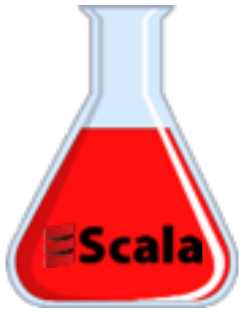


On the chemistry of typestate-oriented actors

Formal methods … in Action!

Silvia Crafa

Università di Padova, Italy

CurryOn - July 19th 2016

In distributed systems the

coordination of concurrent entities

is a key issue

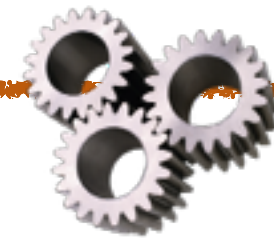
Protocol-Oriented Programming

thinking (hence programming)
in terms of
communication protocols

it requires:

- **high-level support to declare/express** the coordination protocol
- **support to check it!** …hopefully statically…(gradual) typing?

Easy to think,
Easy to reason about



Expressiveness
Performance

productively **interoperate**
with the other language abstractions

what are **suitable abstractions** for
Protocol-Oriented Programming?

shared memory
***data*-centric**

fits a ***centralised control***, of
distributed entites.

top-down *implementation of a*
global protocol
e.g., **session types**'s methodology

message passing
***communication*-centric**

which work with ***strong isolation principle***, ***locality*** of info and decisions

Actors

harmonic ***bottom-up composition***
of local entities

Overview

by examples in Scala + Akka

1. actors can be defined in a **TypeState-Oriented** style:
    - a TSOP (***different messages*** iterfaces *in different states*
    - a TSOP actor has *different behaviours in different states*

Scala Compiler

2. we let **types prevent protocol violations**:
    - types represent *actor interfaces*
    - *typed references* represent ***stateful*** actors
    - *Continuation-Passing* to keep track of the dynamic state-change: both explicit (à la Akka Typed, but…) and **implicit** (***Continuation Monad***)
    - is *robust with concurrent accesses* by ***mixin-in*** **Chemical Semantics**

Overview

by examples in Scala + Akka

1. actors can be defined in a **TypeState-Oriented** style:
   - a TSOP (**different *messages*** iterfaces in different states*
   - a TSOP actor has *different behaviours in different states*

Scala Compiler

only intended msgs at the intended states

clean logic natural def.

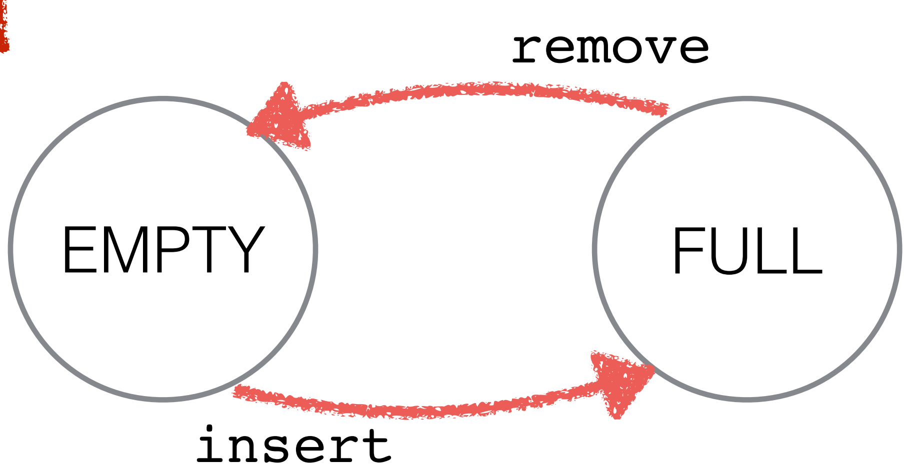
2. we let **typ**
   - types repres
   - *typed references* represent stateful actors
   - *Continuation-Passing* to keep track of the dynamic state-change: both explicit (à la Akka Typed, but…) and **implicit** (***Continuation Monad***)
   - is *robust with concurrent accesses* by ***mixin-in*** **Chemical Semantics**

1-place buffer ***as TSOP***

Buffer's protocol:

- 2 **states**, EMPTY and FULL
- in state EMPTY the interface only contains **`insert`**, that moves the state to FULL
- in state FULL the interface only contains **`remove`**, that moves the state to EMPTY

we can **express it as a FSM**



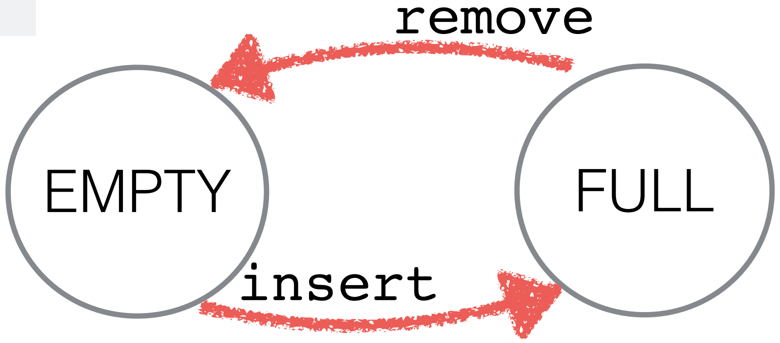
1-place buffer as TSOAP

```
case class insert[T](value:T)
case class remove()
```

each behaviour /state
defines **only**
the intended messages!

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }

  def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

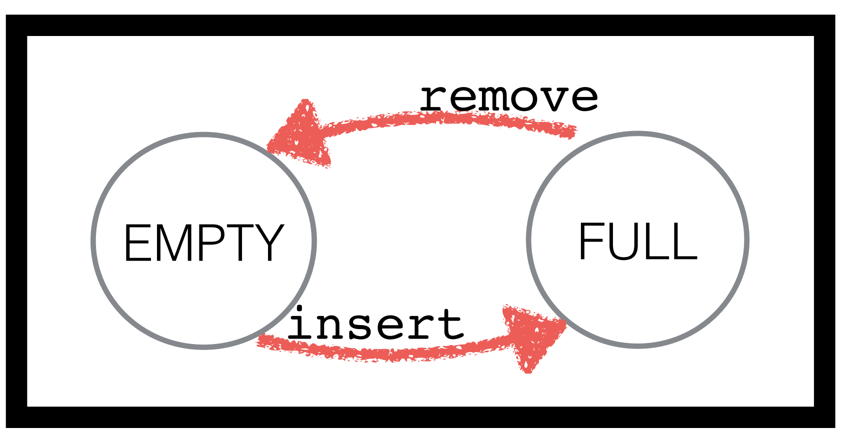
no defensive programming



1-place buffer as TSO**A**P

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

each behaviour /state
defines **only**
the intended messages!



bad msg!!

