Specifying and Checking Stateful Software Interfaces (Lecture 3)

Manuel Fähndrich maf@microsoft.com
Microsoft Research

2005 Summer School on Reliable Computing
Eugene, Oregon

Lecture 2 (recap)

- Frame axiom
- Type-states using capabilities
- Vault: W2K driver case study
- Recursive data structures
- Unifying non-linear data structures and linear data

Lecture 3

- Fractional permissions
- Fugue: Type-states for objects
- Sing# and Singularity

- No more type rules!

Read-only sharing

- Idea by John Boyland: fractional permissions

  $\{ \rho \,!\, h \} \,,\, \frac{1}{2}\{ \rho \,!\, h \} - \frac{1}{2}\{ \rho \,!\, h \}$
- In general

  $\{ \rho \,!\, h \} \,,\, k\{ \rho \,!\, h \} - (1-k)\{ \rho \,!\, h \}$
- Permission $k\{\rho \mapsto h\}$
  - Write if $k = 1$
  - Read-only otherwise
- Can express temporary sharing
- Useful for multiple threads

Fugue (MSR)

- C# + annotations only
- No change in the language
- Type states for objects
- Resource/alias management
- Non-null types
- Checker at MSIL level (standard C# compiler)
- Parts of the analysis are used in FxCop
  - will ship with VS2005

Fugue Demo

Typestates and class invariants

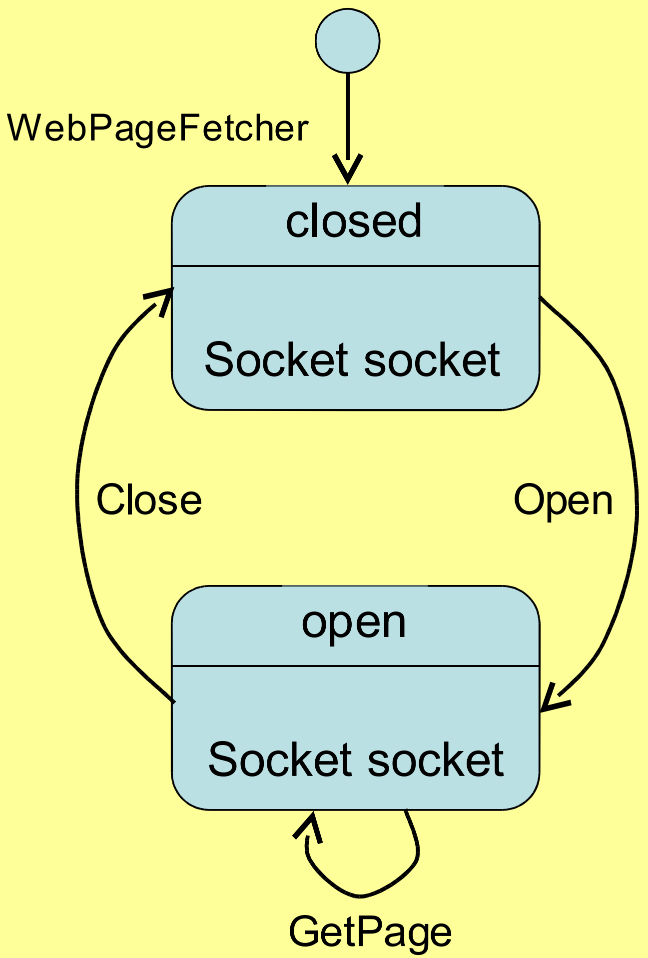
Relate symbolic typestate name with internal class properties

- Gives meaning to typestates

  What do ‘open’ and ‘closed’ mean?

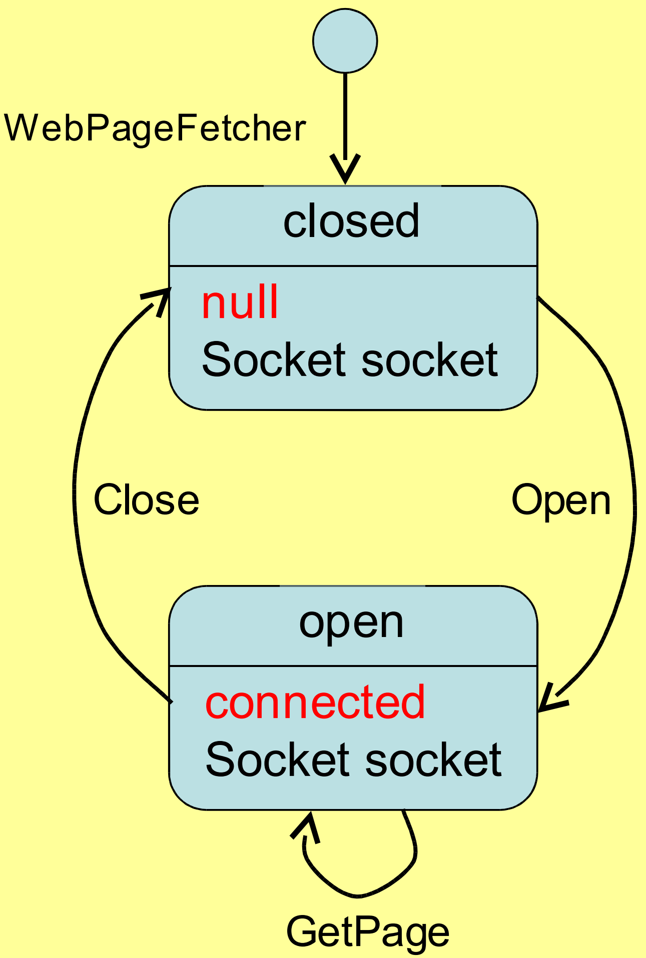
Typestates and class invariants

```
[ WithProtocol("open","closed") ]
class WebPageFetcher
{


  private Socket socket;

  [Creates("closed")]
  public WebPageFetcher ();

  [ChangesState("closed","open")]
  public void Open (string server);

  [InState("open")]
  public string GetPage (string url);

  [ChangesState("open","closed")]
  public void Close ();
}
```



Typestates and class invariants

```
[ WithProtocol("open","closed") ]
class WebPageFetcher
{
   [Null(WhenEnclosingState="closed")]
   [InState("connected",
            WhenEnclosingState="open")]
   private Socket socket;

   [Creates("closed")]
   public WebPageFetcher ();

   [ChangesState("closed","open")]
   public void Open (string server);

   [InState("open")]
   public string GetPage (string url);

   [ChangesState("open","closed")]
   public void Close ();
}
```



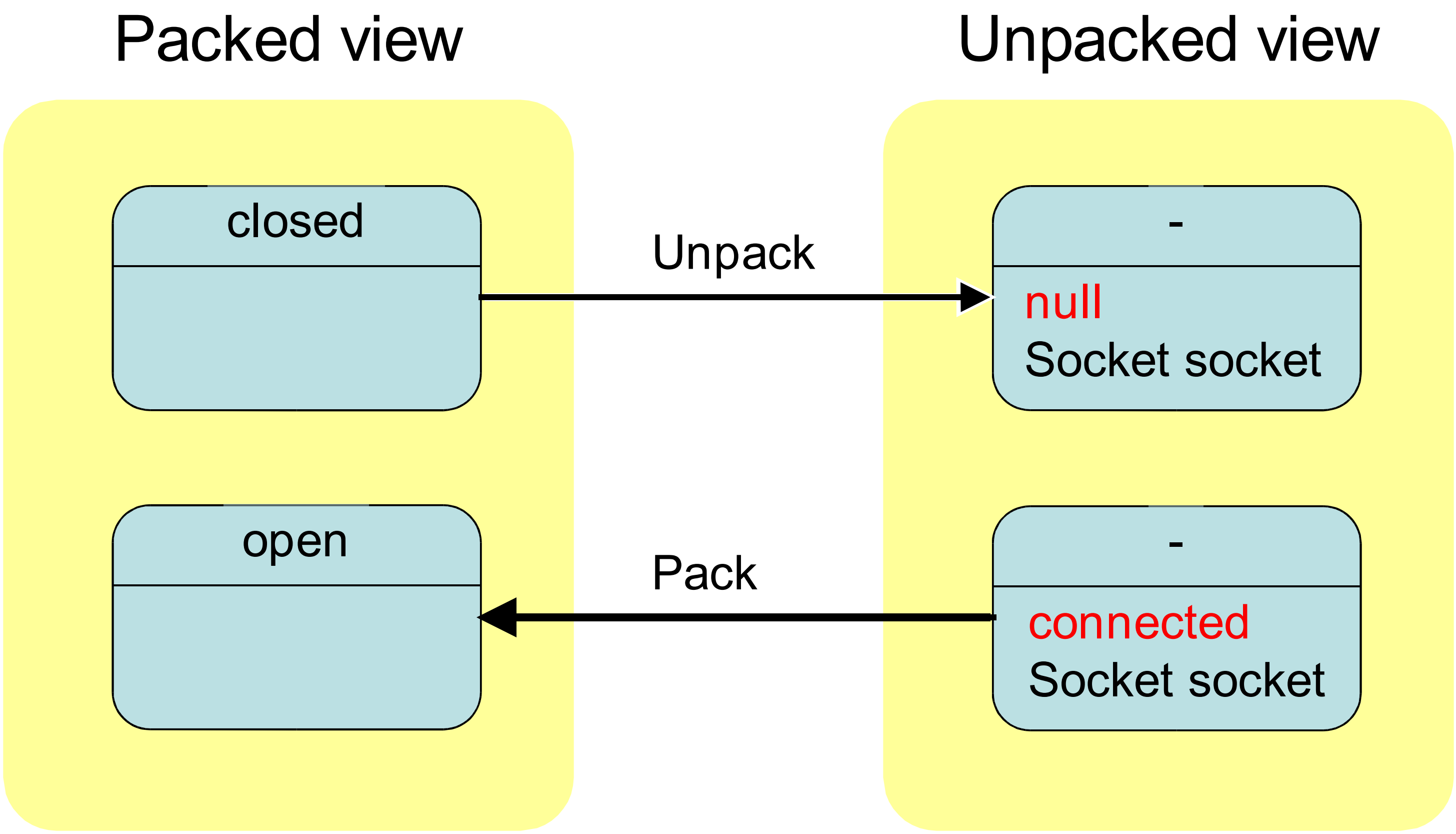
Typestates are predicates

- Named predicate over object state
  - connected, open, closed, etc…

x.state == open ≡ x.socket.state == connected

- Pack and unpack
  - Transitions between abstract named predicate and field knowledge
- Interpreted and abstract views

Pack and unpack



Unpack = apply definition

Pack = prove predicate

Abstract vs. interpreted typestate

In what contexts are pack and unpack allowed?

