Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Modelle und Theorie Verteilter Systeme
Ernst-Reuter-Platz 7
10587 Berlin

Diplomarbeit

Joining Actors in La Scala

Simon Slama, Matrikelnummer: 301129

Berlin, den 11. Januar, 2015

Gutachter:	Prof. Dr. Uwe Nestmann
	Prof. Dr. Peter Pepper
Betreuer:	Christoph Wagner

Contents

List of Figures

Listings

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 11. Januar, 2015

____

Unterschrift

Danke den Geduldigen.

Zusammenfassung

Diese Arbeit beschreibt die Entwicklung der Sprache *JAiLS* und gibt einen Überblick über deren Möglichkeiten. Es werden die theoretischen Grundlagen des Join-Kalküls dargelegt und wie dieser in der Programmiersprache *Scala* als ein dem Kalkül in Syntax möglichst nahekommendes Programmier-Framework umgesetzt wurde. Dafür wurde im Speziellen das von *Scala* bereitgestellte Actor-Modell verwendet.

Neben der Beschreibung der technischen Umsetzung wird anhand von Beispielen die Entwicklung von Programmen mit *JAiLS* erläutert. Ein besonderes Augenmerk wird dabei auf die der diesem speziellen Prozesskalkül intrinsische Eigenschaft der Nebenläufigkeit gelegt, und wie diese mit Hilfe des Frameworks in Programmen umgesetzt werden kann. Ebenso werden andere Konzepte wie Nichtdeterminismus besprochen und umgesetzt.

Abstract

This thesis describes the development of the language *JAiLS* and overviews its potential. It discusses the theoretical foundations of the join-calculus and how it was implemented in the programming language *Scala*. The developed programming-framework *JAiLS* has a syntax that has been implemented as close to the one defined by the calculus as possible. It is based in particular on the actor model, provided by *Scala*.

Apart from the technical aspect and implementation, the thesis explains the usage of *JAiLS* following various examples. Special attention is put upon concurrency as an intrinsic property of this special process calculus. It shows how concepts like concurrency, non-determinism, replication and guarded processes are implemented in *JAiLS*.

1. Introduction

The join-calculus has been identified as a useful tool, in modelling and as a way of thinking in processes and the communication between them. Its simplicity and nonetheless its expressiveness are what accentuate its standing among the process-calculi. Despite its lack of terms to express restriction, reception and replication directly [3], it comes with a new tool that unifies those concepts in one single receptor definition [1].

Even though it defines such an elegant way to express, formalise and model data and processes, it lacks in a way to directly implement in this language. The existing implementations of the join-calculus are quite complicated in their usage and or implementation.

This thesis presents an implementation of the join-calculus, that is not only easy to use, but also characterized by a transparent implementation at the same time. This transparency is based upon the fact, that it is implemented in *Scala* and its way to express concurrency. The actor model provided by *Scala* makes it easy to encapsulate the actual semantic content into very abstract containers, and at the same time benefits of its reactivness This comes quite close to how the join-calculus handles communication between processes.

This thesis starts off with a theoretical overview of the join-calculus, explaining its theoretical foundations and how the discussed single receptor is used to express the foundations of a functional language. The second part of the theorectical foundation discusses the actor model.

The created framework *JAiLS* (*Joining Actors in La Scala*) is discussed in the third chapter. It shows how the elements of the language are used and explains, following the examples of the previous chapter, how the join-calculus is embedded into *Scala*.
When written in the join-calculus, there has to be a reaction semantic in order for the

processes of the program interact with each other. Since this implementation is done in a programming language it is interesting to watch how the reaction comes with the intrinsic property of the programming language to be executed.

The fourth chapter will demonstrates how *JAiLS* was developed. This draws a link to the second chapter, in that sense that it shows how the actor-model that is provided by *Scala* was used.

After concluding in the fifth chapter the thesis will point out in the last chapter how future works could extend the framework and where it could be improved.

2. Theoretical foundations

2.1. The join-calculus

Introduced by Fournet and Gonthier [1] as an extension of the generic chemical abstract machine (*CHAM*) by Berry and Boudol [2], the join-calculus is the set of molecules of the extended *reflexive CHAM*. Floating molecules in a solution represent processes that communicate through messages over defined links. It was introduced as the foundation of a programming language with functional and object-oriented features [1].

Definition 1 (J)**.** *The following grammar defines processes, definitions and join-patterns and therefore the process terms of the join-calculus denoted by $\mathcal{P}_J$. It is given by*

$$\begin{aligned} P &::= 0 \quad\Big|\quad y\langle x\rangle \quad\Big|\quad P_1 \mid P_2 \quad\Big|\quad \textbf{\textit{def}}\, D\, \textbf{\textit{in}}\, P \quad\Big|\quad \checkmark \\ D &::= J \rhd P \quad\Big|\quad D_1 \wedge D_2 \\ J &::= y(x) \quad\Big|\quad J_1 \mid J_1 \end{aligned}$$

for some names $x, y, z \in \mathcal{N}$.

A process P is either the compound of two subprocesses that are running parallel $P_1 \mid P_2$, the asynchronic sending of the message z over the link y or the definition of a receiver on fresh names **def** D **in** P. D itself is compound of several elementary definitions connected by $\wedge$, which means that they are all valid and accessible in the P and the triggered processes that are defined in the elementary definitions. The elementary definitions $J \rhd P$ are pieced together: One by a join-pattern J that itself can be composed of several reception patterns connected by $|$. The other part of the elementary definition is the triggered Process P which is a process that is able to access the received variables from the elementary definition it was triggered. It is also within the scope of the defined reception patterns from D.

An elementary definition $J \rhd P$ expresses the condition under which the Process P is triggered. Each of the receiving patterns $y_i(x_i)$ waits for a message on its defined link

y_i. If and only if each pattern of this elementary definition has received a message over its defined channel, the specified process is triggered.

The two constant processes 0 and ✓ are added in [3] as the empty process and a special process that states *successful termination* or just *success*.

2.1.1. variables

In regard to the implementation of the join-calculus, it is inevitable to consider how defined(dv), received(rv) and free(fv) variables behave and how their scoping works. This is recursivly defined in [1] by the following definitions:

$$\begin{aligned}
rv(y(x)) &\triangleq \{x\} \\
rv(J_1 \mid J_2) &\triangleq rv(J_1) \uplus rv(J_2) \\[1em]
dv(y(x)) &\triangleq \{y\} \\
dv(J_1 \mid J_2) &\triangleq dv(J_1) \cup dv(J_2) \\[1em]
dv(J \triangleright P) &\triangleq dv(J) \\
dv(D_1 \wedge D_2) &\triangleq dv(D_1) \cup dv(J_2) \\[1em]
fv(J \triangleright P) &\triangleq dv(J) \cup (fv(P) - rv(J)) \\
fv(D_1 \wedge D_2) &\triangleq fv(D_1) \cup fv(D_2) \\[1em]
fv(y(x)) &\triangleq \{x, y\} \\
fv(\mathbf{def}\ D\ \mathbf{in}\ P) &\triangleq (fv(D) \cup fv(P)) - dv(D) \\
fv(P_1 \mid P_2) &\triangleq fv(P_1) \cup fv(P_2)
\end{aligned}$$

Note. *This implies that two reception patterns cannot define the same name for the received variable.*

2.1.2. Some examples

As shown in [1] and stated in [3] the construct $\textbf{def}\, D\, \textbf{in}\, P$ unifies the concept of restriction, input capabilities and replication that were a native part of the pi-calculus [11]. In [1], Fournet and Gonthier give some examples that clarify this statement and will be used to show the possibilities of the developed language *JAiLS*. The examples are slightly extended so they are closer to the following implementation.

$$\textbf{def}\ x(u) \triangleright y\langle u\rangle\ \textbf{in}\ x\langle 1\rangle \tag{2.1}$$

$$\textbf{def}\ y(u) \triangleright x\langle u\rangle\ \textbf{in}\ \textbf{def}\ x(u) \triangleright y\langle u\rangle\ \textbf{in}\ x\langle 2\rangle \tag{2.2}$$

$$\textbf{def}\ x_1(u) \mid x_2(v) \triangleright x\langle u, v\rangle\ \textbf{in}\ x_1\langle 1\rangle \mid x_2\langle 7\rangle \tag{2.3}$$

$$\textbf{def}\ \text{ready}(\text{printer}) \mid \text{job}(\text{file}) \triangleright \text{printer}\langle\text{file}\rangle\ \textbf{in}\ \text{ready}\langle\text{laser}\rangle \mid \text{job}\langle\text{doc}\rangle \tag{2.4}$$

$$\textbf{def}\ s() \triangleright P \wedge s() \triangleright Q\ \textbf{in}\ s\langle\rangle \tag{2.5}$$

$$\textbf{def}\ \text{once}() \mid y(v) \triangleright x\langle v\rangle\ \textbf{in}\ y\langle 1\rangle \mid y\langle 2\rangle \mid y\langle 3\rangle \mid \text{once}\langle\rangle \tag{2.6}$$

$$\textbf{def}\ \text{loop}() \triangleright P \mid \text{loop}\langle\rangle\ \textbf{in}\ \text{loop}\langle\rangle \mid Q \tag{2.7}$$

2.1 This first example shows how the value 1 is wired from an inner process over the link x to an outside process that waits for a message on the link y.

2.2 Extends the first example with an outside process that waits for a message on link y and then sends it to a link x that is not the same one as the inner x.