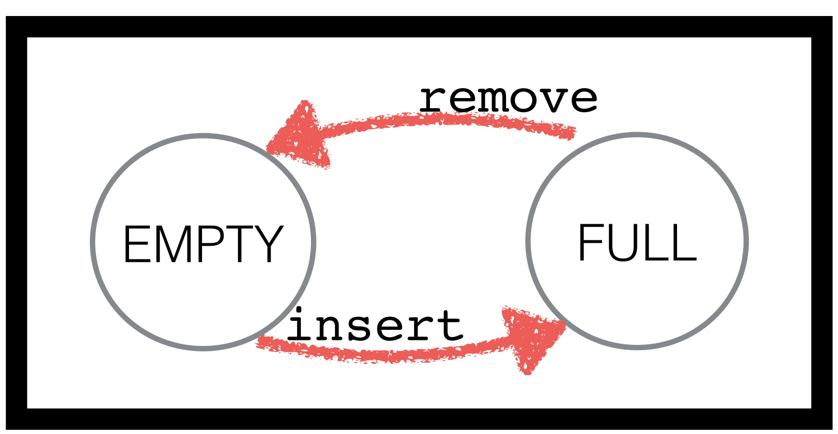
```
buffer ! “hello”
buffer ! insert(1)
buffer ! remove()
buffer ! remove()
```

bad state!!

1-place buffer as TSO**A**P

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

each behaviour /state
defines **only**
the intended messages!



Producer

```
insert(1)
insert(2)
insert(3)
…
```

```
remove()
remove()
remove()
…
```

Consumer

`insert` msg can arrive in the buffer's mailbox while the buffer is **in state FULL** !

we still want to ensure that `insert` / `remove` are served in the correct state!

1-place buffer ***as TSOP***

Buffer's protocol **Properties:**

1. **no other messages** but `insert` and `remove`
2. `insert` and `remove` **must be alternated** since they make the buffer switch state

```
buffer ! “hello”
```

`ActorRef` declares
`def !(msg:Any)`
which is *always well-typed*

add
a layer of typing!

encapsulates an
actor at state T

```
class TypedRef[-T](r:ActorRef) {
  def tyTell(msg:T) =  r ! msg
}
```

statically type-checks
that sent messages
belong to the interface `T`

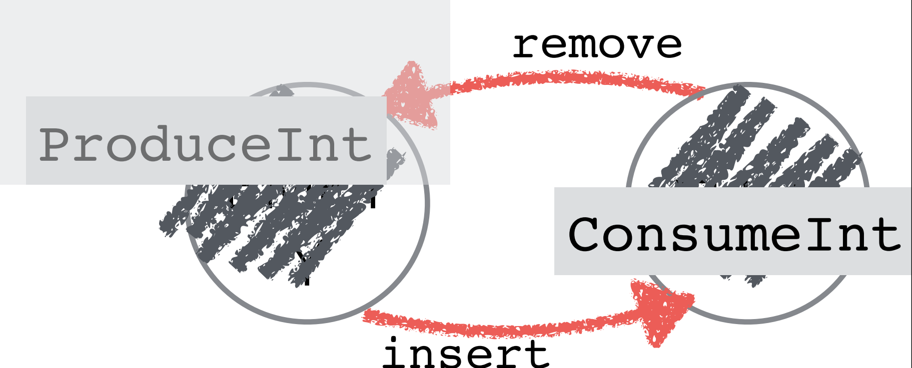
1-place buffer …*with typed reference*

```
trait BufferInterf
trait ProduceInt extends BufferInterf
trait ConsumeInt extends BufferInterf

case class insert[T](value:T) extends ProduceInt
case class remove()  extends ConsumeInt

class Buffer[T] extends Actor {  …as before… }
```

Nominal typing: a Type for each set of allowed messages, i.e.
an Interface for each state

```
val s = ActorSystem()
val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
val buffer = new TypedRef[ProduceInt](untypedBuffer)

val user = s.actorOf(Props(new Actor{
            buffer tyTell insert(4)    ok
            buffer tyTell  “hello”     does not compile

}))
```

1. **no other messages** but `insert` and `remove`

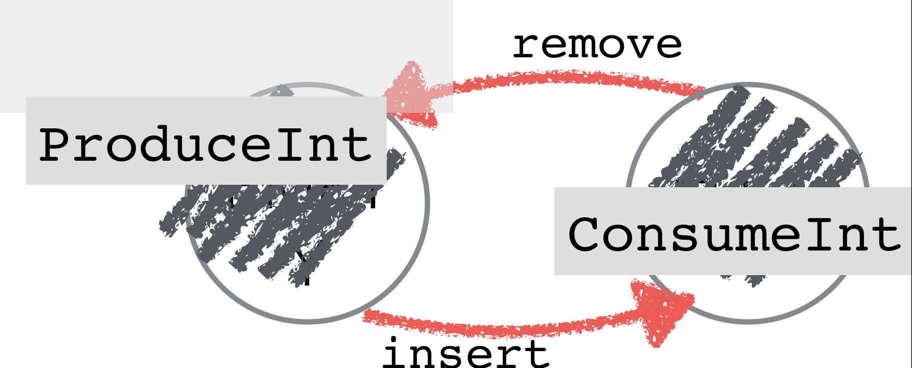


1-place buffer …*with typed reference*

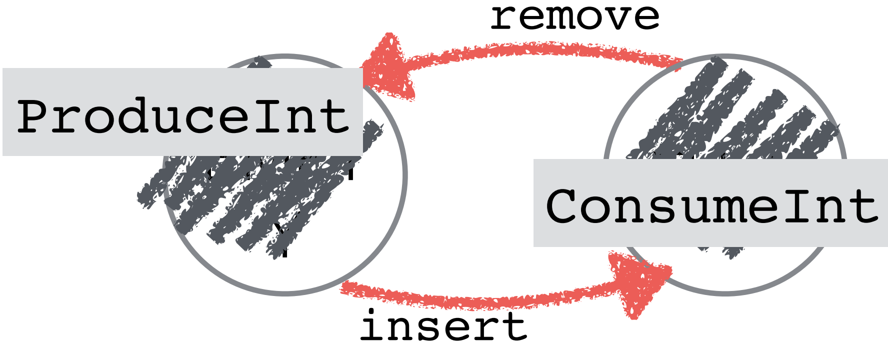
```
trait BufferInterf
trait ProduceInt extends BufferInterf
trait ConsumeInt extends BufferInterf

case class insert[T](value:T) extends ProduceInt
case class remove()  extends ConsumeInt

class Buffer[T] extends Actor {  …as before… }
```

Nominal typing: a Type for each set of allowed messages, i.e.
an Interface for each state

```
  val s = ActorSystem()
  val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
  val buffer = new TypedRef[ProduceInt](untypedBuffer)

  val user = s.actorOf(Props(new Actor{
               buffer tyTell insert(4)    ok
               buffer tyTell remove()    does not compile
})) 
```



1-place buffer …***with typed reference***



the **`buffer`** reference
dynamically changes its type
between `TypedRef[`**`Producent`**`]`
and `TypedRef[`**`ConsumeInt`**`]`

statically, we can only approximate these changes by a common supertype

…or… at any change we take **a new reference, statically typed with the new type**

