Scala Join (version 0.3)
User's Manual

Jiansen HE

The Scala Join Library (version 0.3) improves the scala joins library [2] implemented by Haller and Cutsem. Main advantages of this library are: (i) providing uniform *and* operator, (ii) supporting pattern matching on messages, (iii) supporting theoretically unlimited numbers of join patterns in a single join definition (iv) using simpler structure for the **Join** class, and most importantly, (v) supporting communications on distributed join channels.

1 Using the Library

1.1 Sending messages via channels

An elementary operation in the join calculus is sending a message via a channel. A channel could be either asynchronous or synchronous. At the caller's side, sending a message via an asynchronous channel has no obvious effects in the sense that the program will always proceed. By contrast, when a message is sent via a synchronous channel, the current thread will be suspended until a result is returned.

To send a message m via channel c, users simply apply the message to the channel by calling $c(m)$. For the returned value of a synchronous channel, users may want to assign it to a variable so that it could be used later. For example,

```
val v = c(m) // c is a synchronous channel
```

1.2 Grouping join patterns

A join definition defines channels and join patterns. Users define a join definition by initializing the **Join** class or its subclass. It is often the case that a join definition should be globally *static*. If this is the case, it is a good programming practice in Scala to declare the join definition as a *singleton object* with the following idiom:

```
object join_definition_name extends Join{
  //channel declarations
  //join patterns declaration
}
```

A channel is a singleton object inside a join definition. It extends either the **AsyName**[**ARG**] class or the **SynName**[**ARG**, **R**] class, where **ARG** and **R** are generic type parameters. Here, **ARG** indicates the type of channel parameter whereas **R** indicates the type of return value of a synchronous channel. The current library only supports unary channels, which only take one parameter. Fortunately, this is sufficient for constructing nullary channels and channels that take more than one parameter. A nullary channel could be encoded as a channel whose argument is always **Unit** or any other constants. For a channel that takes more than one parameter, users could pack all arguments in a tuple.

Once a channel is defined, we can use it to define a join pattern in the following form

```
case <pattern> => <action>
```

The <*pattern*> at the left hand side of ⇒ is a set of channels and their formal arguments, connected by the infix operator *and*. The <*action*> at the right hand side of ⇒ is a sequence of Scala statements. Formal arguments declared in the <*pattern*> must be pairwise distinct and might be used in the <*action*> part. In addition, each join definition accepts one and only one group of join patterns as the argument of its *join* method. Lastly, like most implementations for the join calculus, the library does not permit multiple occurrences of the same channel in a single join pattern. On the other side, using the same channel in an arbitrary number of different patterns is allowed.

We conclude this section by presenting a sample code that defines and uses join patterns.

Listing 1: Example code for defining join patterns (join_test.scala)

```
import join._
import scala.concurrent.ops._ // spawn
object join_test extends App{// for scala 2.9.0 or later
  object myFirstJoin extends Join{
    object echo extends AsyName[String]
    object sq extends SynName[Int, Int]
    object put extends AsyName[Int]
    object get extends SynName[Unit, Int]

    join{
      case echo("Hello") => println("Hi")
      case echo(str) => println(str)
      case sq(x) => sq reply x*x
      case put(x) and get(_) => get reply x
   }
  }

  spawn {
    val sq3 = myFirstJoin.sq(3)
    println("square(3) = "+sq3)
  }
  spawn { println("get: "+myFirstJoin.get()) }
  spawn { myFirstJoin.echo("Hello"), myFirstJoin.echo("Hello World") }
  spawn { myFirstJoin.put(8) }
}
```

One possible result of running the above code is:

```
>scalac join_test.scala
>scala join_test
square(3) = 9
Hi
Hello World
get: 8
```

1.3 Distributed computation

With the distributed join library, it is easy to construct distributed systems on the top of a local system. This section explains additional constructors in the distributed join library by looking into the code of a simple client-server system, which calculates the square of an integer on request. The server side code is given at Listing 2 and the client side code is given at Listing 3.

