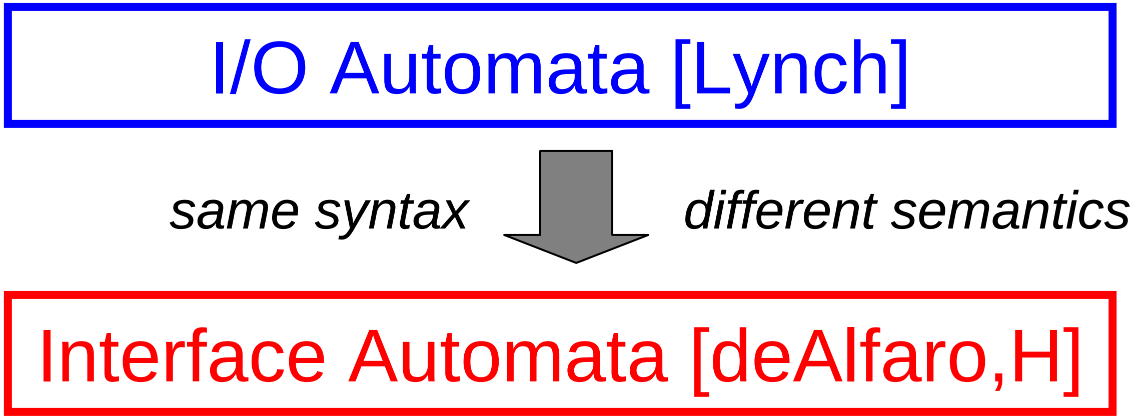
Interface-based Design 4

Tom Henzinger
EPFL and UC Berkeley

An interface algebra can be built around any (?) model of concurrency.

Example:



I/O Automata [Lynch]

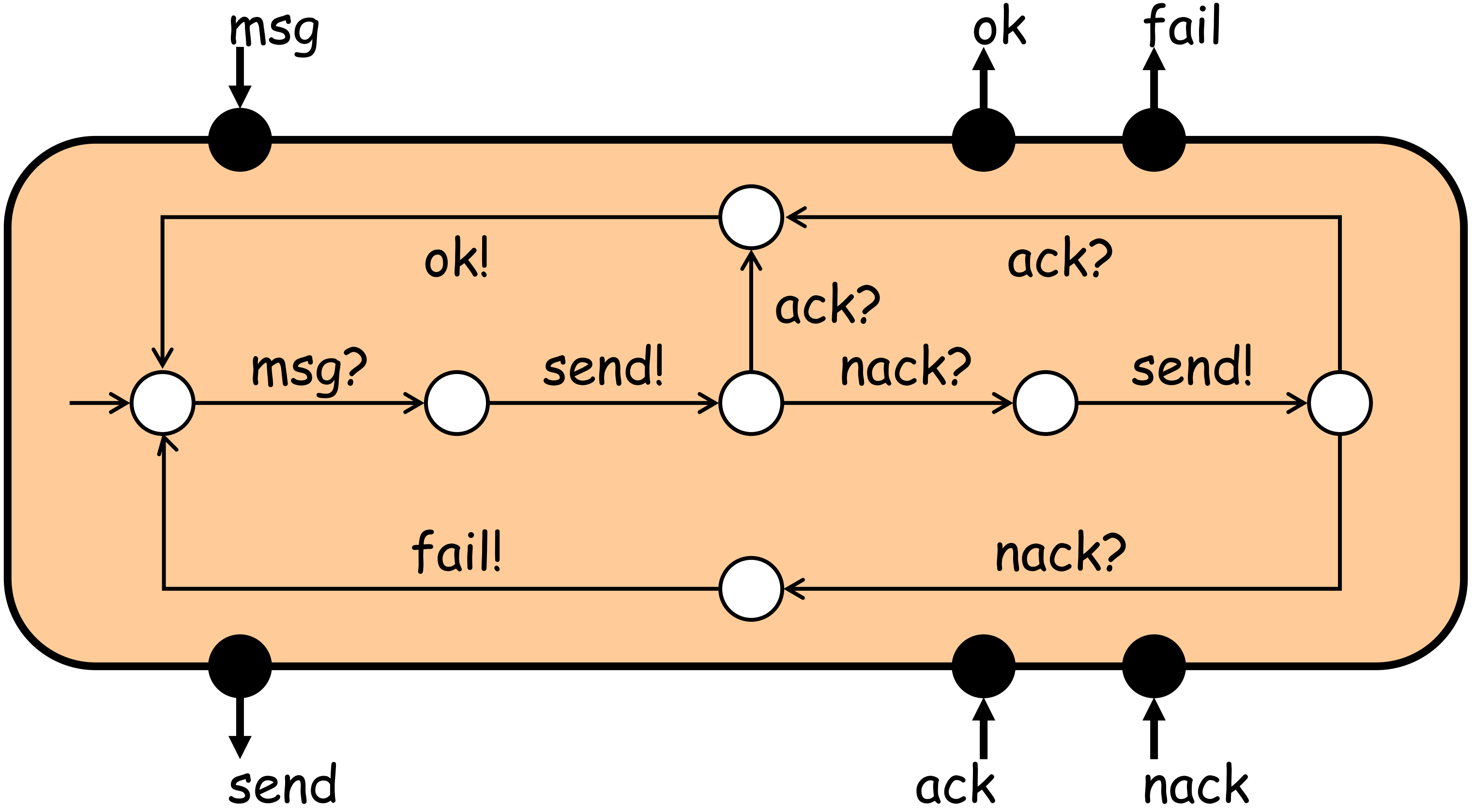
same syntax

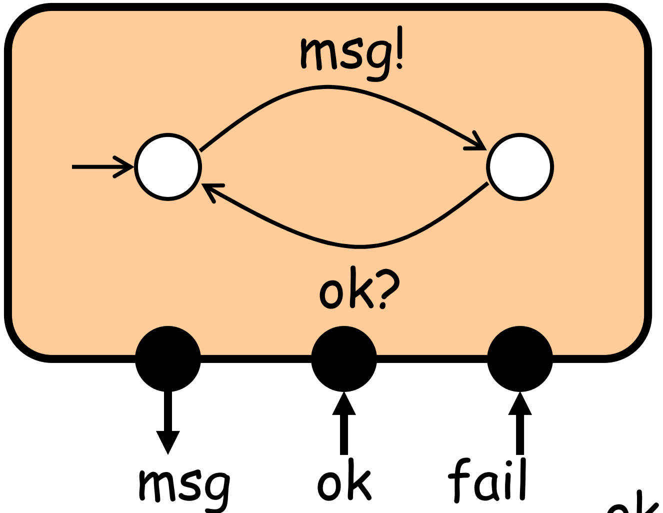
different semantics

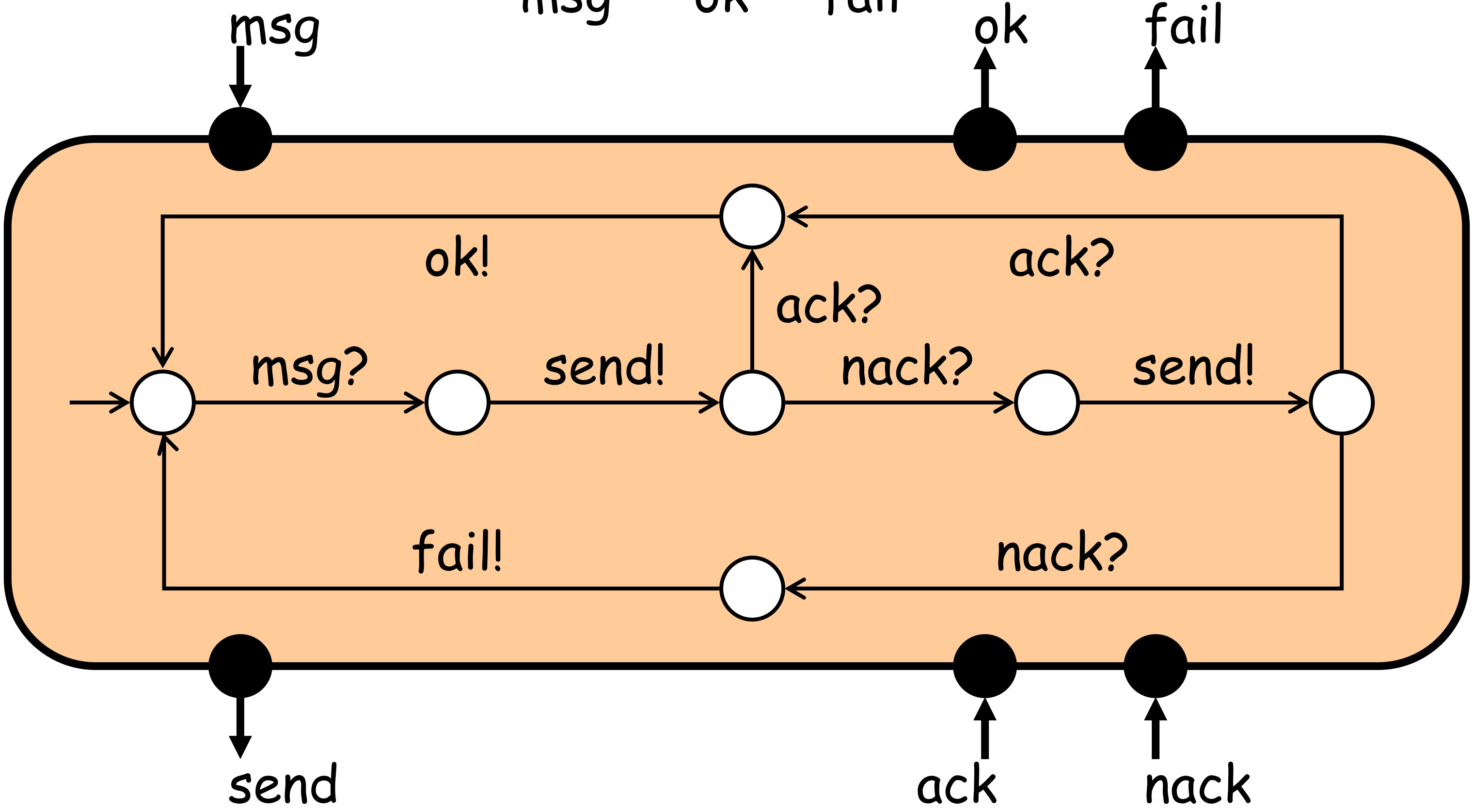
Interface Automata [deAlfaro,H]