Continuation-passing Style

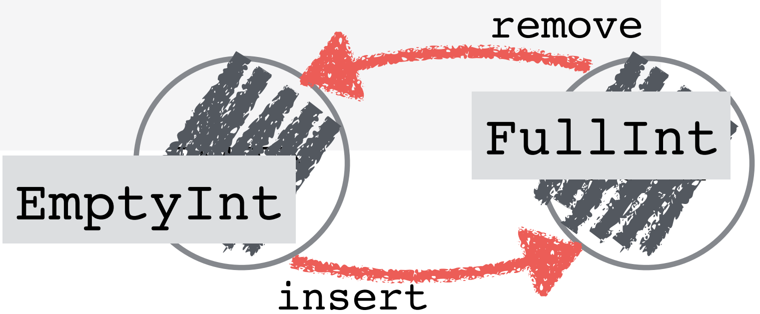
```
val s = ActorSystem()
val untypedBuffer = s.actorOf(Props(new Buffer[Int]
val buffer = new TypedRef[ProduceInt](untypedB

val user = s.actorOf(Props(new Actor{
            buffer tyTell insert(4)    ok
            buffer tyTell remove()    does not compile
}))
```

1-place buffer *with explicit continuations*

```
case class insert[T](value:T, replyTo:ActorRef) extends ProduceInt
case class remove(replyTo:ActorRef) extends ConsumeInt
case class insertReply(o:TypedRef[ConsumeInt])
case class removeReply[T](v:T, o:TypedRef[ProduceInt])
```

reply messages carry the continuation reference with suitable type

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T,r) => context.become(FULL(x))
                            r ! insertReply(new TypedRef[FullInt](self))
  }
  def FULL(x:T):Receive = {
      case remove(r) => context.become(EMPTY)
                        r ! removeReply(x, new TypedRef[EmptyInt](self))
  }
  def receive = EMPTY
}
```



1-place buffer ***with <u>explicit</u> continuations***

```
case class insert[T](value:T, replyTo:ActorRef) extends ProduceInt
case class remove(replyTo:ActorRef) extends ConsumeInt
case class insertReply(o:TypedRef[ConsumeInt])
case class removeReply[T](v:T, o:TypedRef[ProduceInt])
```

reply messages
carry the continuation reference
with suitable type

```
  val s = ActorSystem()
  val untypedBuffer = s.actorOf(Props(new Buffer[Int]),"buff")
  val buffer = new TypedRef[ProduceInt](untypedBuffer)

  val user = s.actorOf(Props(new Actor{
       buffer tyTell insert(1,self)
       def run(v:Int) :Receive = {
           case insertReply(o) =>  o tyTell remove(self)
           case removeReply(x,o) => o tyTell insert(v+1,self)
                                    context.become(run(v+1))
       }
       def receive = run(0)
}))
```

à la
Akka Typed

1-place buffer ***with <u>implicit</u> continuations***

```
case class insert(value:Int) extends ProduceInt
case class remove() extends ConsumeInt
```

```
  val s = ActorSystem()
  val untypedBuffer = s.actorOf(Props(new Buffer),"buff")
  val buffer = new ProtRef[ProduceInt](untypedBuffer, Buffer.protocol)

  val user = s.actorOf(Props(new Actor{
       for {
           o <- buffer tyTell insert(1)
           o <-  o tyTell remove()
           o <-  o tyTell insert(2)
           o <-  o tyTell remove()
           o <-  o tyTell insert(3)
               // o tyTell insert(4) compiler error
           o <-  o tyTell remove()
       } yield print("END")

       def receive= PartialFunction.empty
}))
```

(Typed) Monad

1-place buffer ***with <u>implicit</u> continuations***

```
case class insert(value:Int) extends EmptyInt
case class remove() extends FullInt
```

```
  val s = ActorSystem()
  val untypedBuffer = s.actorOf(Props(new Buffer),"buff")
  val buffer = new ProtRef[ProduceInt](untypedBuffer, Buffer.protocol)

  val user = s.actorOf(Props(new Actor{
        for {
             o <- buffer tyTell insert(1)
             o <-  Buffer.afterInsert(o) tyTell remove()
             o <-  Buffer.afterRemove(o) tyTell insert(2)
             o <-  Buffer.afterInsert(o) tyTell remove()
             o <-  Buffer.afterRemove(o) tyTell insert(3)
                 //Buffer.afetrInsert(o) tyTell insert(4) compiler error
             o <-  Buffer.afetrInsert(o) tyTell remove()
        } yield print("END")

       def receive= PartialFunction.empty
}))
```

(Typed) Monad

encapsulates both **the current state** and the **state transitions**

```
class ProtRef[-T](r:ActorRef, protocol: T => Continuation)

   def tyTell(msg:T) : Continuation = { … }
```

a pair

`(Promise[ProtRef[NextState]], Future[ProtRef[NextState]])`

where `NextState` is an **Abstract Type**

the **receiver** will asynchronously complete the ***promise***

while the **user** continues its protocol using the ***future*** (with for-notation)

```
val buffer = new ProtRef[ProduceInt](untypedBuffer, Buffer.protocol)
val user = s.actorOf(Props(new Actor{
       for {
            o <- buffer tyTell insert(1)
            o <- Buffer.afterInsert(o) tyTell remove()
            o <- Buffer.afterRemove(o) tyTell insert(2)
            o <- Buffer.afterInsert(o) tyTell remove()
            o <- Buffer.afterRemove(o) tyTell insert(3)
            // Buffer.afterInsert(o) tyTell insert(4)   compiler error
            o <- Buffer.afetrInsert(o) tyTell remove()
       } yield print("END")

     def receive= PartialFunction.empty }))
```

```
class ProtRef[-T](r:ActorRef, protocol: T => Continuation)

   def tyTell(msg:T) : Continuation = { … }
```

a pair

`(Promise[ProtRef[NextState]], Future[ProtRef[NextState]])`

where `NextState` is an **Abstract Type**

…cly

while the **user** continues its protocol using the ***future*** (with for-notation)

the compiler statically types the user code with an abstract type,
hence we need "phantom types"-casts

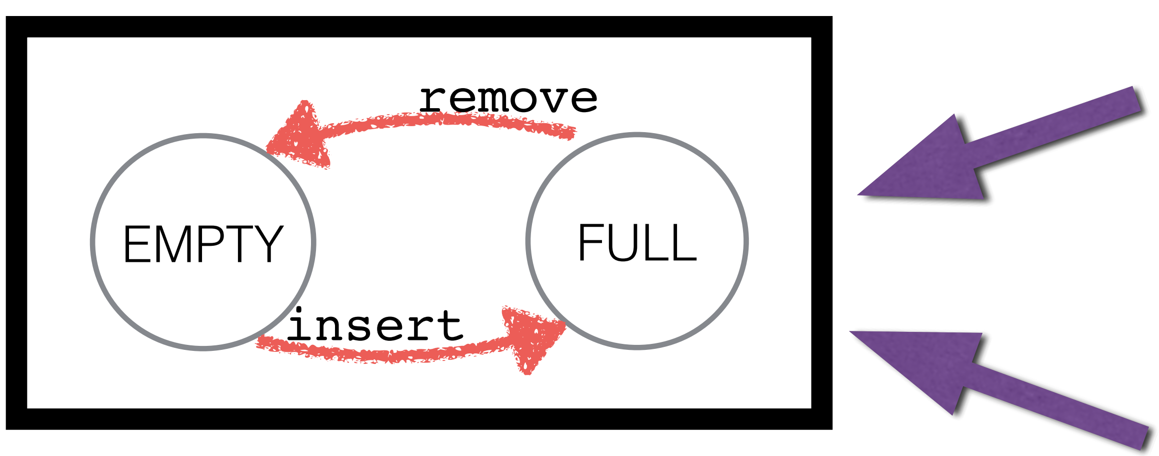
…boilerplate code produced by the protocol