- No unpack or pack:
  - Completely abstract object view.
  - By name matching
- Unpack allowed
  - Object invariant visible anywhere
- Pack allowed
  - State changes allowed anywhere

Prototype design

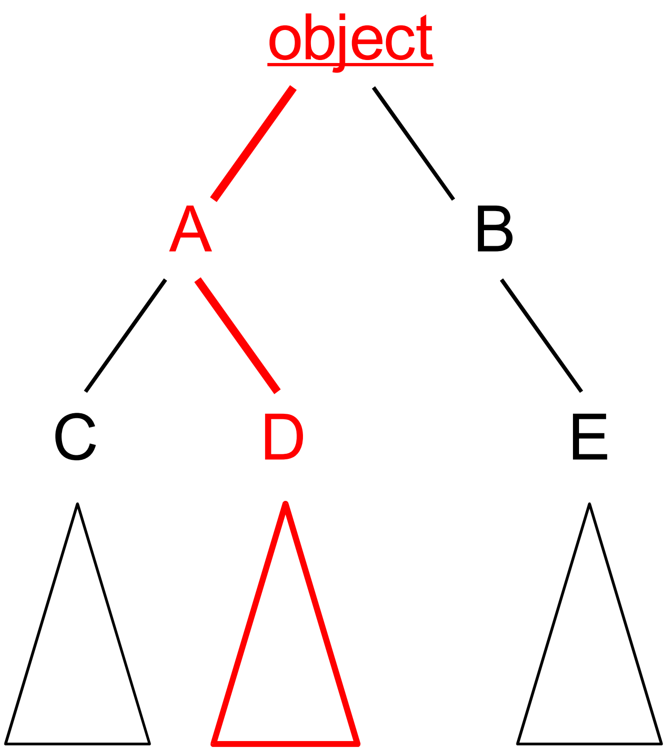
- Unpack anywhere
- Pack within scope of class

Reasoning about objects

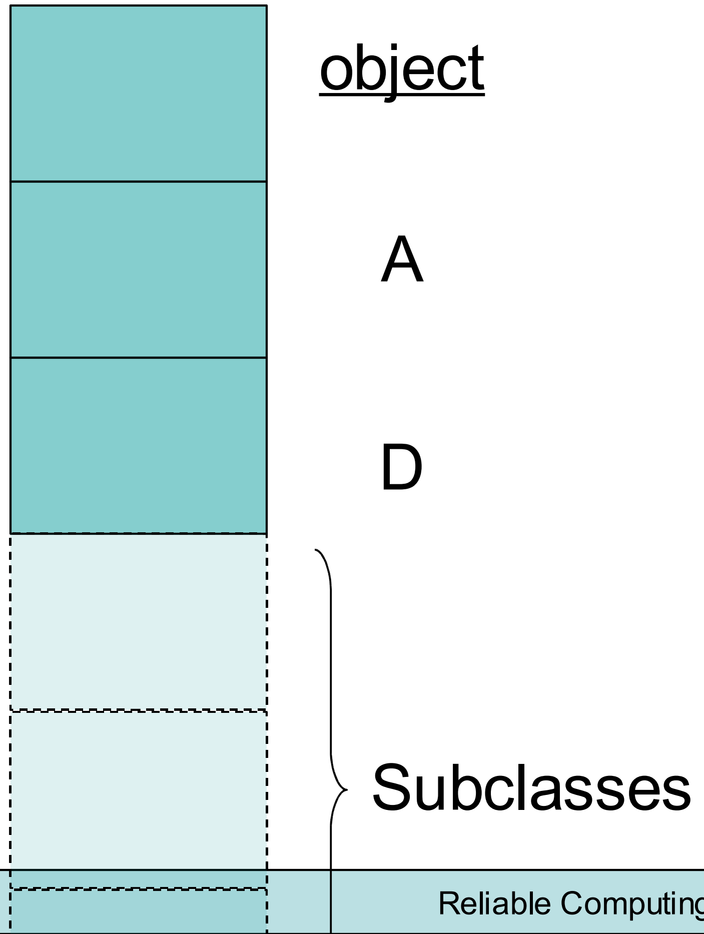
- Frame stack and subclass state
- Up- and down-casts
- Object extensions (subclassing)
  - Open state model. By default, every predicate on a frame means true.
- Sliding method semantics
  - Example: Caching WebPageFetcher