Listing 2: Server.scala

```
import join._
object Server extends App{
  val port = 9000
  object join extends DisJoin(port, 'JoinServer){
    object sq extends SynName[Int, Int]
    join{ case sq(x) => println("x:"+x); sq reply x*x }
    registerChannel("square", sq)
  }
  join.start()
}
```

Listing 3: Client.scala

```
object Client{
  def main(args: Array[String]) {
    val server = DisJoin.connect("myServer", 9000, 'JoinServer)
    //val c = new DisSynName[Int, String]("square", server)
    //java.lang.Error: Distributed channel initial error:
    //        Channel square does not have type Int => java.lang.String ...
    val c = new DisSynName[Int, Int]("square", server)//pass
    val x = args(0).toInt
    val sqr = c(x)
    println("sqr("+x+") = "+sqr)
    exit()
}}
```

```
> scala ServerTest
Server 'JoinServer Started...
                                                    >scala Client 5
x:5
                                                    sqr(5) = 25
                                                    >scala Client 7
x:7
                                                    sqr(7) = 49
```

In Server.scala, we constructed a distributed join definition by extending class **DisJoin**(**Int**, **Symbol**). The integer is the port where the join definition will listen and the symbol is used to identify the join definition. The way to declare channels and join patterns in **DisJoin** is the same as the way in **Join**. In addition, channels which might be used at remote site are registered with a memorizable string. At last, different from initializing a local join definition, a distributed join definition has to be explicitly started.

In Client.scala, we connect to the server by calling DisJoin.connect. The first and second arguments are the hostname and port number where the remote join definition is located. The last argument is the name of the distributed join definition. The hostname is a String which is used for the local name server to resolve the IP address of a remote site. The port number and the name of join definition should be exactly the same as the specification of the distributed join definition.

Once the distributed join definition, server, is successfully connected, distributed channels could be initialized as instances of DisAsyName[ARG](channel_name, server) or DisSynName[ARG, R](channel_name, server). Using an unregistered channel name or declaring a distributed channel whose type is inconsistent with its referring local channel will raise a run-time exception during the channel initialization. In later parts of the program, the client is free to use distributed channels to communicate with the remote server. The way to invoke distributed channels and local channels are the same.

2 Implementation Details

2.1 Case Statement, Extractor Objects and Pattern Matching in Scala

In Scala, a partial function is a function with an additional method: $isDefinedAt$, which will return $true$ if the argument is in the domain of this partial function, or $false$ otherwise. The easiest way to define a partial function is using the case statement. For example,

```
scala> val myPF : PartialFunction[Int,String] = {
     |  case 1 => "myPF apply 1"
     | }
myPF: PartialFunction[Int,String] = <function1>

scala> myPF.isDefinedAt(1)
res1: Boolean = true

scala> myPF.isDefinedAt(2)
res2: Boolean = false

scala> myPF(1)
res3: String = myPF apply 1

scala> myPF(2)
scala.MatchError: 2
       at $anonfun$1.apply(<console>:5)
       ...
```

In addition to basic values and case classes, the value used between *case* and ⇒ could also be an instance of an *extractor object*: object that contains an *unapply* method[1]. For example,

```
scala> object Even {
     |   def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
     | }
defined module Even

scala>  42 match { case Even(n) => Console.println(n) } // prints 21
21

scala>  41 match { case Even(n) => Console.println(n) } // prints 21
scala.MatchError: 41
       ...
```

In the above example, when a value, say x, attempts to match against a pattern, $Even(n)$, the method $Even.unapply(x)$ is invoked. If $Even.unapply(x)$ returns $Some(v)$, then the formal argument n will be assigned with the value v and statements at the right hand side of ⇒ will be executed. By contrast, if $Even.unapply(x)$ returns $None$, then the current case statement is considered not matching the input value, and the pattern examination will move towards the next case statement. If the last case statement still does not match the input value, then the whole partial function is not defined for the input. Applying a value outside the domain of a partial function will rise a *MatchError*.