2.3 Shows how multiplexing is implemented in the join-calculus as both values 1 and 7 are received via one distinct link each and are then sent to an outside process as a tuple $(1, 7)$

2.4 This is an example to show how a message can be also used as a link. After receiving *laser* via the link *printer* and the file *doc* via the link *job*, *laser* is used as a link and *doc* is sent to it.

2.5 The fiths example shows how non-determinism can be expressed in the join calculus. A message on the link s is caught by either one of the two occurences. Depending on which one consumed the message, P or Q is executed.

2.6 Another way of expressing non-determinism is to have four processes runnnig parallel to eachother, three of them sending a value via the link y. The fourth process just sends an empty message on the link *once*. As this link is used only once, the process from the elementary definition is triggered only once. The value that is received on the link y is randomly choosen between 1, 2 and 3.

2.7 This example shows how replication is expressed. A message on the link *loop* triggers a process P and at the same time sends a message on the same link again so P is triggered an indefinite amount of times.

As those examples show, the join-calculus as a process language has the power to express the invocation of functions, the use of parameters and replication and can therefore be used as the foundation of a functional language.

2.2. The actor model

In 1973 the actor model was first introduced by Carl Hewitt, Peter Bishop and Richard Steiger [5] as a model for the architecture of artificial intelligence that is conceptually based, as they state, on a single kind of objects: Actors. This conceptual singularity is not a restriction. Quite the contrary, an actor can be any form of data structure on the inside and it is not possible to determine the real implementation. It was defined as an active agent that was following a script. They stated that data structures, functions, semaphores and various other synchronization patterns are special cases of actors. They introduced the method of actor induction, which is a generalization of structural induction as it is defined over the meta object that actors are. Inspired by languages like *Simula* and *Smalltalk* [6] the actor model in its original form has characteristics of an object-oriented language. Privacy and protection fall out as an intrinsic property of actors as their use is only allowed to other actors that have the authority of using them.

Extended by Agha [8] in 1977, actors are defined as computational agents. Each incoming communication is mapped to a 3-tuple that is compounded of:

1. a finite set of communications sent to other actors
2. a new behavior
3. a finite set of new actors created.

With that in mind, it is stated that actors are a more powerful computational paradigm than sequential or functional programming languages. As said in the introduction, he states that it is possible to create an arbitrary sequential process by a suitable actor system but the opposite direction is not possible.

The interaction between agents can be classified in two groups:

1. Shared variables and
2. communication.

The first group would stand diametrically opposed to the closed idea of actors, so that the interaction between different agents is done exclusivly by communication. Communication and the reception of messages are bound together in an actor-system. Actors start acting only after receiving a message in their FIFO mailbox [7].

The extensiveness of the usage of actors differs from implementation to implementation. To give an example, in Erlang everything to the most basic objects as strings or integers are actors [10]. In comparison *Scala* every object *can* be defined as an actor. Since in *Scala* literally everything is an object, this leads to exactly the paradigm that was intended in [5] but does not imply the use of the actor model as a mandatory programming paradigm. It can be chosen where needed. Nevertheless, they form the fundamental instrument of concurrency in *Scala*[1].

Documenting the use of actors in *Scala* is not the target of this thesis but however needs to be taken into account as a side effect of the implementation of *JAiLS* explained in 4.

[1]As *Scala* is based upon *Java*-Bytecode and even *Java*-code can be used inside of *Scala* programms, this is not entirely true but holds for the native language tools.

3. *JAiLS* – a join-calculus language

The language *JAiLS* is the attempt to create an easy to use framework for the join-calculus on top of the *Scala* provided actor model. As *Scala* is an object-oriented language in its pure form [9] and leaves the developer almost all the liberty to define methods, it was possible to create a join-calculus programming framework in which the join-calculus aficionado can start developing without (almost) any knowlege of *Scala*. The first section will introduce *JAiLS* with an easy example that presents the basic classes. The following sections will explain how to use the framework, class by class, following some examples (in particular the ones presented in section 2.1.2). To increase the intelligibility, this chapter follows a bottom-up approach, in the same manner in which whole language is built up (cf. figure 3.1). The chapter concludes with a section about concurrency and how this is done in *JAiLS*.

This chapter contains two different varieties of code. The first one is simple code. It is encapsulated in a frame and the line numbers are shown.

```
var s: String = "This␣is␣a␣normal␣code␣snippet"
```

When something is defined, the second type of code would be the execution of an expression and the output on the console. The executed command follows a console symbol (#>).

```
#> println(s)
This is a normal code snippet
```

3.1. Introduction

The first implementation will be a simple counter given by the following join code

$$\textbf{def}\ count(x) \triangleright count\langle x+1\rangle\ \textbf{in}\ count\langle 1\rangle$$

Listing 3.1 is a simple implementation in *JAiLS*. To show the base classes of JAiLS it is written quite unhandy. This will be improved in the forthcoming examples.

```scala
import jails._

object UnhandyCounter {
  def main(args: Array[String]) {

    val receiver: JoinPattern     = new Receiver("count", "x")
    val triggeredProcess: Process = Add1
    val sender: Process           = new Sender("count", 1)
    val definition: Definition    = new ElementaryDefintion(receiver, triggeredProcess)
    val counter: Process          = new DefinitionInProcess(definition, sender)

    counter.start

  }
}

object Add1 extends MyProcess {
  def body {
    val input: Integer = rv("x").asInstanceOf[Integer]
    val outputLink: Receiver = dv("count")

    println(input)

    Thread.sleep(1000)

    outputLink ! input + 1
  }
}
```

Listing 3.1: Unhandy Counter

The outermost object found in this code is

```scala
val counter: Process = new DefinitionInProcess(definition, sender)
```

which corresponds to the *definition* in the join-calculus. As given in Definition 1, a *definition* process term itself is compound of two elements. In this example the first one is an elementary definition;

```scala
val definition: Definition = ElementaryDefinition(receiver, triggeredProcess
```

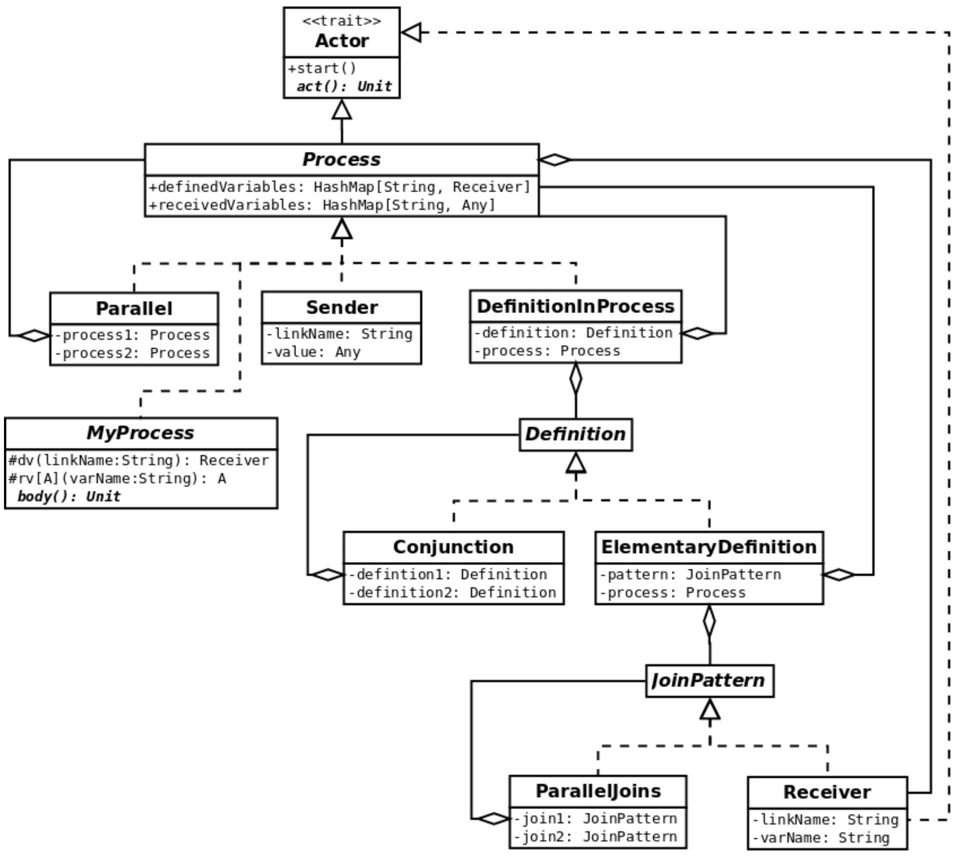


Figure 3.1.: Class diagram of *JAiLS*

```
)
```

whereby the initial process the definition is valid in:

```
val sender: Process = new Sender("count", 1)
```

The message sent by this sender is caught by the receiver defined in line 6:

```
val receiver: JoinPattern = new Receiver("count", "x")
```

Which at last triggers the Process `Add1`. This part requires some further explanation. The part $count\langle x+1\rangle$ has to be evaluated as the incrementation itself is not part of the language. To do so, an object `Add1` was created, which extends the abstract class `MyProcess`. This abstract class has a function `body: Unit` that needs to be implemented. This is where any *Scala*-code can be inserted. As the definition states, the process `Add1` can access both the received variable x and the defined link *count*. This is handled as follows; The abstract class `MyProcess` provides 2 functions. `rv(varName: String): Any` returns a receivedVariable for the given name and `dv( linkName: String): Receiver` whose result is a link for a given string. These two functions are used to get the received variable in line 19: `rv("x")`. To get the link the next value should be sent to in line 20: `dv("count")`. Finally, the value of $x + 1$ is sent to the link in line 26 using the *bang* operator `!` provided by the *Scala* actor library: `outputlink ! x + 1`.

