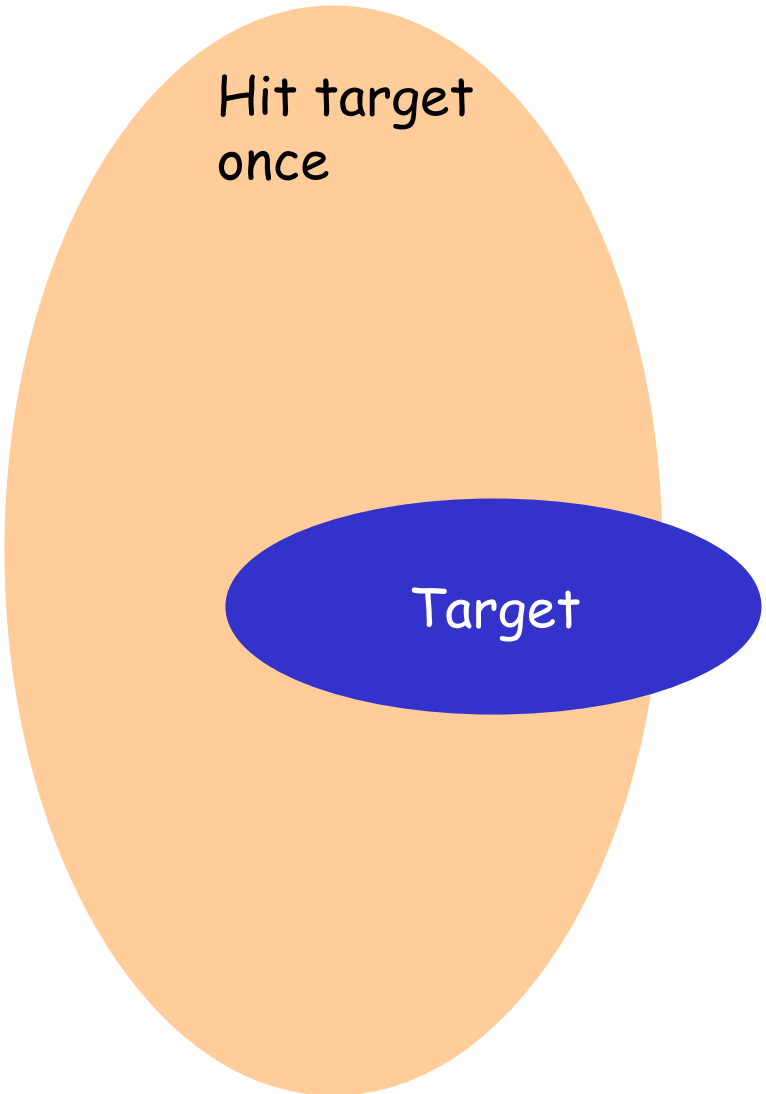
LINEAR TIME (P-COMPLETE)

Asynchronous Interfaces lead to Buechi Games

Environment should not block interface moves:

Does the OR player have a strategy to hit the target states infinitely often?

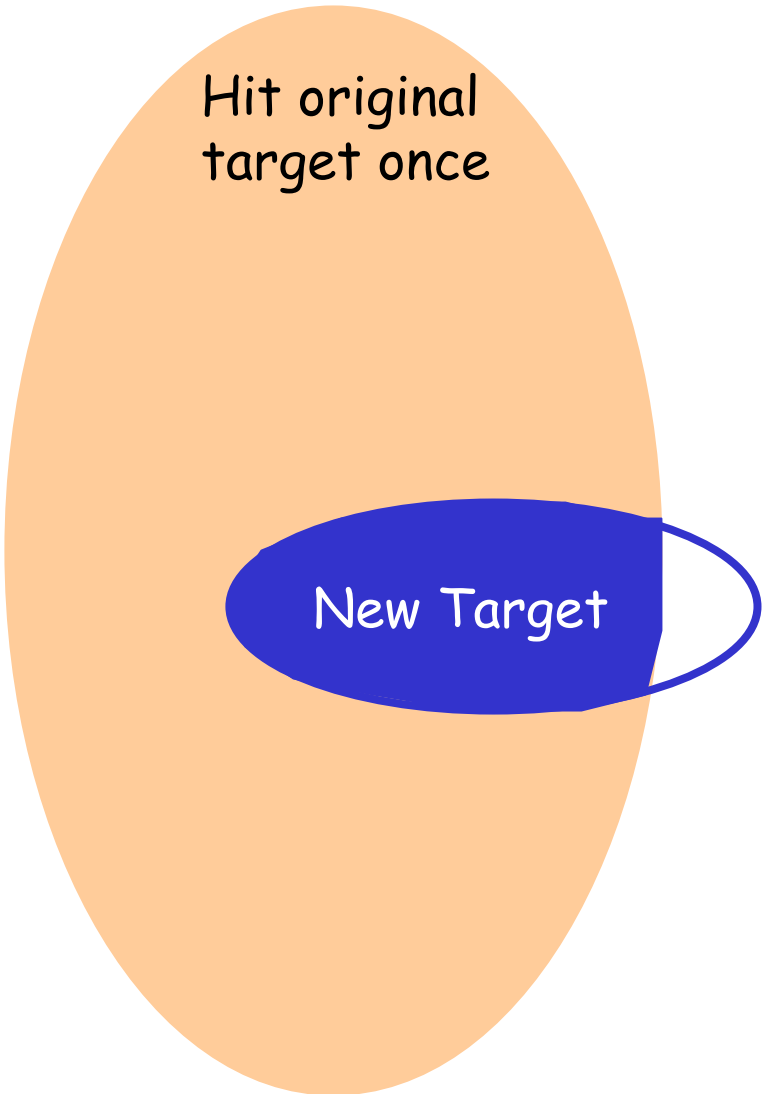
Asynchronous Interfaces lead to Buechi Games



Environment should not block interface moves:

Does the OR player have a strategy to hit the target states infinitely often?

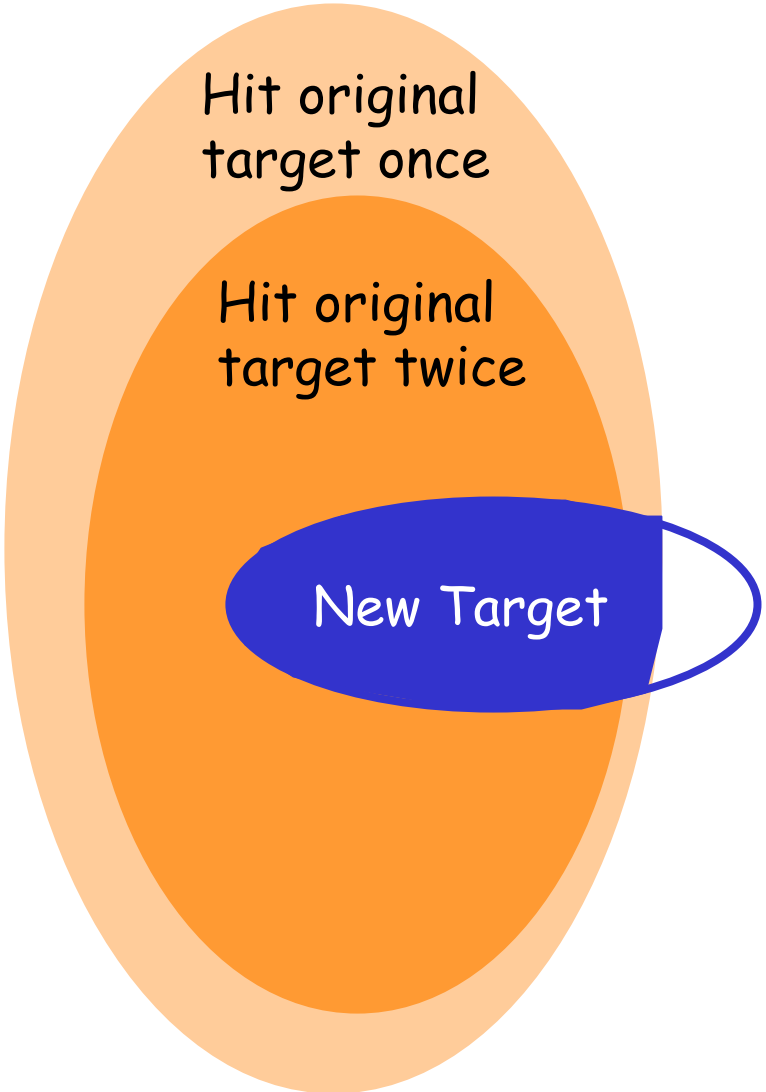
Asynchronous Interfaces lead to Buechi Games



Environment should not block interface moves:

Does the OR player have a strategy to hit the target states infinitely often?

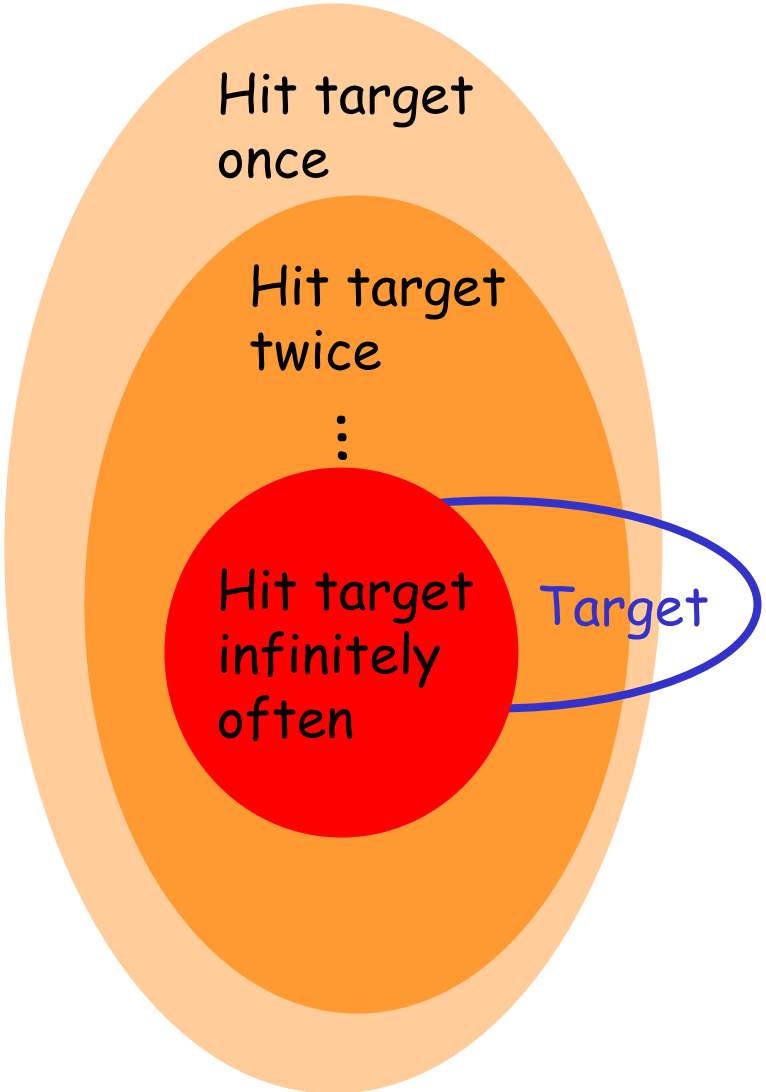
Asynchronous Interfaces lead to Buechi Games



Environment should not block interface moves:

Does the OR player have a strategy to hit the target states infinitely often?