Modeling object state

Inheritance hierarchy



Frame stack



Modeling object state

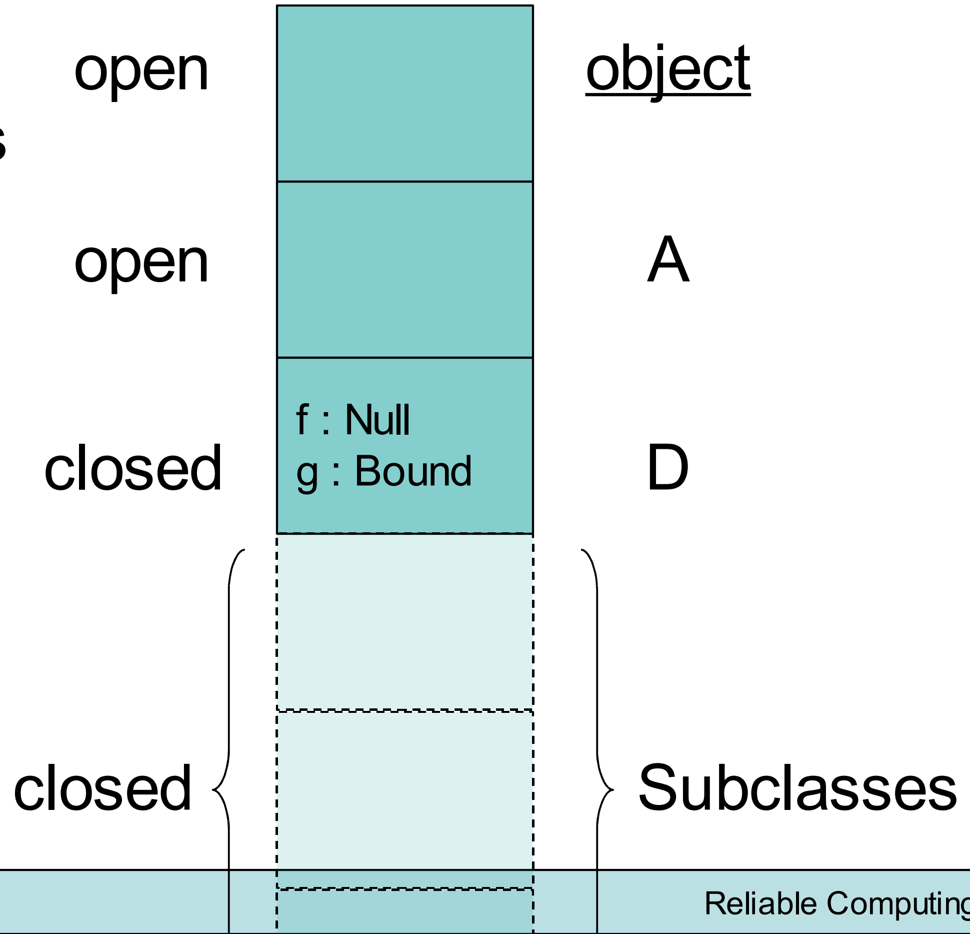
Typestate

- One per class frame
- Single symbolic state for unknown subclasses

Each frame

- Packed, or
- Unpacked

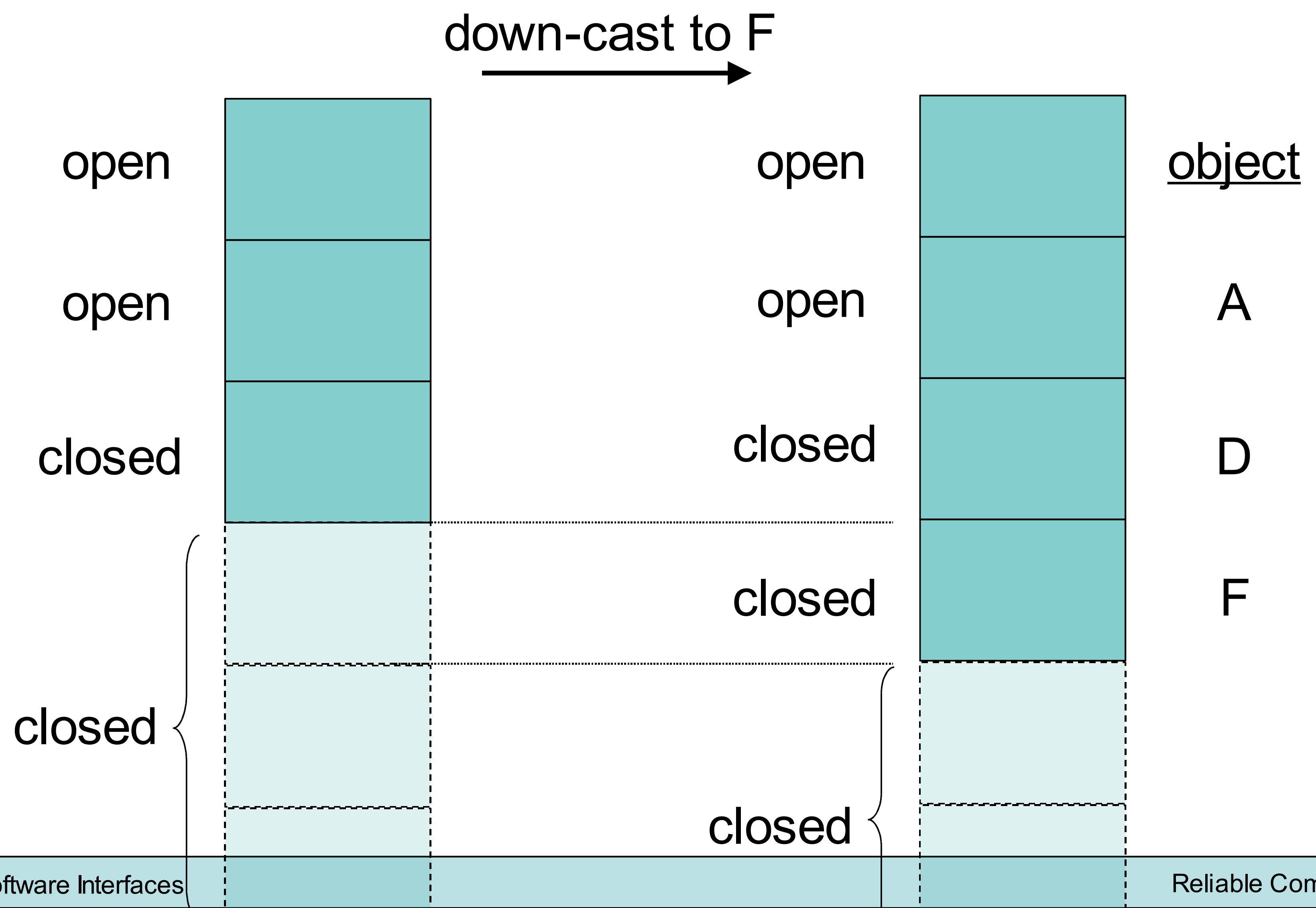
Frame stack



Motivation for frame stacks

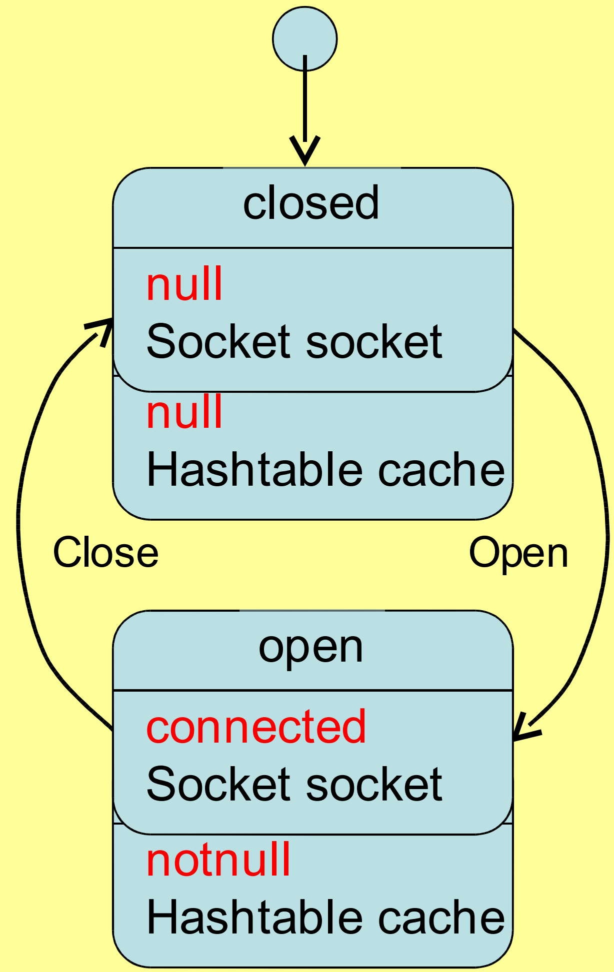
- Characterize complete object state
  - Including unknown subclasses
  - Needed for casts
  - Modularity
    - Invariants do not span frames
    - Extensibility : subclasses can interpret typestate individually
  - State changes
    - How to change state of entire object?
    - Code can only directly manipulate concrete frames

Up- and down-casts



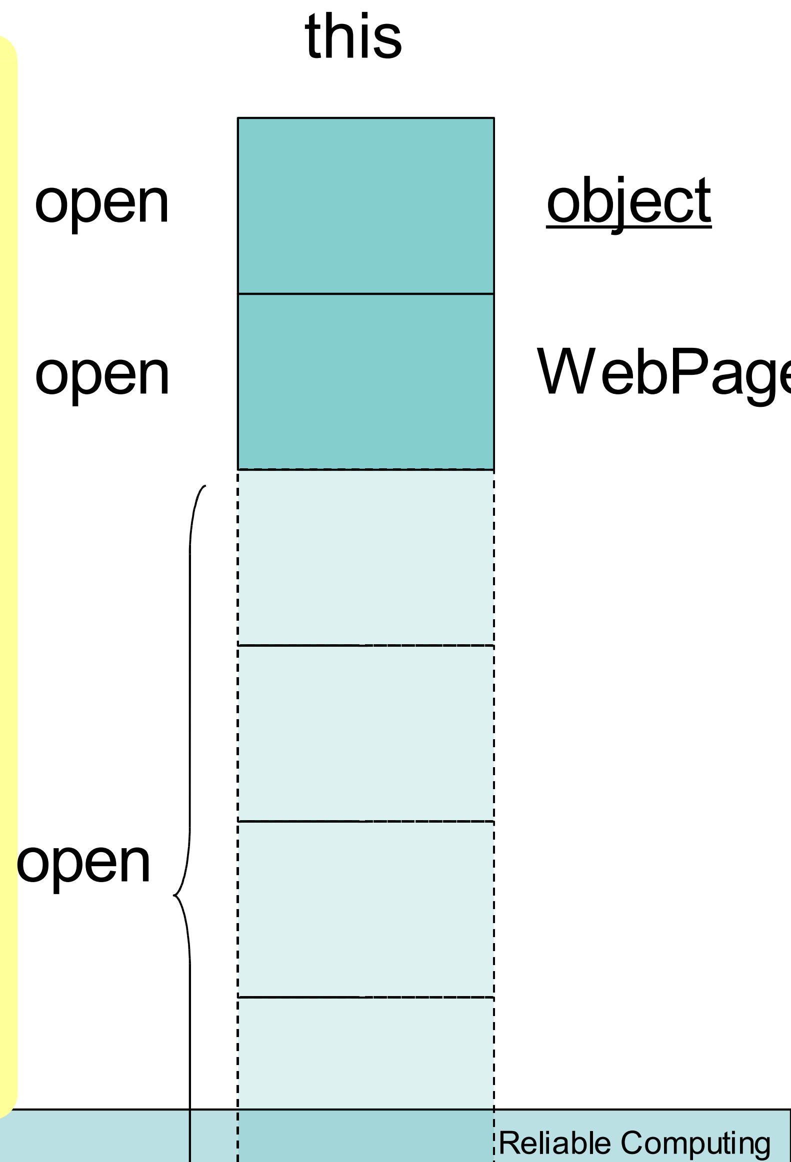
Typestate and subclassing

```
class CacheWebPageFetcher
 : WebPageFetcher
{
  [Null(WhenEnclosingState="closed")]
  [NotNull(WhenEnclosingState="open")]
  private Hashtable cache;

  [Creates("closed")]
  CachedWebPageFetcher ();

  [ChangesState("closed","open")]
  override void Open (string server);

  [InState("open")]
  override string GetPage (string url);

  [ChangesState("open","closed")]
  override void Close ();
}
```



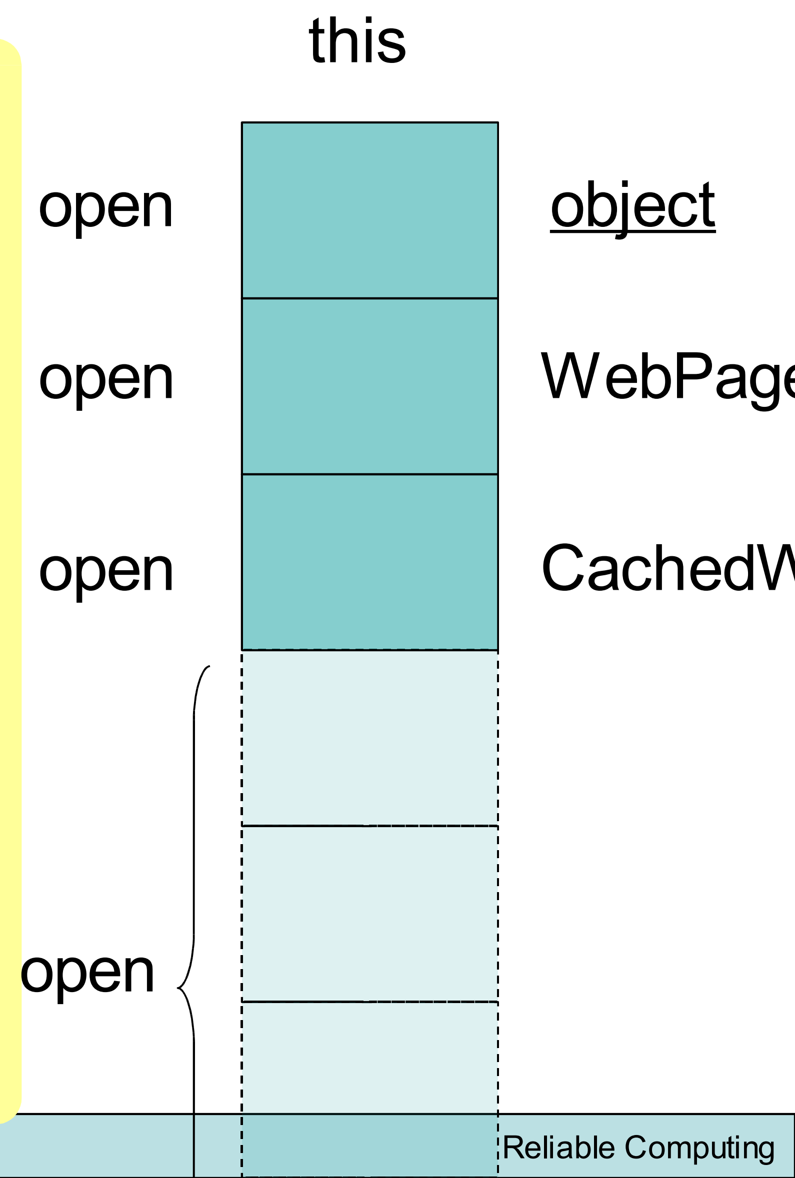
GetPage method

```
class WebPageFetcher {
  [InState("open")]
  virtual string GetPage (string url) {
    … this.socket …
  }
}


class CachedWebPageFetcher :
   WebPageFetcher {

  [InState("open")]
  override string GetPage (string url) {
    … this.cache …
    … base.GetPage(url) …
  }
}
```