```
             …Ref[ProduceInt](untypedBuffer, Buffer.protocol)
      …     …torOf(Props(new Actor{
        for {
            o <- …uffer tyTell insert(1)
            o <- Bu…fer.afterInsert(o) tyTell remove()
            o <- Buffer.afterRemove(o) tyTell insert(2)
            o <- Buffer.afterInsert(o) tyTell remove()
            o <- Buffer.afterRemove(o) tyTell insert(3)
            // Buffer.afterInsert(o) tyTell insert(4)  compiler error
            o <- Buffer.afetrInsert(o) tyTell remove()
        } yield print("END")

      def receive= PartialFunction.empty }))
```

1-place buffer as TSOAP

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

different states with **<u>different interfaces</u>**



Producer

```
insert(1)
insert(2)
insert(3)
…
```

```
remove()
remove()
remove()
…
```

Consumer

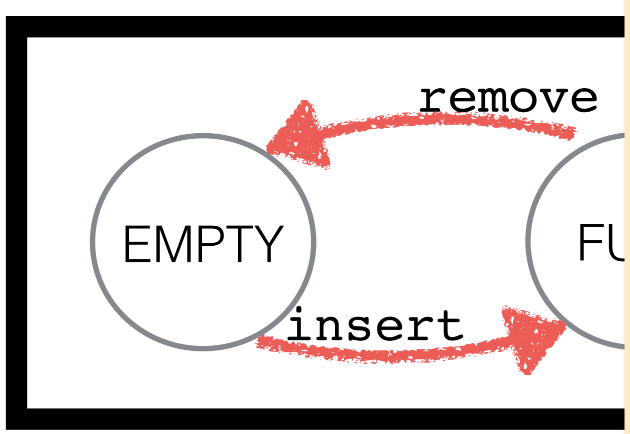
`insert` msg can arrive in the buffer's mailbox while the buffer is **in state FULL** !

we still want to ensure that `insert` / `remove` are served in the correct state!

1-place buffer as TSOAP

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case insert(x:T) => context.become(FULL(x))
 }
 def FULL(x:T):Receive = {
      case remove() => context.become(EMPTY)
  }
  def receive = EMPTY
}
```

different states with **different interfaces**



Akka Typed

each dynamic behaviour *must handle the whole set of messages…*

different behaviours but the same interface

`insert` msg can arrive in the buffer's mailbox while the buffer is **in state FULL** !

we still want to ensure that `insert` / `remove` are served in the correct state!

Chemical model of concurrency (Berr… ol’92)

just the messages
***eventually processed*:**
`ProtocolMsg`

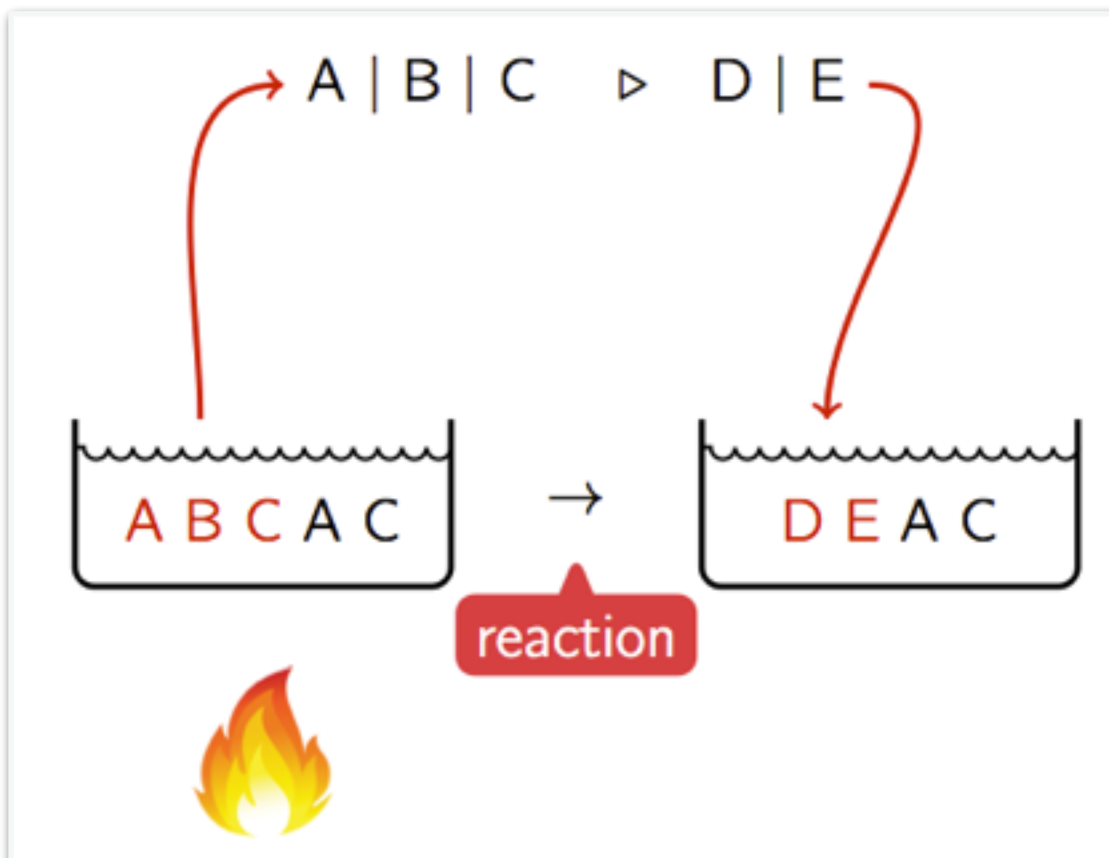
The **behaviour** is described by
reaction rules
that consume some molecule
and produce new ones

incoming message

keep a ~~molecule~~
until a reaction
is ~~triggered~~!

the correct state
is entered

A **state** is a
soup of molecules
(received messages)



