University of Bergen
Department of Informatics

# Specifying Actor-Based Computer Games

*Author:* Tor Andreas Røsæg
*Supervisors:* Crystal Chang Din, Mikhail Barash

UNIVERSITETET I BERGEN
*Det matematisk-naturvitenskapelige fakultet*

August, 2023

## Abstract

The *actor* model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to implement and control concurrent and distributed systems. Among all the languages which support actors, we choose Scala as the one in this thesis work to implement computer games. While the complex thread management using explicit locking and unlocking is avoided in the implementation, not all the Scala syntax is specifically relevant for the design of computer games. It can be challenging for the game designers to master the Scala language in order to concretize their ideas through programming. To assist the game designers even further, we create a Domain-specific Language (DSL), i.e, *GameLang*, for specifying actor-based computer games in Scala. GameLang is designed based on the characteristic and the concurrency criteria of computer games. The game developers can utilize this DSL with the syntax such as `player`, `actions` and `from-to` statements to specify the logistics of a game. Our DSL hides the accidental complexity of the Scala code. The concurrency-related implementation details in Scala is abstract away in our DSL. We discuss about possible approaches to generate Scala code from GameLang specification, and leave the complete code generation to the future work.

**Acknowledgements**

Thank you to my supervisor Associate Professor Crystal Chang Din for your excellent guidance and deep knowledge on concurrency. Thank you for the countless meetings filled with discussions, status updates, laughter and joys. You truly made me look forward to every meeting and each stepping stone. Your academic guidance have also been invaluable and I could not have done this without you.

Thank you to my co-supervisor Associate Professor Mikhail Barash for your excellent guidance on DSL and code generation. Thank you for the countless hours you dedicated to our meetings and supporting me. Your input on both code generation and concurrency issues have been invaluable as well as the many past papers and previous experiences have been great and likewise I could not have done this without your help.

Thank you to my girlfriend Molly Keeble for the motivational and emotional support. The long nights and many hours of work could not have been done without your support. The words of encouragement and the amount of love and care I have received have been invaluable.

Thank you to my friends and family for all the support I have received.

Tor Andreas Røsæg

Tuesday 15th August, 2023

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

Concurrency in programming is a great mechanism to divide the overall task into several smaller ones that can be executed concurrently. In computer games, multiple elements can interact with each other at runtime. It is therefore very difficult to predict what elements and objects will interact with each other. Without knowing what elements are interacting with each other, it becomes very difficult to control the behaviour of the system.

Multi-threaded programming is a common approach to implement concurrent programs. However, it is notoriously difficult to decide where to lock and unlock threads in the programs, especially when the dynamics of the program execution increases. In this thesis, we will give an insight into the difficulties of using explicit locking and unlocking, specifically in the domain of computer game design, and how it is utilized today. We will get an insight of a possible paradigm shift that might help solve this issue. Specifically, we will explain what actor-based paradigm is and why we choose to use it instead of explicit locking and unlocking in the multithreaded programming.

However, writing actor-based programs can also be a non-trivial task for the game designers who are not necessary programmers, especially when they need to grasp a new programming language that supports the actor-based programming paradigm. Take an example of implementing games using Scala actors, the game designers need to understand the meaning of *behaviours* in Scala, how to switch between *behaviours*, and why it is needed to switch between *behaviours*. The concept of *behaviours* is language specific and is not relevant for the game design. As a response to this difficulty, we created a Domain-specific Language (DSL), i.e., *GameLang*, for the game design. GameLang seeks to ease the design and control of concurrency in computer games. It hides irrelevant part of the Scala syntax that is not relevant for the game design. This means that the users of the DSL only need to care about the game design and the flow of the game. They can concentrate on specifying what messages are going to be sent or received and how the game will respond when those messages are received.