Asynchronous Interfaces lead to Buechi Games



Environment should not block interface moves:

Does the OR player have a strategy to hit the target states infinitely often?

Linear number of finite games:

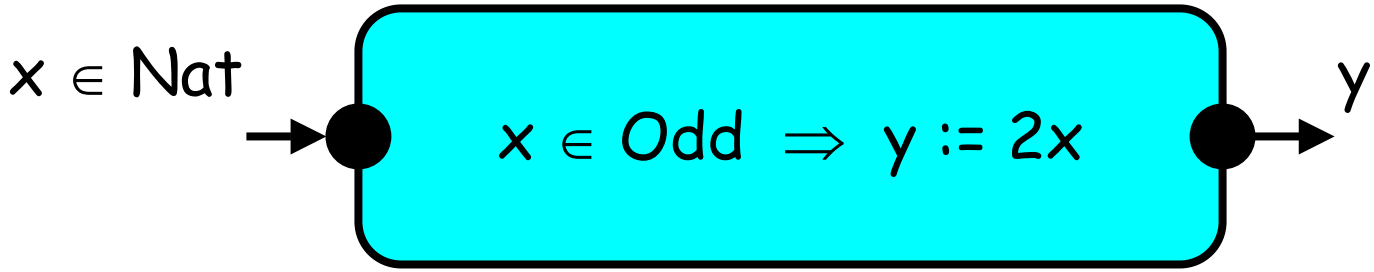
QUADRATIC TIME

Can this be improved ???

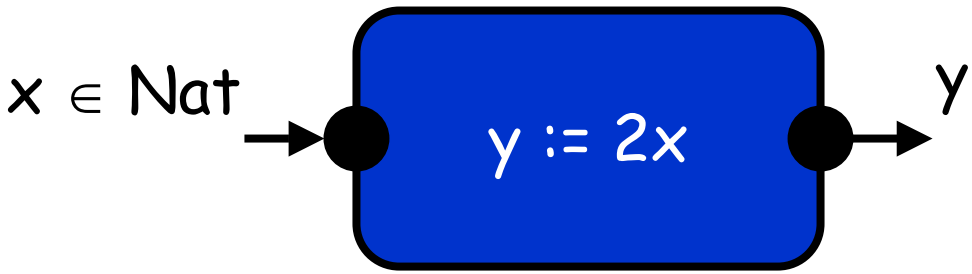
Interface Refinement:

The Block-Diagram View

Component Refinement

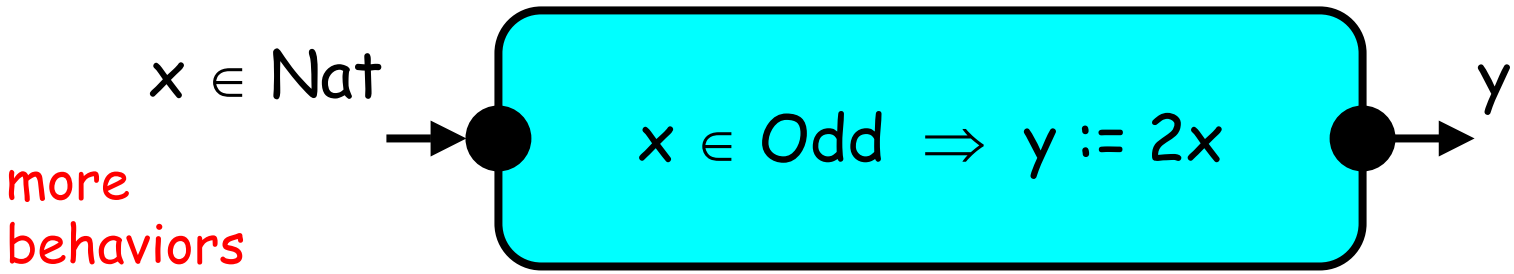


VI



Refinement is implication (simulation; trace containment).