Chemical semantics ....

```
class Buffer[T] extends Actor {
  def EMPTY:Receive = {
      case (insert(x),p:Promise[…]) => context.become(FULL(x))
                              p success new ProtRef[ProduceInt](self,Buffer.protocol)
  }
  def FULL(x:T):Receive = {
      case (remove(),p:Promise[…]) => context.become(EMPTY)
                              p success new ProtRef[ConsumeInt](self,Buffer.protocol)
  }
  def receive = EMPTY
}
```

different states with **different interfaces**

Chemical semantics ***by mix-ins***

```
class Buffer[T] extends Actor with Chemical {
  def EMPTY:Receive = chemReact {
      case (insert(x),p:Promise[…]) => chemBecome(FULL(x))
                             p success new ProtRef[ProduceInt](self,Buffer.protocol)
  }
  def FULL(x:T):Receive = chemReact {
      case (remove(),p:Promise[…]) => chemBecome(EMPTY)
                             p success new ProtRef[ConsumeInt](self,Buffer.protocol)
  }
  def receive = EMPTY
}
```

different states with **different interfaces**

```
trait ProtocolMsg

trait Chemical extends Actor with Stash {
  private def check() = { unstashAll() }
  private def keep :Receive = { case (msg:ProtocolMsg,p) => stash() }
  def chemBecome(newState:Receive)={ check(); context.become(newState) }
  def chemReact(behave:Receive):Receive = behave orElse keep
}
```

```
class Producer(buffer:ProtRef[ProduceInt]) extends Actor {
   for {
      o <- buffer tyTell insert(0)
      o <- Buffer.afterInsert(o) tyTell insert(10)
      o <- Buffer.afterInsert(o) tyTell insert(20)
      o <- Buffer.afterInsert(o) tyTell insert(30)
      o <- Buffer.afterInsert(o) tyTell insert(40)
   } yield println("End Producer")
  def receive = PartialFunction.empty
}
```

```
class Consumer(buffer:ProtRef[ConsumeInt]) extends Actor{
   for {
      o <- buffer tyTell remove()
      o <- Buffer.afterRemove(o) tyTell remove()
      o <- Buffer.afterRemove(o) tyTell remove()
      o <- Buffer.afterRemove(o) tyTell remove()
   } yield println("End Consumer")

  def receive = PartialFunction.empty
}
```

```
val s = ActorSystem()
val bufferUntyped = s.actorOf(Props(new Buffer[Int]),”buffer”)
val buffer = new ProtocolRef[BufferInterf](bufferUntyped,Buffer.protocol)
val producer1 = s.actorOf(Props(new Producer(buffer,1)))
val consumer1 = s.actorOf(Props(new Consumer(buffer," pippo")))
val consumer2 = s.actorOf(Props(new Consumer(buffer," pluto")))
```

contravariance
of ProtRef[-T]

```
class Producer(buffer:ProtRef[ProduceInt]) extends Actor {
   for {
      o <- buffer tyT
      o <- Buffer.aft
      o <- Buffer.aft
      o <- Buffer.aft
      o <- Buffer.aft
   } yield println("Er
  def receive = Partia
}
```

```
val s = ActorSystem(
val bufferUntyped =
val buffer = new Pro
val producer1 = s.ac
val consumer1 = s.ac
val consumer2 = s.ac
```

```
produce 1
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 1
removed 1
produce 11
 pippo consumed 1
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 11
produce 21
removed 11
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 21
removed 21
 pippo consumed 11
 pluto consumed 21
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
produce 31
inserted 31
produce 41
removed 31
 pippo consumed 31
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 41
removed 41
End Producer n.1
 pippo consumed 41
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
```

That's it !

let's recap
the *programming pattern*
on a bigger example

Bookshop Server

Conclusions

TSOP actors:

- they **dynamically change both behaviour and interface**;
  "*no" defensive programming*
- it scales to concurrent accesses by mixing-in Chemical semantics
- the user code can take advantage of the monad to keep a clean logic and type safety
- a lot of **boilerplate code** can be generated from the protocol declaration

clean logic
natural def.

Protocol compliance:

- protocol as a type(d expression) (from sequence diagrams, FMS,…)
- the compiler checks that **stateful actors** will only handle **intended messages at the intended states**
- compliance with ***protocol obligations*** (intended msgs are eventually sent) ***requires linear typing***, which is not supported by Scala type system.

only
intended msgs at
the intended states

Conclusions

TSOP actors:

- they **dynamically change both behaviour and interface**;
  "*no" defensive programming*
- it scales to concurrent accesses by mixing-in Chemical semantics

- **Do you see a *killer application* for this programming style?**
- **What kind of *properties* would you like to be checked by the compiler in this scenario?**

Protocol compliance:

- protocol as a type(d expression) (from sequence diagrams, FMS,…)
- the compiler checks that **stateful actors** will only handle **intended messages at the intended states**
- compliance with *protocol obligations* (intended msgs are eventually sent) *requires linear typing*, which is not supported by Scala type system.

References

Formal foundation

OOPSLA'15

The Chemical Approach to Typestate-Oriented Programming

Silvia Crafa
Università di Padova, Italy

Luca Padovani
Università di Torino, Italy

- **sound** behavioural type system for the Join calculus
- currently working at obtaining the same result in the Actor model,

full version of this presentation

On the chemistry of typestate-oriented actors

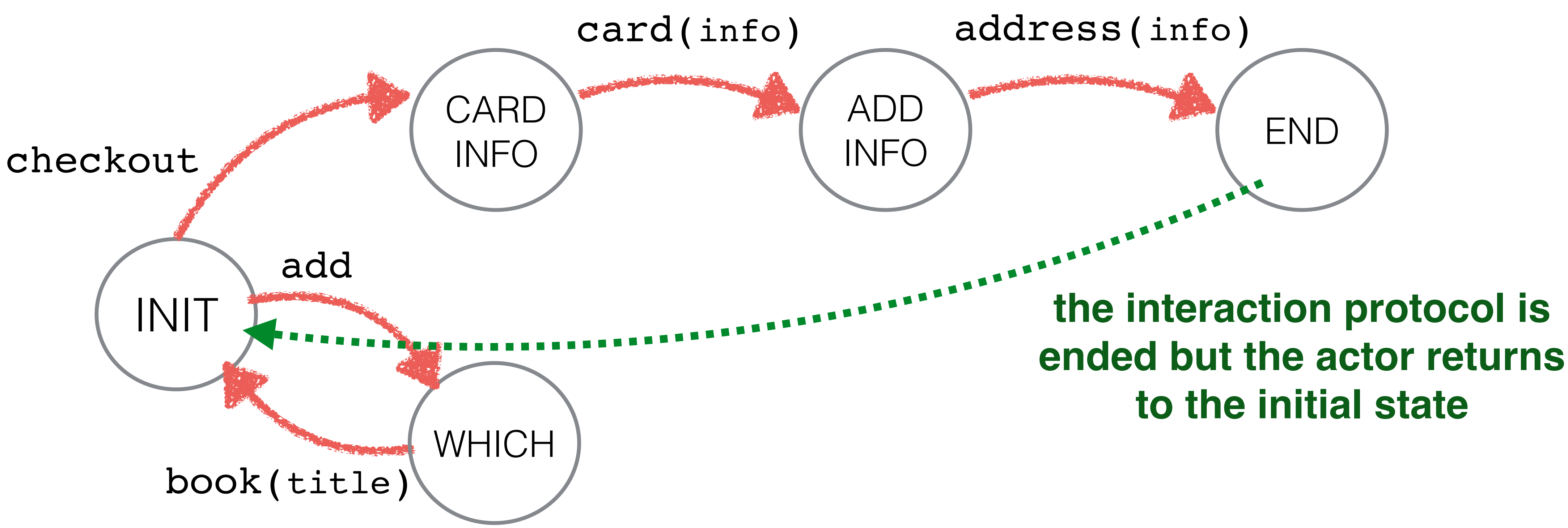
Silvia Crafa
Università di Padova, Italy