*GameLang* is a stepping stone in the game design. The ultimate goal of using *GameLang* is to generate Scala code from the given *GameLang* programs. A mapping table from the *GameLang* syntax to the equivalent Scala syntax is provided in Section 4.2. What should appear in the Scala code but is hidden in the *GameLang* programs for the game designers should be generated by the code generator. For example, the code generator will utilize the concurrency specification given by the *GameLang* users to infer what messages of an actor should be grouped together in one identifiable behaviour. The algorithm presented in Section 4.2.2 describes this grouping process. In Section 4.2.3 we discuss the possible approaches for implementing the code generator. The complete implementation of the code generator is however left in the future work.

The thesis is structured as follows: Chapter 1 gives an introduction of the work, Chapter 2 provides the background knowledge, Chapter 3 introduces the syntax of the DSL language *GameLang*, Chapter 4 explains the implementation details of *GameLang*, Chapter 5 presents the related work, and Chapter 6 concludes the thesis and discusses future work.

# Chapter 2

# Background

In this chapter, we provide sufficient background knowledge for the readers to understand the contribution of this master thesis. The topics include multithreading, the current state of art in game engines, actor-based concurrency, domain-specific languages and language workbenches.

## 2.1 Multithreading

Multithreading [2] allows an application to create a small unit of tasks to execute concurrently. A thread executes the code statements one by one in sequence. A single processor computer can run multiple threads concurrently, i.e., every thread runs sequentially but interleaves with each other. A multicore computer can execute threads in parallel on separate processors with each core processing only one instruction at a time. Note that we only focus on the concurrency-related issues in this work and do not consider multicore or the performance of computation.

A thread by itself is not a program. It cannot run on its own [7]. Instead it runs within a program as we can see in Figure 2.1. When a thread is created and started, it is in the *Runnable* state. Only when it is scheduled for execution, the state is changed from *Runnable* to *Running*. The scheduled thread either runs until it finishes the execution and ends in the *Dead* state, or it is *Blocked* when it is suspended, sleeping, or waiting for the resources that are held by another thread. A scheduler can reschedule a thread and move the thread from the *Blocked* state back to the *Runnable* state or kill the thread so the thread is *Dead*.