GetPage method

```
class WebPageFetcher {
  …
  [InState("open")]
  virtual string GetPage (string url) {
    … this.socket …
  }
}


class CachedWebPageFetcher :
   WebPageFetcher {
  …
  [InState("open")]
  override string GetPage (string url) {
    … this.cache …
    … base.GetPage(url) …
  }
}
```

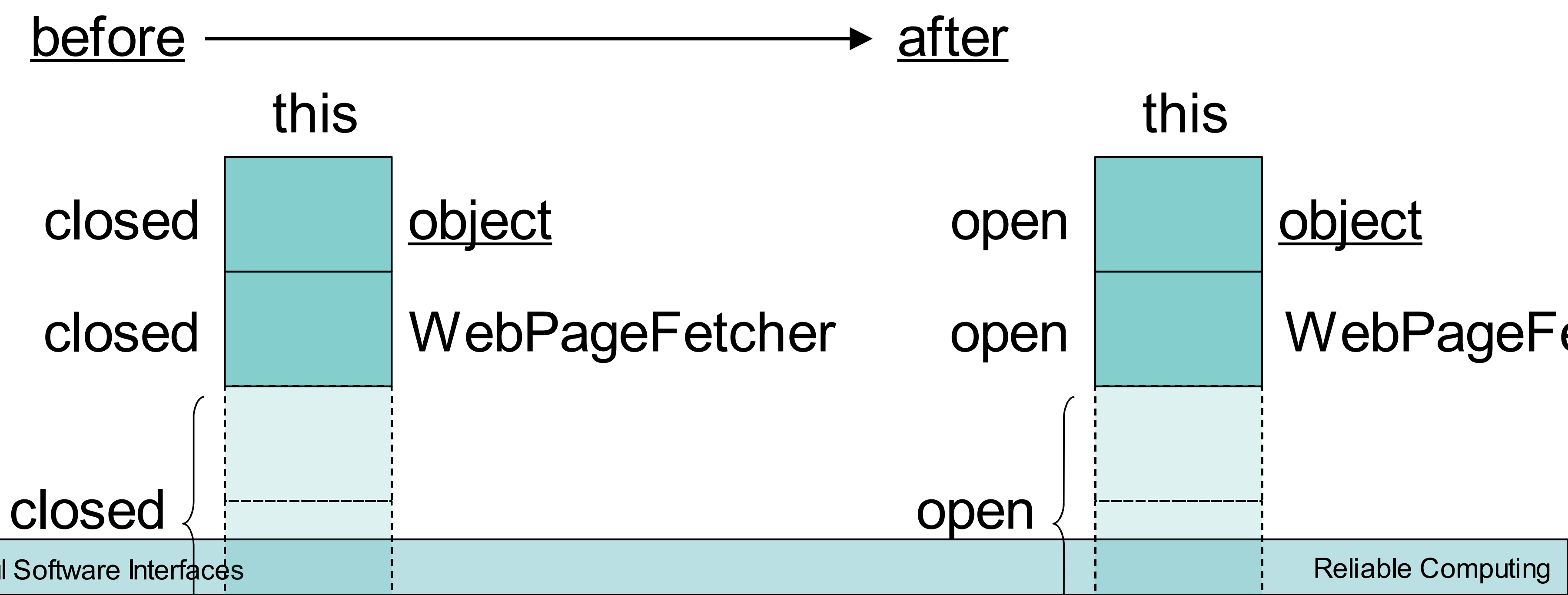


Establish new typestates

- **`GetPage`** leaves object in same typestate
- **`Open`** method must change frames from ‘closed’ to ‘open’
- How can a method change the typestate of all frames?

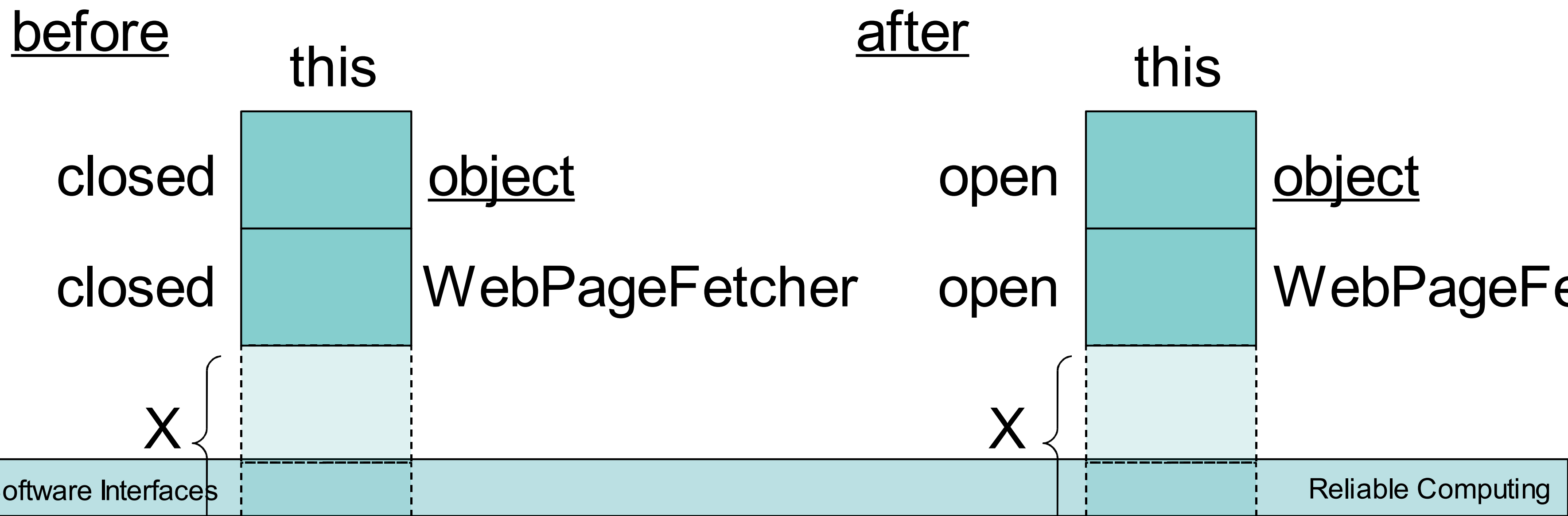
Open method (client view)

```
class WebPageFetcher {
 …
  [ChangesState(“closed”,”open”)]
  virtual void Open (string server);
}
```



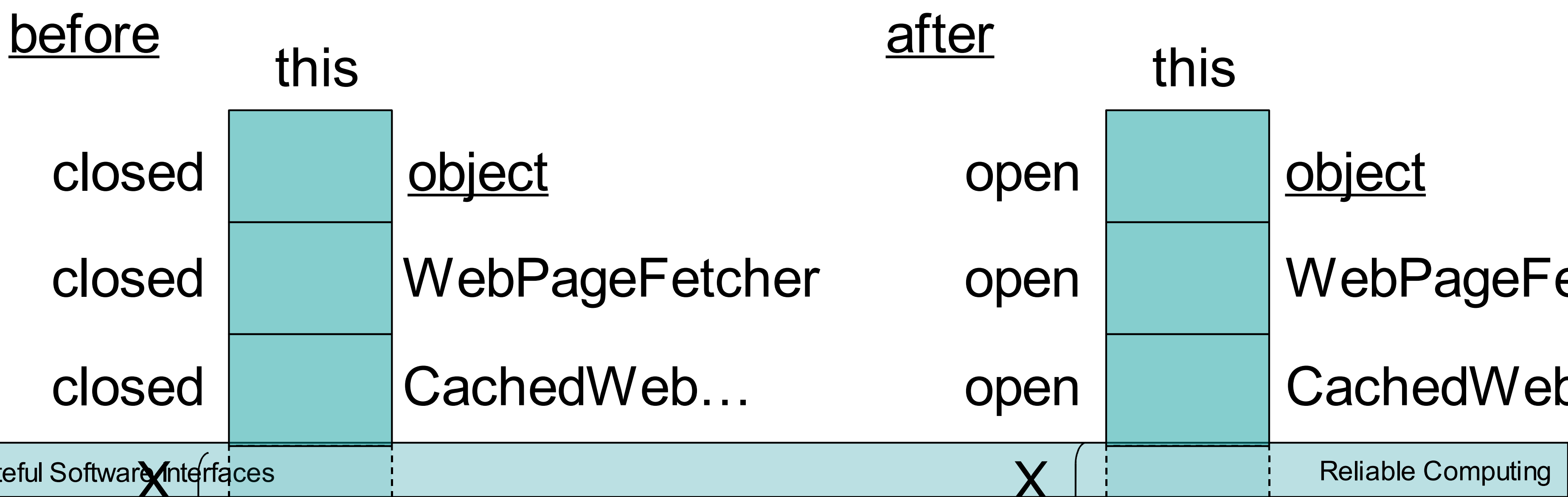
Open method (implementation)

```
class WebPageFetcher {
 …
  [ChangesState("closed","open")]
  virtual void Open (string server) {
    … this.socket = new Socket();
    …
  }
}
```