3.2. JoinPattern

As previously stated, the language will be explained in a bottom-up manner, building up from the inner most class. The grammar of the join calculus defines the join pattern as follows (cf. Definition 1)

$$J ::= y(x) \quad \Big| \quad J_1 \mid J_1$$

.

A join pattern can either reflexivly consist of 2 parallel join patterns itself, or be defined as a reception pattern that receives a message over a specified link (see figure 3.2). To define such a join pattern in *JAiLS* the join pattern class is used.

```
sealed abstract class JoinPattern
```

Being an abstract class the proper classes to use are

```
class ParallelJoins(join1: JoinPattern, join2: JoinPattern)
  extends JoinPattern
```

which acts as a container for 2 subpattern and

```
class Receiver(linkName: String, varName: String)
  extends JoinPattern
```

to define a receiver that waits on a defined link for a message and stores it in a variable. The following code snippet shows how to define some receivers and then adding them together to a chain of join pattern.

```
1 val J1: JoinPattern = new Receiver("link1", "x")
2 val J2: JoinPattern = Receiver("link2", "y")
3 val J3: JoinPattern = ><("link3", "z")
4
5 val J4: JoinPattern = ParallelJoins(J1, J2)
6 val J5: JoinPattern = J4 | J3
```

Listing 3.2: Defining `JoinPattern`

The evaluation of `J5` looks like this

```
#> J5

res0: JAiLS.JoinPattern = ParallelJoins(ParallelJoins(Receiver(link1,x),
   Receiver(link2,y)),Receiver(link3,z))
```

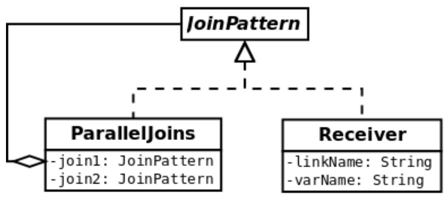


Figure 3.2.: Class diagram of `JoinPattern`

The first thing to take in to account is the fact that `J1` is defined with a **`new`** construct and `J2` is not. This is some syntactic sugar offered by *Scala* itself which comes handy in keeping the written code shorter. Even shorter is the definition of `J3`. This is defined as a syntactic imitation of the join-calculus' formalism. The direction of the chevrons symbolize the incoming message (cf. section 3.4.1). There are two ways to combine several join pattern as line 5 and 6 show. The first one uses the constructor of the class and the second one was added again to stay closer to the join-calculus.
To create the same join pattern one could have also written

```
#> ><("link1", "x") | ><("link2", "y") | ><("link3", "z")$\\

res1: JAiLS.JoinPattern = ParallelJoins(ParallelJoins(Receiver(link1,x),
   Receiver(link2,y)),Receiver(link3,z))
```

It is noted that the operator | between several join pattern is commutative and associative [1], as the combination of receivers are more of a syntactical nature than having any semantical significance. The definition of the join calculus and the inductive definition of the received and defined variables state that any two join pattern, which form one bigger join pattern together, cannot store the received message in the same variable which forbids an interaction between two join pattern on a shared memory object. The other form of interaction where two join pattern would affect each other is when two receivers wait on the same linkname for a message

$$link(x) \mid link(y)$$

.

This is not forbidden, but creates a nondeterministic situation having the reflexive chemical abstract machine in mind where floating molecules are consumed by each other. This would mean, that one of both receivers consumes the message before the other one. In the worst case, this could lead to a situation where all the messages are consumed by the same receiver and the combined join pattern is never satisfied. This pathological case can be observed creating such a program in *JAiLS* and will be discussed in section 3.5.3.

3.3. Definition

A definition aggregates join pattern and processes in a causal manner. The join pattern receives messages and triggers the process that is given in the definition.

$$J \triangleright P$$

As seen in the previous section, elementary definitions can be conjoined in a similar way as join pattern to more complex definitions. This is given by the following part of the join-calculus grammar.

$$D ::= J \triangleright P \quad \Big| \quad D_1 \wedge D_2$$

The use of the definition classes in *JAiLS* is akin to the one of join pattern (see figure 3.3, cf. figure 3.2).

A simple elementary definition in *JAiLS* is created by a join pattern and a process. This code snippet connects a simple receiver with the `Success` Process (the `Success` and `Empty` process are two trivial processes that are used, as they do not do much. They are later discussed in section 3.4.1).

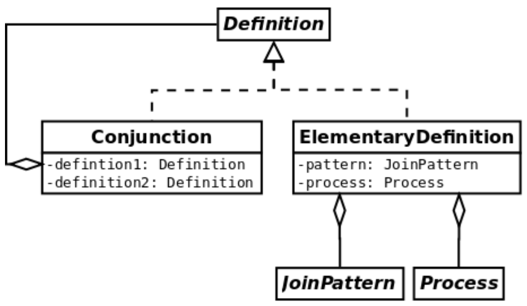


Figure 3.3.: Class diagram of `Definition`

```
val J: JoinPattern = ><("link3", "z")
val D: Definition  = Definition(J, Success)
```

And to stay in the mood of the join-calculus, the syntactic sugar is added to make te definition even shorter.

```
#> val D = J |> Success

D  : JAiLS.ElementaryDefinition = ElementaryDefinition(Receiver(link3,z),
   JAiLS.Success$@282ceded)
```

To conjoin several elementary definitions to one complex, such as this example

$$J_1 \triangleright P \wedge J_2 \triangleright Q \wedge J_3 \triangleright R$$

, there is in addition to the classes constructor;

`Conjunction(definition1: Definition, definition2: Definition)`

as well as the shorter version with the `&`-operator.

```
val D = (><("link1", "x") |> Success) &
        (><("link2", "x") |> Success) &
        (><("link3", "x") |> Empty)
```

Listing 3.3: Conjoined Definitions

Note. *Each elementary definition must be encapsulated in brackets for the scala interpretator to recognize the operator correctly.*

What happens after the process is triggered and how the variables can be accessed is discussed in section 3.4.2.

3.4. Process

As the join-calculus is part of the family of process calculi, processes are its spine. The same applies for *JAiLS*. Processes are the fundamental class of this implementation. This section is split into two parts. The first demonstrates the language concept of processes themselves, and the processes that are provided by *JAiLS*. The second shows how to create arbitrary processes.

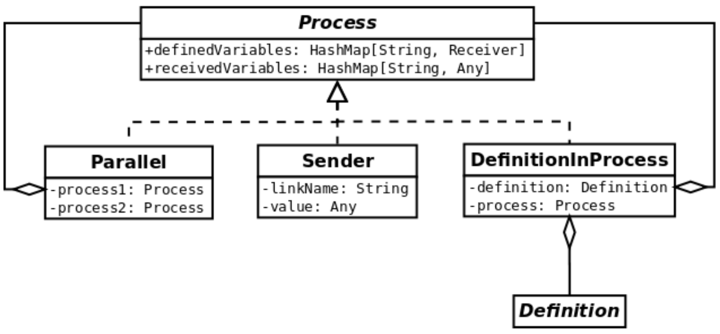


Figure 3.4.: Class diagram of `Process`

3.4.1. Processes in *JAiLS*

JAiLS already provides the programmer with the basic processes that are needed to write join-code. The basic class `Process` is an abstract class. As seen in figure 3.4 it is then implemented by various other classes. This is derived directly from the grammar that defines processes.

Empty and Success

As defined in [3] and discussed in this thesis there are two trivial processes; 0 and ✓. These have a representation in *JAiLS*. `Empty` and `Success` are two special objects extending the class `Process`. The `Empty` process does nothing, as it can be omitted [3] and the `Success` process prints `"success"` to the console. These will be used in this section to explain processes in *JAiLS*.

Parallel (Processes)

As discussed in secion 2.1, a process can be compound of two subprocesses.

$$P = P_1 \mid P_2$$

The *JAiLS* equivalent is the class

```
case class Parallel(process1: Process, process2: Process) extends Process
```

It works as a container for the subprocesses and can be used as follows

```
#> var P = Parallel(Empty, Success)

P   : JAiLS.Parallel = Parallel(JAiLS.Empty$@68beaed3,JAiLS.
    Success$@3ec7d45e)
```

Similar to the already seen concept of parallel `JoinPattern`, two existing processes can be joined to mark their concurrent execution with the the `|`-operator.

```
#> var Q = Success | Success

Q   : JAiLS.Process = Parallel(JAiLS.Success$@4f124609,JAiLS.
    Success$@4f124609)
```

Sender

The receiver has been explained in a previous section. This special process is the counterpart. Its purpose is to send a message v over a link x.

$$P = x\langle v \rangle$$

To create a simple sender (specialized senders can always be created as explained later in section 3.4.3) *JAiLS* offers the class

```
case class Sender(linkName: String, value: Any) extends Process
```

The sender is created with the name of the link and the value that should be sent to this link. If the link is not available at the time of the execution of the process, a `NoSuchElementException` is thrown.

```
#> val P = Sender("x", "v")

P   : JAiLS.Sender = Sender(x,v)
```

The earlier explained syntactic formalism to create a receiver is now used in a oppositional way. The chevrons in this case point to the exterior since they represent the sending process.

```
#> val P = <>("x", "v")

P   : JAiLS.Sender = Sender(x,v)
```

DefinitionInProcess

The most intriguing class of processes is the equivalent to the definition pattern in the join calculus.

$$P = \textbf{def}\ D\ \textbf{in}\ Q$$

It combines the definition discussed in section 3.3 and a process in which this definition is valid. It can be called.

$$\textbf{def}\ J \triangleright P\ \textbf{in}\ Q$$

As defined in section 2.1.1, the defined variables in this expression are the ones given by J. This join pattern, defines the various receivers that wait for messages. They are made public to the Process Q by this construction, so that Q itself or any of its parts can send messages to those links from the defined variables. To complete the syntactic formalism of *JAiLS* the last operator is introduced. With the `in`-operator a definition can be made public to a process.