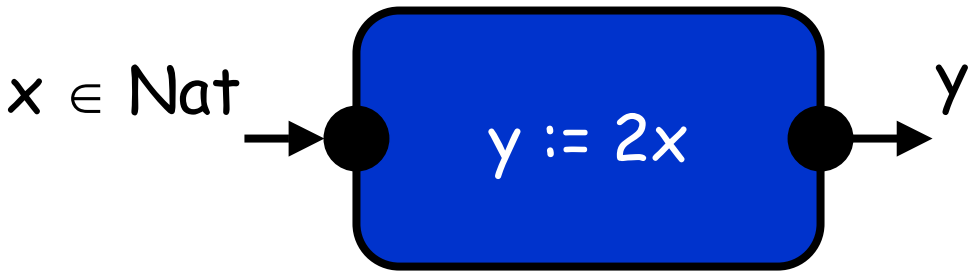
Component Refinement



more behaviors

UI

VI

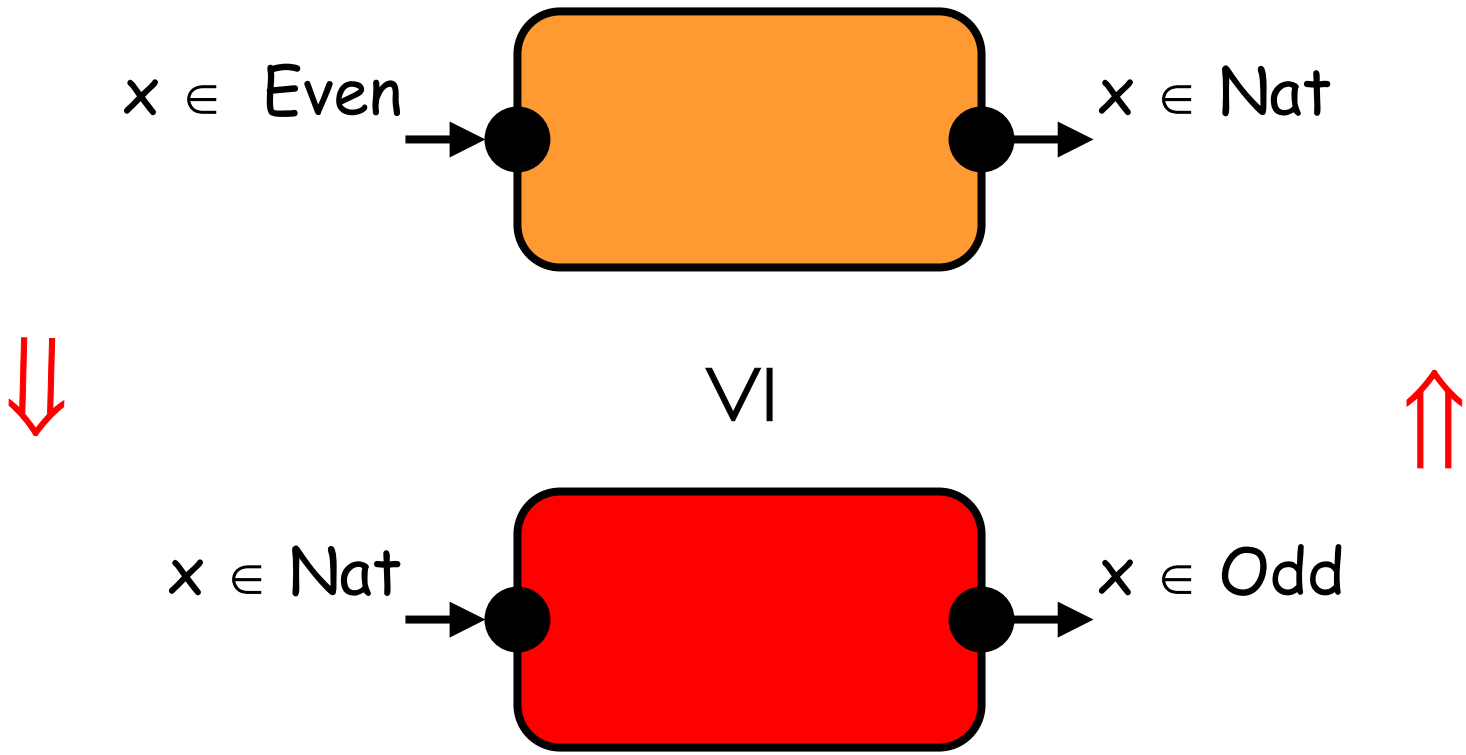


fewer behaviors

⇑

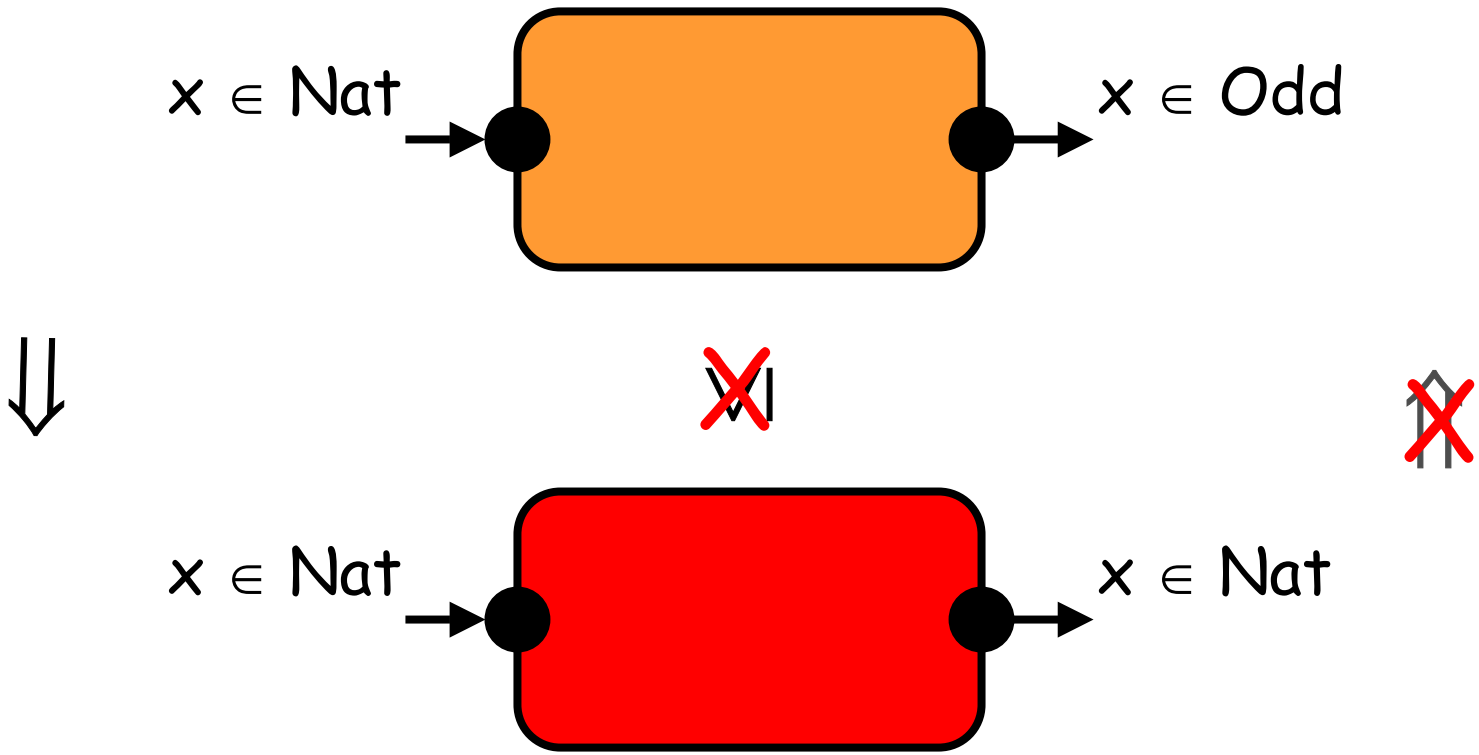
Refinement is implication (simulation; trace containment).

Interface Refinement



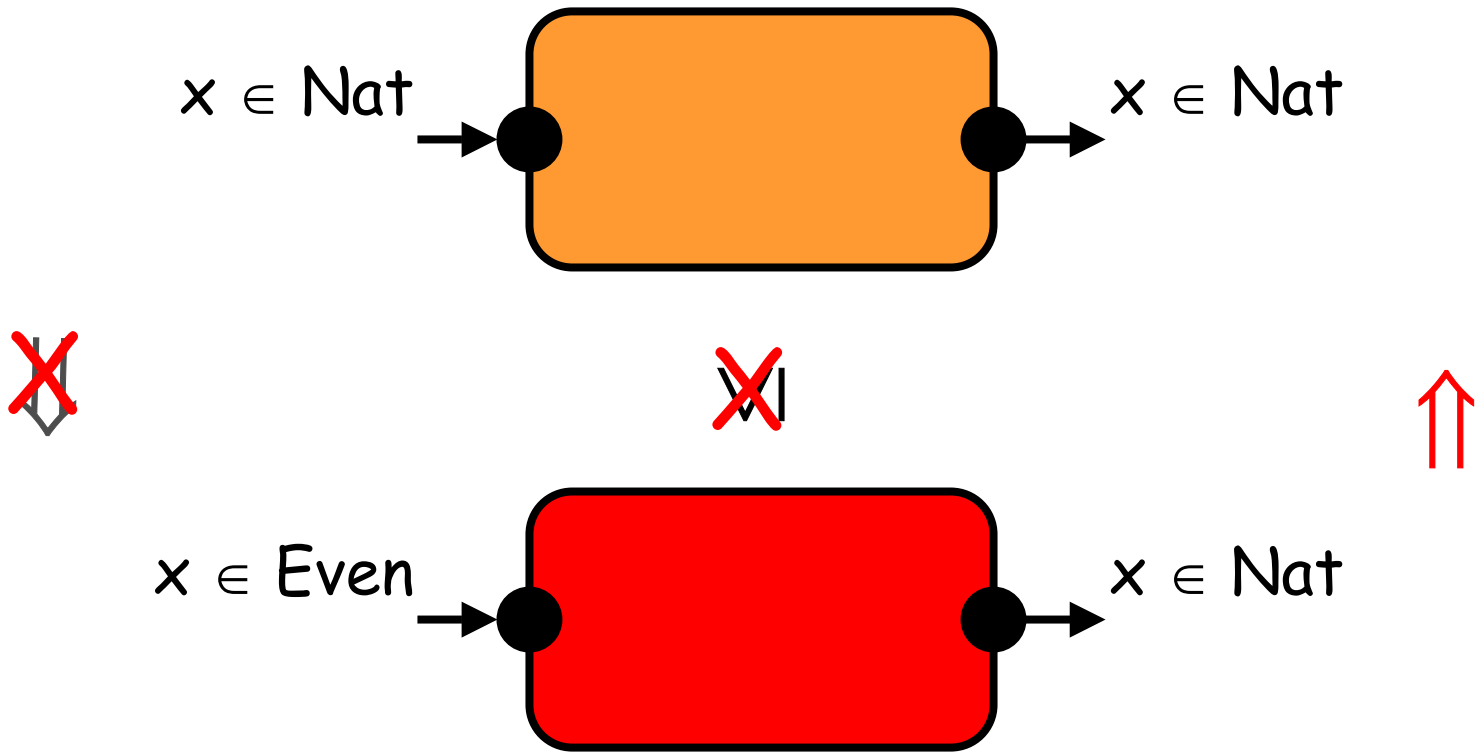
Like subtyping, refinement is I/O contravariant.

Interface Refinement



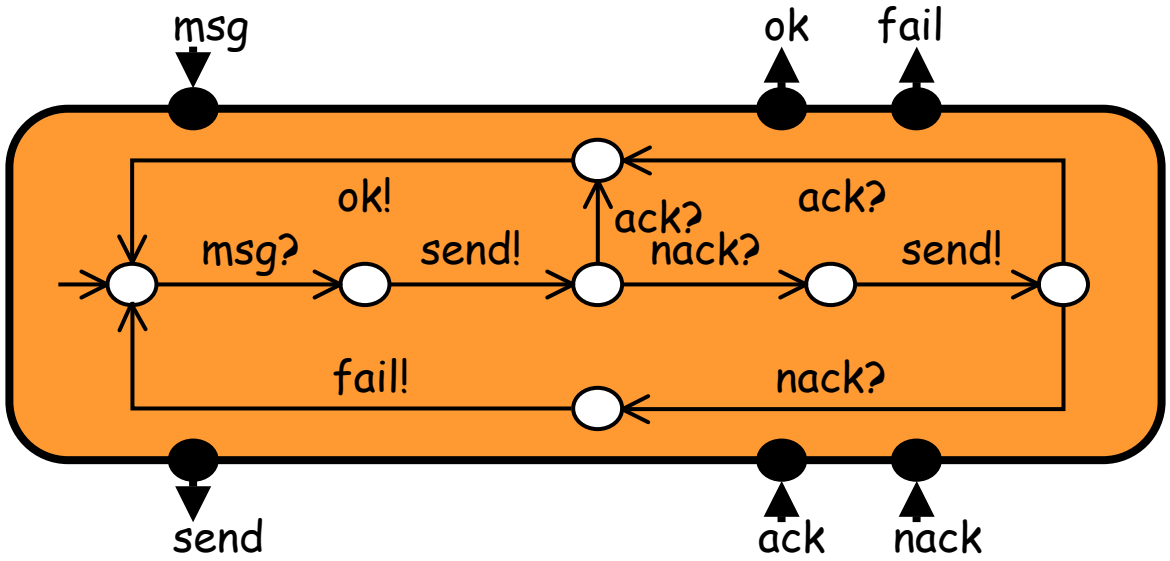
1. Implementation must obey output guarantee.