Open method (override)

```
class CachedWebPageFetcher {
 …
 [ChangesState("closed","open")]
  override void Open (string server) {
    … base.Open(server);
    … this.cache = new Hashtable();
  }
}
```



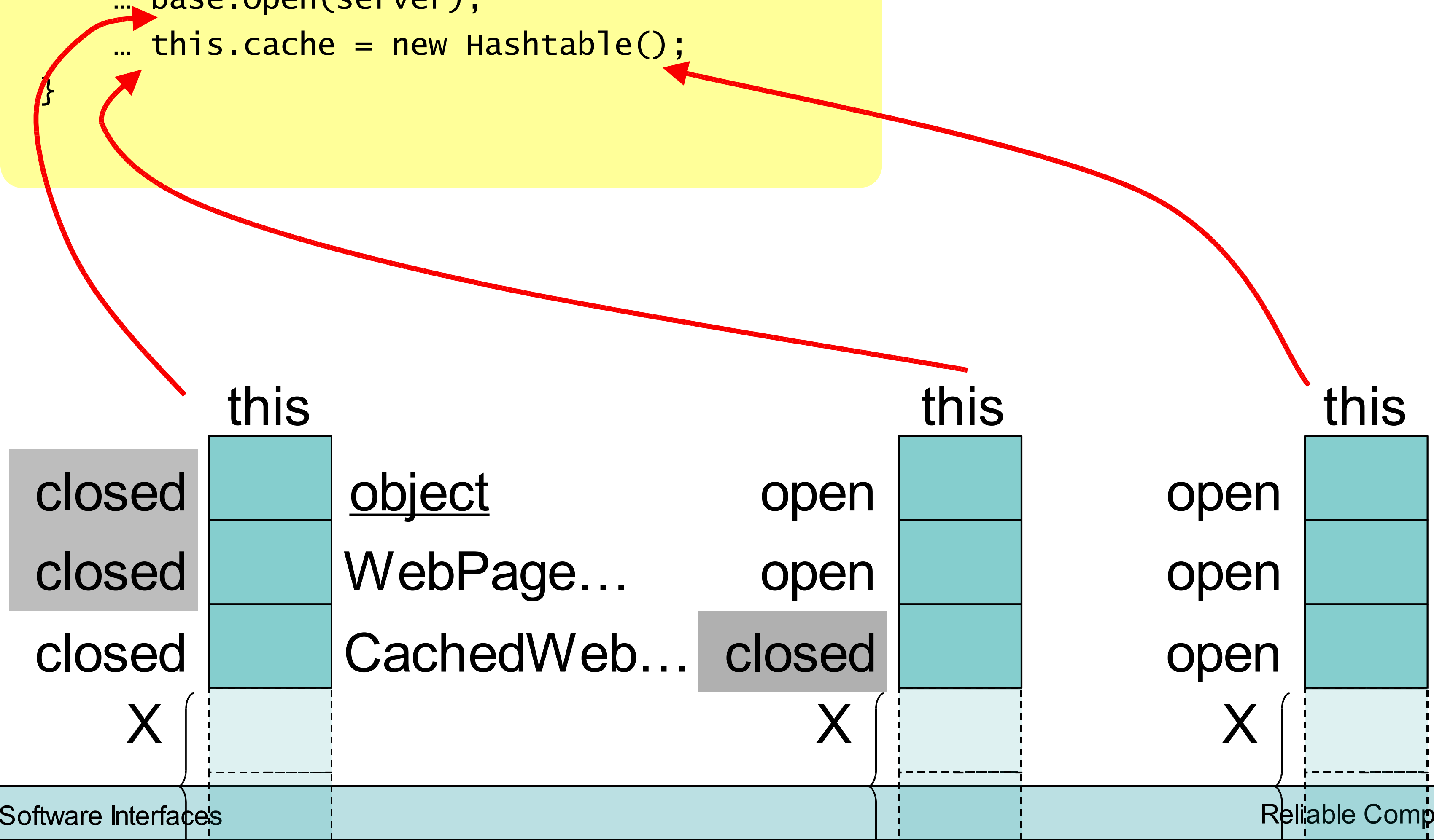
Sliding methods

Method signatures differ:

- Virtual call (entire object changes)
- Method specs for C.m (and non-virtual call)
  - only prefix including C changes
  - frames below C do not change

Open method (override)

```
[ChangesState("closed","open")]
override void Open (string server) {
    … base.Open(server);
    … this.cache = new Hashtable();
}
```



Object type-states summary

- Break object state into class frames
- Each frame has individual type state
- Symbolic treatment of type states of subclasses (ECOOP04)

- Related work: Spec# object invariants
  - Also frame based
  - Invariants allowed to depend on base-class fields
  - Requires suffix unpacking
  - See Journal of Object Technology (JOT article)

Singularity

- Research agenda: how do we build reliable software?
- Singularity OS
  - Based on type safe language and IL verification
  - Strong process isolation
  - Communication solely by messages

But: message-based code is difficult to write

  - Message not understood errors
  - Deadlocks

Goal:

  - Provide language and tool support for message-based programs and systems programming
  - Compile time detection of errors

Sing# Language

- Channel contracts
  - Specify typed message passing and valid protocol sequences
  - Provide efficient implementation based on pre-allocated receipt buffers
- rep structs
  - Hierarchical structures safe to exchange over channels
- Custom heaps
  - Explicit, but compiler verified, resource management for endpoints and other exchangeable data
- Switch-receive
  - asynchronous event pattern matching
- Overlays
  - Type safe structural casts for arrays of scalars

- Deadlock prevention methodology

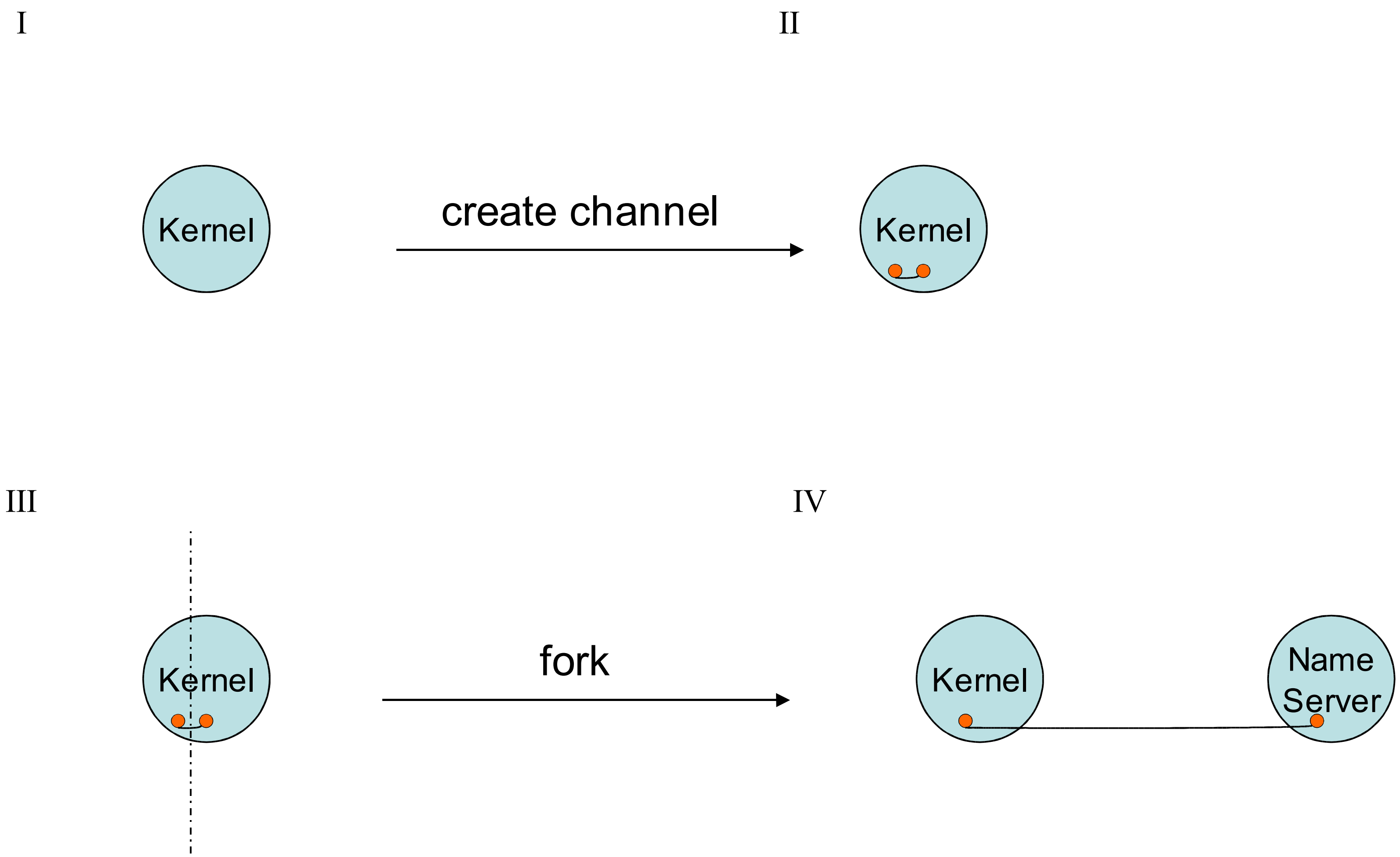
Deadlock prevention

- … in dynamically configured communication networks.

