On the chemistry of typestate-oriented actors

Silvia Crafa
Università di Padova, Italy

Formal methods ... in Action!

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?
What are suitable abstractions for Protocol-Oriented Programming?

- Shared memory, data-centric fits a centralised control, of distributed entities.
- 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. Harmonic bottom-up composition of local entities.

Easy to think, Easy to reason about

Expressiveness, Performance

Productively interoperate with the other language abstractions.
Overview

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

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

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

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**
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()

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
}
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!!
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
}

1-place buffer as TSOAP

Producer
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!

Consumer
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
```scala
def !(msg:Any)
```
which is *always well-typed*

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

*encapsulates an actor at state T*

*add a layer of typing!*

*statically type-checks that sent messages belong to the interface T*
1-place buffer ... *with typed reference*

```scala
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... }

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
}))

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

```scala
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
}))
```

Continuation-passing Style
1-place buffer *with explicit continuations*

```scala
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])
```

```scala
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
}
```

reply messages carry the continuation reference with suitable type
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])

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

reply messages carry the continuation reference with suitable type
1-place buffer with implicit 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 implicit 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.afterInsert(o) tyTell insert(4) compiler error
    o <- Buffer.afterInsert(o) tyTell remove()
  } yield print("END")
}

def receive= PartialFunction.empty
}))

(Typed) Monad
def tyTell(msg:T) :
Continuation = {
...
}

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

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

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.afterInsert(o) tyTell remove()
  }
} yield print("END")

  def receive= PartialFunction.empty })))
class `ProtRef[-T](r:ActorRef, protocol: T => Continuation)`

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

...boilerplate code produced by the protocol

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

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

```scala
val buffer = new ProtRef[ProduceInt](untypedBuffer, Buffer.protocol)
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.afterInsert(o) tyTell remove()
  } yield print("END")

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

1-place buffer as TSO

Producer

Consumer

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

insert msg can arrive in the buffer’s mailbox while the buffer is in state FULL!

different states with different interfaces

we still want to ensure that insert/remove are served in the correct state!
1-place buffer as TSO

The class `Buffer[T]` extends Actor with two receive cases:

1. **EMPTY**
   - `case insert(x:T) => context.become(FULL(x))`
2. **FULL**
   - `case remove() => context.become(EMPTY)`

The `receive` method sets the initial state to `EMPTY`.

Akka Typed

- each dynamic behaviour must handle the whole set of messages...
- different behaviours but the same interface

**Different states with different interfaces**

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

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

The correct state is entered just the messages eventually processed: ProtocolMsg.
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
}
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
}

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")))
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)
    } 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")))

discarding 1
producing 1
keeping msg remove(Actor[akka://default/user/$b#-1611031969])
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 1
removed 1
producing 2
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 21
removed 21
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 31
removed 31
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
inserted 41
removed 41
keeping msg remove(Actor[akka://default/user/$c#-1876841699])
End Producer n.1
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

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.
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

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,

On the chemistry of typestate-oriented actors

Silvia Crafa
Università di Padova, Italy

Luca Padovani
Università di Torino, Italy

Formal foundation

full version of this presentation

Technical Report

http://www.math.unipd.it/~crafa/Pubblicazioni/
**Protocol**: a user adds a number of books to the basket and finally checks-out

The interaction protocol is ended but the actor returns to the initial state

rephrased as Chemical-TSOAP:

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>add</td>
<td>WHICH</td>
</tr>
<tr>
<td>WHICH</td>
<td>book</td>
<td>INIT</td>
</tr>
<tr>
<td>INIT</td>
<td>checkout</td>
<td>CINFO</td>
</tr>
<tr>
<td>CINFO</td>
<td>card</td>
<td>ADDINFO</td>
</tr>
<tr>
<td>ADDINFO</td>
<td>address</td>
<td>END</td>
</tr>
</tbody>
</table>
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
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

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 }
  }
}
trait ShopInterface
trait IniInterf extends ShopInterface with ProtocolMsg
trait WhichInterf extends ShopInterface
trait CInfoInterf extends ShopInterface
trait AddInfoInterf extends ShopInterface
trait EndInterf extends ShopInterface

case class add(usrName:String) extends IniInterf with ProtocolMsg
case class checkout(usrName:String) extends IniInterf
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 = IniInterf
      val p=Promise[ProtRef[IniInterf]]
      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 }
  }
}
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)
    }

......

rephrased as Chemical-TSOA:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>add</td>
<td>--&gt; WHICH</td>
</tr>
<tr>
<td>WHICH</td>
<td>book</td>
<td>--&gt; INIT</td>
</tr>
<tr>
<td>INIT</td>
<td>checkout</td>
<td>--&gt; CINFO</td>
</tr>
<tr>
<td>CINFO</td>
<td>card</td>
<td>--&gt; ADDINFO</td>
</tr>
<tr>
<td>ADDINFO</td>
<td>address</td>
<td>--&gt; END</td>
</tr>
</tbody>
</table>

keeps the add messages arriving in the meanwhile

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

rephrased as Chemical-TSOAP:

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

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
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()
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(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", ...)), "user1")
val user2 = s.actorOf(Props(new User(shop,"Jane", ...)), "user2")
val user3 = s.actorOf(Props(new User(shop,"Alice", ...)), "user3")

Thread.sleep(6000); s.shutdown()
let increase the interleaving between users

interleave the *shopping* phases and the *checkout* phases
trait ShopInterface
trait InitInterf extends ShopInterface with ProtocolMsg
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 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[ProtRef[InitInterf]]) => ... context.become(WHICH)
    case (checkout(n),p:Promise[InitInterf]) => ... context.become(CINFO)
  }
  def WHICH :Receive = chemReact {
    case (book(n,b),p:Promise[ProtRef[InitInterf]]) => ... 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)
  }
}
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
}

def 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()

Mary please chose a book
  keeping msg add[Alice]
  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[Alice]
Odissea put in shopping basket of Mary
Alice please chose a book
  keeping msg add[Alice]
  keeping msg checkout[Mary]
Java8 put in shopping basket of Alice
Alice please chose a book
  keeping msg add[Alice]
  keeping msg checkout[Mary]
Scala put in shopping basket of Alice
start payment process for Alice
  keeping msg add[Alice]
  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
  keeping msg checkout[Mary]
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