2.2 Implementing local channels

Both asynchronous channel and synchronous channel are subclasses of trait *NameBase*. The reason why we introduced this implementation free trait is that, although using generic types to restrict the type of messages pending on a specific channel is important for type safety, a uniform view for asynchronous and synchronous channels simplifies the implementation at many places. For example, the three methods listed in Listing 4 are common between those two kinds of channels and are important for the implementation of Join and DisJoin class.

Listing 4: The NameBase trait

```
trait NameBase{ // Super Class of AsyName and SynName
  def argTypeEqual(t:Any):Boolean
  def pendArg(arg:Any):Unit
  def popArg():Unit
}
```

Listing 5: Code defines local asynchronous channel

```
class AsyName[Arg](implicit owner: Join, argT:ClassManifest[Arg]) extends NameBase{
  var argQ = new Queue[Arg] //queue of arguments pending on this name

  override def pendArg(arg:Any):Unit = {
    argQ += arg.asInstanceOf[Arg]
  }

  def apply(a:Arg) :Unit = synchronized {
    if(argQ.contains(a)){ argQ += a }
    else{
      owner.trymatch(this, a) // see if the new message will trigger any pattern
    }
  //other code
}
```

Asynchronous channel is implemented as Listing 5. The implicit argument *owner* is the join definition where the channel is defined. The other implicit argument, **argT**, is the descriptor for the run time type of **Arg**. Although **argT** is a duplicate information for **Arg**, it is important for distributed channels, whose erased type parameter might be declared differently between different sites. We postpone this problem until §2.4.

As shown in the above code, an asynchronous channel contains an argument queue whose element must have generic type **Arg**. Sending a message via a channel is achieved by calling its *apply* method, so that $c(m)$ could be written instead of c.apply(m) for short in Scala. Based on the linear assumption that no channel should appear more than once in a join pattern, reduction is possible only when a new message value is pending on a channel. Therefore, if the new message has the same value as another pended message, it should be attached to the end of the message queue; Otherwise, the join definition will be notified to perform a pattern checking and fire a possible pattern, if there is one.

As listing 6 shows, in addition to firing a pattern or pending a message to the message queue, an invocation on synchronous channel also needs to return a result value to the message sender. Since many senders may be waiting for a return value at the same time, for each reply invocation, the library need to work out which message the result is replied for. To this end, messages with the same value is tagged with different integers. The library uses msgTags to store the message that matches current fireable pattern. When a reply method is called, the channel inserts a integer-message pair and its corresponding reply value to the result queue and notifies all fetch threads that are waiting for a reply. With the help of the synchronized method, only one thread could attempt to fetch the reply value at a time.

Listing 6: Code defines local synchronous channel

```
class SynName[Arg, R](implicit owner: Join, argT:ClassManifest[Arg], resT:ClassManifest[R])
 extends NameBase{
  var argQ = new Queue[(Int,Arg)] // argument queue
  var msgTags = new Stack[(Int,Arg)] // matched messages
  var resultQ = new Queue[((Int,Arg), R)] // results

  private object TagMsg{
    val map = new scala.collection.mutable.HashMap[Arg, Int]
    def newtag(msg:Arg):(Int,Arg) = {
      map.get(msg) match {
        case None =>
          map.update(msg,0)
          (0,msg)
        case Some(t) =>
          map.update(msg, t+1)
          (t+1, msg)
   }}}

  def pushMsgTag(arg:Any) = synchronized {
    msgTags.push(arg.asInstanceOf[(Int,Arg)])
  }

  def popMsgTag:(Int,Arg) = synchronized {
    if(msgTags.isEmpty) { wait(); popMsgTag }
    else{ msgTags.pop }
  }

  def apply(a:Arg) :R = {
    val m = TagMsg.newtag(a)
    argQ.find(msg => msg._2 == m._2) match{
      case None => owner.trymatch(this, m)
      case Some(_) => argQ += m
    }
    fetch(m)
  }

  def reply(r:R):Unit = spawn {synchronized {
    resultQ.enqueue((msgTags.pop, r))
    notifyAll()
  }}

  private def fetch(a:(Int,Arg)):R = synchronized {
    if (resultQ.isEmpty || resultQ.front._1 != a){
      wait(); fetch(a)
    }else{ resultQ.dequeue()._2 }
  }
  //other code
}
```