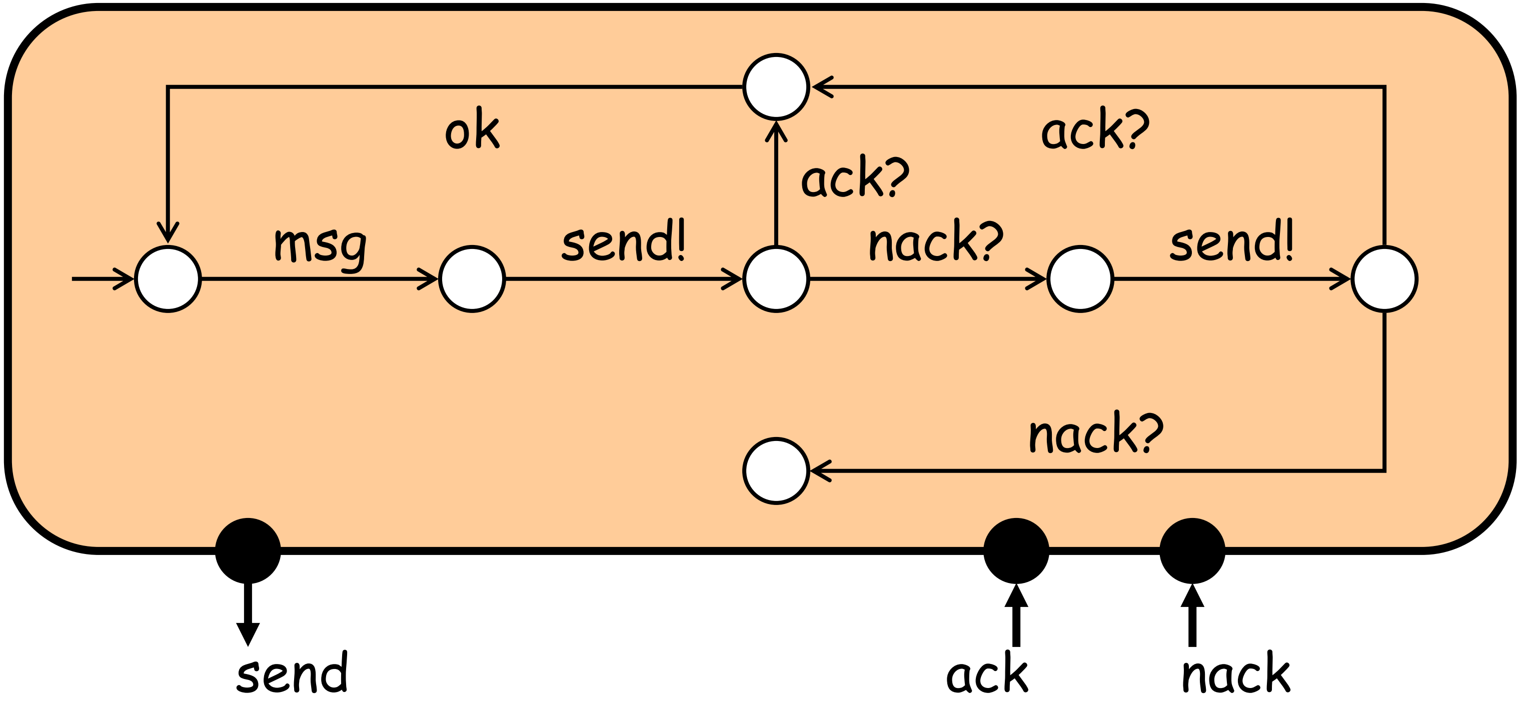
-Total composition
-A process algebra

-Partial composition
-Compatibility check
-An interface algebra

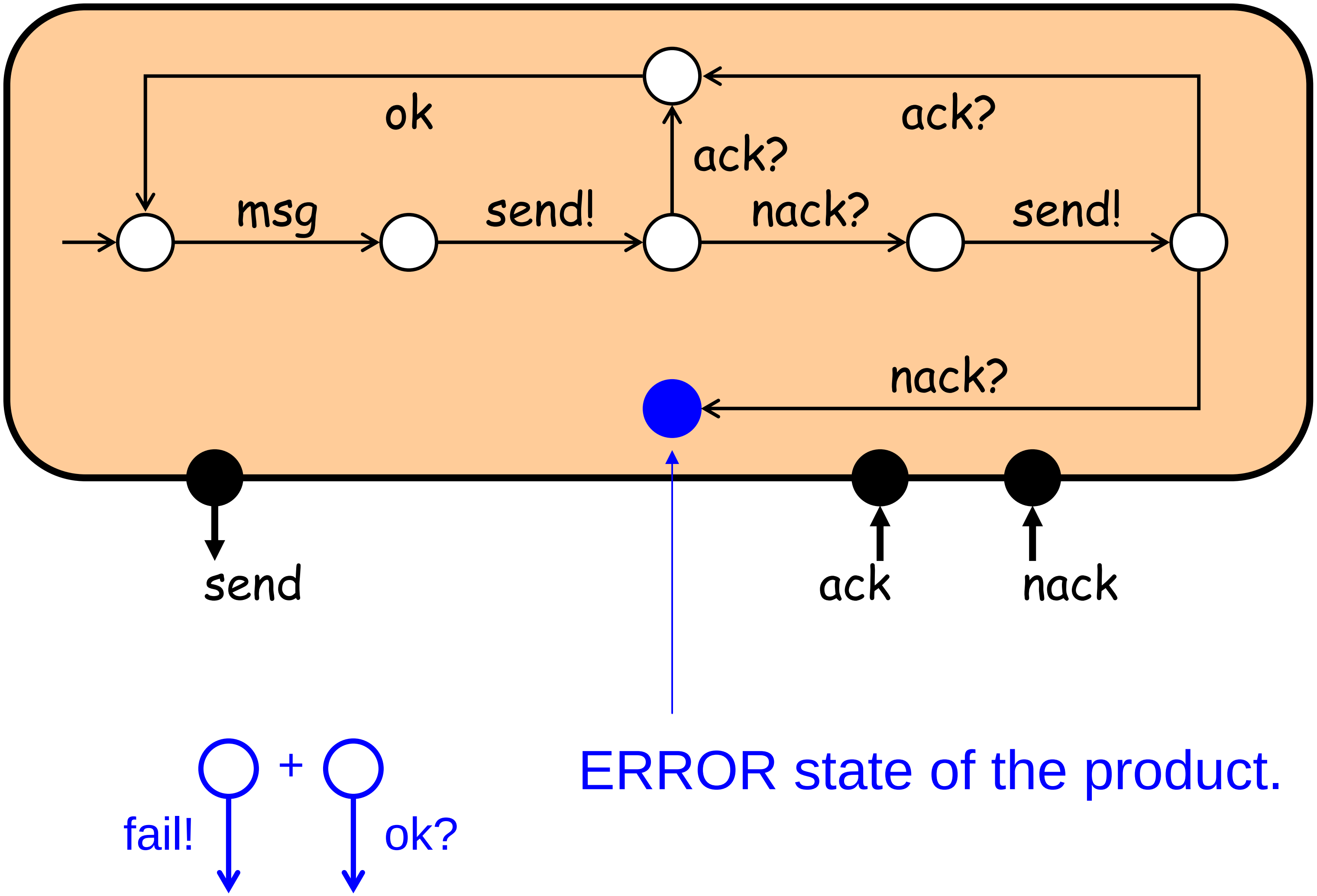

msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack


msg!
ok?
msg
ok
fail

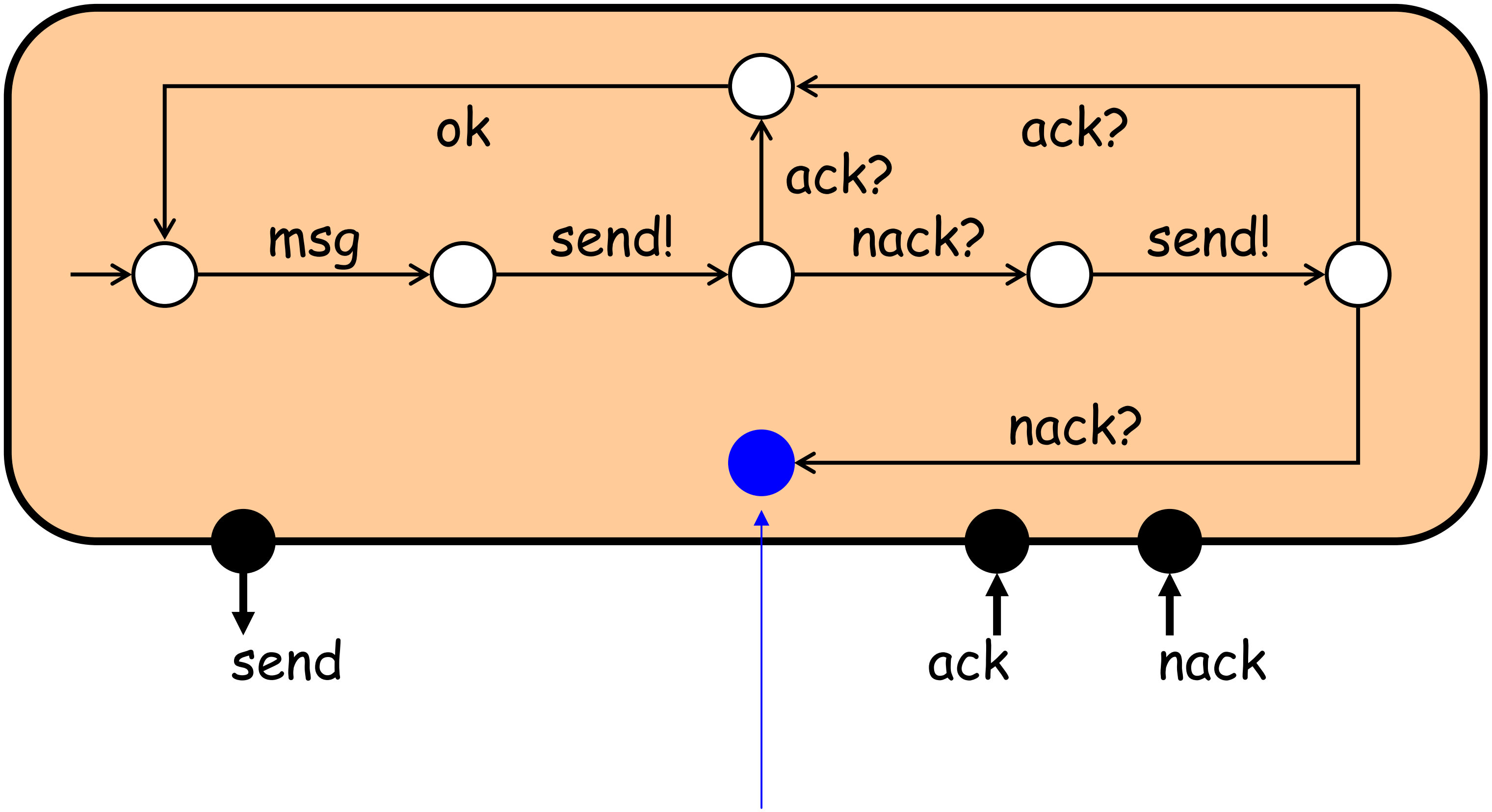

msg
ok
fail
ok!
ack?
ack?
msg?
send!
nack?
send!
fail!
nack?
send
ack
nack



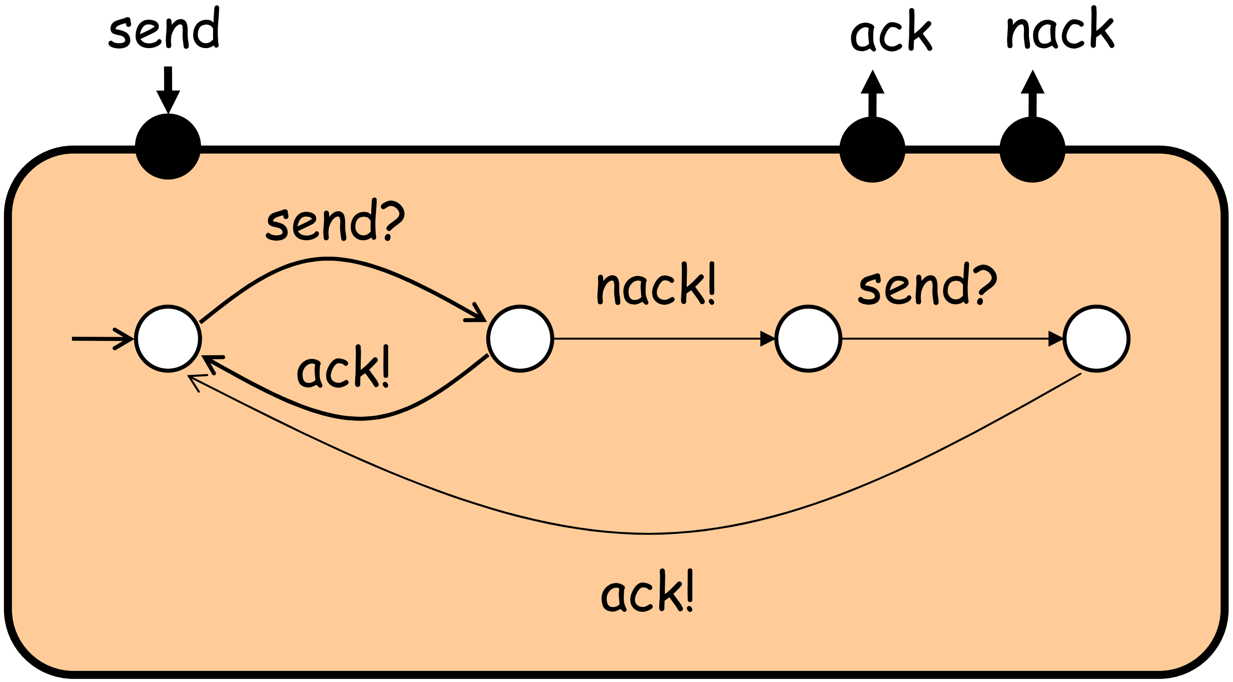
The product automaton.



ERROR state of the product.



Environment can avoid this state.



The most general helpful environment.

Computing the Composite Interface Automaton

1. Construct product automaton.

This procedure computes the most general helpful environment as the most general strategy of the environment to avoid error states.

Computing the Composite Interface Automaton

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

This procedure computes the most general helpful environment as the most general strategy of the environment to avoid error states.

Computing the Composite Interface Automaton

1. Construct product automaton.
2. Mark ERROR 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.

This procedure computes the most general helpful environment as the most general strategy of the environment to avoid error states.