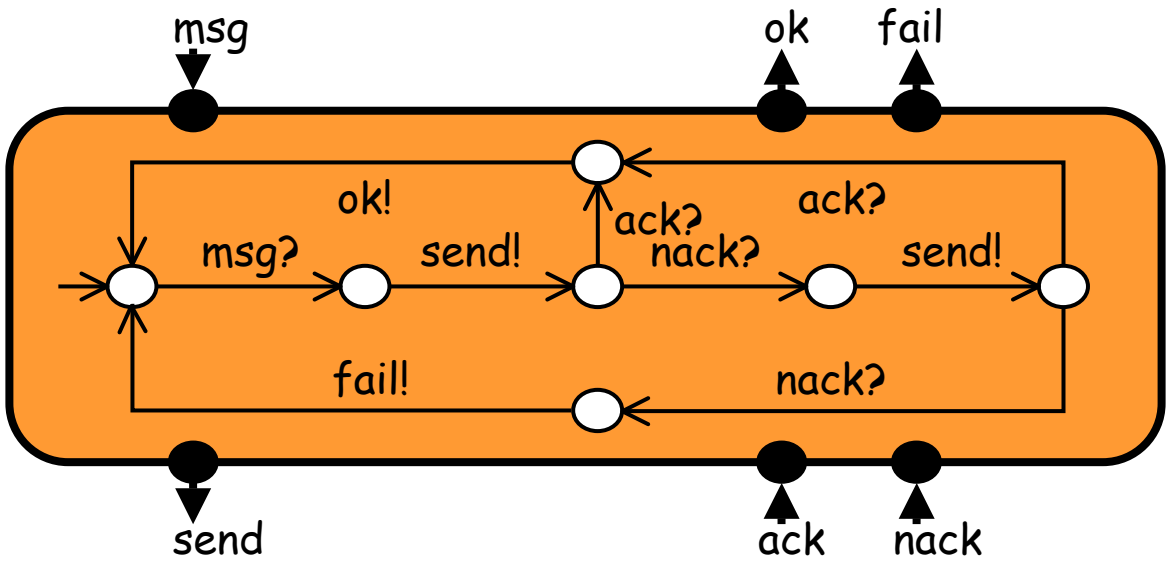
Interface Refinement



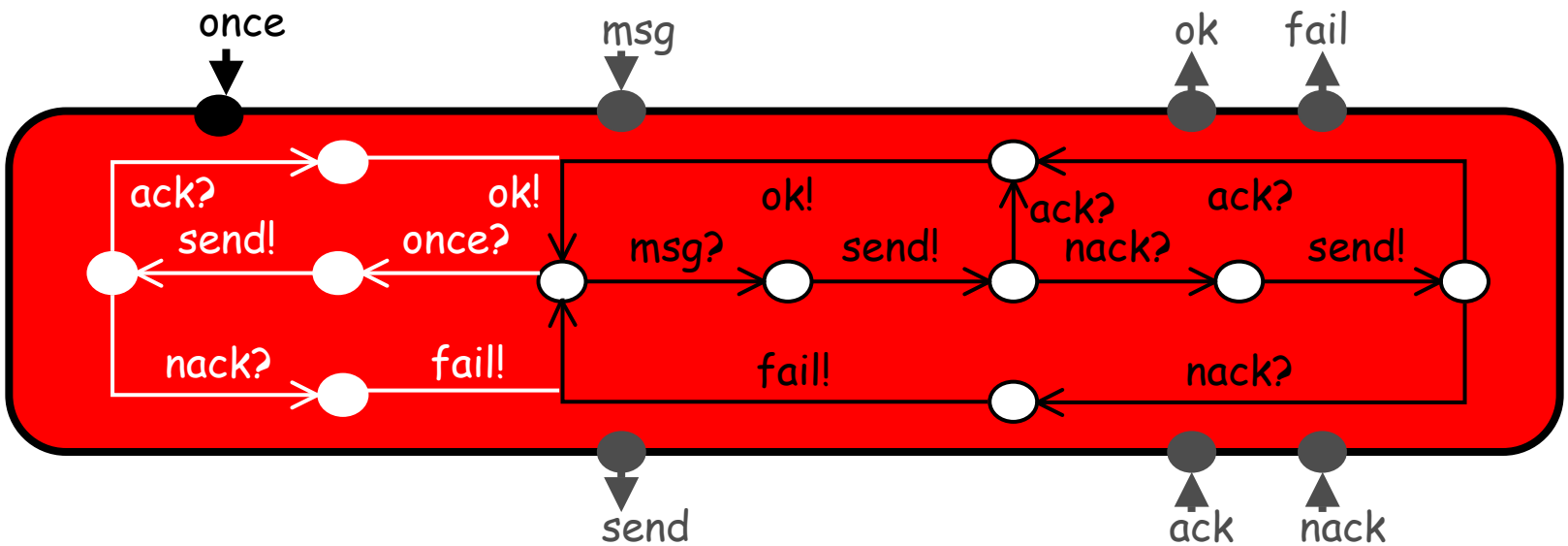
2. Implementation must accept all permissible inputs.

Interface Refinement:
The State-Transition View


msg
ok
fail
ok!
msg?
send!
ack?
nack?
ack?
send!
fail!
nack?
send
ack
nack


msg
ok
fail
ok!
msg?
send!
ack?
nack?
ack?
send!
fail!
nack?
send
ack
nack

VI


once
msg
ok
fail
ack?
send!
ok!
once?
ok!
msg?
send!
ack?
nack?
ack?
send!
nack?
fail!
fail!
nack?
send
ack
nack

I/O Alternating Simulation

$Q \leq q$

iff

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

and

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

I/O Alternating Simulation

$Q \leq q$

iff

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

and

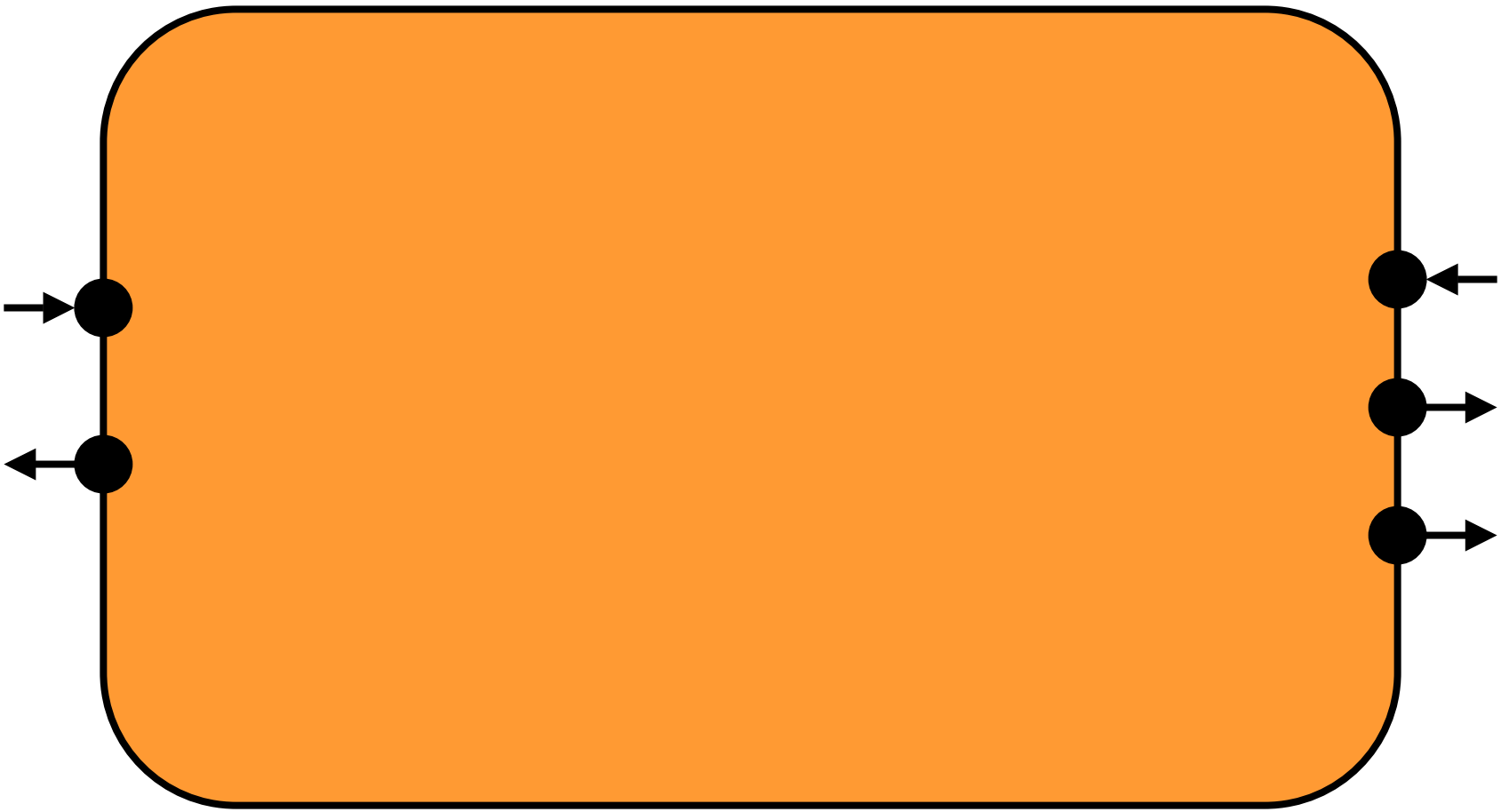
2. for all outputs o, if Q –o!-> Q' , then there exists q' such that q –o!-> q' and $Q' \leq q'$.

If there is a helpful environment at q, then there is a helpful environment at Q. This can be checked in quadratic time [Alur/Henzinger/Kupferman/Vardi '98].

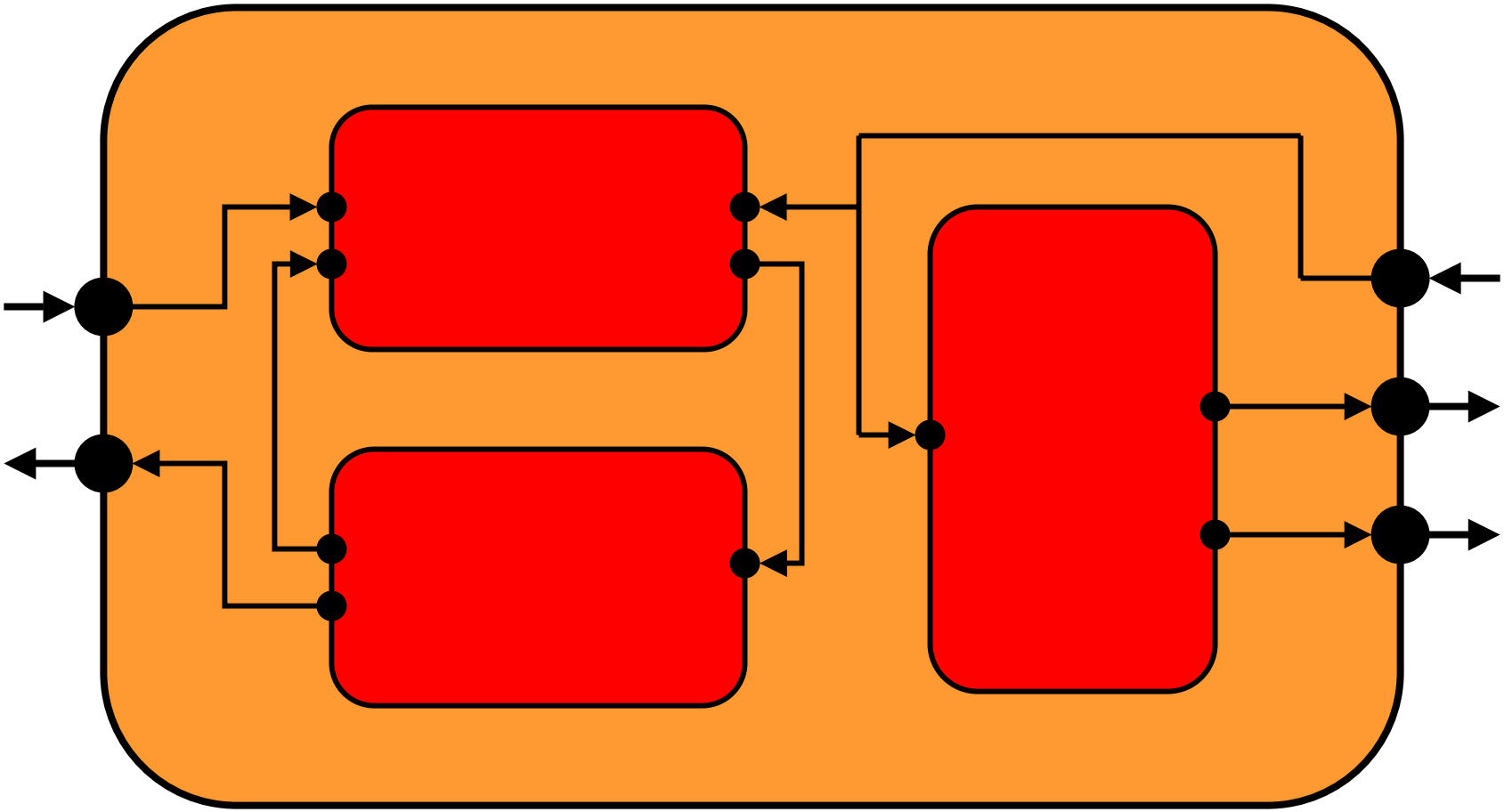
Compositionality:

The Interplay between Composition and Refinement

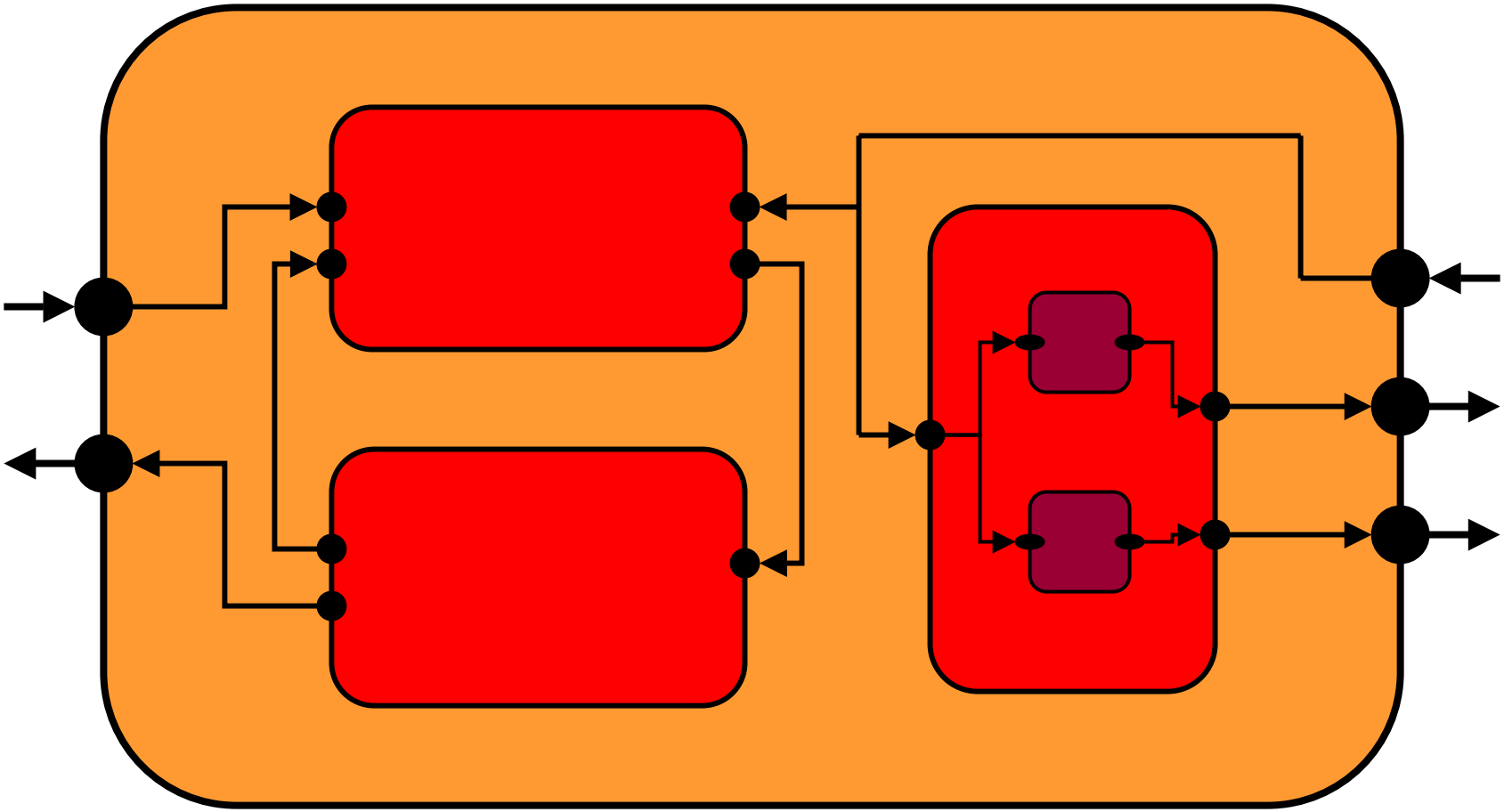
Top-down Design



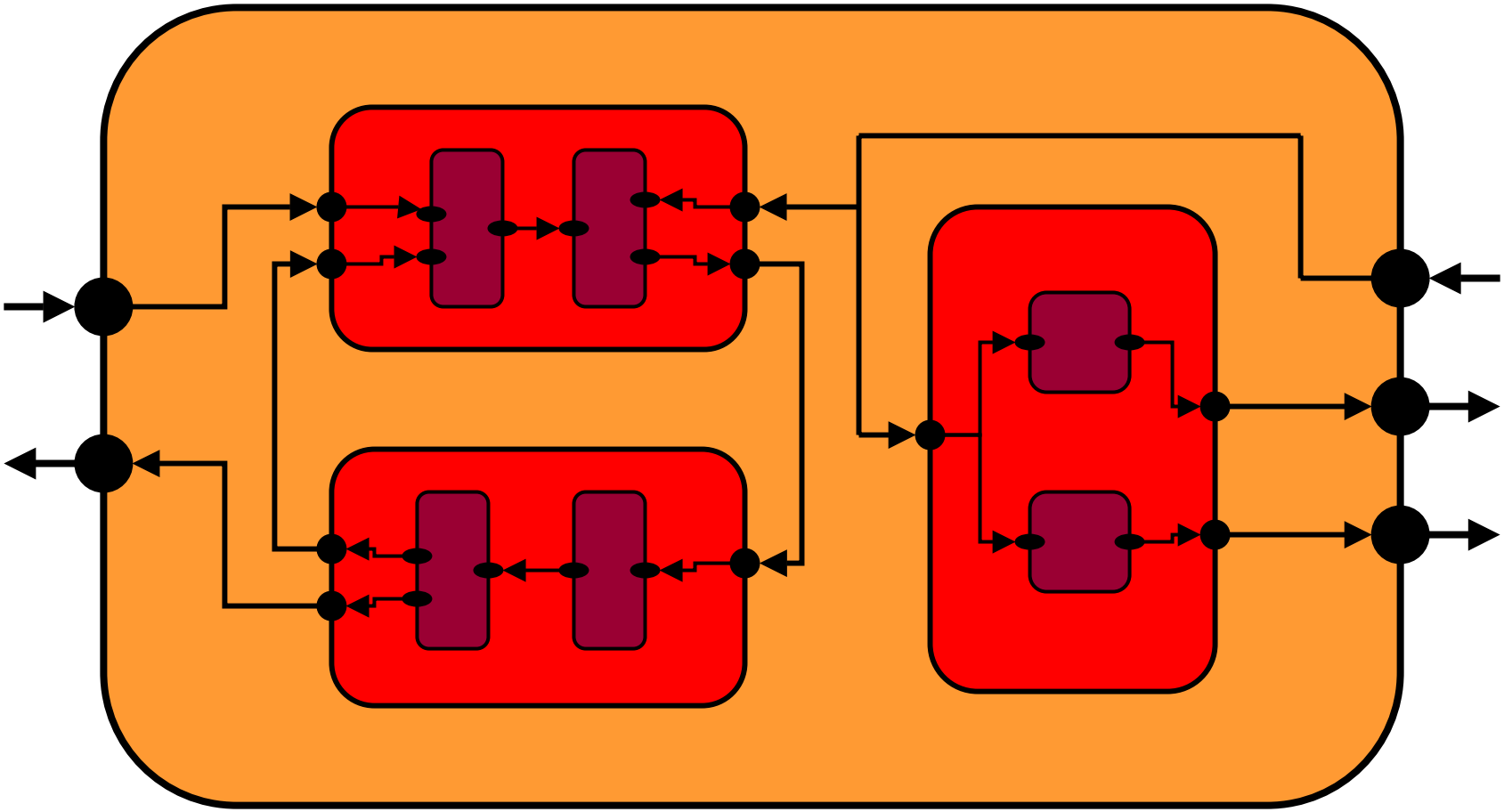
Top-down Design



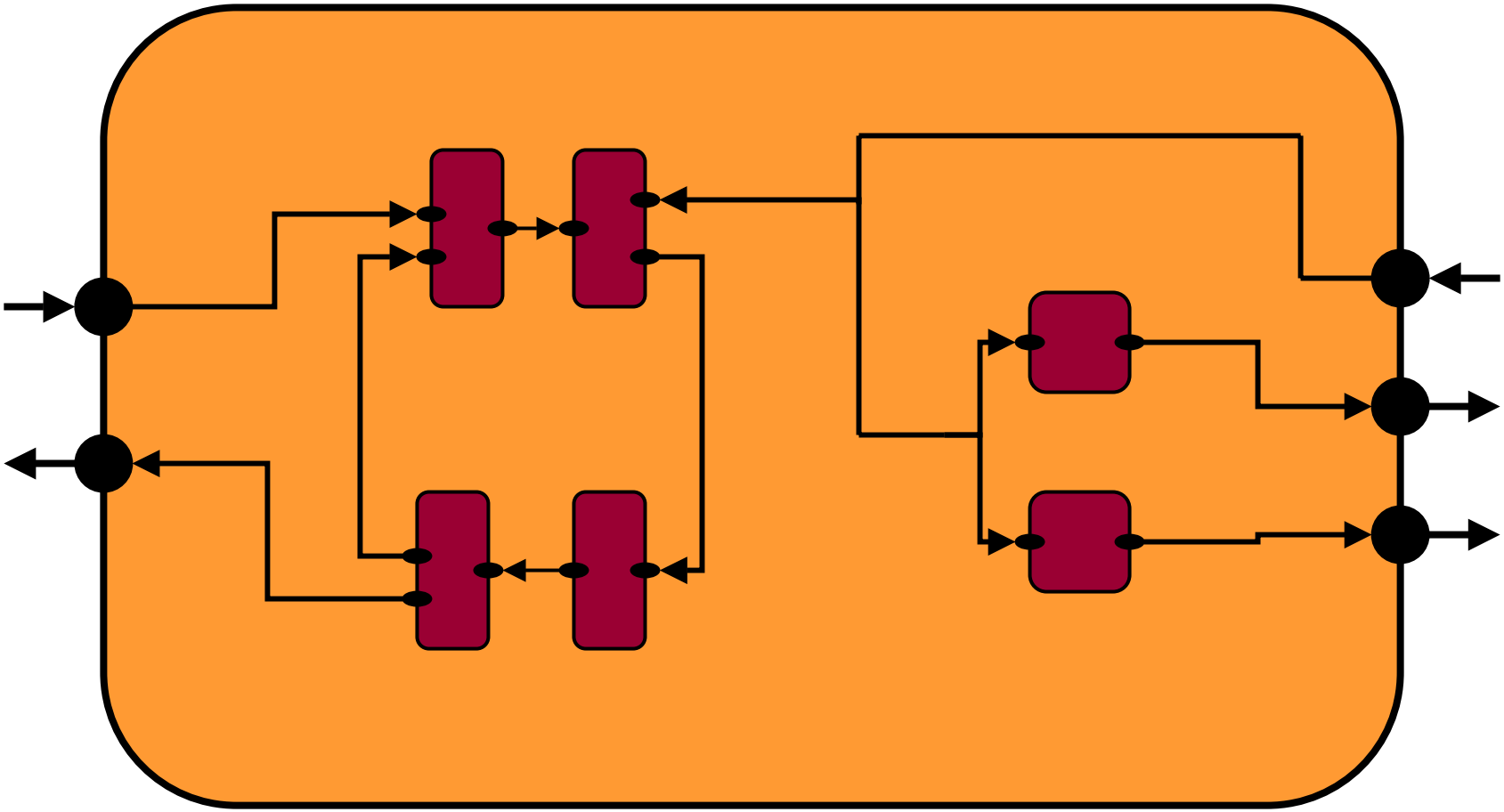
Top-down Design



Top-down Design



Top-down Design



Compositionality for Top-down Design

If $a||b$ is defined and $A \leq a$ and $B \leq b$,
then $A||B$ is defined and $A||B \leq a||b$.

-enables independent interface implementation
-examples: subtypes, interface automata