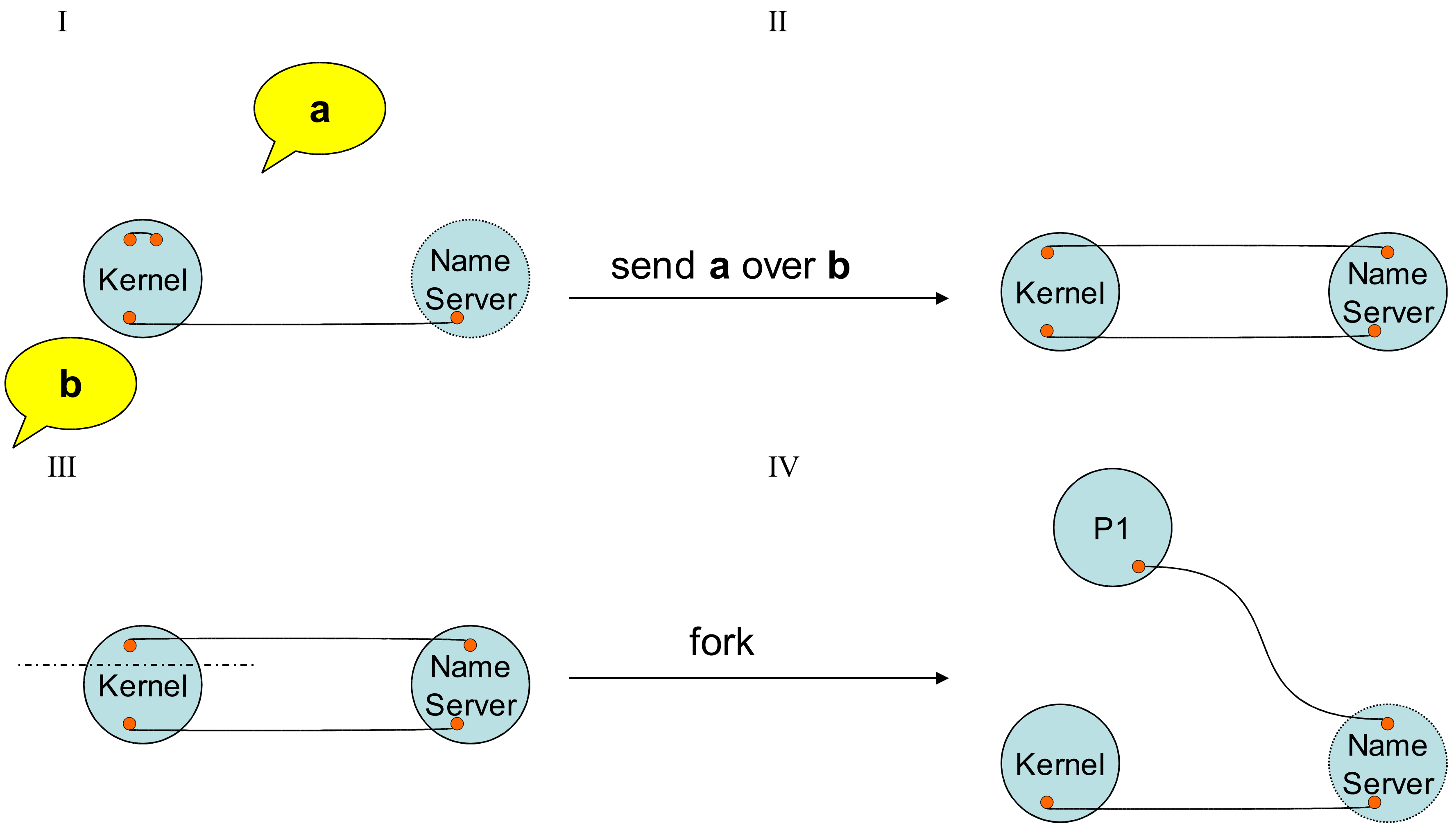
Communication model

- Inter-process communication via messages.
- Messages are exchanged over channels
  - Assume synchronous
- Channels are point-to-point
  - Two endpoints
  - Each owned by exactly one process
  - Bidirectional
- Endpoints can be sent over a channel
- Processes can fork, partitioning their endpoints

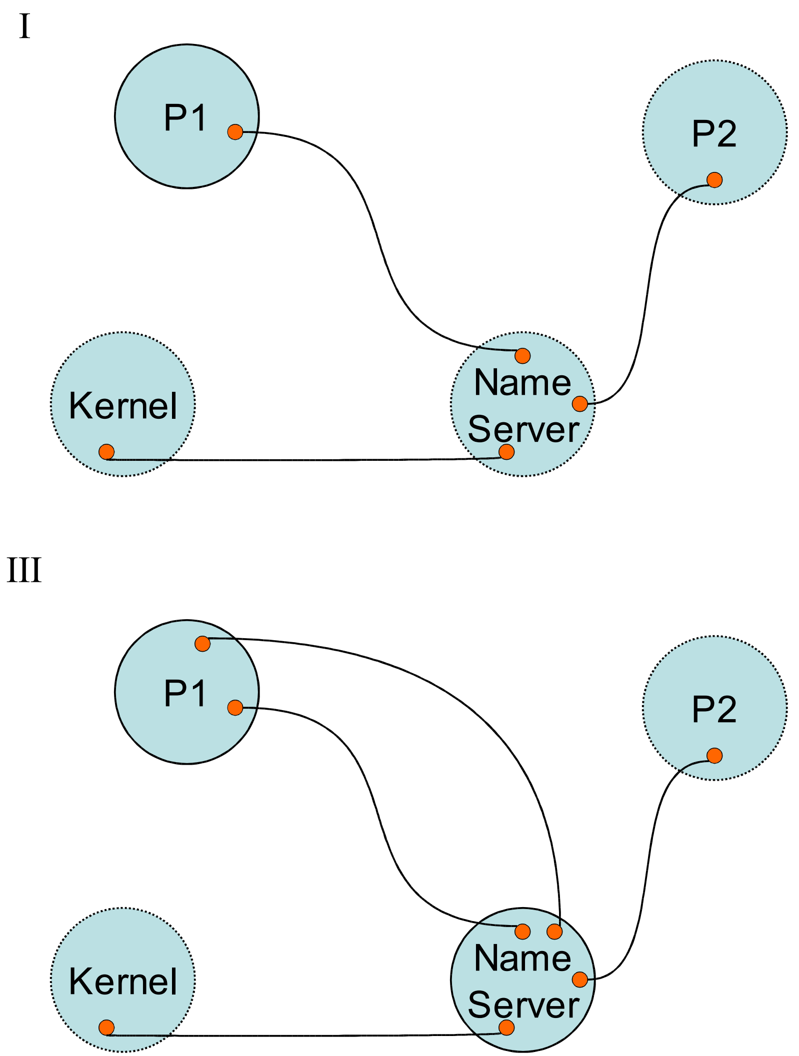
Communication model explained



Kernel creates a process



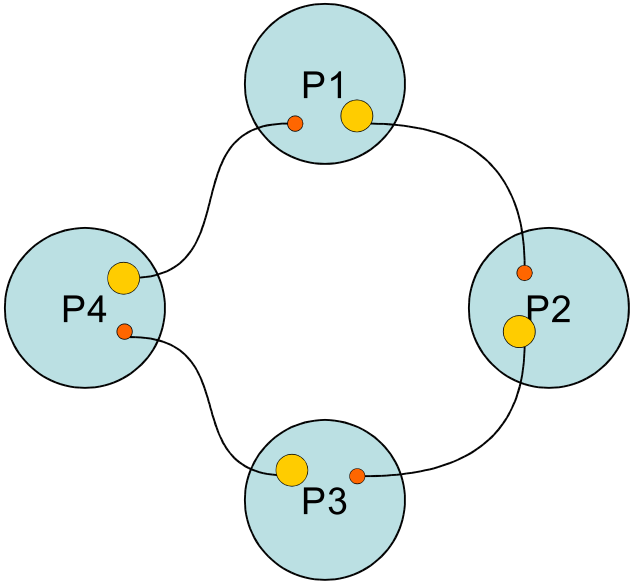
2 processes connect



Operational semantics

- At each step of the configuration, each process chooses one of three actions:
  1. Create channel
  2. Fork
  3. Communicate
     (by selecting a non-empty subset of its endpoints)
- Deadlock:
  - Every process wants to communicate,
    but no channel has both endpoints selected.

A dead lock

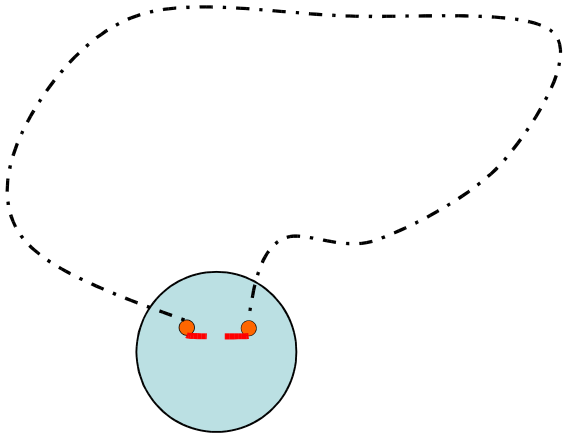


Basic idea: Obstructions

Configuration invariant:

At any point during execution,
for each cycle C in the graph,
there exists at least one process that witnesses C

- Witness process is responsible for breaking cycle
- A process witnesses a cycle via an obstruction,
  ie., a pair of endpoints local to a process connected by a path.