Computing the Composite Interface Automaton

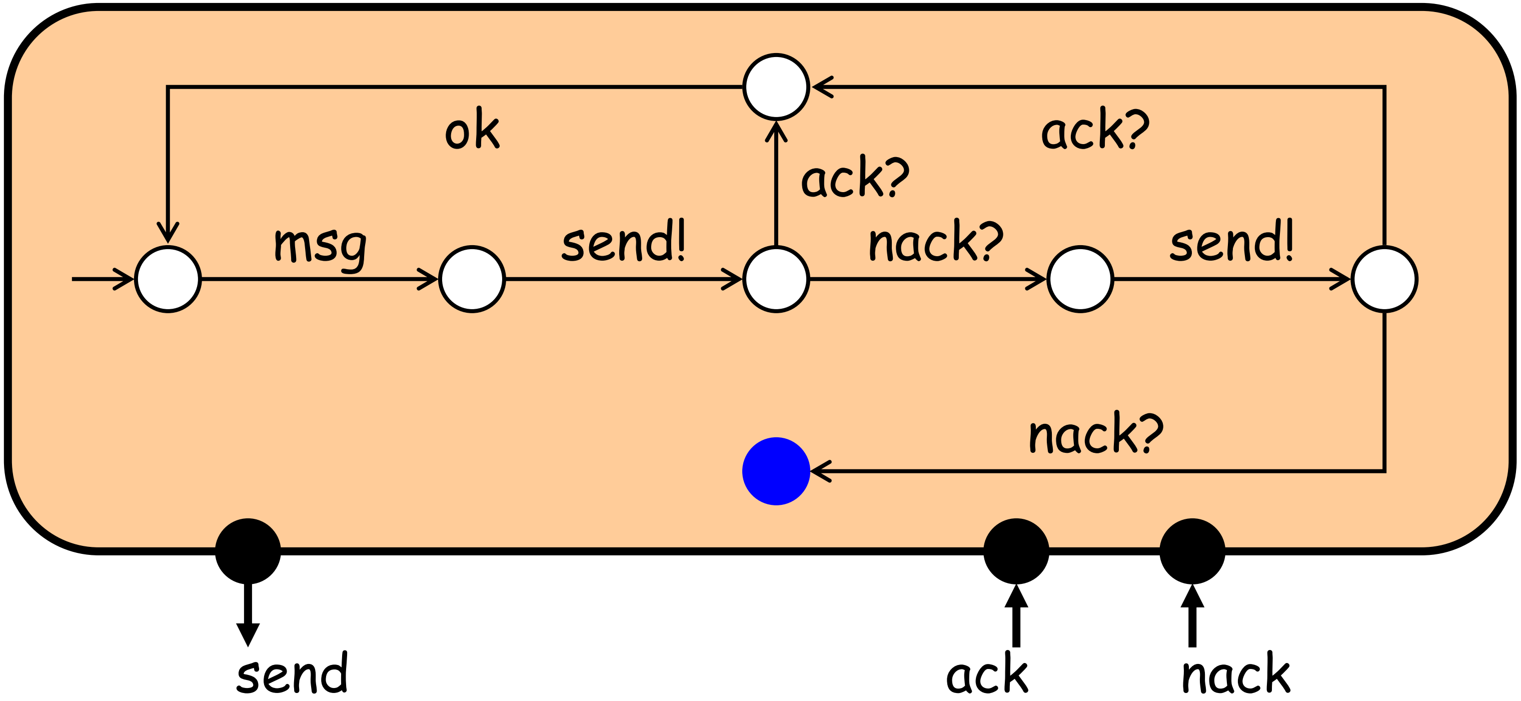
1. Construct product automaton.
2. Mark ERROR states as incompatible.
3. Until no more incompatible states can be added: mark state q as incompatible if there is an internal or output action from q to an incompatible state.

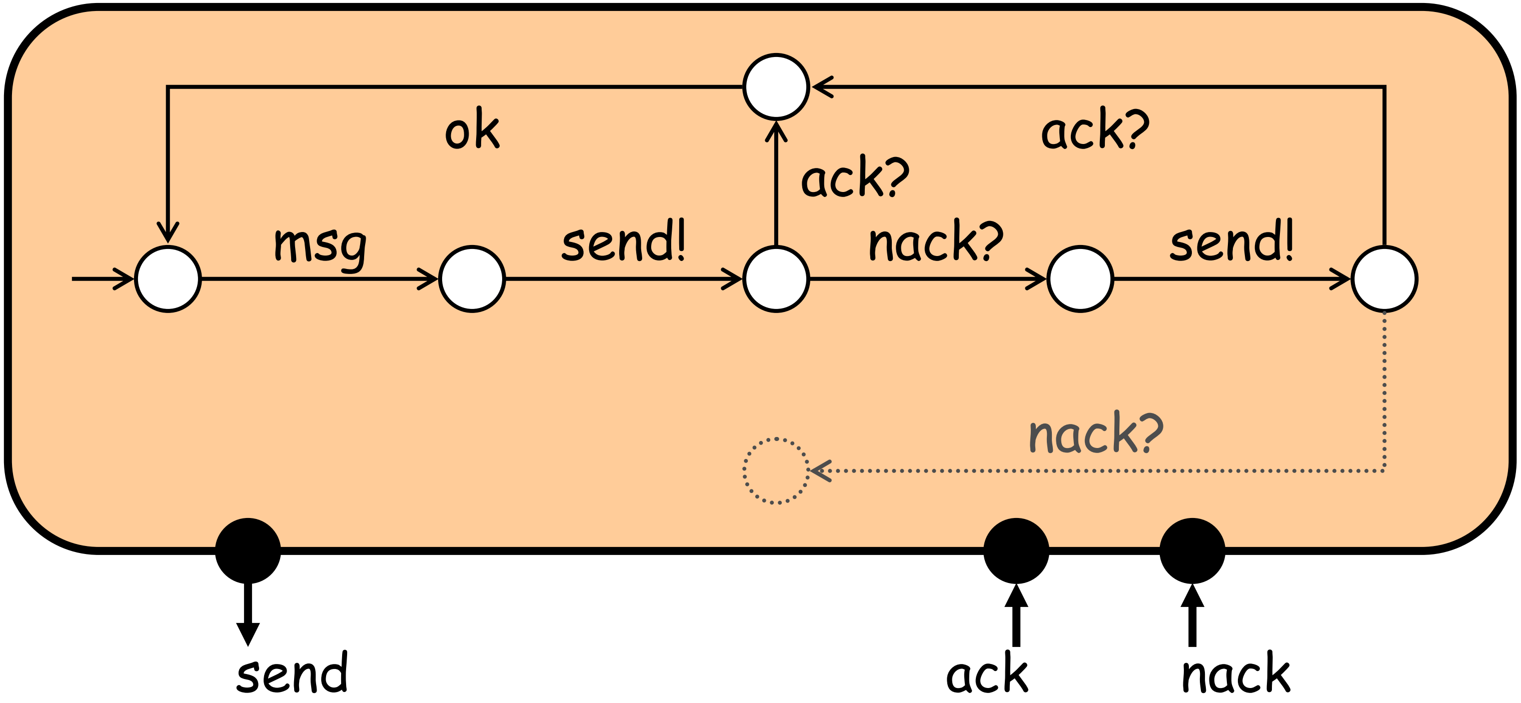
This procedure computes the most general helpful environment as the most general strategy of the environment to avoid error states.

Computing the Composite Interface Automaton

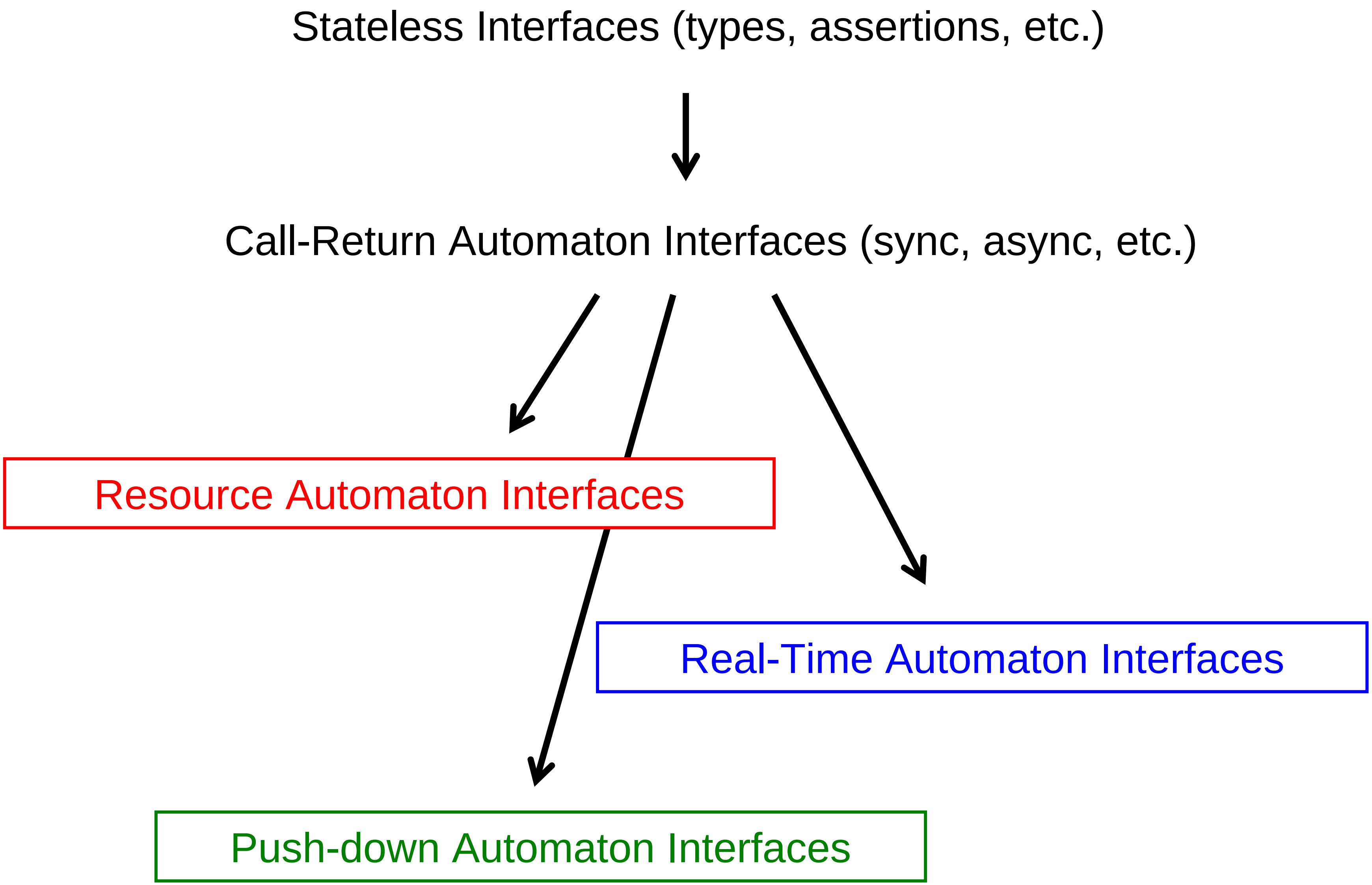
1. Construct product automaton.
2. Mark ERROR states as incompatible.
3. Until no more incompatible states can be added: mark state q as incompatible if there is an internal or output action from q to an incompatible state.
4. If the initial state is incompatible, then the two interfaces are incompatible. Otherwise, the composite interface is the product automaton without the incompatible states.

This procedure computes the most general helpful environment as the most general strategy of the environment to avoid error states.


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



The composite interface automaton.



Stateless Interfaces (types, assertions, etc.)

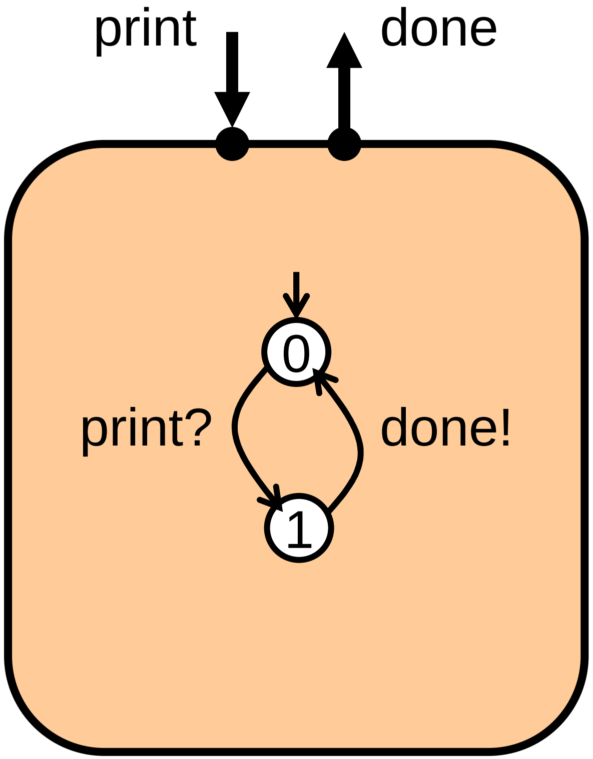
Call-Return Automaton Interfaces (sync, async, etc.)