2.3 Implementing the join pattern using extractor objects

2.3.1 The unapply method for local synchronous channel

In this library, join patterns are represented as a partial function. To support join patterns and pattern matching on message values, the library provides the unapply method for local channels. The unapply method for synchronous channel is given in Listing 8. The unapply method for

asynchronous channel is almost the same as the synchronous version, except that it does not need to deal with message tags.

Listing 7 gives the core of the unapply method of synchronous channel. The five parameters sent to the unapply method are:

(i) *nameset*: channels that could trigger the first fireable pattern.

(ii) *pattern*: the join pattern itself.

(iii) $fixedMsg$: a map from channels to corresponding message values. If the current channel is a key of the map, the unapply method returns its mapped value.

(iv) *dp*: an integer indicates the depth of pattern matching. The *dp* is useful for optimizations and debugging.

(v) $bandedName$: a banded channel name. If the current channel is the same as the $bandedName$, the unapply method returns None.

When a channel is asked to select a message that could trigger a *pattern*, it first check rule (iii) and (v). If neither rule applies, the channel returns the first message that matches the pattern and adds this channel to the *nameset*, if such a message exists. We consider a message of the current channel triggers a join pattern if the join pattern cannot be fired without the presence of messages on the current channel and will be fired when that message is bound to the current channel.

Listing 7: Core of the unapply method of local synchronous channel

```
case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
          fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase) => {
        //other code
        if (this == banedName) {return None}
        if(fixedMsg.contains(this)){
          Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2)
        }else{

          def matched(m:(Int,Arg)):Boolean = {
          //    pattern cannot be fired without the presence of message on current channel
          //and pattern can be fired when m is bound to the current channel
              (!(pattern.isDefinedAt(nameset, pattern, fixedMsg+((this, m)), dp+1, this))
           && (pattern.isDefinedAt((nameset, pattern, fixedMsg+((this, m)), dp+1, banedName))))
          }

        var returnV:Option[Arg] = None

        argQ.span(m => !matched(m)) match {
           case (_, MutableList()) => { // no message pending on this channel may trigger the pattern
             returnV = None
           }
           case (ums, ms) => {
             val arg = ms.head // the message could trigger a pattern
             nameset.add(this)
             if(dp == 1) {pushMsgTag(arg)}
             returnV = Some(arg)
           }
        }
        returnV
      }
}
```

The above code implements the core algorithm and could be improved for better efficiency. Firstly, if the value of a message has been proved not to trigger the join pattern, the *matched* method invoked by the span iteration does not need to run complex test for that value. To this end, a HashSet $checkedMsg$ could be introduced to record checked message values. The set should be cleared after the span iteration. Secondly, when a message is selected, popping it to the head of the message queue will save the later work of removing that message from the queue. Lastly, each channel that triggers a pattern only needs to be added to the *nameset* once. Although inserting an element to a hashset is relatively cheap, the cost could be further reduced to the cost of comparing two integers. The full implementation for the unapply methods of synchronous channel is given at Listing 8. The first case statement is used for improving the efficiency of code involving singleton pattern, where a pattern only contains one channel. More explanation for this decision will be given in §2.3.2.