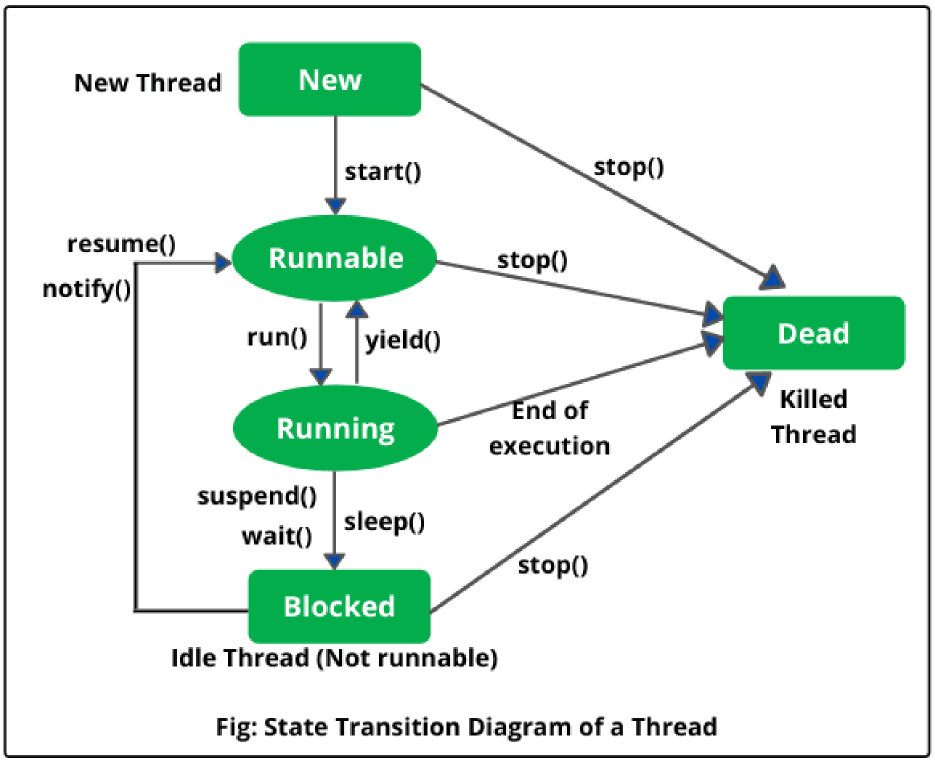

Figure 2.1: Life of thread in Java [40]

### 2.1.1 Explicit Locking and Unlocking Mechanism

The basic requirement in concurrent programming is coordinating the execution of multiple threads so that whatever order they are executed in, they produce the correct answer. Given that threads can be started and preempted non-deterministically, any moderately complex program will have essentially an infinite number of possible orders of execution. These systems are not easy to test [23].