Resource Automaton Interfaces

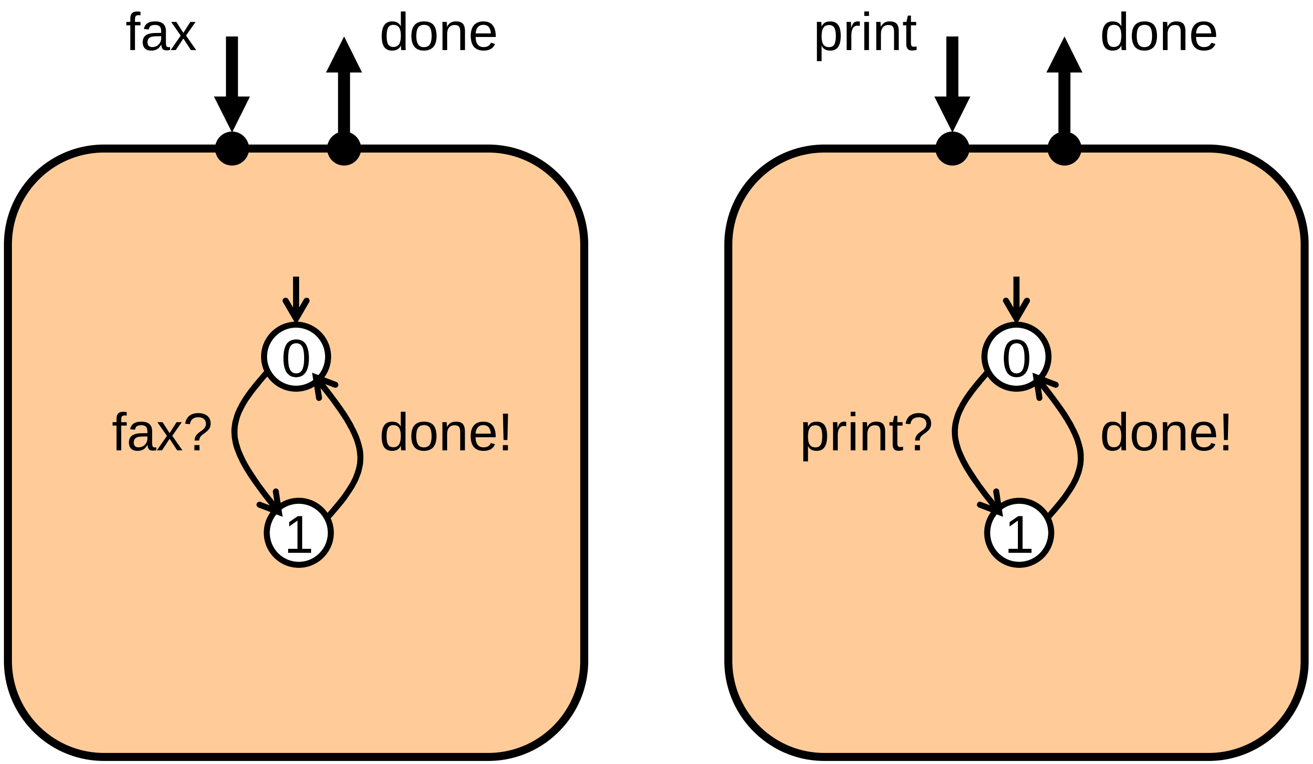
Real-Time Automaton Interfaces

Push-down Automaton Interfaces

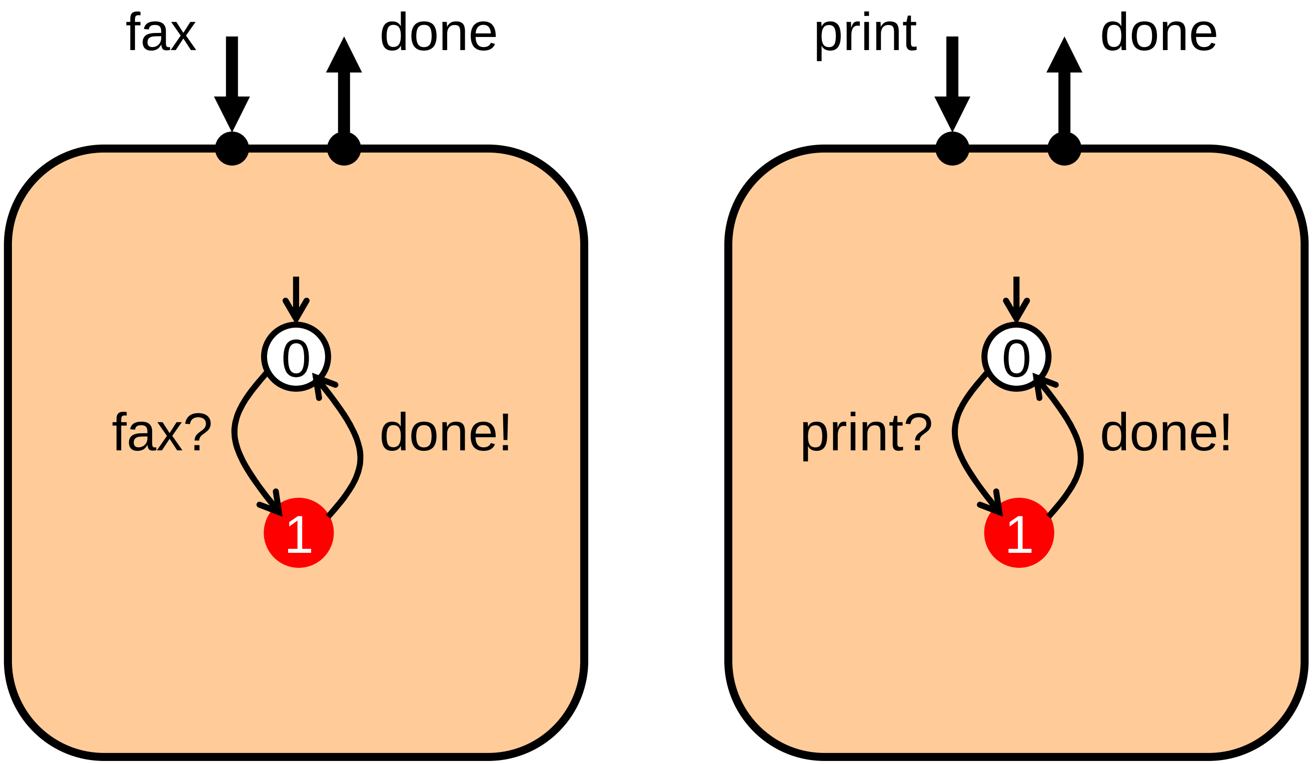
A Mutex Interface



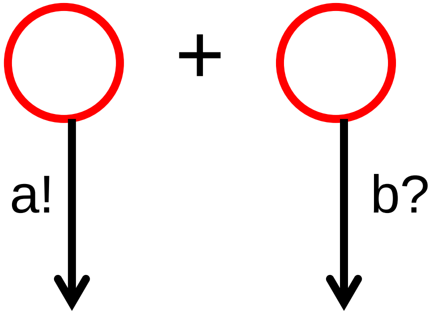
Two Mutex Interfaces



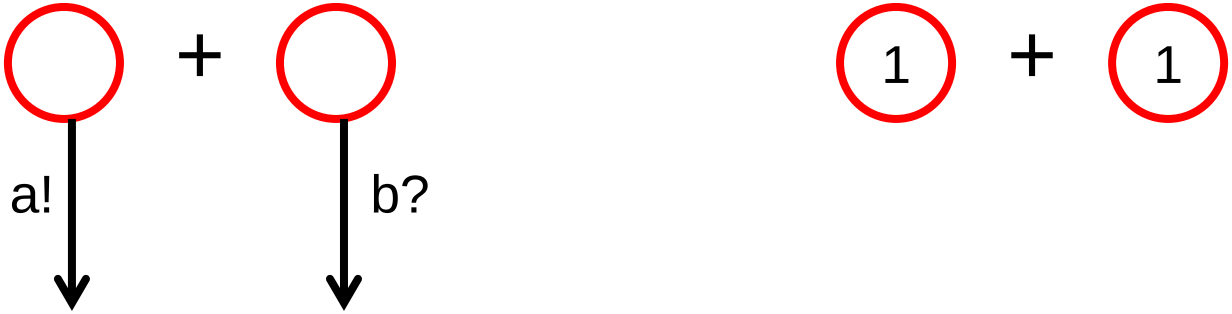
Mutex Interface Incompatibility



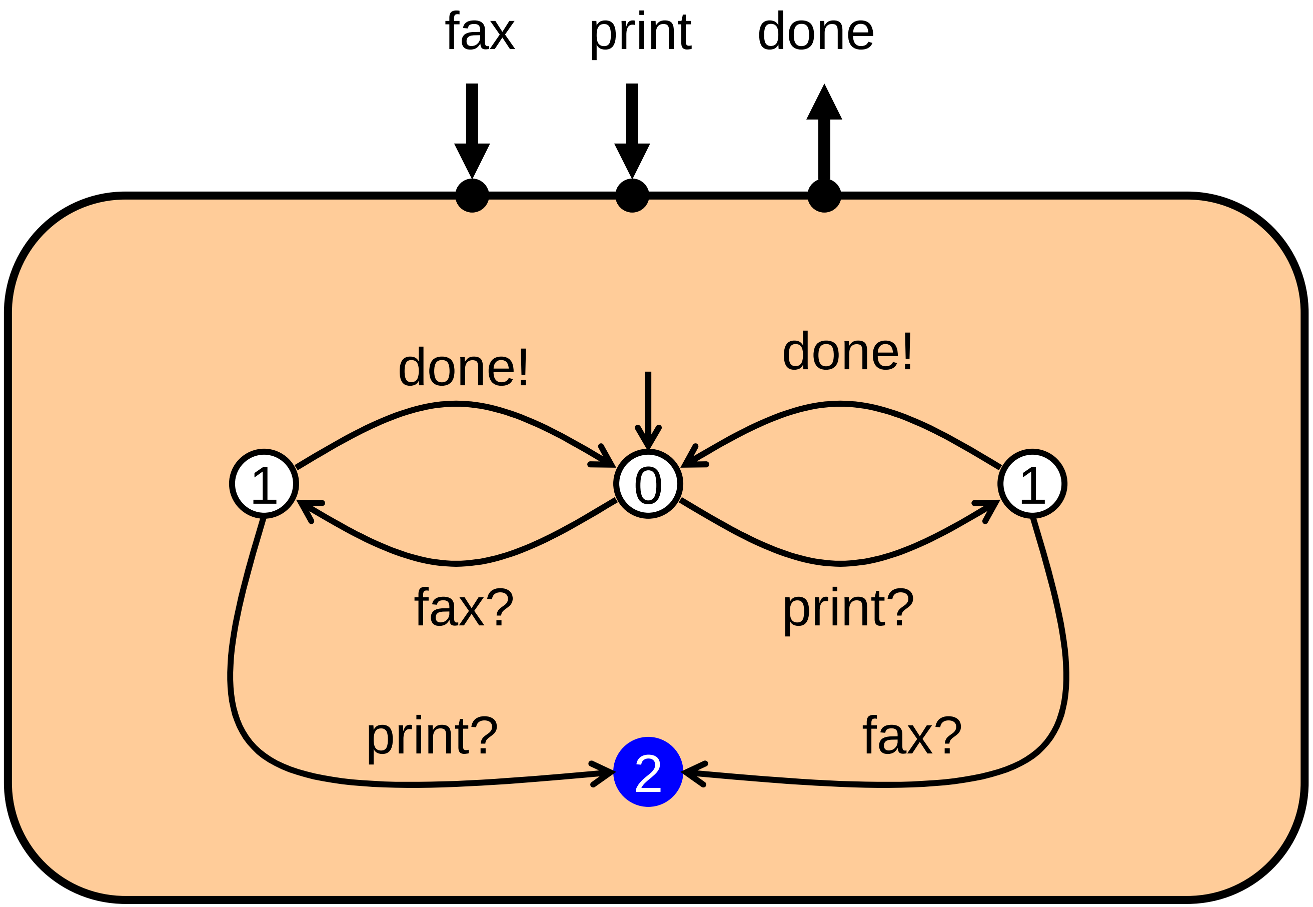
Call-Return Interface Incompatibility



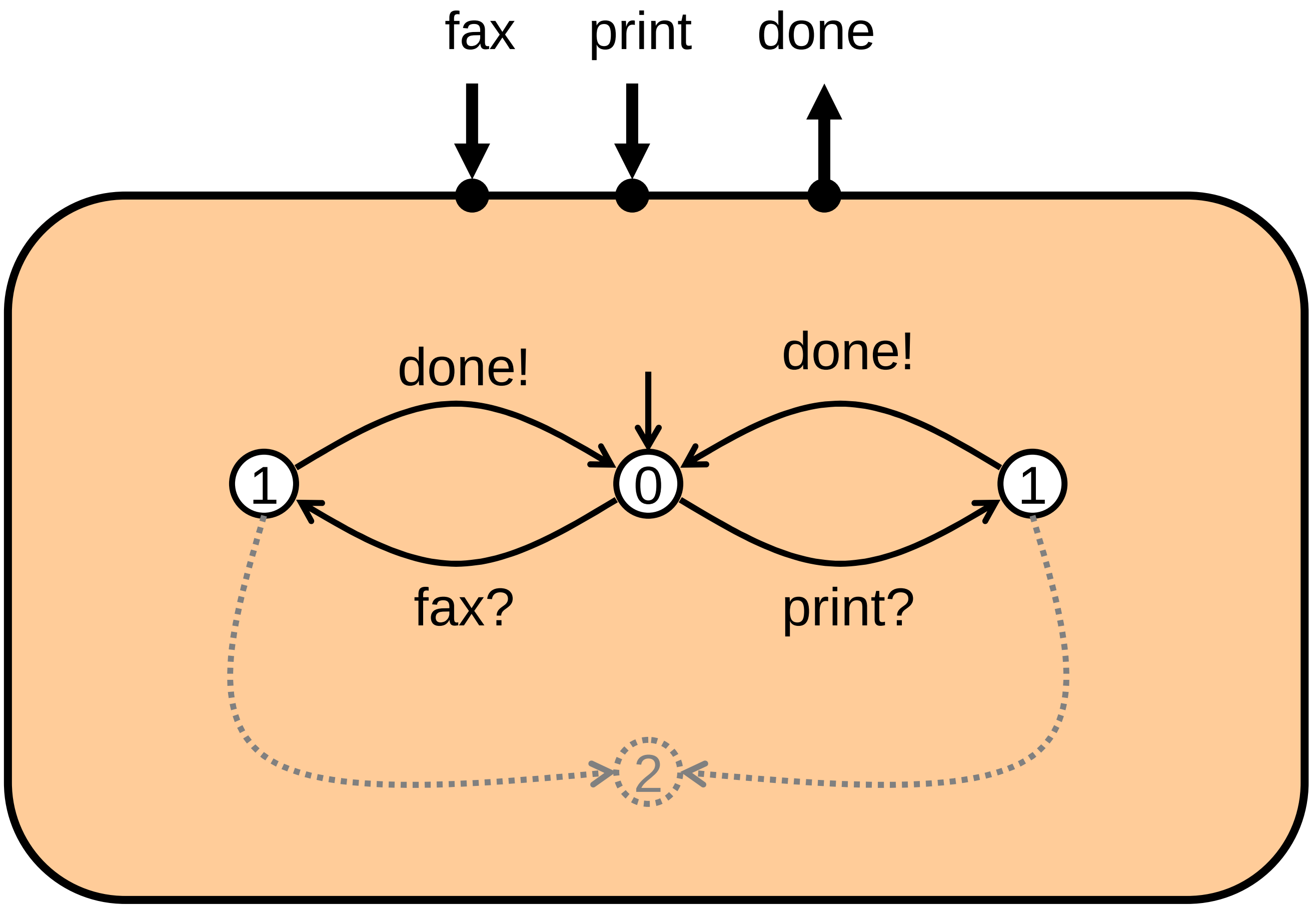
Mutex Interface Incompatibility



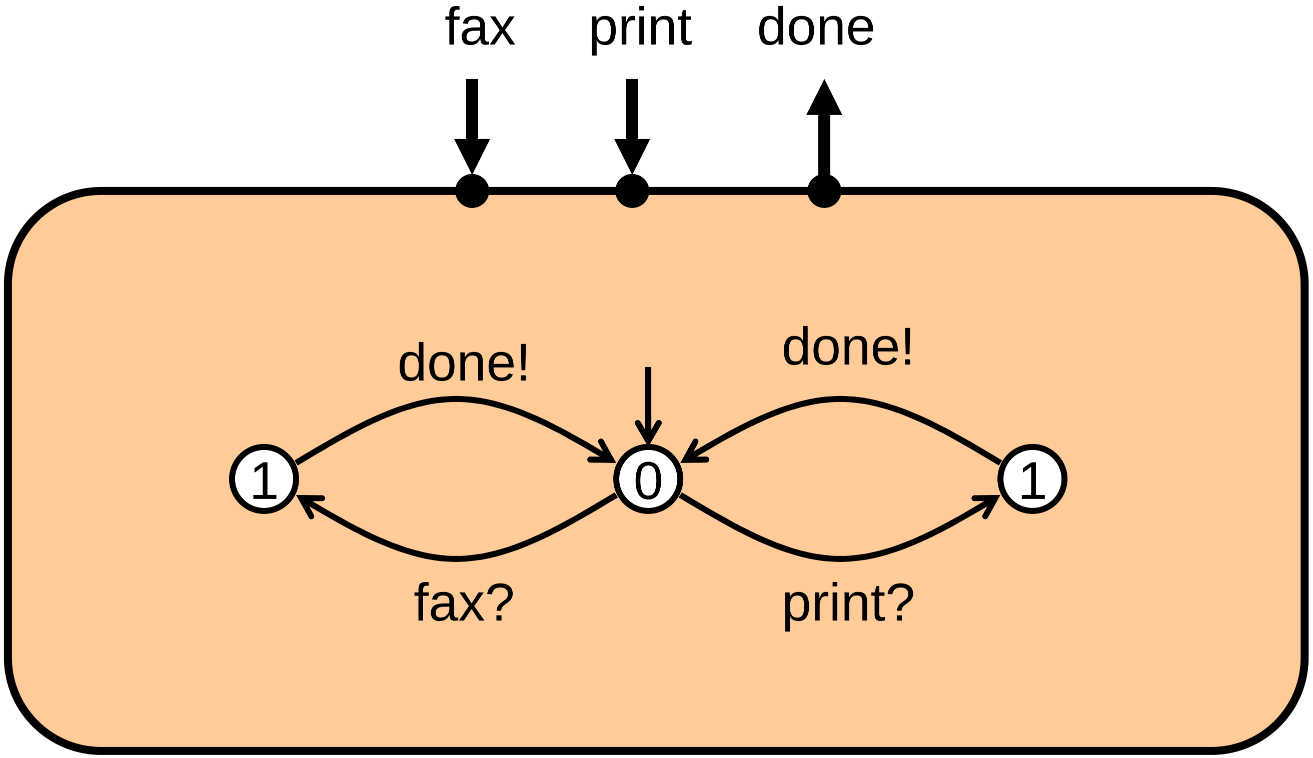
Mutex Interface Product



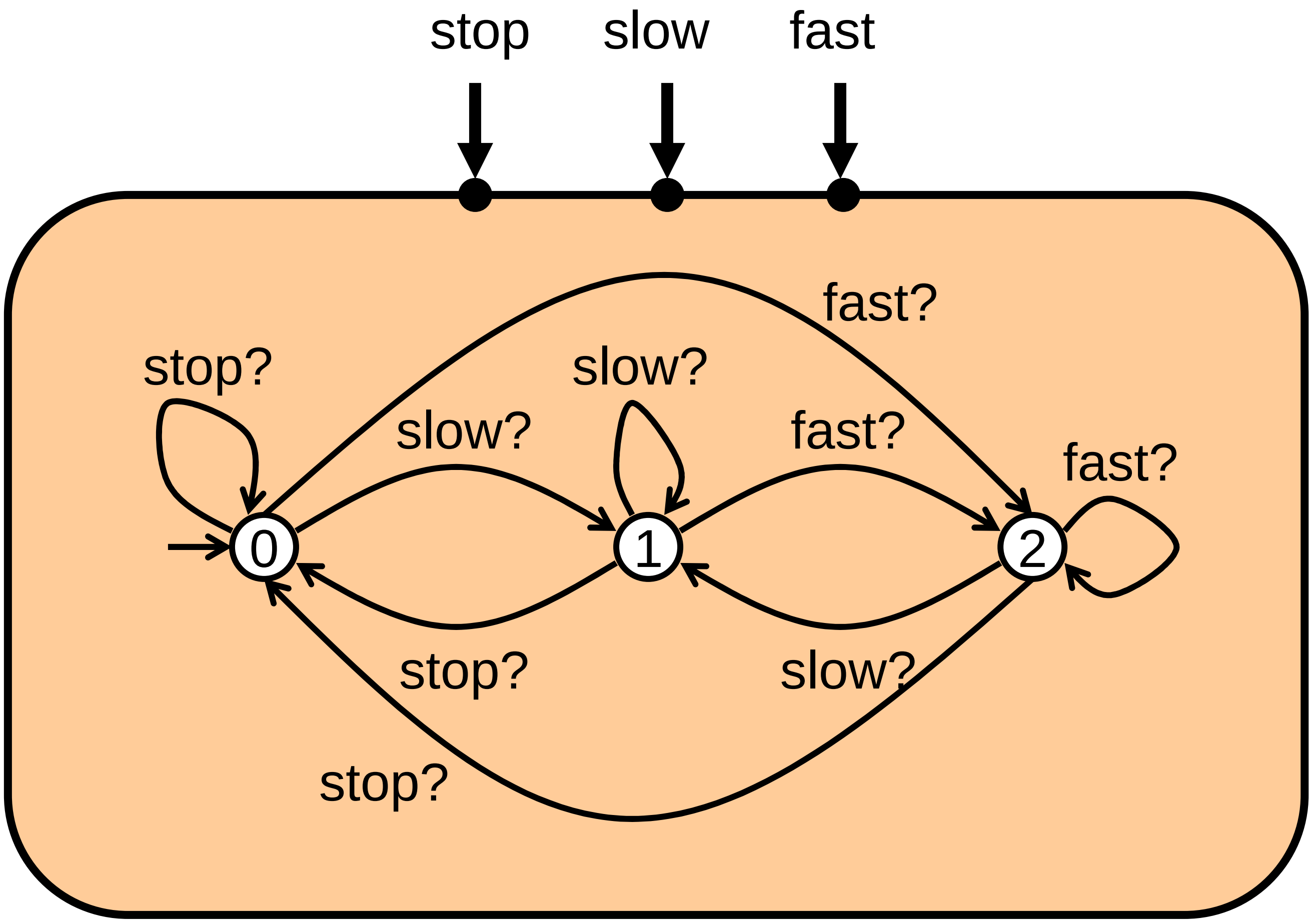
Mutex Interface Composition



The Composite Interface



A Power Resource Interface



Motor driver.

Resource Interface Composition

Node Limit Resource Interfaces
(e.g. mutex, limited buffer size, limited peak power):