Compositionality for Top-down Design

If a||b is defined and $A \leq a$ and $B \leq b$,
then A||B is defined and $A||B \leq a||b$.

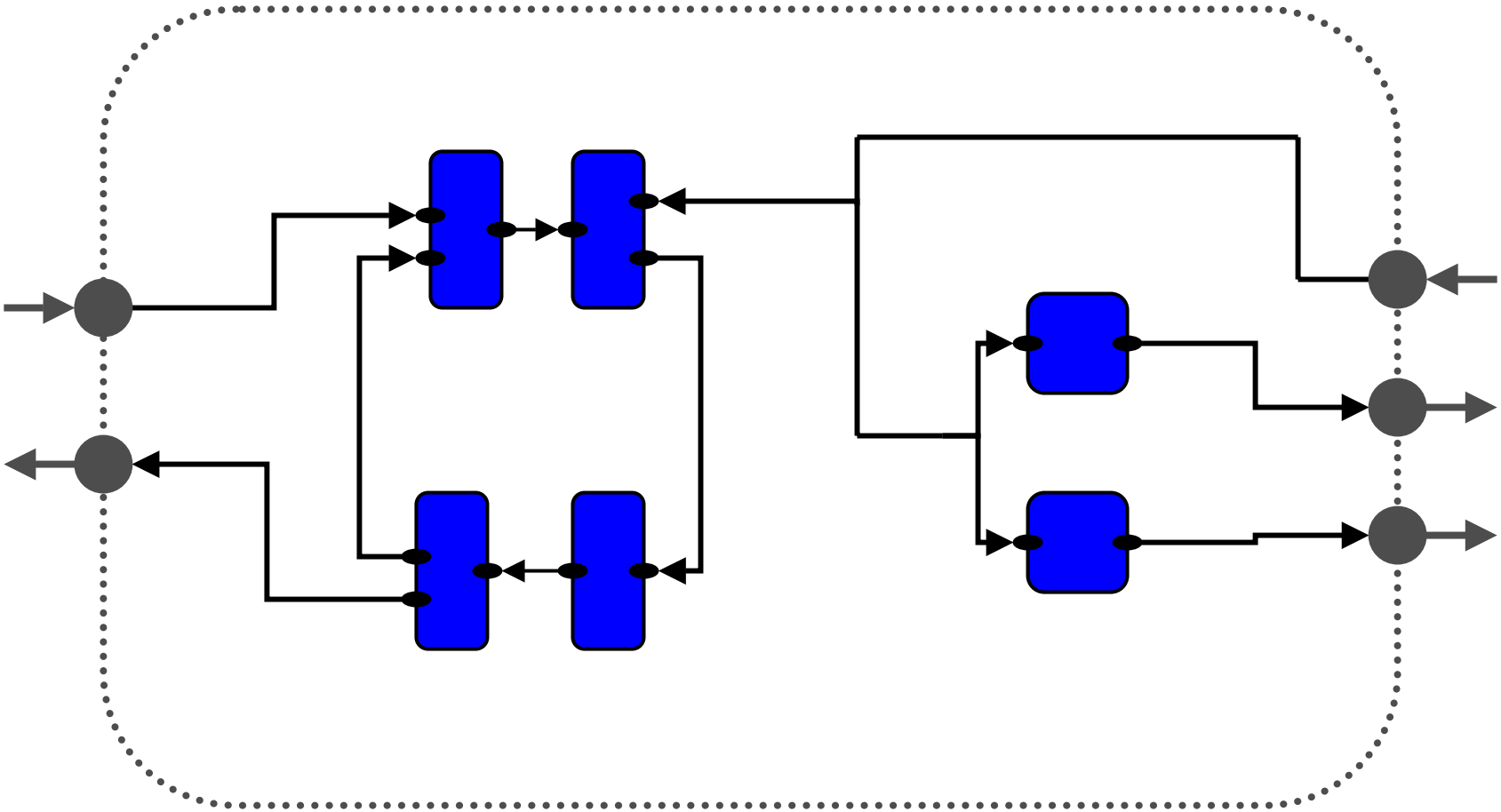
-enables independent interface implementation
-examples: subtypes, interface automata

Compositionality in Formal Methods

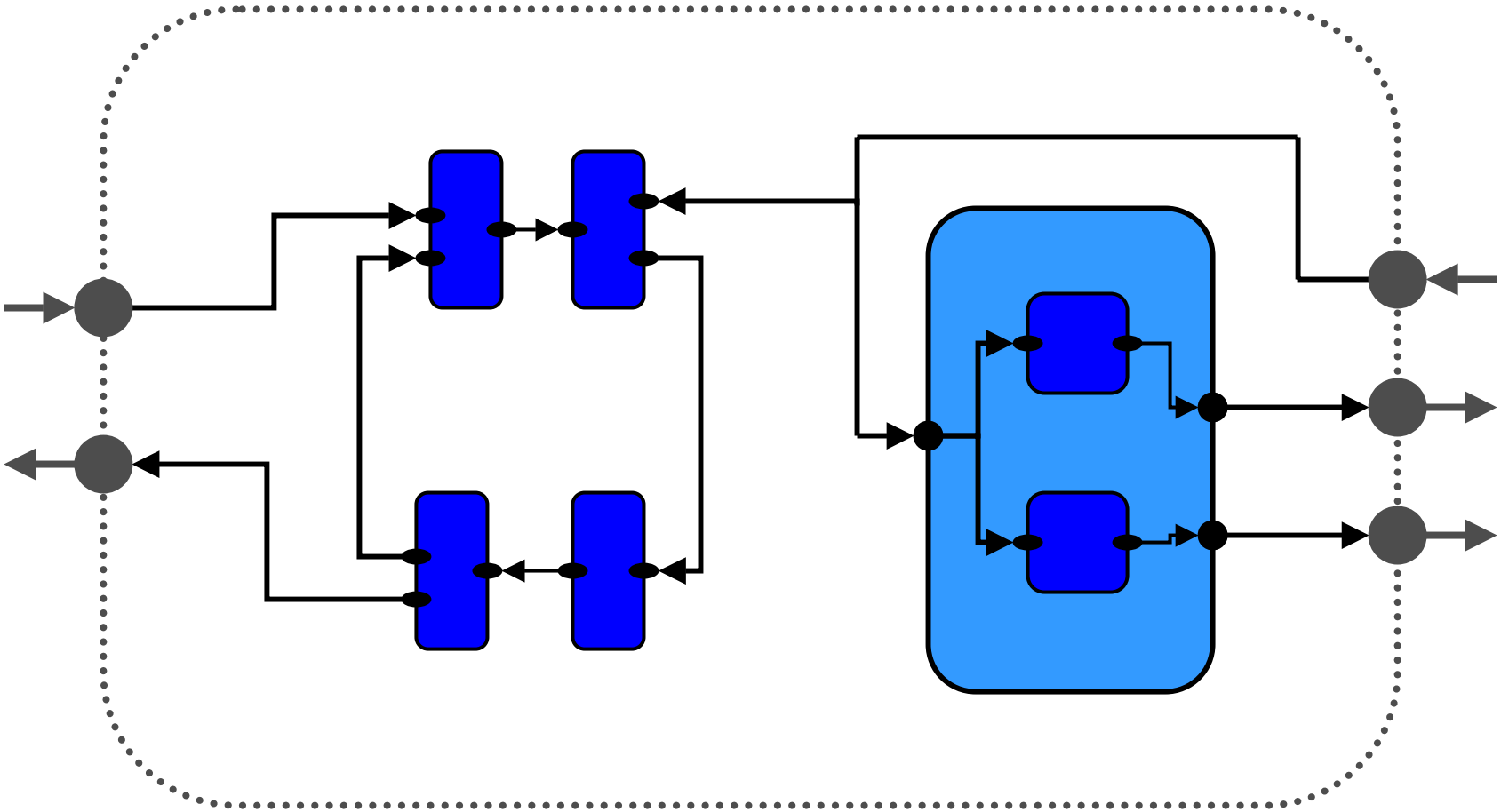
If A||B is defined and $A \leq a$ and $B \leq b$,
then a||b is defined and $A||B \leq a||b$.

-enables independent component verification
-examples: I/O Automata, CSP, Reactive Modules, etc.