The following code snippet shows how the first of the examples (see 2.1.2) can be written using the syntactic sugar *JAiLS* has to offer.

```
><("x", "u") |> <>("y", "u") in <>("x", 1)
```

It is quite close to the formalism of the join-calculus it was derived from

$$\mathbf{def}\, x(u) \triangleright y\langle u\rangle \,\mathbf{in}\, x\langle 1\rangle$$

.

3.4.2. Triggered processes and received variables

The last example is a poor one to execute in *Scala*, since it is a example of how messages are passed to an outer process and this process lacks of exactly that. An outside process that receives the message stored in variable `u` and receives it on channel `y` just does not exist. In order to extend this example to an executable one, a process `Printer("v")` is added. How this process is programmed, is explained in the following section regarding arbitrary processes. But for now, it is a process that takes the content of the variable `v` and prints it to the console.

```
val ex1 = ><("y", "v") |> Printer("v") in
          (><("x", "u") |> <>("y", "u") in <>("x", 1))
```

Listing 3.4: First CHAM example

When executed:

```
#> ex1.start
1
```

Explanation and message passing

When a process is started the innermost process P, that is bound by a **def** D **in** P statement is executed. It marks the starting point of the execution. In this example it is the part `<>("x", 1)`. When this is executed it sends the value `1` to the link `x` which has been defined by the definition `><("x", "u") |> <>("y", "u")`.

As there is only one join pattern in this statement and it has received a message over the channel `x`, all the preconditions are satisfied to trigger the process `<>("y", "u")`.

One could suspect, that this triggered process now sends the string `"u"` over the channel `y`. But the predefined sender class from *JAiLS* checks for the equality of names and replaces the value that is sent with the one it got from the receiver as a received variable.

At the end, the outermost receiver `><("y", "v")` receives a message on the channel `y` and makes it public over the received variables. This triggers the process `Printer("v")`, which prints the value of the received variable to the console. The whole flow of message passing is illustrated in figure 3.5.

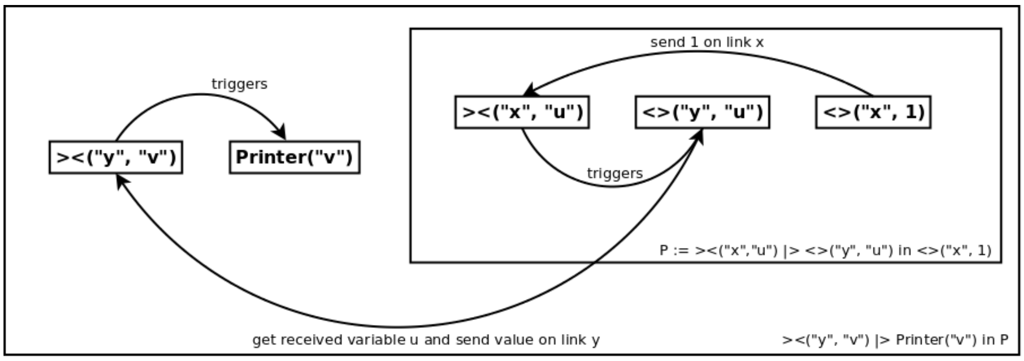

send 1 on link x
triggers
><("x", "u")
<>("y", "u")
<>("x", 1)
triggers
><("y", "v")
Printer("v")
P := ><("x","u") |> <>("y", "u") in <>("x", 1)
get received variable u and send value on link y
><("y", "v") |> Printer("v") in P

Figure 3.5.: Message passing

Scoping

The second example 2.2 makes it necessary to pay attention to the scoping of the variables.

$$\textbf{def}\ y(u) \triangleright x\langle u\rangle\ \textbf{in}\ \textbf{def}\ x(u) \triangleright y\langle u\rangle\ \textbf{in}\ x\langle 2\rangle$$

Again, this example lacks the outside world that receives the last message passed from the inside. So this is the executable version in *JAiLS*.

```
val ex2 = ><("x", "a") |> Printer("a") in
              (><("y", "u") |> <>("x", "u") in
                (><("x", "u") |> <>("y", "u") in <>("x", 2)))
```

Listing 3.5: Second CHAM example

The defined variable `x` as a link is used on two different levels and it is of fundamental importance, that the value `2` is not sent directly to the outside world. To show that this is not the case, a debug message was added in the receiver class. This is the outcome when executed:

```
#> ex2.start
received 2 over link x
received 2 over link y
received 2 over link x
2
```

A second example shows that the problem of the name clash is also solved for the triggered processes. It also illustrates, how the defined variables are passed down the structure into the triggered sub processes.

```
val rv = ><("y", "a") |>
             (><("y", "v") |> Printer("v") in <>("y", "a")) in
               <>("y", 42)
```
Listing 3.6: Message passing

The message first sent from `<>("y", 42)` is received by the outer most receiver and then passed into an inner process. This itself is a process of the form **def** D **in** P. This inner process is then sending a message on a link that has the same name as the outer process, but is routed correctly to the inner receiver as it has been overwritten.

Figure 3.6 demonstrates how the passing of messages works.

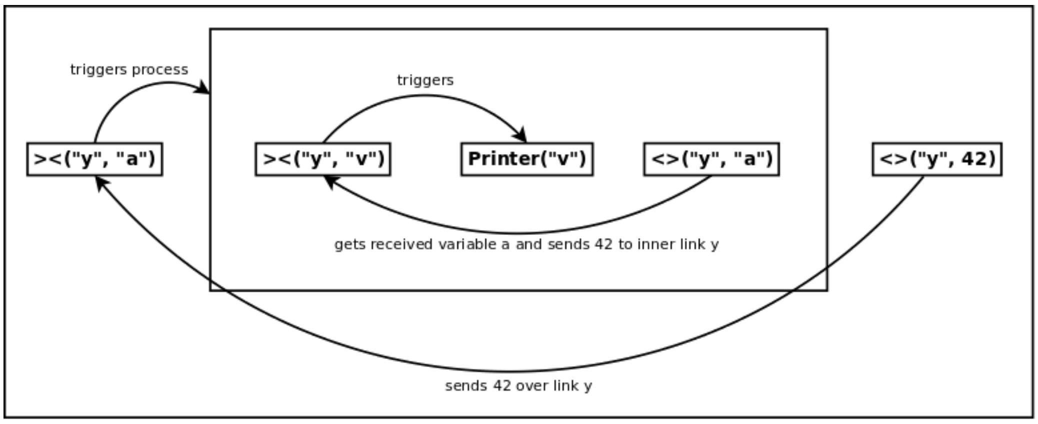


Figure 3.6.: Message passing

Using the received variables as a link

To show the use of received variables as a link, example 2.4 is implemented in JAiLS.

$$\textbf{def}\ \text{ready(printer)} \mid \text{job(file)} \rhd \text{printer}\langle\text{file}\rangle\ \textbf{in}\ \text{ready}\langle\text{laser}\rangle \mid \text{job}\langle\text{doc}\rangle$$

In this example there a printer was added on the outside to receive a file. Both message and printer are first sent over seperate links `><("ready", "printer")` and `><("job", "file")`. After both messages are received, the message that has been received as a file is forwarded on the link that has been stored in `printer`.

```
val laser    = ><("laser", "x") |> Printer("x")
val dispatch = ><("ready", "printer") | ><("job", "file") |> <>("
   printer", "file")
val spool    = laser in (dispatch in <>("ready", "laser") | <>("job"
   , "datei"))
```

Listing 3.7: Fourth CHAM example – Spooler

The *JAiLS* standard sender manages the evaluation of names automaticly so the execution of the spooler results with debug output of the receiver.

```
#> spooler.start

received datei over link job

received laser over link ready

received datei over link laser

datei
```

3.4.3. Arbitrary processes

The previous sections have used processes called `Printer` or `Add1` that do obviously more than it is possible to express in the pure join-calculus. Such processes are implemented as an extension of the abstract class `MyProcess` (see figure 3.7).