Player Input must achieve objective without visiting states that exceed threshold.

Path Limit Resource Interfaces
(e.g. limited battery capacity):

Player Input must achieve objective without expending more energy (power times time) than available.

These games can be solved in polynomial time.

Resource Interface Design

Strategy Synthesis
(e.g. resource scheduler, sensornet routing algorithm):

Given a resource bound, how can the objective be achieved?

Resource Synthesis
(e.g. necessary buffer size, battery capacity):

What is the minimum resource requirement so that the objective can be achieved?

Game algorithms can be generalized to answer both.

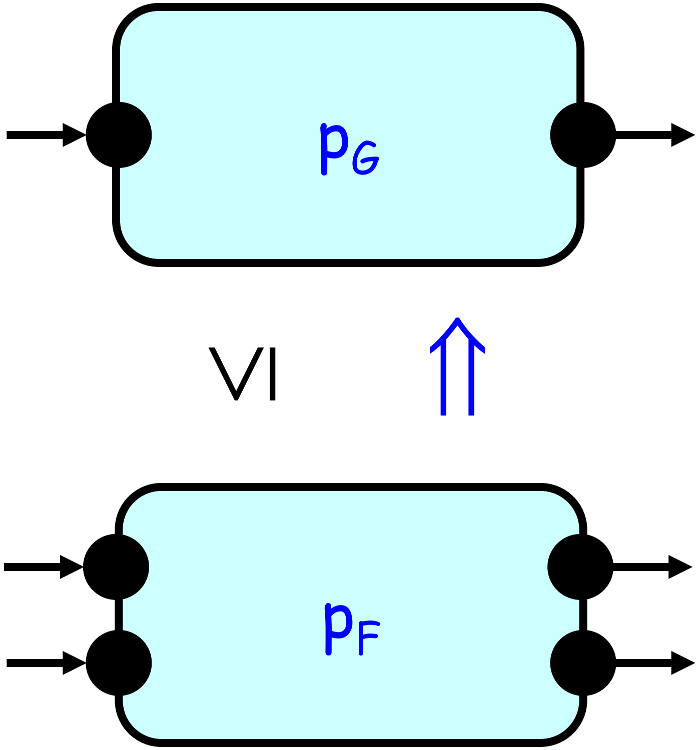
Interface Algebra

If G and G' compatible and G ¸ F and G' ¸ F',
then F and F' compatible and $G\|G'$ ¸ $F\|F'$.