Race condition is one of the common concurrency issues. Threads may share data and this can lead to race conditions if it is done improperly. A race condition is created when two processes share the same memory space. The execution outcome of the program becomes unpredictable because one process may modify a value before the other process read it. One solution to race conditions is to use the locking and unlocking mechanism. In Java the explicit locking and unlocking mechanism can be utilized by the use of semaphores. It can be implemented by using atomic blocks with semaphores to manually lock and unlock threads. This is done by the `acquire` and the `release` statements in Java. The atomic block, i.e., critical section, is defined between the `acquire` and the `release` statements. A thread, which grabs the lock, holds the lock until it finishes executing the critical section. Only after that, another thread can grab the lock and enter the critical section.

```java
public class SemaphoreDemo {

    // creating a Semaphore
    Semaphore s = new Semaphore(1);
    public static class myThread extends Thread{
    // Multiple threads call this method on the same/shared instance of
    ↪  "myThread"
        public void run() throws InterruptedException {
            ...
            // Acquire the permit
            s.acquire();

            // Code that runs inside the critical section

            // Release the permit
            s.release();
            ...
        }
    }

    public static void main(String[] args){
        myThread t1 = new myThread(s);
        myThread t2 = new myThread(s);
        myThread t3 = new myThread(s);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Listing 2.1: Example for how to use semaphore in Java

Listing 2.1 displays how a semaphore is declared and used in Java. The program creates three threads and uses one semaphore to synchronize them. Since only the thread who obtains the semaphore can execute the atomic block, at most one thread can be active in the atomic block at a time. An atomic block is used in order to safely prevent other threads from interfering with the one currently executing the block until the current thread leaves the critical section.

Concurrency can speed up the execution of programs by optimizing the use of resources and waiting time that would be otherwise wasted. This is often seen in the real time apps and video games. However, it is shown to be non-trivial to use multithreading in the game design. In the next section, we will discuss about the concurrency difficulties in the design of computer games.

## 2.2 Concurrency in Computer Games

Most game engines today use *C++* or *C#*. Concurrency was first introduced in *C++*11, which was published in 2011. *C++*11 supports threads, mutexes and locks. Among game engines made in *C++*, less than 10% [19] use a version older than *C++*11. Which means the concurrency concept is relatively new in the implementation of game engines. According to a developer at Rockstar[8], a gaming company, their propiatery engine had dedicated threads and a pool of worker threads. Some of the dedicated threads were the Renderer, the Input/Gameplay dispatcher, the Network dispatcher, the physics dispatcher and the animation dispatcher. These dedicated threads would create tasklets to be run on the worker threads. The boundaries between concurrent dispatchers were pretty well-defined, e.g. physics and gameplay must be separated, gameplay and networking must be separated, gameplay and AI must be separated. Once each frame, the visual state of the game is handed over to the renderer, which goes about rendering things while the other threads simulate the next frame. Given those boundaries, you decide on a partial ordering of tasks that theoretically maximizes the saturation of worker threads at all times. The dispatchers themselves aren't really supposed to be doing too much work, although the renderer tends to do plenty because the render passes have to be done in-order. But almost every other operation could be broken down into chunks worthy of multiplexing.

**The Difficulty of Concurrency in Computer Games** The difficulty of concurrency in computer games today is that they are very single-threaded, or are utilizing very few cores. This is due to programming concurrency is difficult while many different things can happen during the program execution. As a result, we propose to utilize actor-based concurrency over the explicit locking and unlocking mechanism in our concurrent programming as the former one is much easier to control. Besides, we can easily scale the number of actors according to the game design.

## 2.3 Actor-Based Concurrency

Actor-based concurrency is a different concurrency paradigm where actors use message passing to communicate with each other. Actors send and receive messages concurrently with the use of a mailbox system. When an actor receives a message, it can either change its state, send messages to other actors or create a finite number of child actors. By looking at the example in Section 2.1.1 which has to lock and unlock threads explicitly in the program code, the Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to implement and control concurrent and distributed systems. The Actor Model was defined in 1973 by Carl Hewitt [27] but has been popularized by the Erlang language [4, 10, 11], and used at Ericsson with a great success to build highly concurrent and reliable telecommunication systems.

Akka [15, 26, 42] is a scalable library for implementing actors on top of Scala and Java. The API of Akka actors has borrowed some of its syntax from Erlang. Interactions between actors in Akka only use message passing. Akka actors interact in the same way regardless whether they are on the same or separate hosts, communicate directly or through routing facilities, run on a few threads or many threads, and so on. Such details may be altered at deployment time through a configuration mechanism, allowing a program to make use of more (powerful) servers without modification. Akka is also well suited for hybrid cloud architectures and the elastic scaling of cloud platforms. Based on the strength of Akka, we choose Akka as our actor-based concurrency framework and specifically we choose Akka Scala [6, 26, 42] as the programming language for constructing computer games.

In Akka, an actor is the primitive unit of computation. It receives a message and performs computation based on the message received. The idea is very similar to what is used in object-oriented languages: an object receives a method call and does something depending on which method is invoked. The main difference is that actors are completely isolated from each other and they do not share memory. It is worth noting that an actor can maintain a private state that can never be changed directly by another actor, except through message passing. It is also important to know that the order of messages an actor sends to another actor is preserved at the receiving side.

Although multiple actors can run in parallel, an actor can only process one message at a time. This means that even if there are three messages sent to the same actor, the actor

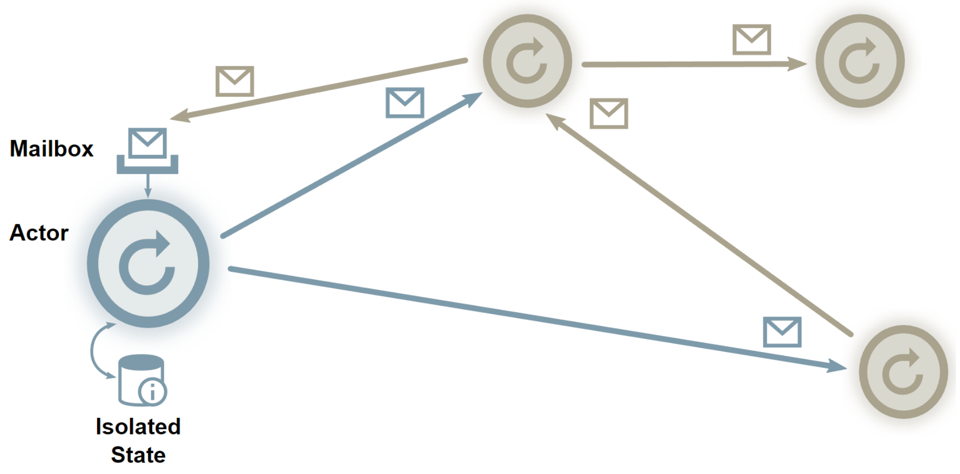

Figure 2.2: How actors use message passing [16].

will just execute one at a time. To have these three messages being executed in parallel, three actors need to be created, one for each message. Messages are sent asynchronously to an actor, that means the messages being sent need to be stored somewhere while the receiving actor is processing another message. As described in Figure 2.2, actors use message passing to communicate with each other. Every actor has a mailbox and an isolated state, as presented in the blue part of the figure. This mailbox preserves the order of the messages received in. The isolated state presents the current execution state.

**Actor-based concurrency in Akka Scala** We provide a simple example of actor-based concurrency in Akka Scala in Listing 2.2 and explain it in details as the following.