Luca Padovani
Università di Torino, Italy

Techical Report **http://www.math.unipd.it/~crafa/Pubblicazioni/**

Bookshop Server (Gay & Vasconcelos JFP 2010)

Protocol: a user adds a number of books to the basket and finally checks-out



rephrased as Chemical-TSOAP:

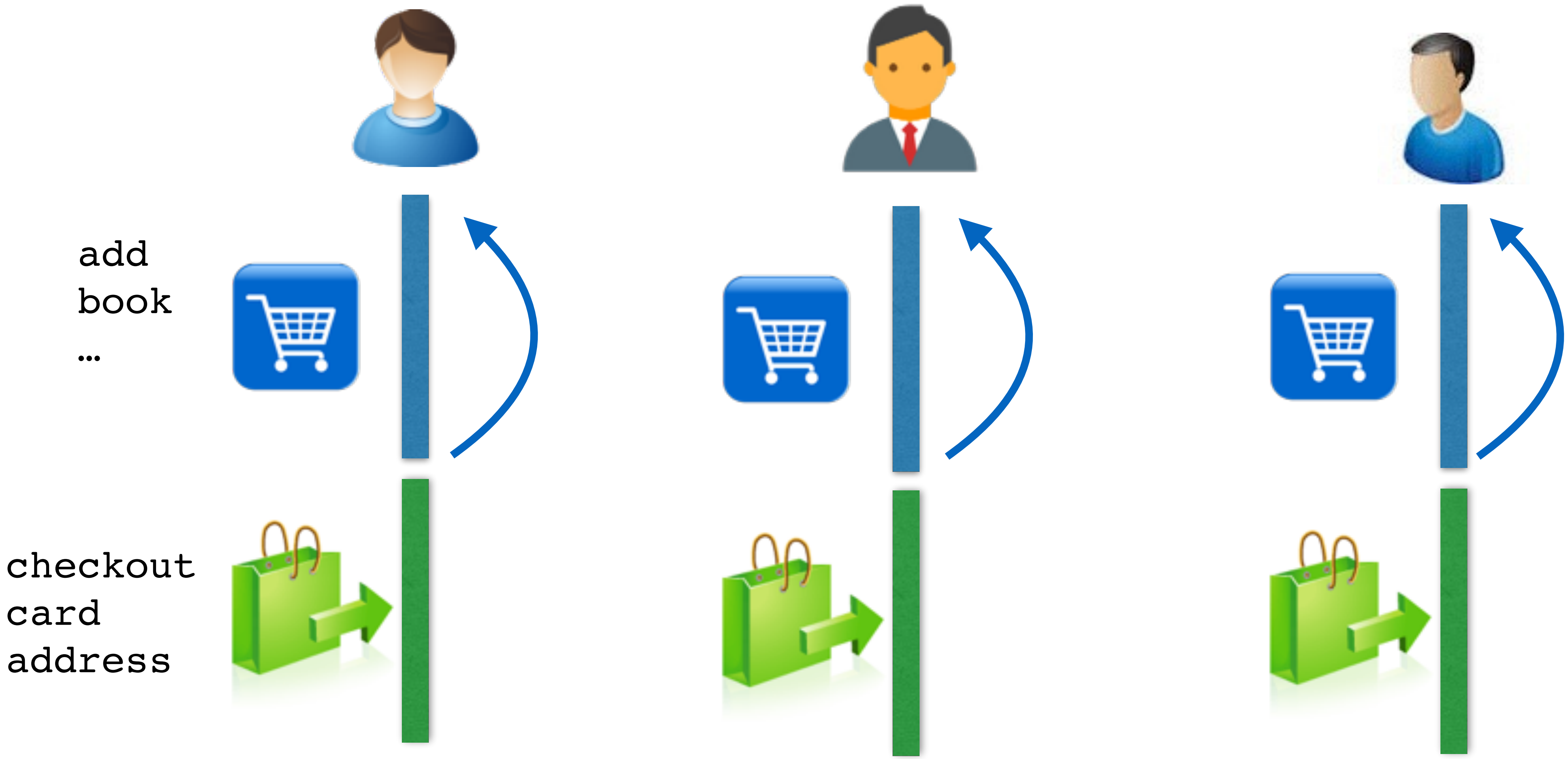
```
INIT    | add      --> WHICH
WHICH   | book     --> INIT
INIT    | checkout --> CINFO
CINFO   | card     --> ADDINFO
ADDINFO | address  --> END
```

a user is fully served before serving another one

(but in the meanwhile incoming requests are collected)

let increase the throughput by

interleave the *shopping* phases and the *checkout* phases



protocol

declare it!

```
trait ShopInterface
trait InitInterf extends ShopInterface
trait WhichInterf extends ShopInterface
trait CInfoInterf extends ShopInterface
trait AddInfoInterf extends ShopInterface
trait EndInterf extends ShopInterface


case class add(usrName:String) extends InitInterf with ProtocolMsg
case class checkout(usrName:String) extends InitInterf
case class book(usrName:String,title:String) extends Which...rf
case class card(usrName:String,cardNum:String) extends CI
case class address(usrName:String,add:String) ext
```

we want the Shop to handle **multiple concurrent users**: `add` msg are ***accepted at any time and wait (non-blocking) to be served at the right time***, i.e. when the Shop is back at `INIT` state

```
object Shop {
 def protocol :ShopInterface => Continuation =
   case add(n) => new Continuation {
          type T = WhichInterf
          val p=Promise[ProtRef[WhichInterf]]
           val f=p.future }
   case checkout(n) => new Continuation {
          type T = CInfoInterf
          val p=Promise[ProtRef[CInfoInterf]]
           val f=p.future }
   case book(n,b) => new Continuation {
         type T = InitInterf
         val p=Promise[ProtRef[InitInterf]]
         val f=p.future }
   case card(n,cn) => new Continuation {
         type T = AddInfoInterf
          val p=Promise[ProtRef[AddInfoInterf]]
         val f=p.future }
   case address(n,add) => new Continuation {
            type T = EndInterf
            val p=Promise[ProtRef[EndInterf]]
            val f=p.future }
  }
}
```