To implement an arbitrary process only the method `body: Unit` has to be implemented. Apart from this abstract, parameterless method the abstract class `MyProcess` offers two concrete functions
final protected def `dv(linkName: String): Receiver`
and
final protected def `rv[A](varName: String): A`

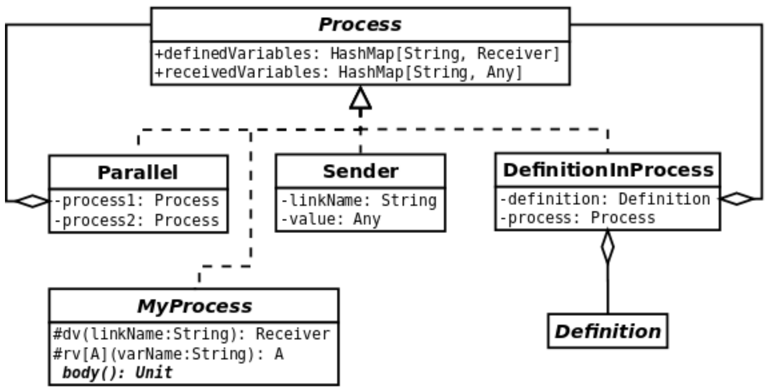


Figure 3.7.: Class diagram of `Process` including arbitrary processes `MyProcess`

to access the defined and the received variables. These are the ones that are accessible at this point, i.e. the process is within the scope of those variables. Due to the fact that the defined variables are uniformly instanciations of the class `Receiver` any message can be sent directly to this receiver with the *bang*-operator !. This is because as the underlying message passing system is the one provided by the actor model (see chapter 4). Since the received variables on the other hand have the type `Any`, an explicit conversion would be necessary to continue working with them. To keep the complexity low and the code neat and clean, `rv` is defined as a typed function, that implicitly does the explicit conversion.

This is the implementation of the printer class that was used in the earlier examples;

```
case class Printer(inputName: String) extends MyProcess {
  def body {
    println(rv(inputName))
  }
}
```

Listing 3.8: Implementation of a `Printer`-class

Note. *Defining arbitrary methods as case classes reduces the programming effort and increases recyclability.*

As *JAiLS* is based on *Scala*, it would be possible to just create objects that extend the `MyProcess` class without defining a class. For example this special printer:

```
object Printer extends MyProcess {
  def body {
    println(rv("x"))
  }
}
```

Listing 3.9: Printer as object

This object can obviously be used as printer, but is only able to work on a fixed variable because objects cannot be parameterized. On the other hand, flagging a class as a **`case class`**, creates some methods internally and a companion singleton object [9] for the created class. This object offers a factory-method that returns an instanciaded object without using the **`new`**-operator. This is how `Printer("x")` can be used as a Process in the *JAiLS*-code.

The third example (see example 2.4) needs a process that multiplexes the received variables and sends them as a tuple to a desired target-link. This could be managed by the following implementation.

```
case class Multiplexer(c1: String, c2: String, target: String)
   extends MyProcess {
  def body {
    dv(target) ! (rv(c1), rv(c2))
  }
}
```

Listing 3.10: Implementation of a `Multiplexer`-class

3.5. Nondeterminism and replication

3.5.1. Nondeterminism (in conjoined definitions)

As discussed earlier, there are different ways to express nondeterminism in the join-calculus. The first variant to take into account is example 2.5.

$$\textbf{def}\ s() \triangleright P \wedge s() \triangleright Q\ \textbf{in}\ s\langle\rangle$$

It uses two conjoined definitions, each one defining a join pattern waiting for messages on a link s. Sending a message to this link will either result in the execution of P or Q (`Println(line: String)` being a process that prints a given string).

```
val P = Println("P")
val Q = Println("Q")
val ex5 = (><("s") |> P) &
          (><("s") |> Q) in <>("s")
```
Listing 3.11: Nondeterminism in conjoined definitions

Note. *Sender and Receiver can also be used without a specific message. The receiver is satisfied just by calling it.*

3.5.2. Nondeterminism II (as guarded access)

A second form of nondeterminism is seen in example 2.6, whereby the same join pattern is called several times but also guarded by one pattern that is only called once.

$$\textbf{def}\ \text{once}() \mid y(v) \triangleright x\langle v\rangle\ \textbf{in}\ y\langle 1\rangle \mid y\langle 2\rangle \mid y\langle 3\rangle \mid \text{once}\langle\rangle$$

In *JAiLS*:

```
val ex6 = ><("once") | ><("y", "v") |> Printer("v") in
          <>("y",1)  | <>("y",2) | <>("y",3) | <>("once")
```
Listing 3.12: Nondeterminism as guarded access

The outcome of this program is either `1`, `2` or `3`, as the whole join pattern is satisfied only once. This is because `<>("once")` is only called once.

3.5.3. Nondeterminism III (in one join pattern)

To get back to the pathological case from the section about join pattern (cf. section 3.2) a third way to express nondeterminism is created by this code snippet:

```
val patho = ><("link", "x") | ><("link", "y") |>
            (Printer("x") | Printer("y")) in
              <>("link", 1) | <>("link", 2)
```
Listing 3.13: Nondeterminism in one join pattern

The outcome of this program is highly unpredictable. One outcome is that both messages are caught by the same join pattern, in which case the conjoined pattern is not satisfied and no process is triggered. But even when both messages get distributed fairly to both join pattern it is not determined which received variable is set to which message. To complete the nondeterministic charade the triggered process itself is compound of 2 parallel processes executed concurrently, which means it is not to be foreseen which process gets to print first.

3.5.4. Replication

As mentioned earlier the join calculus can express replication as well. The fifth example will be extended to create a loop.

```
val P = Print("P") | <>("s")
val Q = Print("Q") | <>("s")
val ex5R = (><("s") |> P) &
           (><("s") |> Q) in <>("s")
```

Listing 3.14: Replication

Processes `P` and `Q` are both containers for two subprocesses executed concurrently. One sends a message on the link `s`, resulting in an infinite loop of random invocations of the processes `P` and `Q`. Therefore the resulting output of this process is something like this:

```
#> ex5R.start

QQPQQPQQPPQQPQPPQQQQQQQPQPPPQPQQQQQQQQQPQPPPQQPPPPQPQQPQQPPQPPPQ
```

3.6. Parallelizability

3.6.1. A problem in the best of all possible worlds

Runnning processes concurrently is of crucial importance in dealing with the join calculus. The previous example showed different processes running at the same time. Until now, the discussed problems and *JAiLS*-programs were not noticeably effected by a problem regarding cocurrency. The problem is basically avoidable if a process is formed out of well-separated subprocesses, i.e. each process object occurs exactly one time in the whole construct.

As long as this is the case, all the processes can run concurrently. Let there be a process, that counts up and stops between the steps for a random time. It prints the current number together with an identifier to the console.

```scala
case class Cnt(name: String) extends MyProcess {
  def body {
    var c   = 0
    val rnd = new Random()
    while(true) {
      Thread.sleep(rnd.nextInt(500))
      println(name + ": " + c)
      c += 1
    }
  }
}
```

Listing 3.15: Countingbattle I

This process is now instantiated twice for two wise men betting who would finish first.

```scala
val parallel = ><("start") |> (Cnt("Candide") | Cnt("Pangloss")) in
                <>("start")
```

One arbitrary execution looks like this:

```
#> parallel.start
Candide: 0
Pangloss: 0
Candide: 1
Pangloss: 1
Pangloss: 2
Pangloss: 3
Candide: 2
Candide: 3
Candide: 4
Pangloss: 4
Candide: 5
Pangloss: 5
Candide: 6
Candide: 7
Pangloss: 6
Candide: 8
...
Pangloss: 129
Candide: 123
Candide: 124
Pangloss: 130
Pangloss: 131
Candide: 125
```

```
Candide: 126
Pangloss: 132
...
```

Both processes are obviously running parallel. Taking this example to the next level shows the foundation of the problem. Since Pangloss was Candide's teacher and master, and since Candide was never to eager to question the lessons given by his teacher, but to repeat them, it could have also been the teacher Pangloss starting this counting battle against himself. In *JAiLS*-words:

```
val pangloss = Cnt("Pangloss")
val parallel = ><("start") |> (pangloss | pangloss) in
                 <>("start")
```

Listing 3.16: Countingbattle II

This is how the battle would appear executed:

```
#> parallel.start
Pangloss: 0
Pangloss: 1
Pangloss: 2
Pangloss: 3
Pangloss: 4
Pangloss: 5
Pangloss: 6
Pangloss: 7
Pangloss: 8
Pangloss: 9
Pangloss: 10
...
```

Obviously this did not achieve the desired result. Only one of the processes started counting. While one process was runnning, the other one, which in fact is the same object or process could not start itself a second time.

This problem seems fairly constructed but has quite an impact regarding a question on the next level of concurrency. To visualize this dilemma, let Pangloss not have only one student but a whole class, and start converting them one by one into a copy of himself, joining him into the counting battle.

A model of this situation would be implemented in *JAiLS* as such:

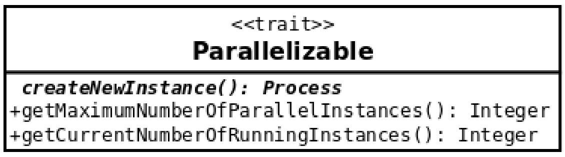
```
val pangloss = Cnt("Pangloss")
val parallel = ><("start") |> (pangloss | <>("start")) in
                <>("start")
```

Listing 3.17: Countingbattle III

The desired outcome would be the replication of various panglosses counting against each other, which of course will not happen as it is always the same process the computer is operating on. The problem that seemed constructed when dealing with a finite number of Processes, becomes severe when trying to work with an infinite number of replications.

3.6.2. Resolution

To solve this dilemma in *JAiLS*, there needs to be some kind of instantiation method. It would be neat, if the process itself would provide such a method. To instantiate itself following the idea of a factory design pattern. In order to turn the process into a process that provides such a method *JAiLS*, offers the trait **`trait`** `Parallelizable`, visualized in figure 3.8.



<<trait>>
Parallelizable
createNewInstance(): Process
+getMaximumNumberOfParallelInstances(): Integer
+getCurrentNumberOfRunningInstances(): Integer

Figure 3.8.: Trait Parallelizable

This trait can be aggregated to any process, ensuring that the method `createNewInstance : Process` is available in the process. As this is an abstract method, it has to be defined by the process.