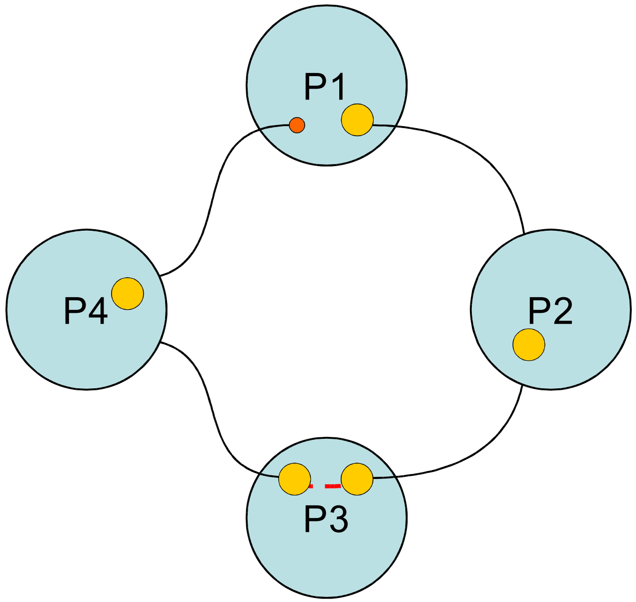


Breaking the cycle

Selection Strategy:

A process **P** wanting to communicate must select at least one endpoint a.

If **a** is obstructed with **b**, **P** must also select **b**.

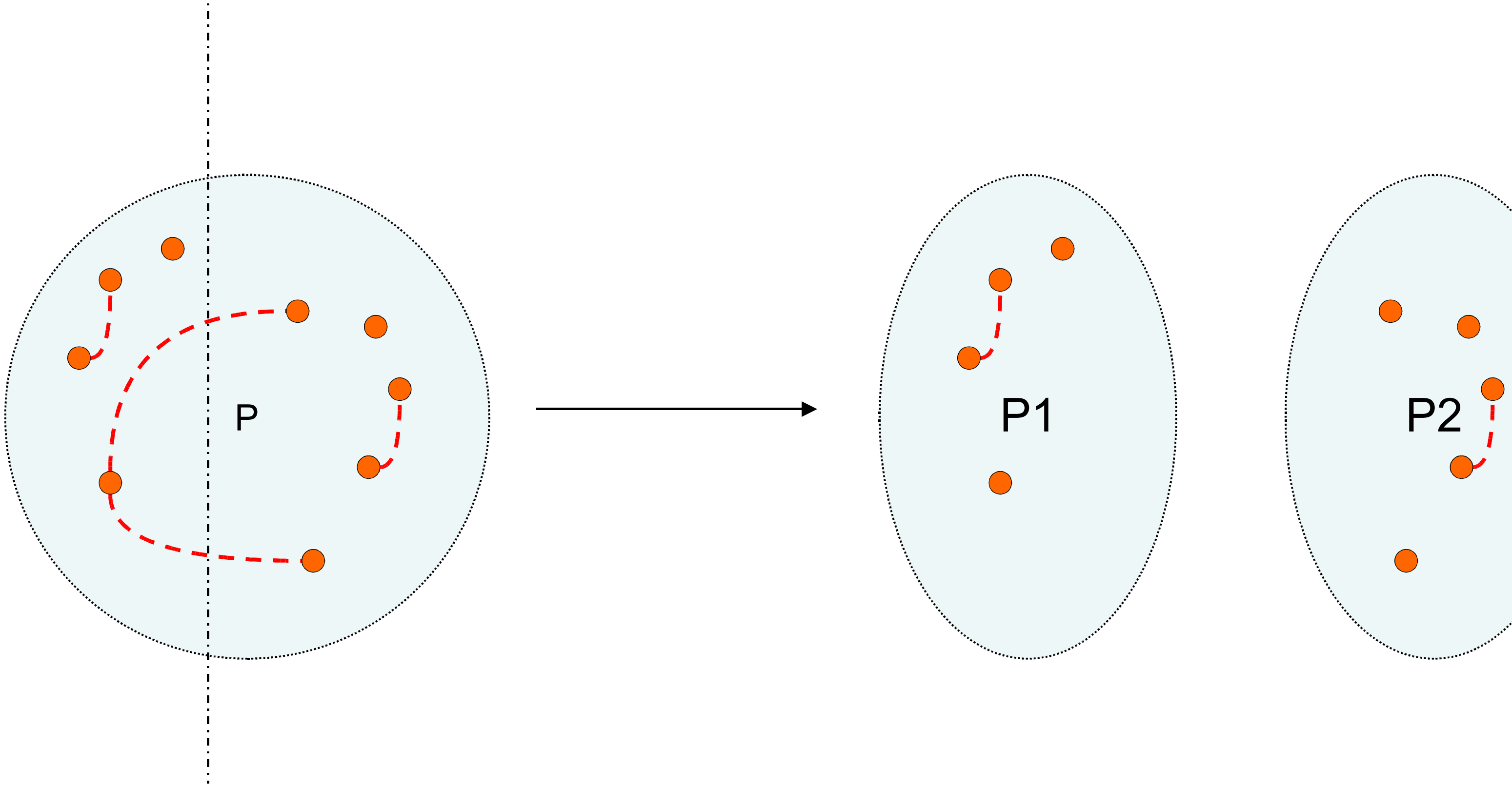


Instrumented Semantics

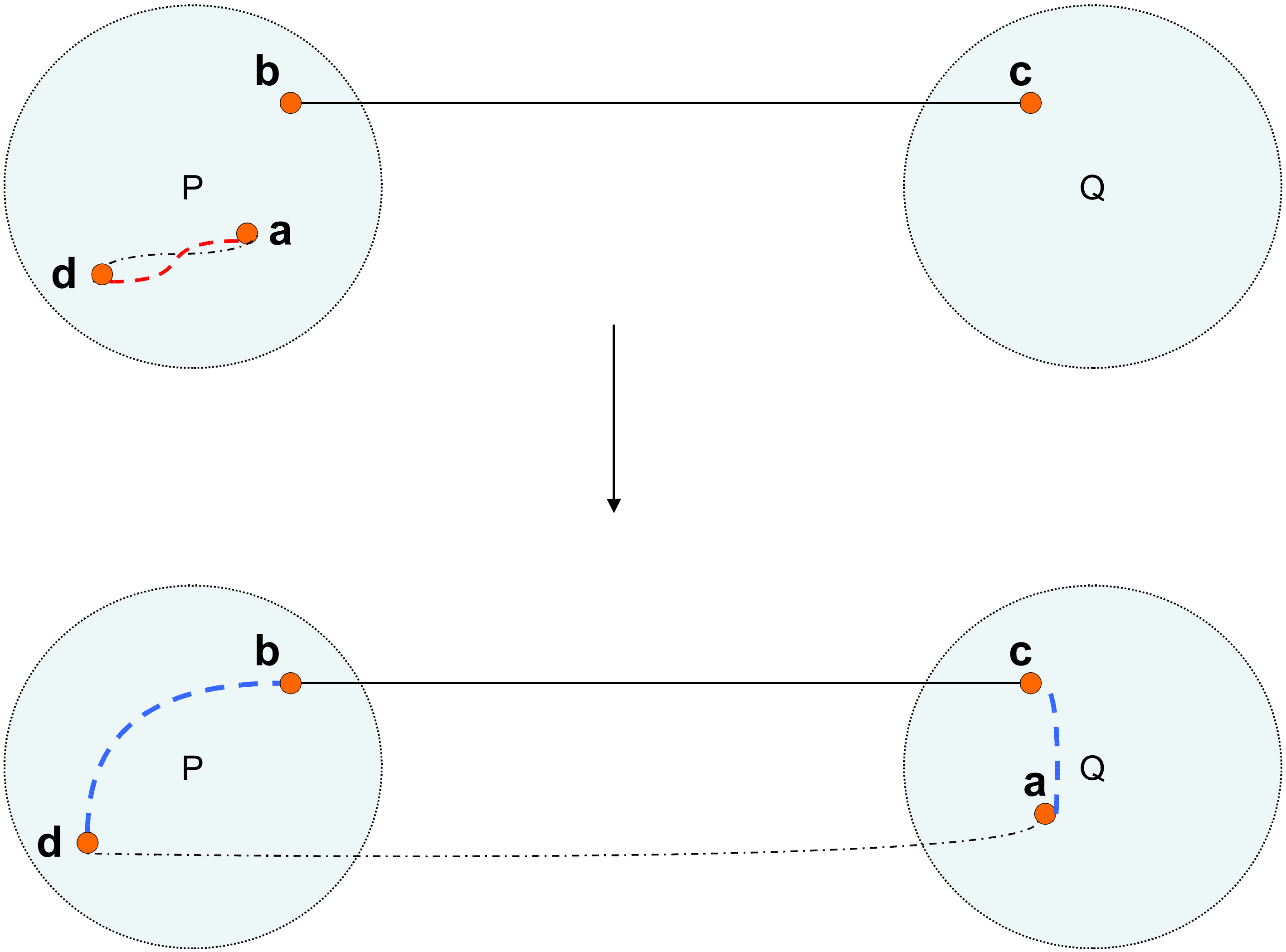
- Configurations contain a set of obstructions
  $O \subseteq E \times E$
- Actions operate on obstructions
  - Create channel
    - Adds obstruction between new endpoints
  - Fork
    - Can split obstructed endpoints
  - Move a over b
    - Sender closure: Add (d, e) if (a,d) and (b,e)
    - Receiver closure: Add (a, f) if (c, f)
    - Add (a,c) or (d, b) for all (d,a)