protocol

declare it!

```
trait ShopInterface
trait InitInterf extends ShopInterface with ProtocolMsg
trait WhichInterf extends ShopInterface
trait CInfoInterf extends ShopInterface
trait AddInfoInterf extends ShopInterface
trait EndInterf extends ShopInterface
```

keeps both **add** and **checkout** msgs, i.e. those that "start a phase"

```
case class add(usrName:String) extends InitInterf with ProtocolMsg
case class checkout(usrName:String) extends InitInterf
case class book(usrName:String,title:String) extends WhichInterf
case class card(usrName:String,cardNum:String) extends CInfoInterf
case class address(usrName:String,add:String) extends AddInfoInterf
```

```
object Shop {
 def protocol :ShopInterface => Continuation = {
   case add(n) => new Continuation {
          type T = WhichInterf
          val p=Promise[ProtRef[WhichInterf]]
           val f=p.future }
   case checkout(n) => new Continuation {
          type T = CInfoInterf
          val p=Promise[ProtRef[CInfoInterf]]
           val f=p.future }
   case book(n,b) => new Continuation {
         type T = InitInterf
         val p=Promise[ProtRef[InitInterf]]
         val f=p.future }
   case card(n,cn) => new Continuation {
        type T = AddInfoInterf
         val p=Promise[ProtRef[AddInfoInterf]]
        val f=p.future }
   case address(n,add) => new Continuation {
          type T = EndInterf
          val p=Promise[ProtRef[EndInterf]]
          val f=p.future }
 }
}
```

keeps the add messages arriving in the meanwhile

rephrased as Chemical-TSOAP:

```
INIT    | add      --> WHICH
WHICH   | book     --> INIT
INIT    | checkout --> CINFO
CINFO   | card     --> ADDINFO
ADDINFO | address  --> END
```

```
class Shop extends Actor with Chemical {

  private val shopBasket:Map[String,String] = Map[String,String]()

  def INIT :Receive = chemReact {
     case (add(n),p:Promise[ProtRef[WhichInterf]]) =>
                              println(n+" please chose a book")
                              p success ProtRef[WhichInterf](self,Shop.protocol)
                              context.become(WHICH)

     case (checkout(n),p:Promise[ProtRef[CInfoInterf]]) =>
                              println("start payment process for "+n)
                              p success ProtRef[CInfoInterf](self,Shop.protocol)
                              context.become(CINFO)
  }

  def WHICH : Receive = chemReact {
     case (book(n,b),p:Promise[ProtRef[InitInterf]]) =>
                              println(b+" put in shopping basket of "+n)
                              if(shopBasket.contains(n))
                                      shopBasket(n) += (" "+b)
                              else  shopBasket += (n->b)
                              p success  ProtRef[InitInterf](self,Shop.protocol)
                              context.become(INIT)
  }
  ......
```

to ensure that the current user completes the shopping without interleaving other users
or use **chemBecome**(INIT)
to allow more interleaving

rephrased as Chemical-TSOAP:

```
INIT    | add      --> WHICH
WHICH   | book     --> INIT
INIT    | checkout --> CINFO
CINFO   | card     --> ADDINFO
ADDINFO | address  --> END
```

```
class Shop extends Actor with Chemical {

  private val shopBasket:Map[String,String] = Map[String,String]()
  …
  def CINFO :Receive = chemReact {
    case (card(n,c),p:Promise[ProtRef[AddInfoInterf]]) =>
                              println("using card n."+c+"of user "+n)
                              p success ProtRef[AddInfoInterf](self,Shop.protocol)
                              context.become(ADDINFO)
  }

  def ADDINFO :Receive = chemReact {
    case (address(n,a),p:Promise[ProtRef[EndInterf]]) =>
                              println("shipping "+shopBasket(n)+" to "+n+" in "+a)
                              shopBasket.remove(n)
                              p success ProtRef[EndInterf](self,Shop.protocol)
                              chemBecome(INIT) //RECHECK SOUP!
  }

  def receive = INIT
}
```

re-install in the Shop's mailbox the add msg that arrived from other users while serving the last one

sends an END-continuation but it is ready for the next client

user code

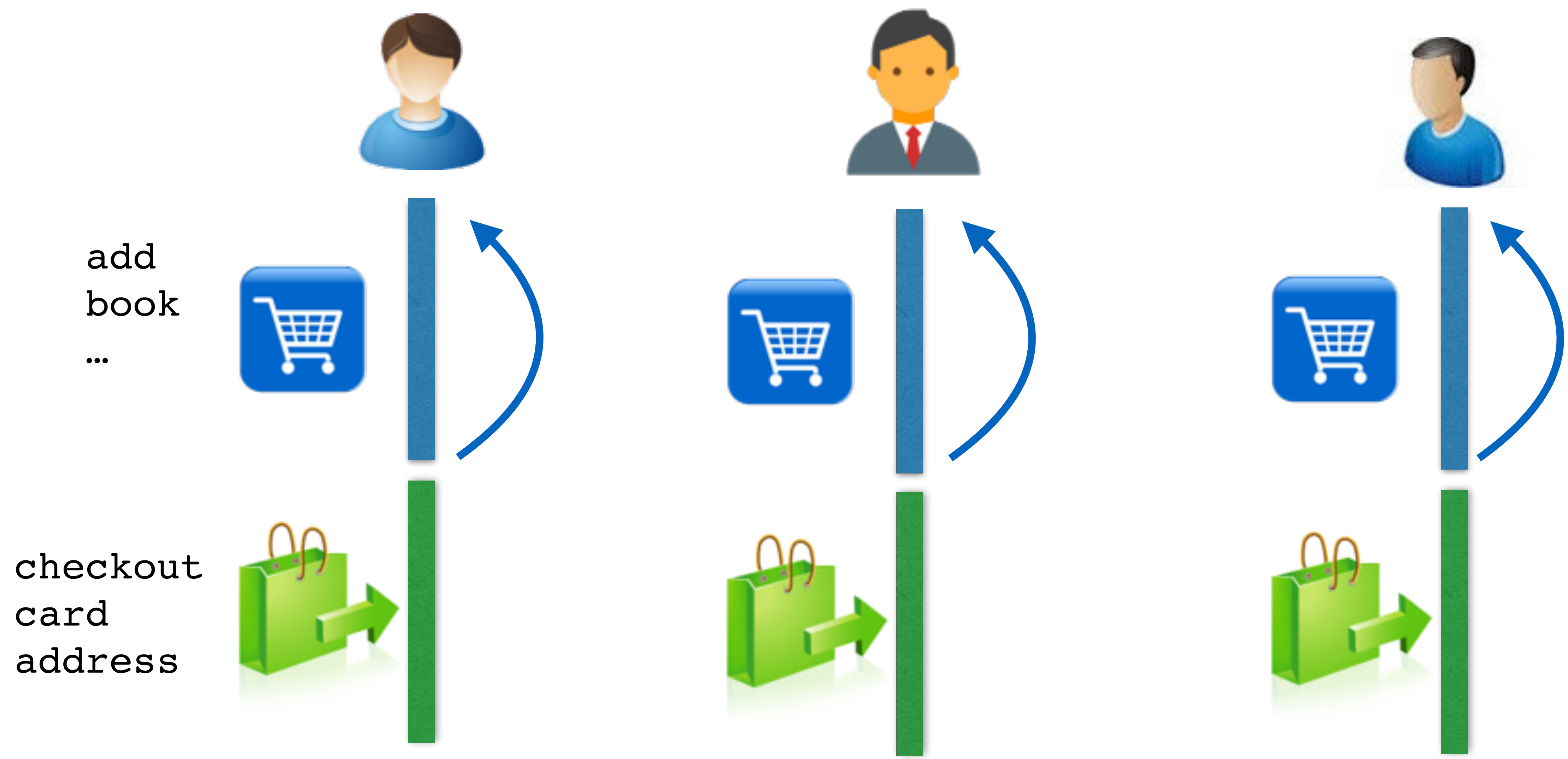
```
class User(shop:ProtRef[InitInterf],name:String,info:Map[String,String]) extends Actor {
   for {
    o <- shop tyTell add(name)
    o <- Shop.afterAdd(o)  tyTell  book(name,info("book1"))
    o <- Shop.afterBook(o) tyTell  add(name)
    o <- Shop.afterAdd(o)  tyTell  book(name,info("book2"))
    o <- Shop.afterBook(o) tyTell  checkout(name)
    o <- Shop.afterCo(o)   tyTell  card(name,info("cc"))
         // Shop.afterCo(o) tyTell address(…)  shipping without paying does not compile
    o <- Shop.afterCard(o) tyTell address(name,info("addr"))
   } yield println(name+" ended shopping")

  def receive=PartialFunction.empty
}


val s = ActorSystem()
val untypedShop = s.actorOf(Props(new Shop),"shop")
val shop = new ProtRef[InitInterf](untypedShop,Shop.protocol)
val user1 = s.actorOf(Props(new User(shop,"Mary", … )))
val user2 = s.actorOf(Props(new User(shop,"Jane", … )))
val user3 = s.actorOf(Props(new User(shop,"Alice", … )))

Thread.sleep(6000); s.shutdown()
```