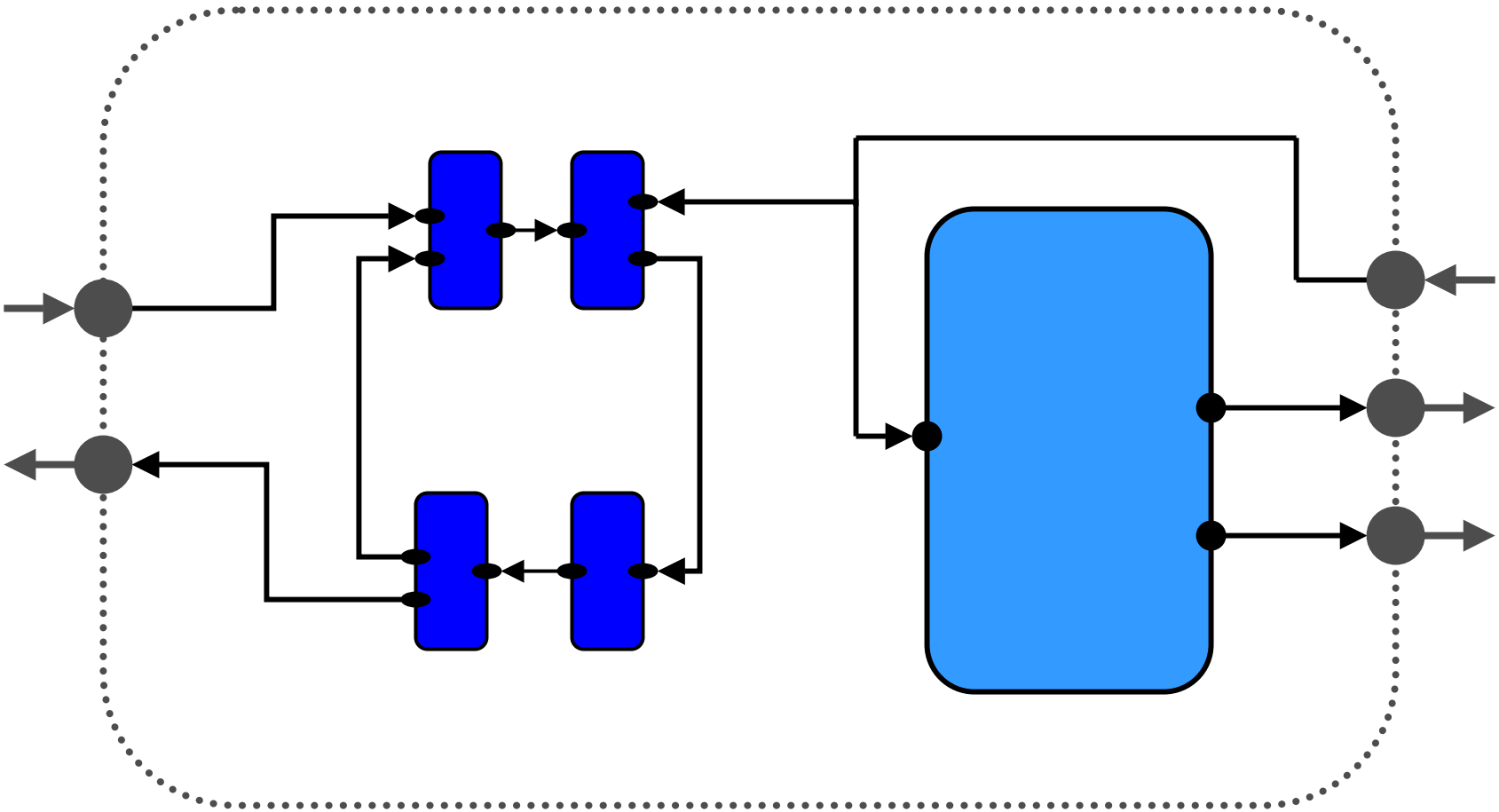
Compositional Verification



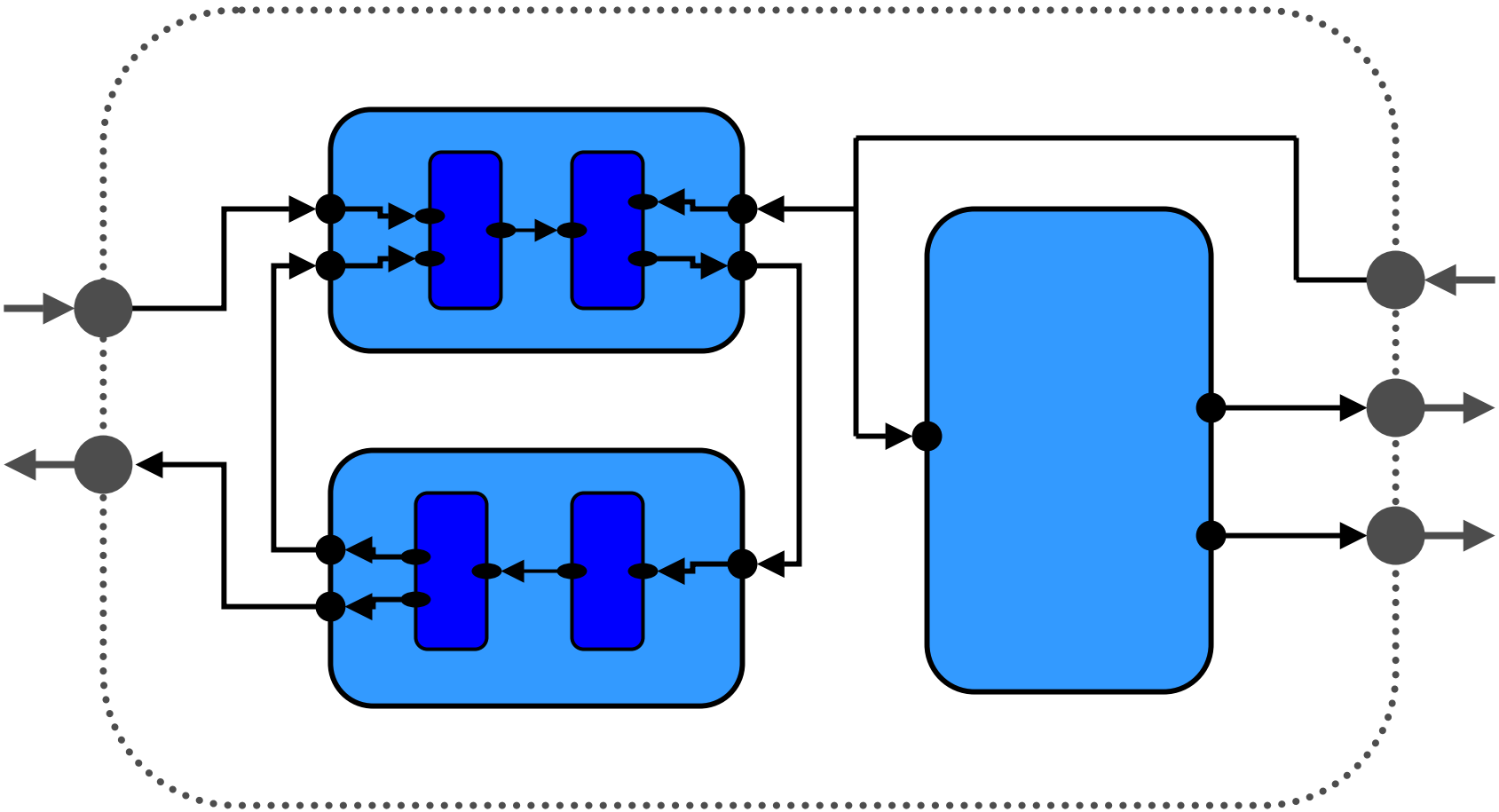
Compositional Verification



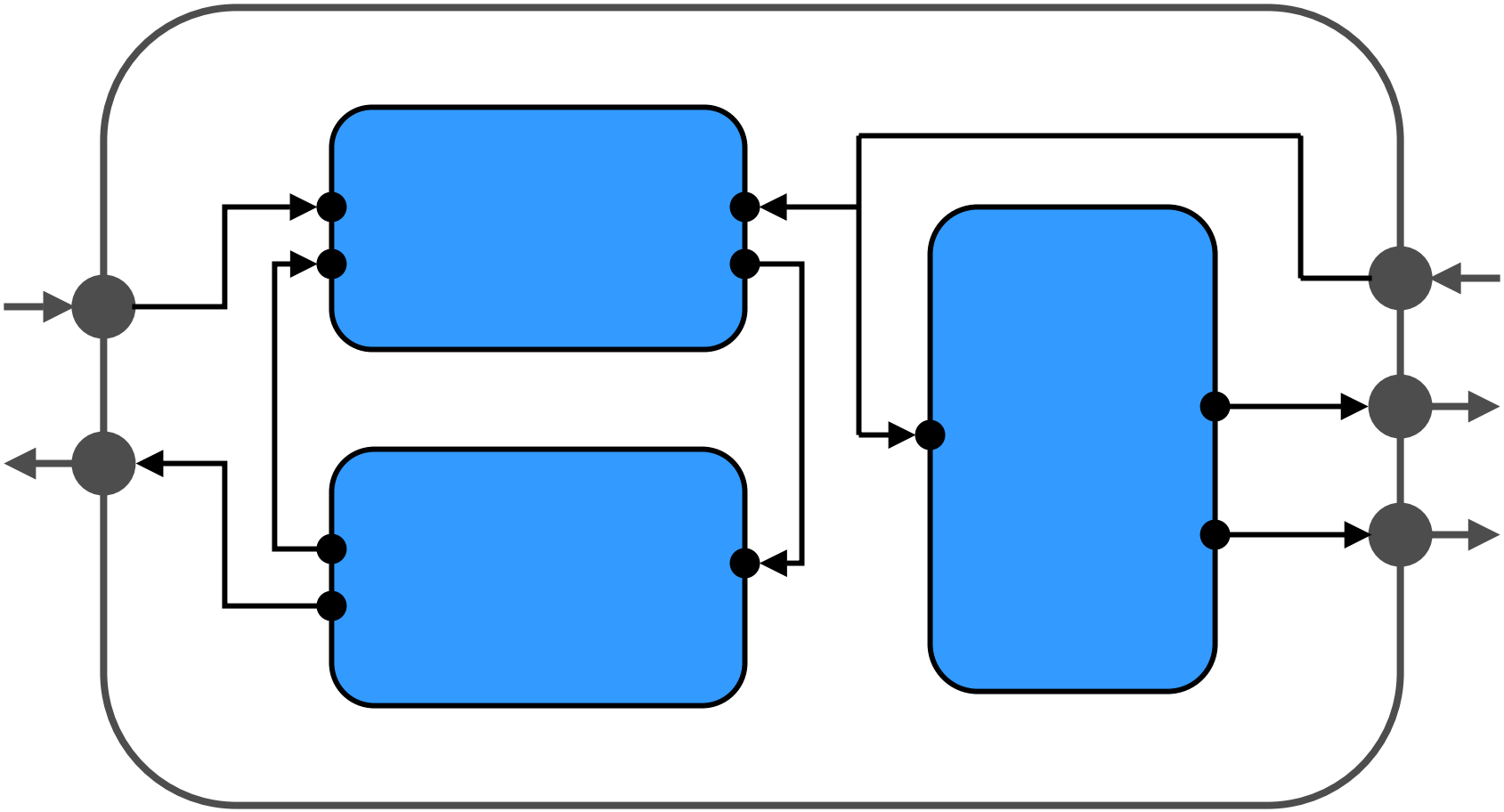
Compositional Verification



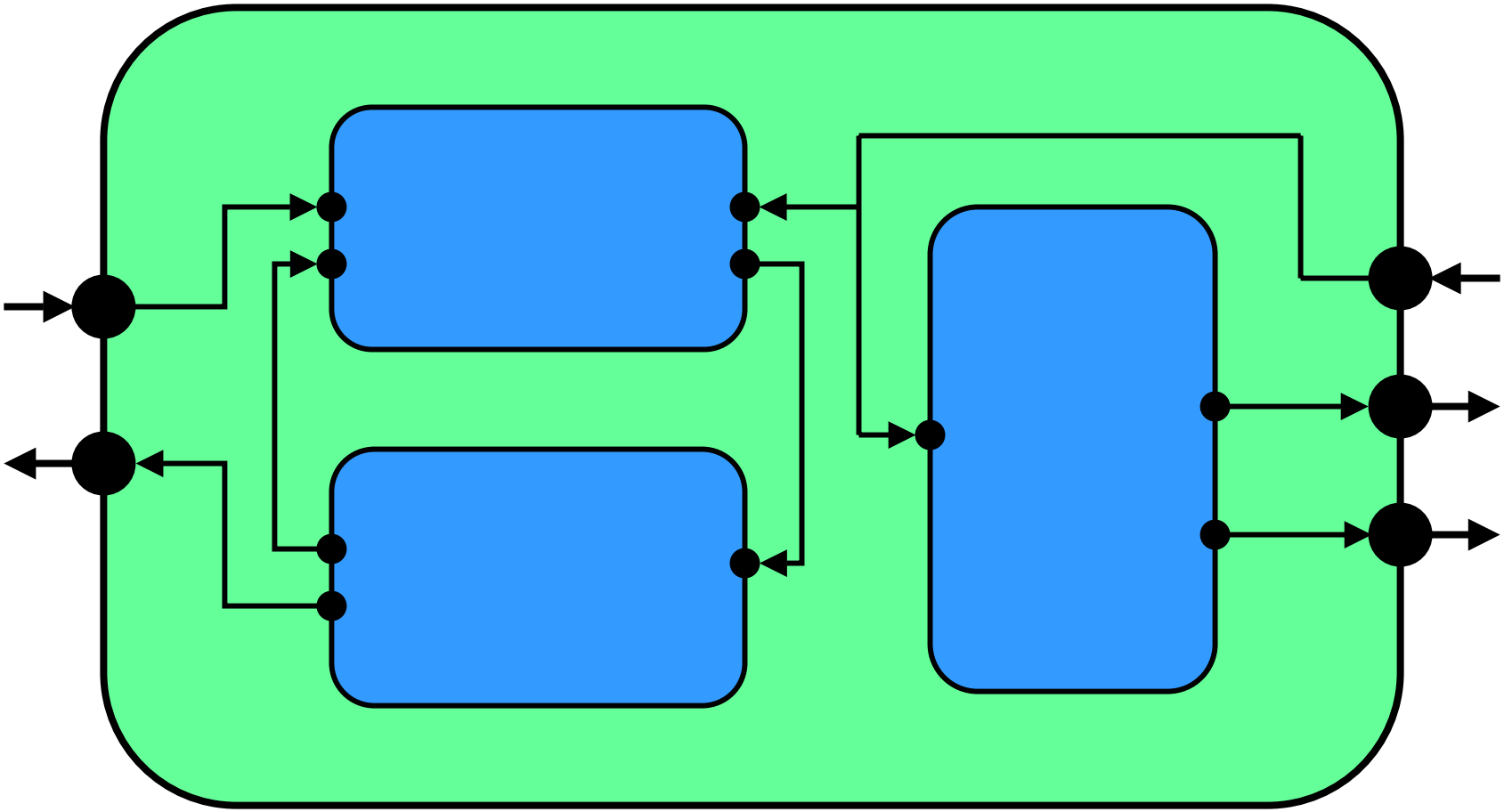
Compositional Verification



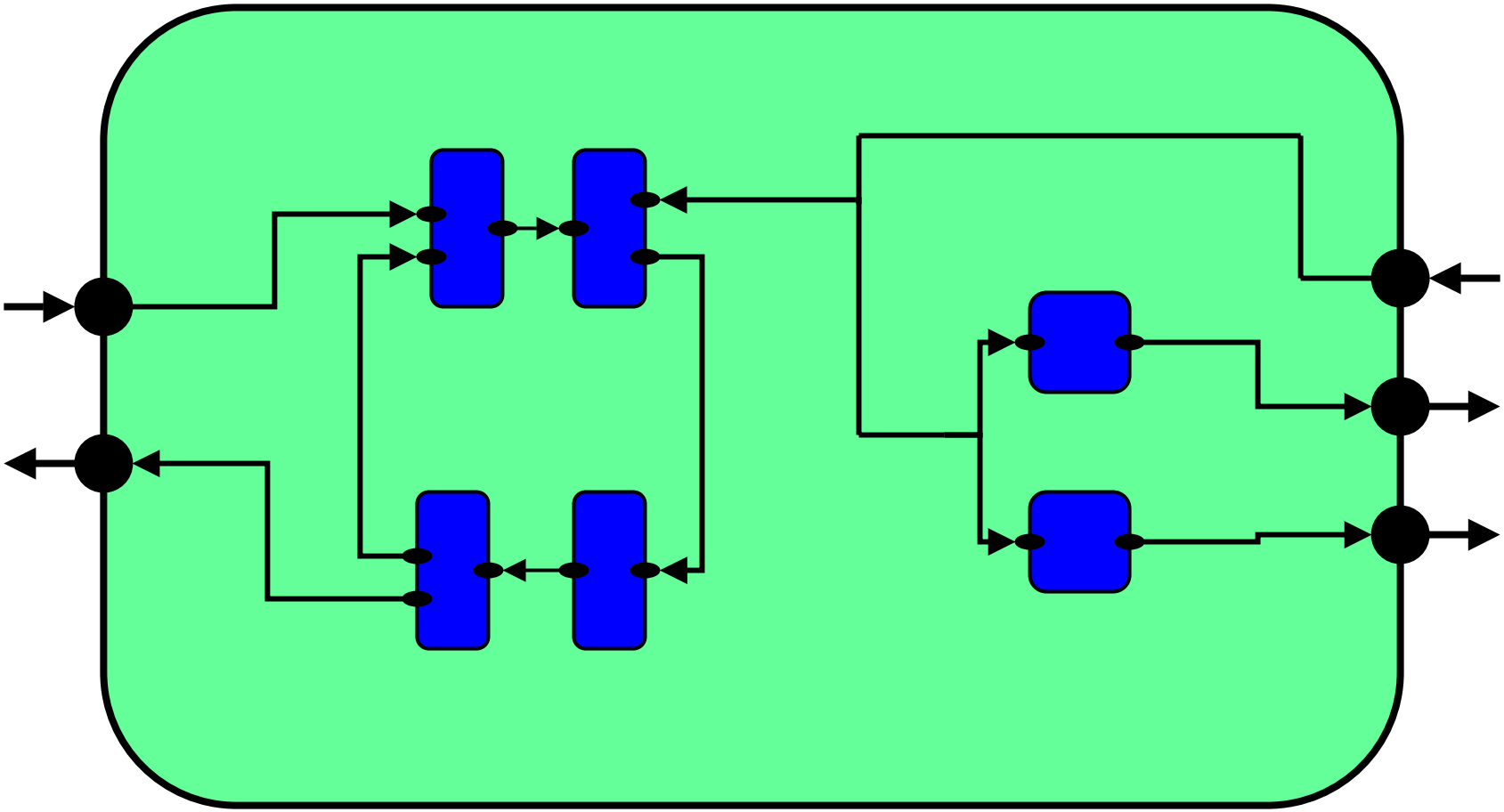
Compositional Verification



Compositional Verification



Compositional Verification



Compositionality for Bottom-up Verification

If $A||B$ is defined and $A \leq a$ and $B \leq b$,
then $a||b$ is defined and $A||B \leq a||b$.

$$\frac{(A \wedge B) \neq \text{false} \quad A \Rightarrow a \quad B \Rightarrow b}{(a \wedge b) \neq \text{false} \quad (A \wedge B) \Rightarrow (a \wedge b)}$$

"Definition" of Component Formalism:
Bottom-up Compositional

If $A||B$ is defined and $A \leq a$ and $B \leq b$,
then $a||b$ is defined and $A||B \leq a||b$.

"Definition" of Interface Formalism:
Top-down Compositional

If $a||b$ is defined and $A \leq a$ and $B \leq b$,
then $A||B$ is defined and $A||B \leq a||b$.

What is yours?

Applications of Interfaces, So Far

Software module interfaces

-Java extension with interface compatibility checking in Jbuilder
-interfaces for TinyOS, an OS for ad-hoc networks [Culler]

Hardware module interfaces

-bidirectional interfaces for PCI bus and clients
-interfaces with timing constraints for TTA, an architecture for safety-critical embedded systems [Kopetz]

References

de Alfaro, Henzinger,
"Interface Automata",
Foundations of Software Engineering (FSE) 2001.

de Alfaro, Henzinger,
"Interface Theories for Component-based Design",
Embedded Software (EMSOFT) 2001.

Chakrabarti, de Alfaro, Henzinger, Jurdzinski, Mang,
"Interface Compatibility Checking for Software Modules",
Computer-Aided Verification (CAV) 2002.

www.eecs.berkeley.edu/~tah