Create Channel

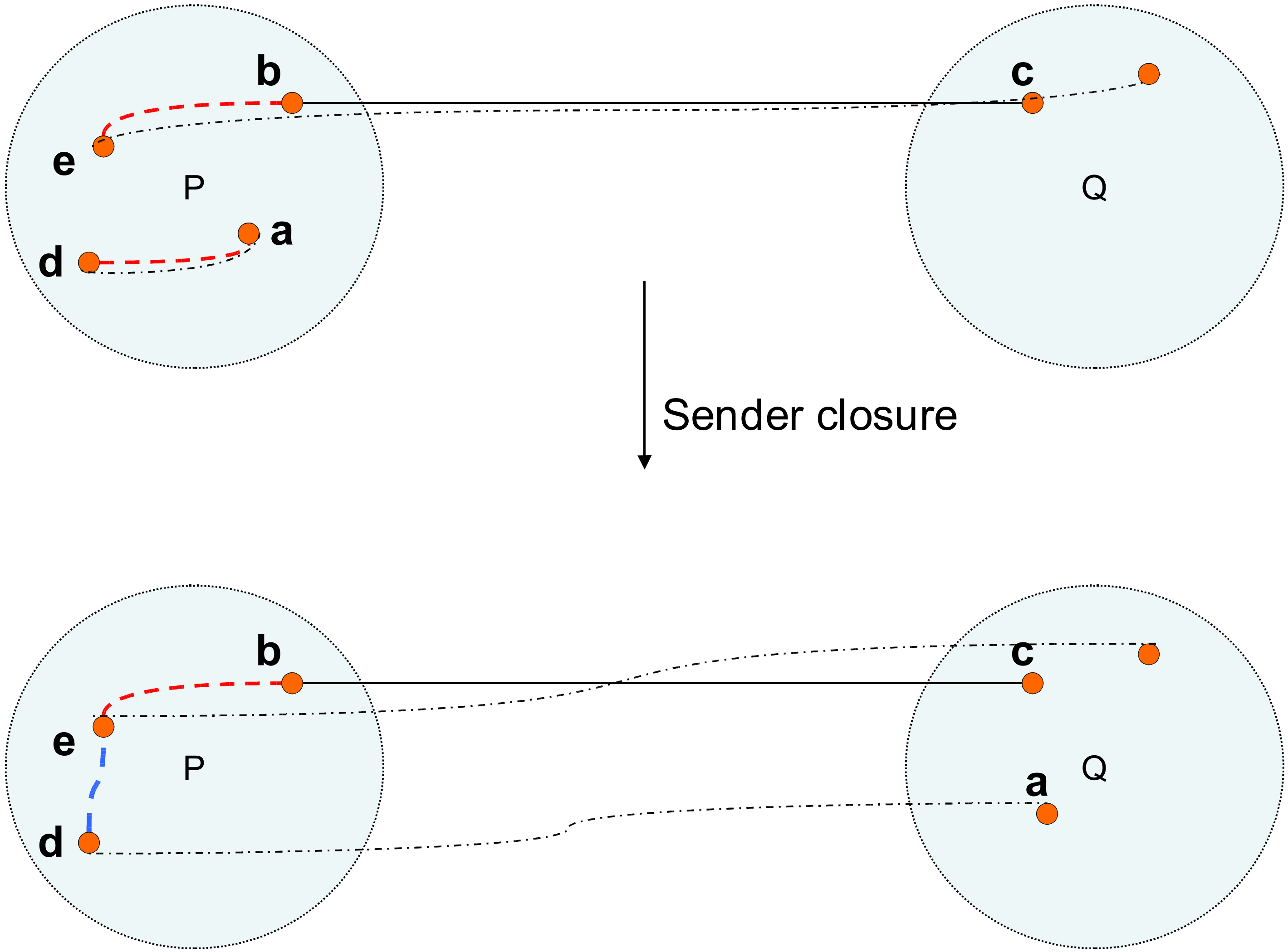
Fork



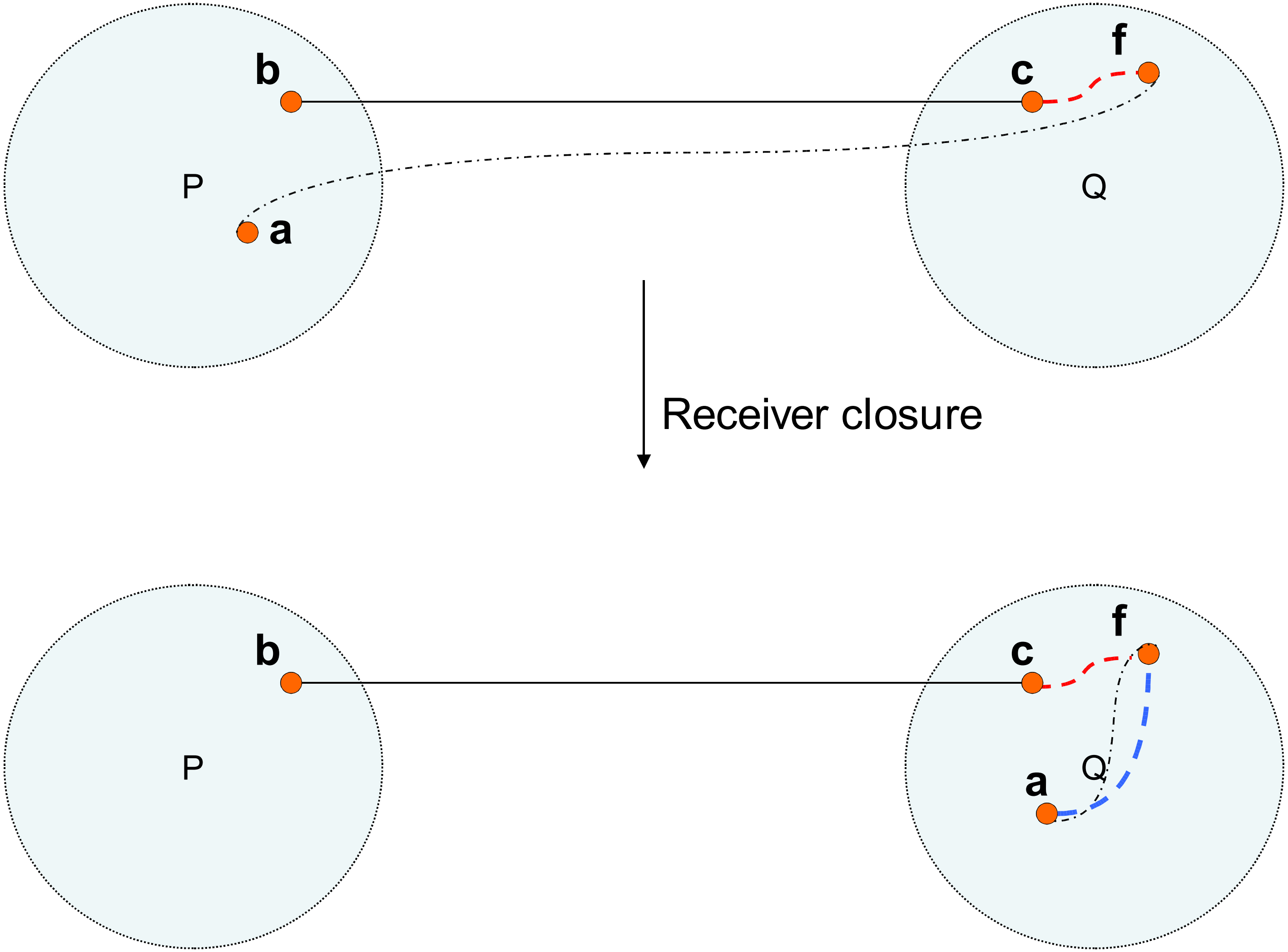
Send a over b (simple)



Send a over b (2)



Send a over b (3)



Type system

- Based on linear treatment of endpoints
- Tracking of obstructions
- Enforcing selection strategy at receives

- Status:
  - Still experimenting with expressiveness

Soundness

- Preservation:
  - Every step in the configuration maintains obstruction invariant: every cycle is covered by an obstruction
- Progress:
  - If all processes want to communicate, the endpoint selection strategy guarantees the existence of an enabled channel

Summary: deadlock prevention

- Surprising result
  - a modular type system, reasoning only locally, guarantees global dead-lock freedom
- Novelty:
  - not based on some locking order
  - network is dynamically changing
  - Network allowed to be cyclic (has to be)

Conclusion

- World is stateful, need ways to deal with changing state
- Type systems based on spatial logics can express many important rules
  - Resource management, type states, locking, etc.
  - Type systems advantage over first-order logics:
    - Higher-order
    - Abstraction over predicates
- Methodology not after-the-fact analysis
  - Language provides a programming model for correct usage
  - Language makes failures explicit
  - Make programmers deal with failures
  - Guide programmer from the beginning
- Programming languages based on such ideas
  - Under development in research community
    - Cyclone, Clay, Sing#, …

Open Research Questions

- Sweet spot in domain of spatial logics
  - Expressive but amenable to automation
  - Combination with traditional theories (arithmetic)
- Finding good abstractions for combining linear and non-linear data
- Dealing with complicated, cross-module heap invariants
  - e.g. Subject-Observer pattern
  - abstraction gets in the way
- Programming Language design

Backups

Clay (Chris Hawblitzel et.al.)

- Explicit memory capabilities and Presburger arithmetic
- Type Mem(i,τ)
- Explicit embedding of Mem in data, function args, return
- Explicit proof terms

$$\frac{C_1 \vdash e_{\mathsf{ptr}} : \mathsf{pt}(i) \qquad C_2 \vdash e_{\mathsf{mem}} : \mathsf{Mem}(i,\tau)}{C_1, C_2 \vdash \mathsf{load}(e_{\mathsf{ptr}}, e_{\mathsf{mem}}) : \tau \otimes \mathsf{Mem}(i,\tau)}$$

- Coercion functions for proof terms
- Case study: copying GC
- (ATS by Xi is similar in style)

Cyclone (Morrisett, Jim, Grossman)

- C replacement, very close to C
- Regions, ref counted and unique pointers
- Region lifetimes are stack like
  - Provides many useful lifetime constraints
- Very nice syntax and defaults