Listing 8: The unapply method of local synchronous channel

```
 def unapply(attr:Any) : Option[Arg]= attr match {
   case (ch:NameBase, arg:Any) => {// For singleton patterns
     if(ch == this){ Some(arg.asInstanceOf[(Int,Arg)]._2))
     }else{ None }
   }
   case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
         fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase) => {
     if (this == banedName) {return None}
     if(fixedMsg.contains(this)){ Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2))
     }else{
       var checkedMsg = new HashSet[Arg]

       def matched(m:(Int,Arg)):Boolean = {
         if (checkedMsg(m._2)) {false} // the message has been checked
         else {
           checkedMsg += m._2
           (!(pattern.isDefinedAt(nameset, pattern, fixedMsg+((this, m)), dp+1, this))
           && (pattern.isDefinedAt((nameset, pattern, fixedMsg+((this, m)), dp+1, banedName))))
         }
       }

       var returnV:Option[Arg] = None
       argQ.span(m => !matched(m)) match {
          case (_, MutableList()) => { returnV = None }
          case (ums, ms) => {
            val arg = ms.head // the message could trigger a pattern
            argQ = (((ums.+=:( arg )) ++ ms.tail).toQueue) // pop this message to the head of message queue
            if(dp == 1) {nameset.add(this); pushMsgTag(arg)}
            returnV = Some(arg._2)
          }
       }
       checkedMsg.clear
       returnV
     }
   }
}
```

2.3.2 The *Join* class and the *and* object

As said in the earlier section, join patterns are represented as a partial function in this library. An instance of the Join class is responsible for storing the join definition and attempting to fire a pattern on request. If the requested channel message association could fire a pattern, all channels

involved in that pattern will be asked to remove the matched message; otherwise, the channel will be notified to pend the message to its message queue.

Although this library encourages using join patterns as a convenience constructor to synchronizing resources, actor model is popular at the time of implementing this library and not all channels need to be synchronized with others. For this reason, this library gives singleton patterns the privilege on pattern examination. Readers may wonder to what extent the efficiency will be affected by the above decision. To answer this question, consider a typical join definition where p patterns are defined and there are m channels on each pattern. At the time of a new message's arrival, there are n messages pending on each channel on average. On average, this library needs $O(p)$ time to check all patterns as if they are singleton patterns before spending $O(pmn)$ time checking all patterns as join patterns. Therefore, the additional checking will not significantly increase the cost of checking join patterns but will benefit programs that use singleton patterns.

Listing 9: Code defines the Join class

```
class Join {
  private var hasDefined = false
  implicit val joinsOwner = this
  private var joinPat: PartialFunction[Any, Any] = _

  def join(joinPat: PartialFunction[Any, Any]) {
   if(!hasDefined){
     this.joinPat = joinPat
     hasDefined = true
   }else{
     throw new Exception("Join definition has been set for"+this)
   }
  }

  def trymatch(ch:NameBase, arg:Any) = synchronized {
    var names: Set[NameBase] = new HashSet
    try{
      if(ch.isInstanceOf[SynName[Any, Any]]) {ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)}
      if(joinPat.isDefinedAt((ch, arg))){// optimization for singleton pattern
        joinPat((ch,arg))
      }else{
        if(ch.isInstanceOf[SynName[Any, Any]]){
          joinPat((names, this.joinPat, (new HashMap[NameBase, Any]+((ch, arg))), 1, new SynName))
          ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)
        }else{
          joinPat((names, this.joinPat, (new HashMap[NameBase, Any]+((ch, arg))), 1, new AsyName))
        }
        names.foreach(n => {
          if(n != ch) n.popArg
        })
      }
    }catch{
      case e:MatchError => {// no pattern is matched
        if(ch.isInstanceOf[SynName[Any, Any]]) {ch.asInstanceOf[SynName[Any,Any]].popMsgTag}
        ch.pendArg(arg)
      }
    }
  }
}
```

The last thing is to define an *and* constructor which combines two or more channels in a join pattern. Indeed, this is surprisingly simple to some extent. Thanks to the syntactic sugar provided by Scala, the infix *and* operator in this library is defined as a binary operator that passes the same

argument to both operands.