Principle of independent implementability of interfaces.

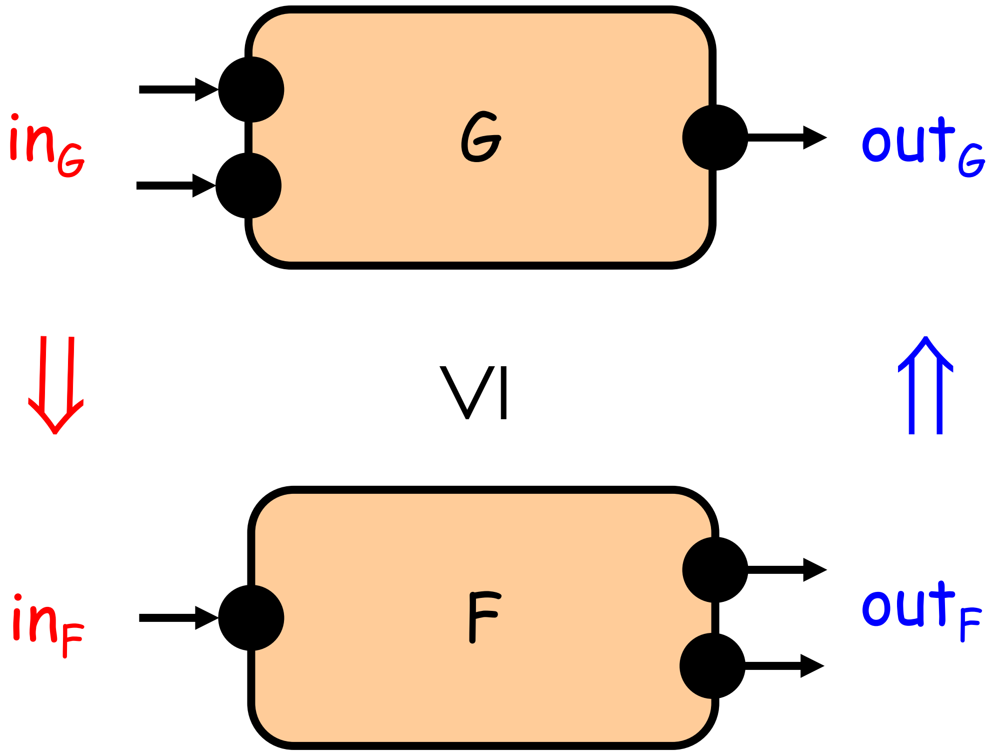
Stateless Process Refinement

More behaviors.



More information.

Stateless Interface Refinement



Weaker input assumption.

Stronger output guarantee.

I/O Automaton Refinement: Simulation

^{1} is a *simulation* relation if

f ^{1} g

iff

for all observable (input and output) actions a, if f –a-> f' , then there exists g' such that g –a-> g' and f' ^{1} g' .

I/O Automaton Refinement: Simulation

1 is a *simulation* relation if

f 1 g

iff

for all observable (input and output) actions a, if f -a-> f' , then there exists g' such that g -a-> g' and f' 1 g' .

If there are internal actions, then replace -a-> by -h*;a-> , where h* is any sequence of internal actions.

I/O Automaton Refinement: Simulation

¹ is a *simulation* relation if

f ¹ g

iff

for all observable (input and output) actions a, if f –a-> f′ , then there exists g′ such that g –a-> g′ and f′ ¹ g′ .

If there are internal actions, then replace -a-> by –h*a-> , where h* is any sequence of internal actions.

F · *G* if there exists a simulation relation ¹ such that q^0_F ¹ q^0_G.

Interface Automaton Refinement: Alternating Simulation

1 is an *alternating simulation* relation if

f 1 g

iff

for all output actions o, if f -o!-> f' , then there exists g' such that g -o!-> g' and f' 1 g' .

Interface Automaton Refinement: Alternating Simulation

1 is an *alternating simulation* relation if

f 1 g

iff

1. for all input actions i, if g –i?-> g' , then there exists f' such that f –i?-> f' and f' 1 g' ,

and

2. for all output actions o, if f –o!-> f' , then there exists g' such that g –o!-> g' and f' 1 g' .

Interface Automaton Refinement: Alternating Simulation

^{1} is an *alternating simulation* relation if

f ^{1} g

iff

1. for all input actions i, if g –i?-> g′ , then there exists f′ such that f –i?-> f′ and f′ ^{1} g′ ,

and

2. for all output actions o, if f –o!-> f′ , then there exists g′ such that g –o!-> g′ and f′ ^{1} g′ .

Interface Automaton Refinement:
Alternating Simulation

1 is an *alternating simulation* relation if

f 1 g

iff

1. for all input actions i, if g –i?-> g′ , then there exists f′ such that f –i?-> f′ and f′ 1 g′ ,

and

2. for all output actions o, if f -o!-> f′ , then there exists g′ such that g -o!-> g′ and f′ 1 g′ .

If there are internal actions, then replace -o?-> by –h*;o?->.

Interface Automaton Refinement: Alternating Simulation

^{1} is an *alternating simulation* relation if

$f \ ^{1}\ g$

iff

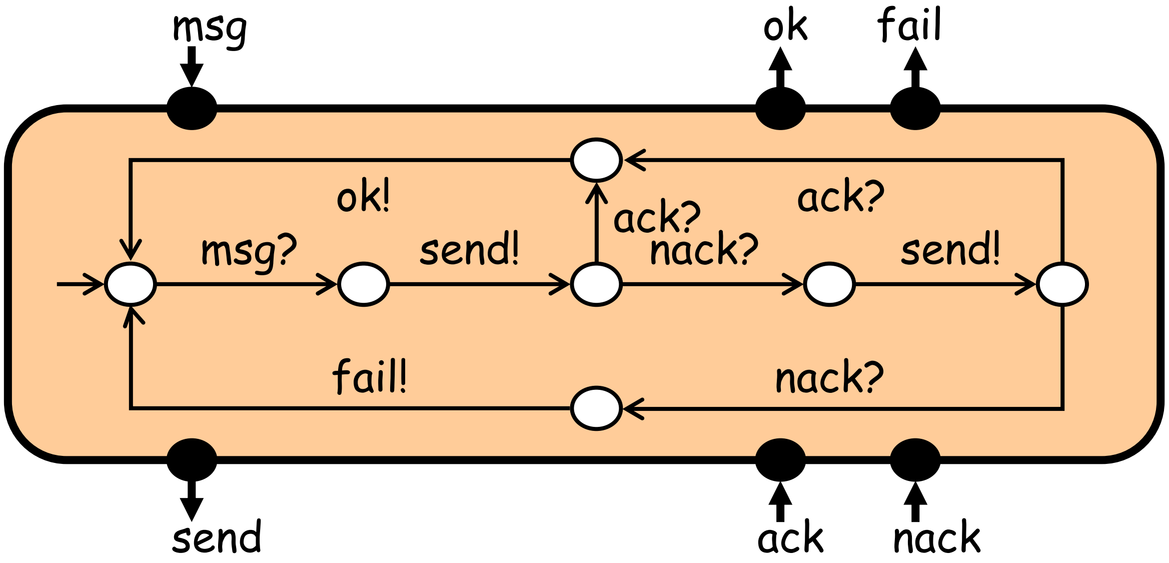
1. for all input actions i, if g –i?-> g′ , then there exists f′ such that f –i?-> f′ and $f' \ ^{1}\ g'$,

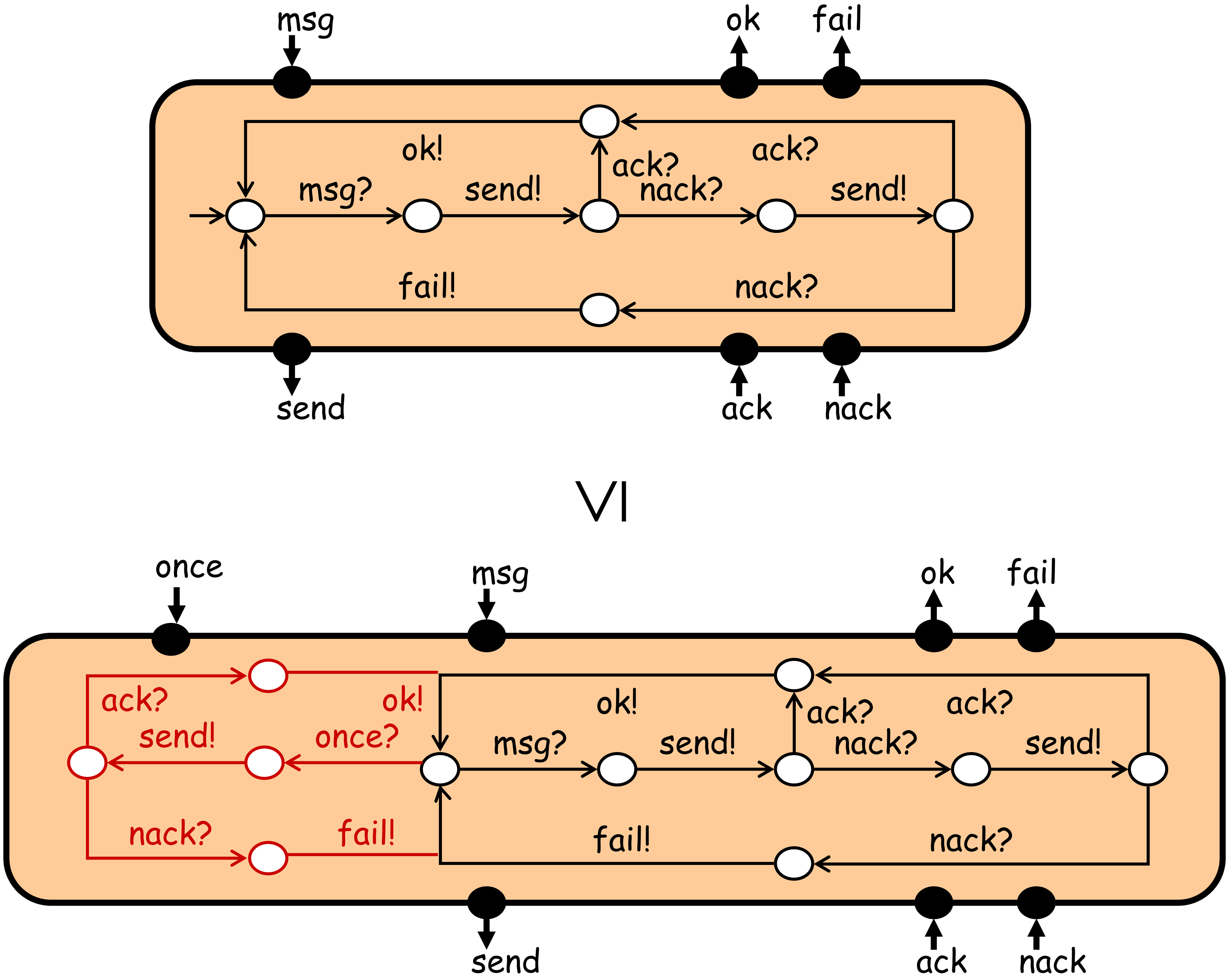
and

2. for all output actions o, if f –o!-> f′ , then there exists g′ such that g –o!-> g′ and $f' \ ^{1}\ g'$.

If there are internal actions, then replace -o?-> by –h*;o?->.

$F \cdot G$ if there is an alternating simulation ^{1} such that $q^{0}_{F} \ ^{1}\ q^{0}_{G}$.


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


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

Alternating Simulation

^{1} is an *alternating simulation* relation if

f ^{1} g

iff

1. for all input actions i, if g –i?-> g′ , then there exists f′ such that f –i?-> f′ and f′ ^{1} g′ ,

and

2. for all output actions o, if f –o!-> f′ , then there exists g′ such that g –o!-> g′ and f′ ^{1} g′ .

If there is a winning environment strategy at g, then there is a winning environment strategy at f [Alur,Kupferman,H,Vardi].

Alternating Simulation

As in the case of simulation, the greatest alternating simulation relation can be computed by successive approximation:

$^{1}{}_{0} = Q_F$ £ Q_G

f $^{1}{}_{k+1}$ g if

0. f $^{1}{}_{k}$ g ,

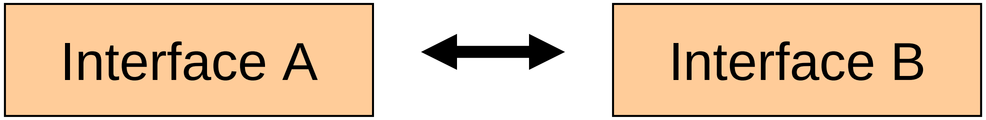
1. for all input actions i, if g –i?-> g′ , then there exists f′ such that f –i?-> f′ and f′ $^{1}{}_{k}$ g′ ,

2. for all output actions o, if f –o!-> f′ , then there exists g′ such that g –o!-> g′ and f′ $^{1}{}_{k}$ g′ .