user code

```
class User(shop:ProtRef[InitInterf]
   for {
    o <- shop tyTell add(name)
    o <- Shop.afterAdd(o)  tyTell
    o <- Shop.afterBook(o) tyTell
    o <- Shop.afterAdd(o)  tyTell
    o <- Shop.afterBook(o) tyTell
    o <- Shop.afterCo(o)   tyTell
         // Shop.afterCo(o) tyTell
    o <- Shop.afterCard(o) tyTell a
   } yield println(name+" ended sho

  def receive=PartialFunction.empty
}


val s = ActorSystem()
val untypedShop = s.actorOf(Props(n
val shop = new ProtRef[InitInterf](
val user1 = s.actorOf(Props(new Use
val user2 = s.actorOf(Props(new Use
val user3 = s.actorOf(Props(new Use

Thread.sleep(6000); s.shutdown()
```

```
Mary please chose a book
  keeping msg add(Jane)
  keeping msg add(Alice)
Pride and Prejudice put in shopping basket of Mary
Mary please chose a book
Odissea put in shopping basket of Mary
start payment process for Mary
using card n.1234of user Mary
shipping Pride and Prejudice Odissea to Mary in Padua
Mary ended shopping
Jane please chose a book
  keeping msg add(Alice)
Ben Hur put in shopping basket of Jane
Jane please chose a book
Pinocchio put in shopping basket of Jane
start payment process for Jane
using card n.1212of user Jane
shipping Ben Hur Pinocchio to Jane in Venice
Jane ended shopping
Alice please chose a book
Java8 put in shopping basket of Alice
Alice please chose a book
Scala put in shopping basket of Alice
start payment process for Alice
using card n.8888of user Alice
shipping Java8 Scala to Alice in NewYork
Alice ended shopping
```

let increase the interleaving between users



interleave the *shopping* phases and the *checkout* phases

protocol

declare it!

```
trait ShopInterface
trait InitInterf extends ShopInterface with ProtocolMsg
trait WhichInterf extends ShopInterface
trait CInfoInterf extends ShopInterface
trait AddInfoInterf extends ShopInterface
trait EndInterf extends ShopInterface
```

keeps both **add** and **checkout** msgs, i.e. those that “start a phase”

```
case class add(usrName:String) extends InitInterf with ProtocolMsg
case class checkout(usrName:String) extends InitInterf
case class book(usrName:String,title:String) extends WhichInterf
case class card(usrName:String,cardNum:String) extends CInfoInterf
case class address(usrName:String,add:String) extends AddInfoInterf
```

```
class Shop extends Actor with Chemical {
  def INIT :Receive = chemReact {
     case (add(n),p:Promise[ProtR ... => ...   context.become(WHICH)
     case (checkout(n),p:Promis ... ) => ...  context.become(CINFO)
  }
  def WHICH :Receive = chemReact {
     case (book(n,b),p:Promise[ProtRef[InitInt ...]]) => ... context.become(INIT)
                                                             chemBecome(INIT)
  }
  def CINFO :Receive = chemReact {
     case (card(n,c),p:Promise[ProtRef[AddInfoInterf]]) => ... context.become(ADDINFO)
  }
  def ADDINFO :Receive = chemReact {
     case (address(n,a),p:Promise[ProtRef[EndInterf]]) => ...chemBecome(INIT)
  }
```

re-checks the soup

code

```
class User                                    String]) extends Actor {
   for {
    o <- s
    o <- S
    o <- S
    o <- S
    o <- S
    o <- S
         /                                    paying does not compile
    o <- S
   } yield

  def rece
}


val s = Ac
val untype
val shop =
val user1 =
val user2 =
val user3 =

Thread.sle
```

```
Mary please chose a book
  keeping msg add(Jane)
  keeping msg add(Alice)
Pride and Prejudice put in shopping basket of Mary
Mary please chose a book
  keeping msg add(Alice)
  keeping msg add(Jane)
Odissea put in shopping basket of Mary
Alice please chose a book
  keeping msg add(Jane)
Java8 put in shopping basket of Alice
Alice please chose a book
  keeping msg add(Jane)
  keeping msg checkout(Mary)
Scala put in shopping basket of Alice
start payment process for Alice
  keeping msg add(Jane)
  keeping msg checkout(Mary)
using card n.8888of user Alice
shipping Java8 Scala to Alice in NewYork
Jane please chose a book
  keeping msg checkout(Mary)
Ben Hur put in shopping basket of Jane
Jane please chose a book
Alice ended shopping
  keeping msg checkout(Mary)
Pinocchio put in shopping basket of Jane
start payment process for Jane
```

```
  keeping msg checkout(Mary)
using card n.1212of user Jane
shipping ... to Jane in Venice
Jane ended shopping
start payment process for Mary
using card n.1234of user Mary
shipping ... to Mary in Padua
Mary ended shopping
```