Now it is possible to have Pangloss compete against himself.

```
case class CntParallel(name: String) extends MyProcess with
   Parallelizable {
  var id = 0

  def body {
    var c   = 0
    val rnd = new Random()
    while(true) {
      Thread.sleep(rnd.nextInt(500))
      println(name + ":␣" + c)
      c += 1
    }
  }

  def createNewInstance(): Process = {
    val instance = CntParallel(name + "␣" + id)
    id += 1
    instance
  }
}

val pangloss = CntParallel("Pangloss")
val parallel = ><("start") |> (pangloss | pangloss) in
                <>("start")
```

Listing 3.18: Countingbattle IV

What occurs, is that each time a process implementing the trait `Parallelizable` should be started, a new instance of the process is spawned and invocated instead of the original process. This makes it possible for the spawned processes to run concurrently. To make it possible to distinguish both processes, an identifier was added.

```
#> parallel.start
Pangloss 0: 0
Pangloss 1: 0
Pangloss 0: 1
Pangloss 1: 1
Pangloss 1: 2
Pangloss 1: 3
Pangloss 0: 2
Pangloss 1: 4
Pangloss 0: 3
Pangloss 1: 5
Pangloss 1: 6
Pangloss 0: 4
Pangloss 1: 7
...
Pangloss 1: 90
Pangloss 0: 107
Pangloss 0: 108
Pangloss 1: 91
Pangloss 0: 109
Pangloss 0: 110
Pangloss 1: 92
Pangloss 1: 93
Pangloss 1: 94
Pangloss 0: 111
Pangloss 1: 95
Pangloss 1: 96
...
```

And finally, Panglosses battle against his, only by the maximum number of parallel processes on the computer limited number, of students.

```
val pangloss = CntParallel("Pangloss")
val parallel = ><("start") |> (pangloss | <>("start")) in
                   <>("start")
```

Listing 3.19: Countingbattle V

```
#> parallel.start
Pangloss 4: 0
Pangloss 6: 0
Pangloss 5: 0
Pangloss 5: 1
Pangloss 6: 1
Pangloss 5: 2
Pangloss 6: 2
Pangloss 0: 0
Pangloss 1: 0
Pangloss 3: 0
Pangloss 6: 3
Pangloss 2: 0
...
```

4. Implementation of *JAiLS*

Actors define a computational concept that is highly reactive [10] as they suspend their work until a communication arrives and they go on with their given task.

This is quite similar to the concept of the definitions in the join-calculus. The join pattern, compound of several reception patterns waits for messages to arrive on their channel. When the messages are received and all the reception pattern are satisfied, a process is triggered. The implementation of *JAiLS* is based upon this equivalence of concepts.

This chapter will show how *JAiLS* was implemented. It starts off with the introduction of the classes the framework is built of and explains the interaction between them. This time the discussion follows a top-down approach (see figure 4.1). From there the discussion will show how a process is executed and how variables are passed on to the substructures of the implementation.

As this implementation is based upon the actor model provided by *Scala*, parts of the implementation that rests on the nature of actors are examined and explained in-depth.

Note. *This chapter will not explain the complete implementation skipping parts where it comes in handy.*

4.1. Process

The most important parts of the join-calculus framework are undeniably the processes. As stated by their inventors [5], actors are some kind of meta-objects that can contain any form of semantic matter. As explained in section 2.2 literally everything in *Scala* can be defined as an actor and this is exactly what this implementation does. Processes are defined as actors.

Actors in *Scala* are created by extending or implementing the trait `scala.Actor` class. As `Process` does not have any other given semantics, but should work as an

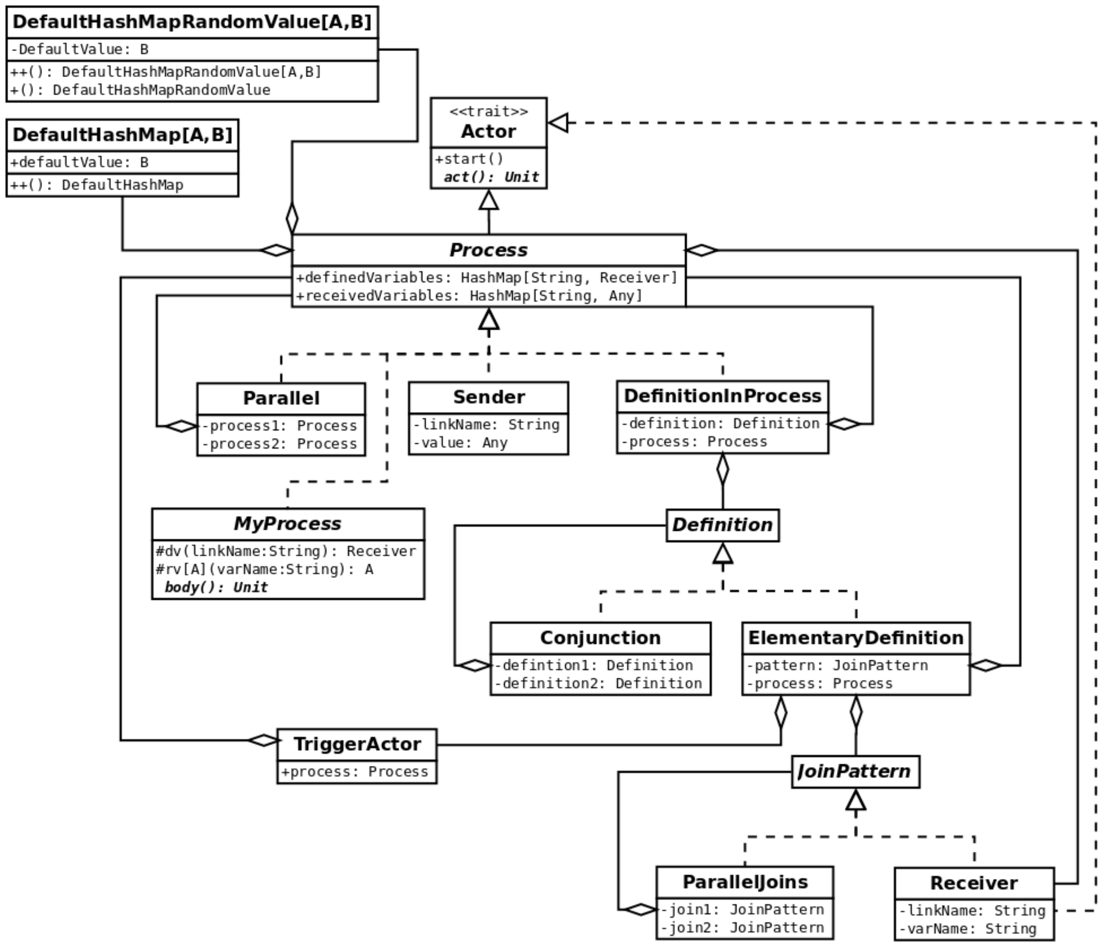


Figure 4.1.: Class diagramm of *JAiLS*

empty container for any code in *JAiLS* it is an abstract class that extends the trait `scala.Actor` (see figure 4.2).

Therefore, processes have to implement the method `act: Unit` which does not really make sense since `Process` is defined as an abstract class.

```
abstract class Process extends Actor{

  // The default definition (interpreted as a non definition)
  var definedVariables = new DefaultHashMapRandomValue[String,
   Receiver](DefaultReceiver)
  var receivedVariables = new DefaultHashMap[String, Any](null)

  def act() {
    // Nothing so far
  }

  override def start: Actor = {
    this.getState match {
      case State.New => super.start
      case _ =>
        while(this.getState != State.Terminated) {
          Thread.sleep(10)
        }
        super.restart
        this
    }
  }
}
```

Listing 4.1: class `Process`

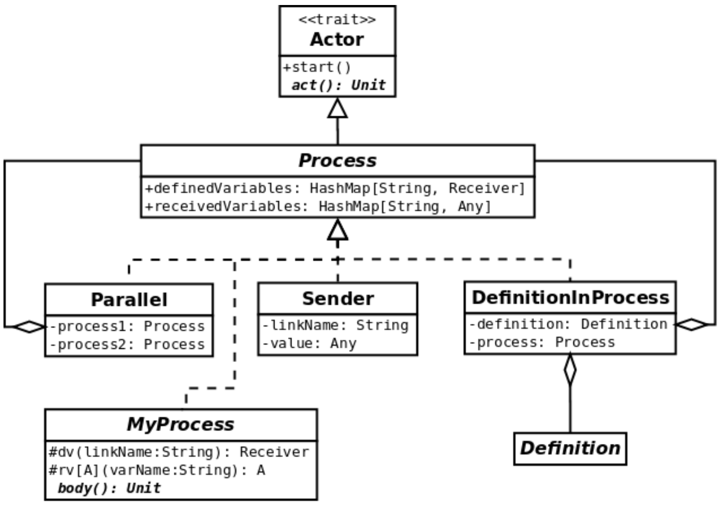


Figure 4.2.: Class diagramm of `Process`

What actually makes sense is to override the method `start: Actor`. This method is usually able to start an actor. Due to the fact that in the join-calculus some processes are run not only once, this uniforms the starting mechanism. Since an actor that has been terminated once, can only be started again by the invocation of the `restart` method instead (this is actually not necessary for processes flagged as `Parallelizable` - cf. section 3.6).