This can be implemented in time quadratic in |F|+|G| .

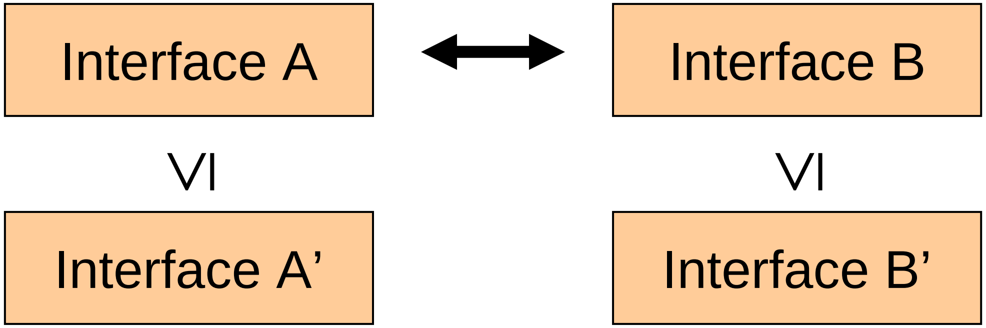
Lesson 4:

Proofs are good.
Algorithms are better.



Interface A

Interface B



Interface A

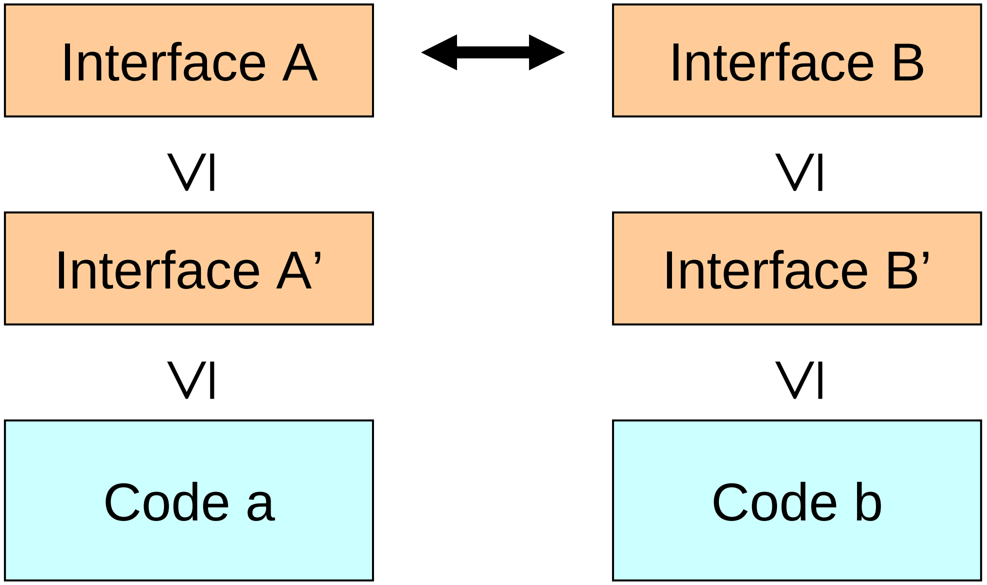
Interface B

VI

VI

Interface A’

Interface B’



Interface A

Interface B

VI

VI

Interface A’

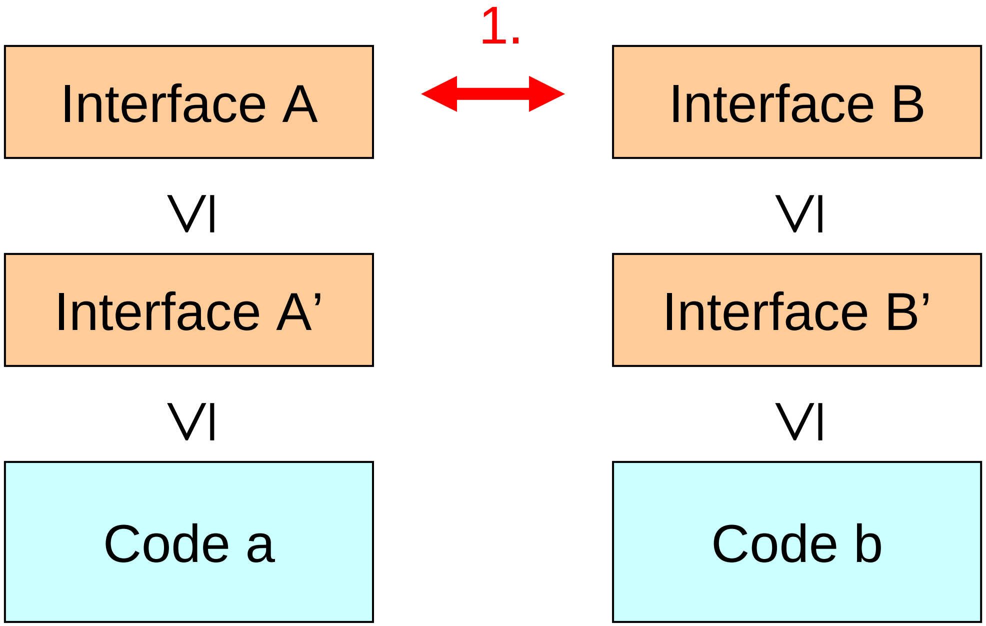
Interface B’

VI

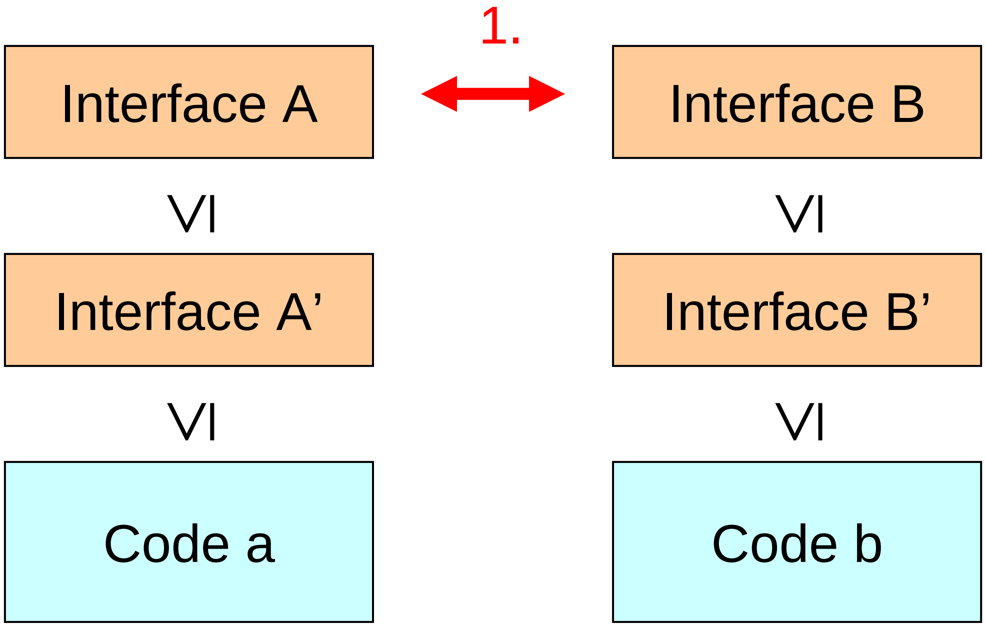
VI

Code a

Code b

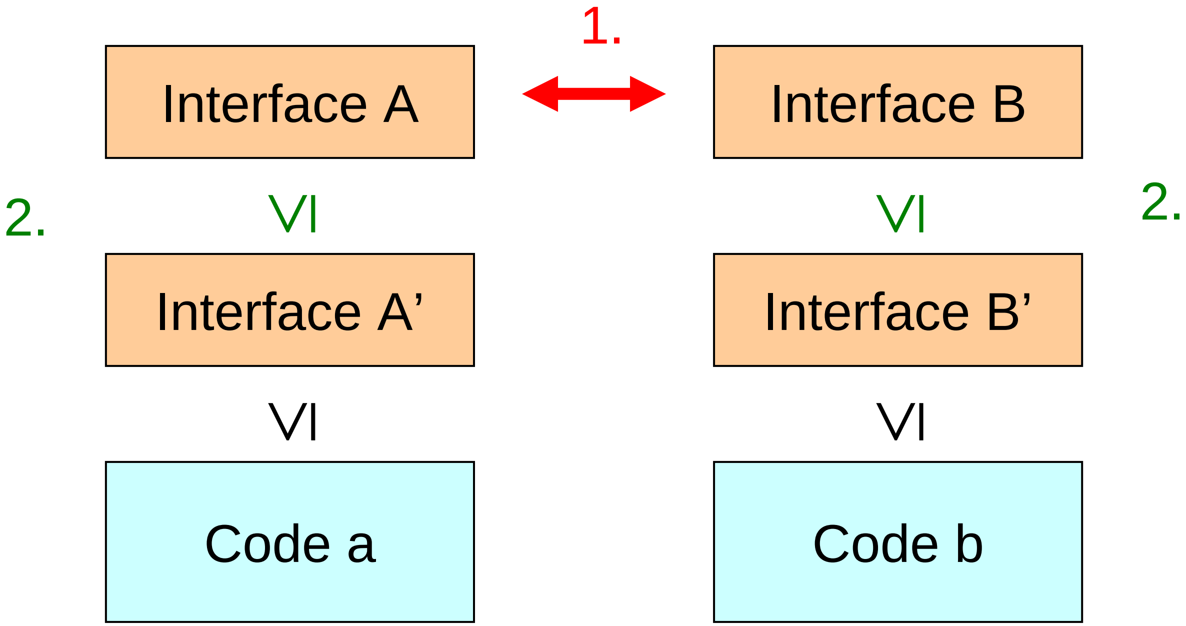


1. Interface compatibility checking:
   solving safety/Buechi games (linear/quadratic)



1. Interface compatibility checking:
   solving safety/Buechi games (linear/quadratic)

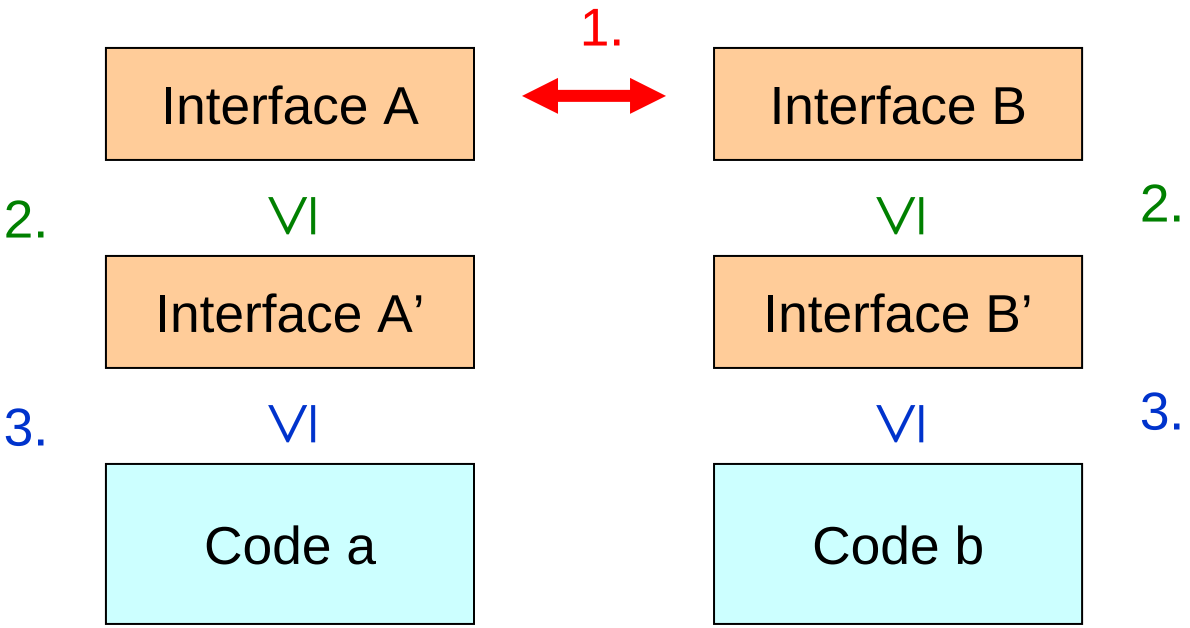
CHIC
[Chakrabarti]



1. Interface compatibility checking:
   solving safety/Buechi games (linear/quadratic)

CHIC
[Chakrabarti]