Listing 10: Code defines the and object

```
object and{
  def unapply(attr:Any) = {
      Some(attr,attr)
  }
}
```

2.4 Implementing distributed join calculus

The **DisJoin** class extends both the **Join** class, which supports join definitions, and the **Actor** trait, which enables distributed communication. In addition, the distributed join definition manages a name server which maps strings to its channels. Compared to a local join definition, a distributed join definition has two additional tasks: checking if distributed channels used at a remote sites are annotated with correct types and listening messages sending to distributed channels. The code of the **DisJoin** class is presented in Listing 11.

Listing 11: Code defines the DisJoin Class

```
class DisJoin(port:Int, name: Symbol) extends Join with Actor{
  var channelMap = new HashMap[String, NameBase] //work as name server

  def registerChannel(name:String, ch:NameBase){
    assert(!channelMap.contains(name), name+" has been registered.")
    channelMap += ((name, ch))
  }

  def act(){
    RemoteActor.classLoader = getClass().getClassLoader()
    alive(port)
    register(name, self)

    loop(
      react{
        case JoinMessage(name, arg:Any) => {
          if (channelMap.contains(name)) {
            channelMap(name) match {
              case n : SynName[Any, Any] => sender ! n(arg)
              case n : AsyName[Any] => n(arg)
         }}}

        case SynNameCheck(name, argT, resT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual((argT,resT)))
          }else{
            sender ! NameNotFound
          }
        }
        case AsyNameCheck(name, argT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual(argT))
          }else{
            sender ! NameNotFound
       }}}
    )
}}
```

In this library, a distributed channel is indeed a stub of a remote local channel. When a distributed channel is initialized, its signature is checked at the place where its referring local channel is defined. Later, when a message is sent through this distributed channel, the message and the channel name is forwarded to the remote join definition where the referring local channel is defined. Consistent with the semantic of distributed join calculus, reduction, if any, is performed at the location where the join pattern is defined. If the channel is a distributed synchronous channel, a reply value will be sent back to the remote caller. Listing 12 illustrates how distributed synchronous channel is implemented. Distributed asynchronous channel is implemented in a similar way.

Listing 12: Code defines distributed synchronous channel

```
class DisSynName[Arg:Manifest, R:Manifest](n:String, owner:scala.actors.AbstractActor){
  val argT = implicitly[ClassManifest[Arg]]//type of arguments
  val resT = implicitly[ClassManifest[R]]//type of return value

  initial()// type checking etc.

  def apply(arg:Arg) :R = synchronized {
    (owner !? JoinMessage(n, arg)).asInstanceOf[R]
  }

  //check type etc.
  def initial() = synchronized {
    (owner !? SynNameCheck(n, argT, resT)) match {
      case true => Unit
      case false => throw new Error("Distributed channel initial error:"+
                                    "Channel " + n + " does not have type "+
                                    argT+ " => "+resT+".")
      case NameNotFound => throw new Error("name "+n+" is not found at "+owner)
}}}
```

Lastly, the library also provides a function that simplifies the work of connection to a distributed join definition.

```
object DisJoin {
  def connect(addr:String, port:Int, name:Symbol):AbstractActor = {
    val peer = Node(addr, port)//location of the server
    RemoteActor.select(peer, name)
  }
}
```

3 Limitations and Future Improvements

3.1 Assumption on linear pattern

As with most of implementations that support join patterns, this library assumes that channels in each join pattern are pairwise distinct. Nevertheless, the current prototype implementation does not check the linear assumption for better simplicity.

Under the current implementation, a non-linear pattern

- will never be triggered if the channel involves a non-linear channel that takes two or more different messages. For example, the pattern $\{case\ c(1)\ and\ c(2) \Rightarrow println(3)\}$ will never fire.
- will work as a linear pattern if the all occurrences of a non-linear channel could take the same message. In this case, one or more variable names could be used to indicate the same

message value. For example, $\{case\ c(m)\ and\ c(n) \Rightarrow println(m+n)\}$ will print 4 when c(2) is called.