4.1.1. Variables

The two members `definedVariables` and `receivedVariables` are defined in the abstract class `Process`. Both pools of variables are separated which comes in handy to handle the different approach of name clashes and scoping. The defined variables are implemented as an extended `HashMap`. To instanciate it, it needs a default value that is returned when the key was not found in the map.

In line 4 the defined variables are instantiated as a

`DefaultHashMapRandomValue[String, Receiver]`

with the default of the `DefaultReceiver`. When a sender is trying to send a message to a non existing link, the message is sent to the `DefaultReceiver` which will throw a `NoSuchElementException`.

Having a closer look at the implementation of the `DefaultHashMapRandomValue` will show how name clashes are handled. The class defines two methods `+: DefaultHashMapRandomValue` and `++: DefaultHashMapRandomValue`. Both conjoin two maps, but differ in the method how they handle double keys. + will overwrite the value with the new one if the key already exists. This is necessary when in a triggered process there is a definition of a new name that has the same name as the one of the outside process (see example 2.2).

The other method ++ does not overwrite the old value. Instead, both values are stored in a list. This proves of use in example 2.5. Both definitions coexist, but will trigger different processes. Which one of the processes P or Q will be triggered is totally random, since the lookup for a defined variable returns randomly one of the receivers.

$$\mathbf{def}\; s() \triangleright P \wedge s() \triangleright Q \;\mathbf{in}\; s\langle\rangle$$

4.1.2. Parallel

When two processes are defined parallel using the |-operator they got aggregated to one object of the class `Parallel`.

```
case class Parallel(process1: Process, process2: Process) extends
   Process {
  override def act() {

    var p1 = process1
    var p2 = process2

    // make the current definition available to the subprocesses
    process1.setDefinedVariables(definedVariables)
    process2.setDefinedVariables(definedVariables)
10      process1.setReceivedVariables(receivedVariables)
11      process2.setReceivedVariables(receivedVariables)
12
13      // check if the process should be started as a new instance (
       created by copy function) or the original
14      process1 match {
15        case p: Parallelizable =>
16          p1 = p.createNewInstance
17          p1.setDefinedVariables(definedVariables)
18          p1.setReceivedVariables(receivedVariables)
19        case _ =>
20      }
21      process2 match {
22        case p: Parallelizable =>
23          p2 = p.createNewInstance
24          p2.setDefinedVariables(definedVariables)
25          p2.setReceivedVariables(receivedVariables)
26        case _ =>
27      }
28
29      // start the subprocesses
30      p1.start
31      p2.start
32    }
33  }
```

Listing 4.2: class `Parallel`

This class works as a container for the subprocesses. This process also shows two things. First, it shows how the defined and received variables are passed on to the subprocesses. The other implementational element, that can be observed, is how there is spawned an instance of the subprocess if it has the type `Parallelizable`. Being a new instance, it is necessary that the variables are made public to it explicitly. At the end, when a parallel process is started, both of the subprocesses have to be started as well (lines 30 and 31).

Note. *The invocation of the method `start: Actor` of an actor will start it concurrently, which means that both of the subprocesses, actors themselves, will be started concurrently.*

4.1.3. Sender

The standard sender [1] is created with two parameters: A linkname to which the message should be sent and the value that will be sent.

[1]How a more specialized sender can be written can be derived from section 3.4.3

```
 1  case class Sender(linkName: String, value: Any) extends Process {
 2    override def act() {
 3
 4      var message = value
 5      var link    = definedVariables(linkName)
 6
 7      // if the value is known from the received variables as a
        variable name
 8      // pass the content on to the receiver
 9      if(receivedVariables(value.toString) != null)
10        message = receivedVariables(value.toString)
11
12      // if the linkName is known from the received variables as a
        link in the defined variables use this link instead
13      if(receivedVariables(linkName) != null)
14        link = definedVariables(receivedVariables(linkName).toString)
15
16      //println("Sending " + value + " over link " + linkName )
17      link ! message
18    }
19  }
```

Listing 4.3: class `Sender`

And this is exactly what it does with the last command. `link ! message` is the actor construct of sending messages asynchronously.

Note. *Extending the* `MyProcess` *class, a synchronous variant of this sender can be written easily using the* `!?`*-operator instead.*

The standard *JAiLS*-sender is able to pass on values or send messages to links only by name-equality. The first case is handled with a lookup of the value as a variablename in the received variable structure in line 9 and the second case by looking up the linkname in the received variables.

4.2. DefinitionInProcess

The third of the trinity of processes does not much more than contain a definition and a process that the definition is valid in.

```
case class DefinitionInProcess(definition: Definition, process:
   Process) extends Process {
  override def act() {
    definition.makeReadyToReceive

    val newDefinedVariables = definedVariables + definition.
    getDefinedVariables

7       process.setDefinedVariables(newDefinedVariables)
8       process.setReceivedVariables(receivedVariables)
9   
10      definition.makeVariablesAvailableForTriggeredProcesses(newDefinedVariables)
11  
12      // check if the process should be started as a new instance (created by copy function) or the original
13      process match {
14        case p: Parallelizable =>
15          val instance = p.createNewInstance
16          instance.start
17        case _ =>
18          process.start
19      }
20    }
21  }
```

Listing 4.4: class `DefinitionInProcess`

Line 5 does what has been discussed before. The already defined variables, which are valid at this point, have to be overwritten by the new definitions in the `definition` this object is compound of. Figure 4.3 illustrates a definition `D` that is valid for a process `P` and therefore for its subprocesses `P1` and `P2`.

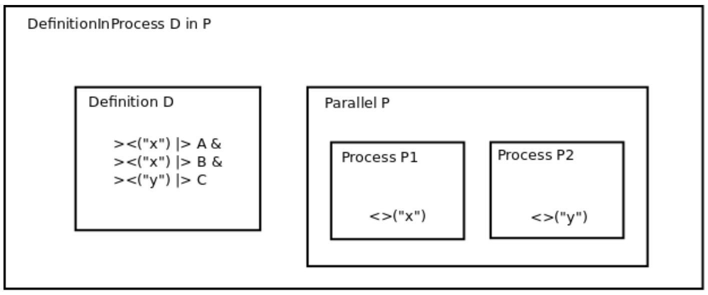


Figure 4.3.: Arbitrary `DefinitionInProcess`-process

Let the process $A := \textbf{def}\ D\ \textbf{in}\ P$ exist in an environment E which is a process that contains A and in which there is already defined a link y. When the start method of A has been invoked, the following steps are taken:

1. A calls a method from D that returns all of the defined links,
2. A joins the map of defined variables with the one that was already defined by the environment,
3. The entry for the key y in the map of defined variables has been overwritten,
4. A sends the defined variables to P and invokes the start method of P,
5. P sends the defined variables to $P1$ and $P2$ and starts both processes.

4.3. Receiver

A join pattern can defined either as bein compound of two sub pattern, or it has the type `Receiver`. The second type was implemented like this:

```
1 case class Receiver(linkName: String, varName: String) extends JoinPattern with Actor {
2   def act() {
3     loop {
4       react {
5         case message => getTrigger ! (varName, message)
6       }
7     }
8   }
9 }
```

Listing 4.5: class `Receiver`

Not only is a receiver an instance of the class `JoinPattern`, but it also implements the trait `Actor`. For every link in the *JAiLS*-system there is one object like this one waiting for incoming messages. They are suspended in line 5 and wait for messages. When a message arrives to this `Receiver`-object it will go to a pattern matching in line 6. As this pattern matches for every message it will always execute the next line. In line 6 the `Receiver` calls a method it inherited from the abstract parent class `JoinPattern`. It will return a special object that exists for every `ElementaryDefinition` object and send it a tuple of the variable name and the message itself. What this trigger object does, is explained during the course of the next two sections.

4.4. Definition

A `Definition` can appear in two different forms, as well. One that conjoins two definitions and one that is an elementary definition. Only the second one is explained shortly.

```
case class ElementaryDefinition(pattern: JoinPattern, process: Process) extends Definition {

  // check if the process should be started as a new instance (created by copy function) or the original
  var triggeredProcess = process match {
    case p: Parallelizable =>
      p.createNewInstance
    case _ =>
      process
  }

  // setting the trigger actor and starting it
  val trigger = new TriggerActor(pattern.getReceivedVariableNames,
   triggeredProcess)
  trigger.start

  // the pattern that compounds this elementary definition needs to
   know this so it can call the trigger
  pattern.setDefines(this)
}
```

Listing 4.6: class `ElementaryDefinition`

The `ElementaryDefinition` object is a compound of a `JoinPattern` and a `Process` object. The process is the one that gets triggered if the join pattern is successfully satisfied with incoming messages on all the links. To manage the triggering of this process, there is defined a special object – the `TriggerActor`. It gets instanciated with a list of all the variable names that are defined in the join pattern and the process that will be triggered.