2. Interface refinement checking:
   alternating simulation (quadratic)

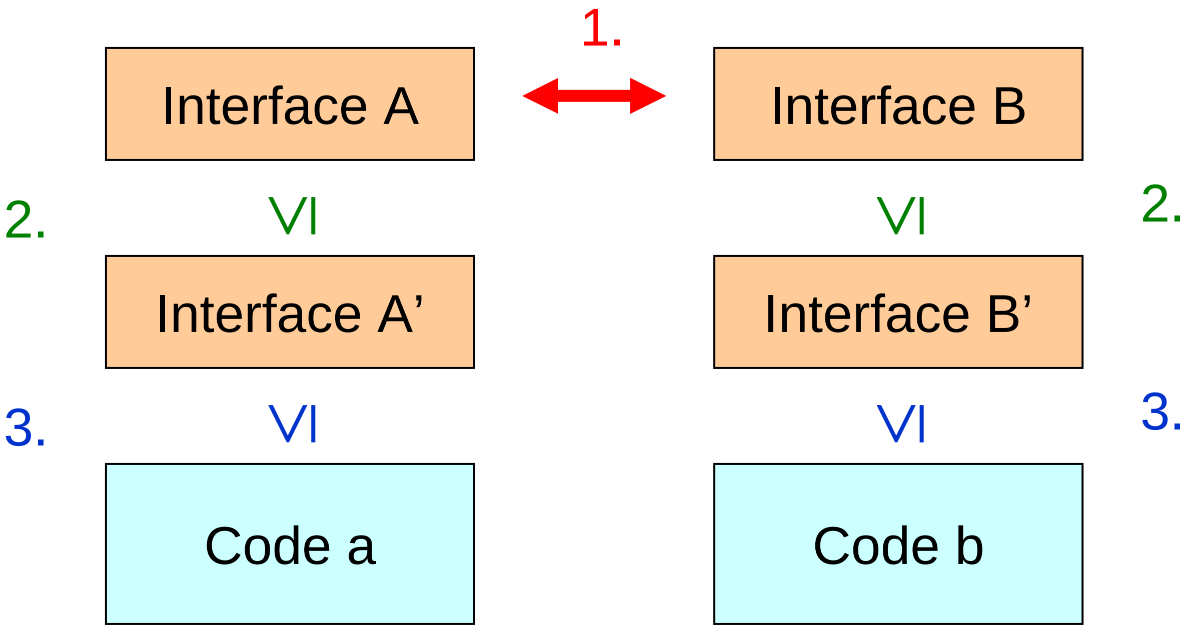


1. Interface compatibility checking:
   solving safety/Buechi games (linear/quadratic)

CHIC
[Chakrabarti]

2. Interface refinement checking:
   alternating simulation (quadratic)

3. Conformance checking of code against interface:
   undecidable



1.

Interface A

Interface B

2.

VI

VI

2.

Interface A’

Interface B’

3.

VI

VI

3.

Code a

Code b

1. Interface compatibility checking:
   solving safety/Buechi games (linear/quadratic)

CHIC
[Chakrabarti]

2. Interface refinement checking:
   alternating simulation (quadratic)

3. Conformance checking of code against interface:
   undecidable

BLAST [Jhala,Majumdar,Sutre]

CHIC

CHecking
Interface
Compatibility

BLAST

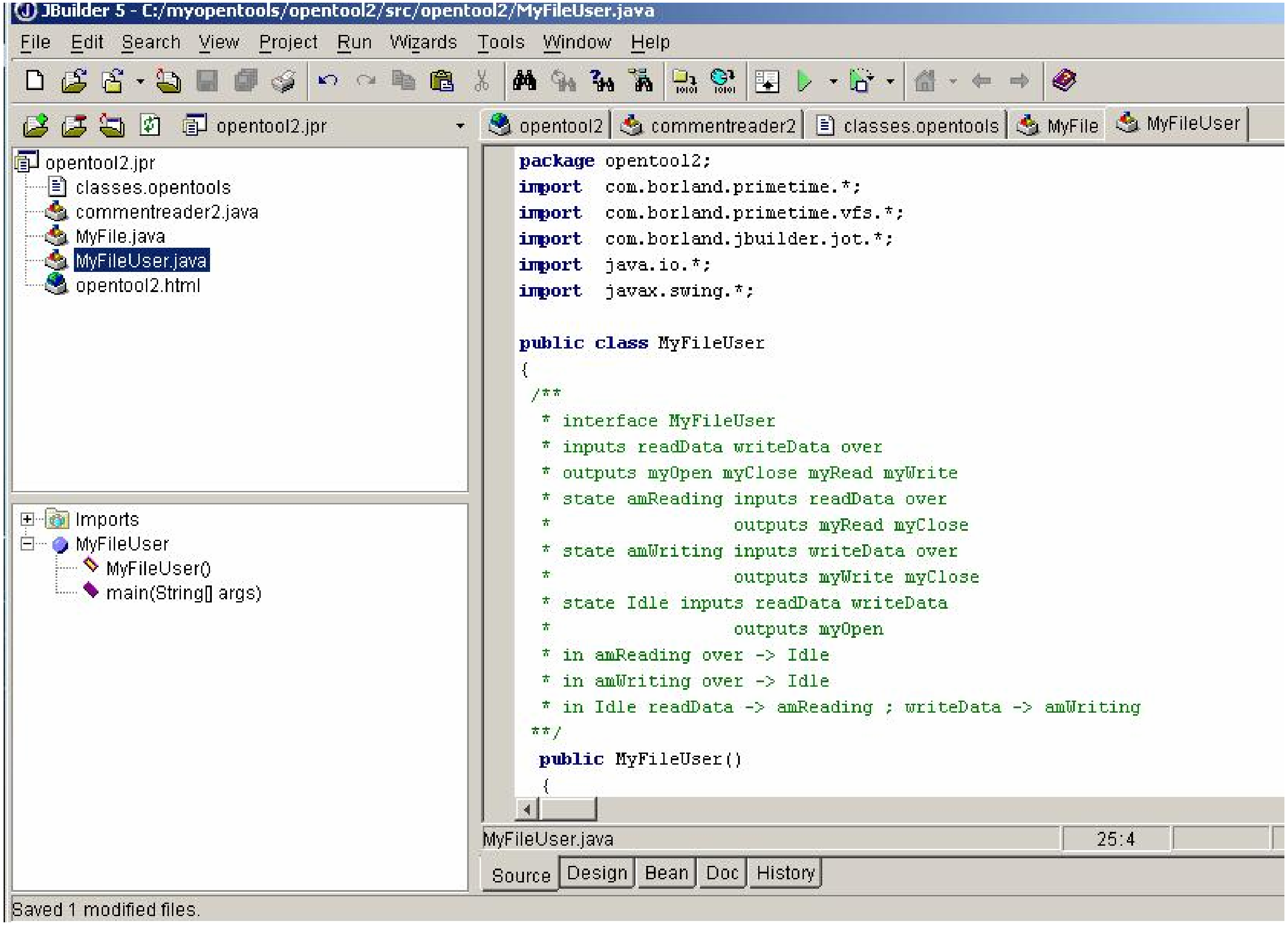
Berkeley
Lazy Abstraction
Software verification
Tool

www.eecs.berkeley.edu/~tah/chic
www.eecs.berkeley.edu/~tah/blast

Try them out!

CHIC

JBuilder plugin for defining and checking automaton interfaces of Java classes.



```
package opentool2;
import  com.borland.primetime.*;
import  com.borland.primetime.vfs.*;
import  com.borland.jbuilder.jot.*;
import  java.io.*;
import  javax.swing.*;

public class MyFileUser
{
 /**
  * interface MyFileUser
  * inputs readData writeData over
  * outputs myOpen myClose myRead myWrite
  * state amReading inputs readData over
  *                 outputs myRead myClose
  * state amWriting inputs writeData over
  *                 outputs myWrite myClose
  * state Idle inputs readData writeData
  *                 outputs myOpen
  * in amReading over -> Idle
  * in amWriting over -> Idle
  * in Idle readData -> amReading ; writeData -> amWriting
 **/
  public MyFileUser()
  {
```

BLAST

-model checker for (multi-threaded) C programs

BLAST

-model checker for (multi-threaded) C programs

-can handle programs with 100K+ lines of code

-supports incremental (“extreme”) model checking

BLAST

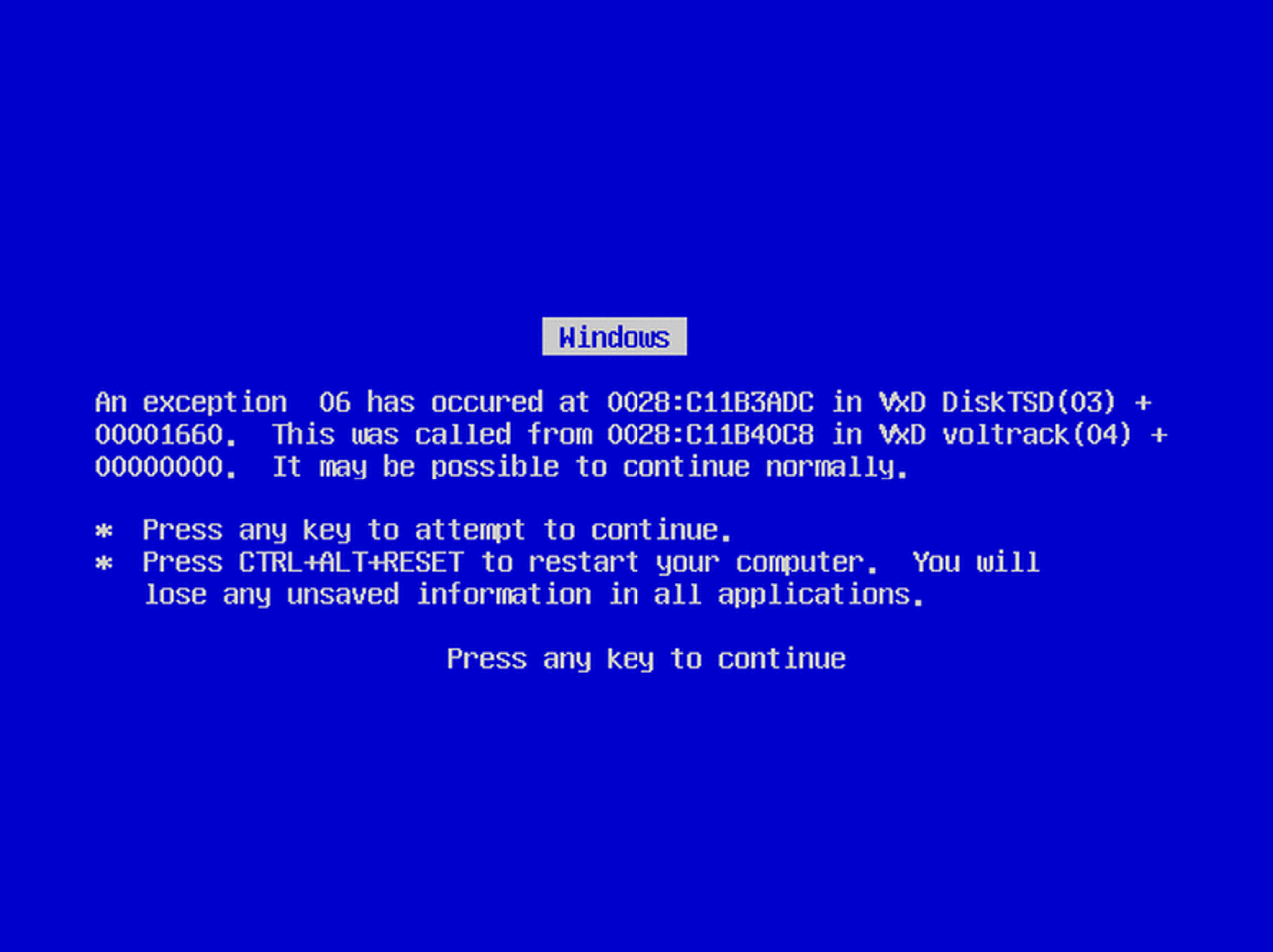
-model checker for (multi-threaded) C programs

-can handle programs with 100K+ lines of code

-supports incremental (“extreme”) model checking

-counterexample-guided predicate abstraction refinement

-inspired by SLAM [Ball,Rajamani] (see next week)



```
                              Windows

An exception  06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660.  This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000.  It may be possible to continue normally.

*  Press any key to attempt to continue.
*  Press CTRL+ALT+RESET to restart your computer.  You will
   lose any unsaved information in all applications.

                    Press any key to continue
```

BLAST

-model checker for (multi-threaded) C programs

-can handle programs with 100K+ lines of code

-supports incremental (“extreme”) model checking

-counterexample-guided predicate abstraction refinement

-inspired by SLAM [Ball,Rajamani] (see next week)

-for interface conformance checking and interface synthesis