```
object ActorDemo {
    sealed trait MessageType
    case class MessageName(from: ActorRef[MessageType]) extends
   ↪ MessageType

    def apply(var: String): Behavior[MessageType] = {
        functionName(var)
    }

    private def functionName(var: String): Behavior[MessageType] =
        Behaviors.receiveMessage {
        case MessageName(from) =>
            if (condition) {
                Behaviors.stopped
            } else {
                from ! OtherActor.MessageName(context.self)
```

```
                    Behaviors.same
               }
          }
}
```

Listing 2.2: An simple example of Akka Scala

The `apply` function initializes the actor in such a way that it is ready to receive messages from other actors. It defines the initial behaviour and the type of message that it will receive. In this example, the `ActorDemo` actor will start in the `functionName` behavior and will receive messages with and only with the `MessageType` type. When a message arrives, the actor will store the message in its mailbox until it can process that message. Then the code inside the matched `case` with message `MessageName` will be executed. The code then checks for a `condition`. If the condition is met, the actor will stop working, i.e., `Behaviours.stopped`. If the condition is not met, the actor will then send an asynchronous message with a name `MessageName` to another actor `from` and retain in the same state, i.e., `Behaviours.same`. In Akka, each specific behaviour can only receive one message type. If that message fails to be understood by the actor, the default behaviour is for the actor to crash and restart.

The default property when sending a message is using at-most-once delivery system. This delivery mechanism means that each message is delivered once or not at all. After the message has been sent, the sender does not require any verification if the message was successfully delivered, nor is there a system for the recipient to handle the unknown messages. In other words, the message will be lost [1]. A different delivery mechanism is at-least-once. This mechanism means for each message sent there could be multiple attempts at delivering it in such a way that at least one succeeds.

At-most-once is very cheap and require the least programming overhead as it just needs to send out the message at most once and it does not care for the delivery status of the message. It can be seen as a "fire and forget" mechanism. At-least-once requires more programming overhead as it is needed to keep track of the lost messages and how to follow up them, while also keeping in mind that multiple messages can be sent when only one message is meant to be sent. Typically this is solved by retrying to send the message to counter the lost of the message and have an acknowledgement[1] mechanism at the receiving end in order to stop the sender from sending more, once the recipient has received the message.

[1]Acknowledgement is a mechanism where the recipient will send a message back to the sender.

### 2.3.1 The Behaviours of Akka Actors

Behaviours of an Akka actor represent the states of the actor as noted in Figure 2.2. One of the core capabilities of the actor model [27] is to react to the message received. Depending on what state an actor is currently in, it can react in different ways when receiving the same message. In this code snippet we have two behaviors: `idle` and `active`. The program execution switches between the two behaviors.