3.2 Limited number of patterns in a single join definition

Due to the limitation of the current Scala compiler (version 2.9.1), the library also has an upper limit for the number of patterns and the number of channels in each pattern. Although the pattern of this limitation is not clear, the writer observed that a "sorted" join-definition may support more patterns.

For example,

```
case A(1) and B(1) and C(1) => println(1)
case A(2) and B(2) and D(2) => println(2)
```

is a "better" join-definition than

```
case A(1) and B(1) and C(1) => println(1)
case D(2) and B(2) and A(2) => println(2)
```

The compiler error: "java.lang.OutOfMemoryError: Java heap space" usually indicates the above limitation.

3.3 Unnatural usages of synchronous channels

Users of the current library may define a join-definition as follows

```
// bad join definition
object myjoin extends Join{
 object A extends AsyName[Int]
 object S extends SynName[Int, Int]
 join {
   case A(1) => S reply 2
   case S(n) => println("Hello World")
 }
}
```

In addition to the linear assumption, the current library further assumes that:

(i) the action part only reply values to synchronous channels that appeared in the left hand side of $\Rightarrow$. For this reason, the first pattern in the above example is invalid.

(ii) all synchronous channels in a pattern, if any, will receive one and only one value when the pattern fires. For this reason, the second pattern in the above example is invalid.

For assumption (i), the writer assumes that a program only needs to send a reply value to a synchronous channel on request. For assumption (ii), we think that all invocations on synchronous channels are expecting a reply value. Unlike some other libraries such as Cω. this library permits multiple synchronous channels in a single pattern.

Violating any of the above assumptions may be accepted by the system but usually causes deadlock or unexpected behaviour at run time.

3.4 Straightforward implementation for synchronous channels

Readers may have noticed that the implementation for synchronous channels are implemented according to its straightforward meaning rather than its formal definition in the join-calculus, which translates synchronous channels to asynchronous channels.

Admittedly, the translation in the join-calculus is a clever strategy to mimic the straightforward meaning of synchronous channels with less constructs. As the Scala programming language provides low-level concurrency constructs, we think that a direct implementation for the synchronous channel is easier than and could be consistent with the indirect translation.

3.5 Type of the join pattern and the unapply methods

As an implementation in a static typed language, users would expect a clear type for the join pattern and the *unapply* methods in **AsyName**, **SynName**, and the *and* object. If the join pattern has type $\mathbf{T} \Rightarrow \mathbf{Unit}$, then the *unapply* methods in **AsyName** and **SynName** should have type $\mathbf{T} \Rightarrow \mathbf{Option}[\mathbf{Arg}]$, and the *unapply* method in the *and* object should have type $\mathbf{T} \Rightarrow \mathbf{Option}[(\mathbf{T}, \mathbf{T})]$. The unusual implementation that passes partial function to the unapply methods indicates that **T** is a recursive type. Furthermore, due to the optimization for singleton patterns, **T** is also a Either type.

For earnest readers, **T** is $\forall\mathbf{T}.\mathbf{Either}[(\mathbf{NameBase}, \mathbf{Any}), (\mathbf{HashSet}[\mathbf{NameBase}], \mathbf{PartialFunction}[\mathbf{T}, \mathbf{Unit}], \mathbf{HashMap}[\mathbf{NameBase}, \mathbf{Any}], \mathbf{Int}, \mathbf{NameBase})]$. Defining such a complex data type in a separate place may not be more helpful for readers than typing all parameters in each case statement. Moreover, general users do not need to understand this complex type to use this library. For above reasons, we simply replace **T** with the **Any** Type and manually verify type correctness of our implementation.

References

[1] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[2] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Rickard Nilsson. Scalacheck, 2011.

[4] Salmon Run. More java actor frameworks compared, jan 2009.