4.5. TriggerActor

The mysterious `TriggerActor` is an actor itself and this is its implementation:

```
case class TriggerActor(receivedVariableNames: List[String], process
   : Process) extends Actor {
  val messageQueueMap = HashMap[String, Queue[Any]]()
  receivedVariableNames.foreach(n => messageQueueMap(n) = Queue[Any
   ]())

  def act {
    loop {
      react {
        case (varName: String, value: Any) =>
          // adding new message to the specified queue
          messageQueueMap(varName) += value

          // checking if the pattern is complete
          var complete = messageQueueMap.values.foldLeft[Boolean](
   true)((b, q) => b && !q.isEmpty)

          if (complete) {

            var triggeredProcess = process

            // create receivedVarMap for the process
            val varMap = new DefaultHashMap[String, Any](null)
21       for((k,v) <- messageQueueMap) varMap(k) = v.dequeue
22
23       process.setReceivedVariables(varMap)
24
25       // if this process is a parallelizable process it has to be instanciated first
26       process match {
27         case p: Parallelizable =>
28           triggeredProcess = p.createNewInstance
29           triggeredProcess.setDefinedVariables(process.definedVariables)
30           triggeredProcess.setReceivedVariables(process.receivedVariables)
31         case _ =>
32       }
33
34       triggeredProcess.start
35     }
36    }
37   }
38  }
39 }
```

Listing 4.7: class `TriggerActor`

The `TriggerActor` holds a `HashMap[String, Queue[Any]]` which has an entry for each `Receiver`. Each variable name is mapped to a queue of values that will hold the messages received on this pattern (see line 2 and 3). When the trigger has been instantiated and started, it will suspend waiting for messages (line 7). If received messages are matched as a tuple $(variableName, value)$, the value will be inserted into its designated queue. Then the trigger will check if all of the join patterns have been satisfied. If so, it will create a received variable map `DefaultHashMap[String, Any](`**`null`**`)` for the triggered process (line 20-21). In lines 26 until 32 it will again check if the process should be started as a new instance or if it can be started without being spawned. Figure 4.4 visualizes how a process is triggered by incoming messages.

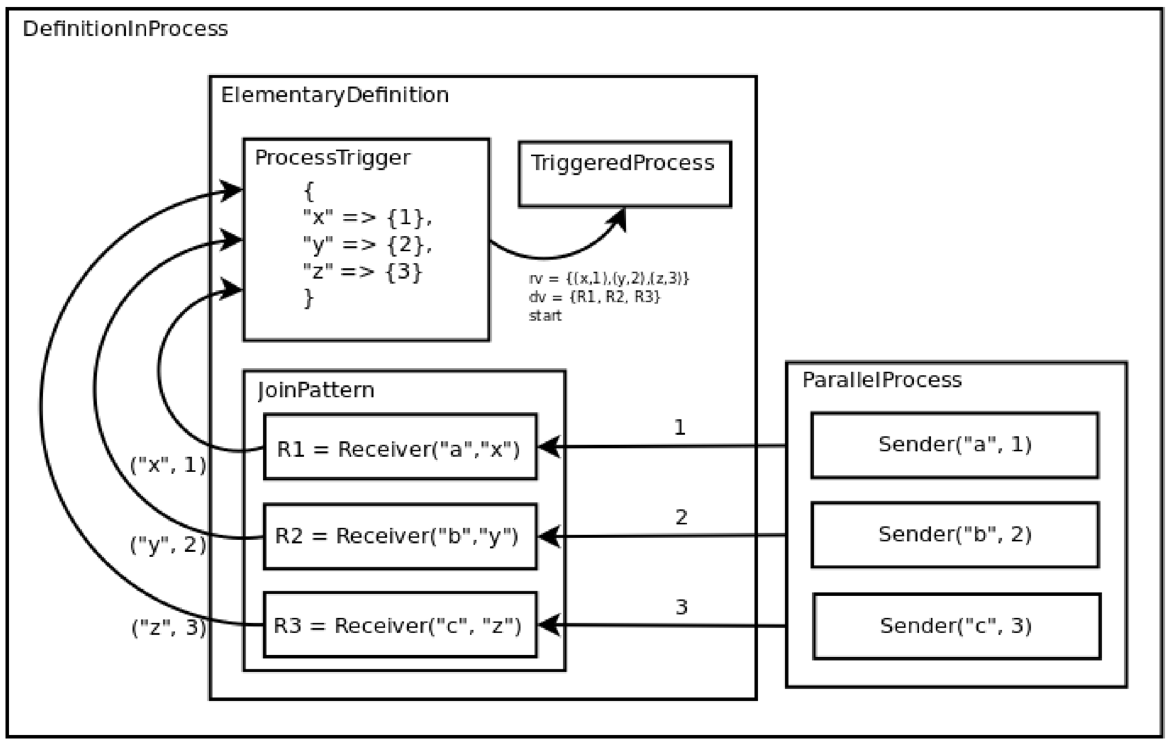


Figure 4.4.: How a process is triggered

5. Conclusion

This thesis showed how the join-calculus as a theoretical construct was implemented in the environment of an actual programming language. As discussed, one important property of programs written in a programming language is that they can be executed. It was discussed how join-calculus programs are created and executed in *JAiLS* as a programming language having a syntax quite close to the one of the theoretical join-calculus and how the underlying interaction between processes operates.

The process of implementation led to some interesting problems, such as the problem of infinite replication, that was solved by a method to spawn new instanciations of processes when needed.

In fact, *JAiLS* comes in handy as a quick tool to implement, execute and test join-calculus programs.

6. Future work

JAiLS was developed with the intention not only to have a join-calculus programming tool, but also develop in an extendable manner. Working with processes as actors, and following an object oriented approach makes it possible to extend this project.

This is a list of ideas that could extend *JAiLS*.

Distributed join-calculus There are several interesting approaches to implement a distributed join-calculus, of which the one proposed by Fournet, Gonthier, Lévy, Maranget and Rémy in [13] and extended by Lévy in [12] would be suitable for the extension of *JAiLS*. They discuss an extension of the grammar that describes locations and how they as first-class objects can be sent through channels. This would complete *JAiLS* to a powerful join-calculus framework.

Typed variables Working with *JAiLS*, makes it necessary from time to time to explicitly cast variables. This could be avoided introducing a type system. It has to be examined how this would restrict the use of defined links.

Migrate to *Akka* *JAiLS* was developed using the trait `scala.Actor`. Since *Scala* version 2.11.0 the *Scala* Actors library was flagged as deprecated. Since then, the default actor framework is *Akka*. In [14], there is described a way to migrate from `scala.actors` to `akka`.

Bibliography

[1] Fournet, C., Gonthier G., The Reflexive CHAM and the Join-Calculus, In: Proc. of POPL. SIGPLANT-SIGACT, pp. 372-385, (1996)

[2] Berry, G., Boudol, G., The Chemical Abstract Machine, In: Proc. of POPL. SIGPLAN-SIGACT, pp. 81-94 (1990)

[3] Peters, K., Nestmann U., Goltz, U., On Distributability in Process Calculi*, In: ESOP 2013, LNCS 7792, pp. 310-329, (2013)

[4] Peters, K., Nestmann U., Goltz, U., On Distributability in Process Calculi* (Appendix), Technical Report, TU-Berlin (2013)

[5] Hewitt, C., Bishop, P., Steiger, R., A Universal Modular Actor Formalism for Artificial Intelligence, In: International Joint Conference on Artificial Intelligence, pp. 235-245 (1973)

[6] Clinger, W. D., Foundations of Actor Semantics, Technical Report 633, Phd thesis, Massachusetts Institute of Technology, AI Laboratory, (1981)

[7] Agha, G., Actors: A Model of Concurrent Computation in Distributed Systems, Technical Report 844, Phd thesis, Massachusetts Institute of Technology, AI Laboratory, (1985)

[8] Agha, G., Mason, I., Smith, S., Talcott, C., A Foundation for Actor Computation, Journal of Functional Programming, (1996)

[9] Odersky, M., Spoon, L., Venners, B., Programming in Scala (Second Edition), artima, (2010)

[10] Haller, P., Sommers, F., Actors in Scala - Concurrent programming for the multicore era, artima, (2011)

[11] Millner, R., communicating and mobile systems: the π-calculus, Cambridge University Press, (1999)

[12] Lévy, J.-J., In: Theoretical Aspects of Computer Software, pp 233-249, (1997)

[13] Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., Rémy, D., A calculus of mobile agents In: Proceedings of the 7th International Conference on Concurrency Theory, Springer, LNCS 1119, (1996)

[14] Jovanovic, V., Haller, P., The Scala Actors Migration Guide, `http://docs.scala-lang.org/overviews/core/actors-migration-guide.html`, (January 11, 2015)

A. *JAiLS* - manual

The usage of *JAiLS* is quite easy. This thesis comes with a version of the current version of *JAiLS*. It is also available on github. To get the complete repository - run the following command on the console.

```
git clone https://github.com/simonslama/jails
```

The revision that has been included in this thesis was tagged with the tag "final". The cloned repository can be set to this version using the following command.

```
cd jails
git checkout final
```

The clone includes three directories.

src This directory contains the core of *JAiLS*. Each of the different classes (`Process`, `Definition`, `JoinPattern`) has its own file that includes the abstract base class and the concrete implementations. The file TriggerActor.scala contains the `TriggerActor` class that handles the triggering of actors. The file Util.scala holds the implementations of the data structures for the received and defined variables.

examples Several example programs (discussed in appendix B).

documentation Contains this thesis.

To create a *JAiLS* program it is sufficient to include the framework with the following two lines of code:

```scala
import jails._
import jails.Process._
```

B. *JaiLS* - list of examples

CHAM.scala This file contains all the examples discussed in 2.1.2 and some extensions.

Counter.scala A basic counter example.

Factorial.scala Implementation of the factorial function in *JAiLS*.

MCPI.scala This contains an implementation of an monte carlo algorithm to calculate pi. There are two versions included. One runs sequential and the other one concurrently. The concurrent version needs some improvement.

Replication.scala Some examples regarding the concept of replication.

UnhandyCounter.scala The first example from the third chapter.