```
object Worker {
    sealed trait Event
    final case class pause() extends Event
    final case class start(obj: Event) extends Event
    private case object stop() extends Event

  // initial state
  def apply(): Behavior[Event] = idle()

  private def idle(): Behavior[Event] =
    Behaviors.receiveMessage[Event] {
      case pause() =>
        Behaviour.same
      case start(evt) =>
        active(evt)
      case _ =>
        Behaviors.unhandled
    }

  private def active(event: Event): Behavior[Event] =
      Behaviors.receiveMessagePartial {
        case pause() =>
         idle()
        case start(evt) =>
          // ... execute event
          active(evt)
        case stop() =>
          behaviour.stopped
      }
}
```

Each behavior has its own set of messages that it can handle. Behavior `idle` can handle the messages `pause` and `start(evt)` and everything else becomes unhandled. Behavior `active` can handle `pause()`, `stop()` and `start(evt)`. We can see that in this example both behaviors can receive `pause()` and `start()`. However, they execute different code depending on the behavior. The object starts by running the `apply()` function, where it initializes the behavior to `idle`. When receiving `start(evt)` it will then switch over to the `active` behaviour and keep the event it is about to handle. If the actor then receives another `start(evt)` message, it will stay in the same behaviour as before and handle the event. The unchanged behavior is represented by simply running the `active(evt)` function while already being inside the `active` state.

In the `idle` state, we see a final case _ with the body of `Behaviors.unhandled`. This is one of three universal behaviors of Akka where it will return to its previous behavior and tell the system that the previous message was not handled. The other two universal behaviors in this example are `Behaviors.same` and `Behaviors.stopped`. The former one is used in the `idle` behavior's `pause()` case. It tells the system to reuse the same behavior and is identical to how we told the system to stay in `active()` while being in the `active` state. The latter one is used in the `active` state's `stop()` case. It shuts down the actor and prevents it from receiving further messages.

## 2.4 Domain-specific Languages and Language Workbenches

A Domain-specific Language (DSL) is a computer language that is targeted to express solutions to a particular kind of problems, unlike general-purpose languages which are aimed at expressing arbitrary software.

A DSL does not attempt to please all. Instead, it is created for a limited sphere of application and use, but it is powerful enough to represent and address the problems and solutions in that sphere. A good example of a DSL is HTML. It is a language for the web application domain. It can't be used for use cases like number crunching, but it is clear how widely used HTML is on the web.

The purpose of a DSL is to capture or document the requirements and behavior of one domain. A DSL's usage might be even narrower for particular aspects within the domain (e.g., commodities trading in finance). This does not imply that a DSL is only for business use. DSLs can bring domain experts and technical teams together. For instance, designers and programmers can use a DSL to communicate and develop an application together [41].

DSLs can be implemented for interpretation or code generation for an addressed domain or problem. Interpretation means to read the DSL script and executing it at run time. Code generation from a DSL is not considered mandatory, as its primary purpose is domain knowledge. But, when it is used, code generation is an advantage in domain engineering. Usually the generated code is in a high level language, such as Java or C.

# Chapter 3

# GameLang

Multithreading can be a very daunting task for new game designers. As demonstrated in the Java language, it is done through creating explicit locks and manually locking and unlocking them. It requires precise knowledge of how programming works, how the software works, how multithreading works and what kind of concurrency issues might occur. Instead of using the explicit locking/unlocking mechanism in multithreading, we simplify the design process for the game designers by utilizing the Actor Model [15, 27], which provides a higher level of abstraction for writing concurrent and distributed systems. However, writing actor-based programs is still not yet trivial for game designers, who are not necessary programmers. They would have to do some form of research to find out which actor-based programming language suits their needs and master the language. This adds an extra layer of complexity to the game design.

In order to assist them further, we develop a domain-specific language (DSL), i.e., *GameLang*, that is easy to use and easy to understand for the game designers, and can be transformed into a real-world actor-based programming language. As shown in Figure 3.1, several programming languages support actor-based concurrency, for instance, Erlang [4, 10, 11], Scala [6, 26], Haskell [5, 29, 30], and ABS [25, 31]. In this thesis, we choose Akka Scala [15, 26, 42] as the end-product language.

## 3.1 The Domain-Specific Language GameLang

*GameLang* is a domain-specific language (DSL) for game designers. The game designers can use *GameLang* to define how the players in a computer game interact with each

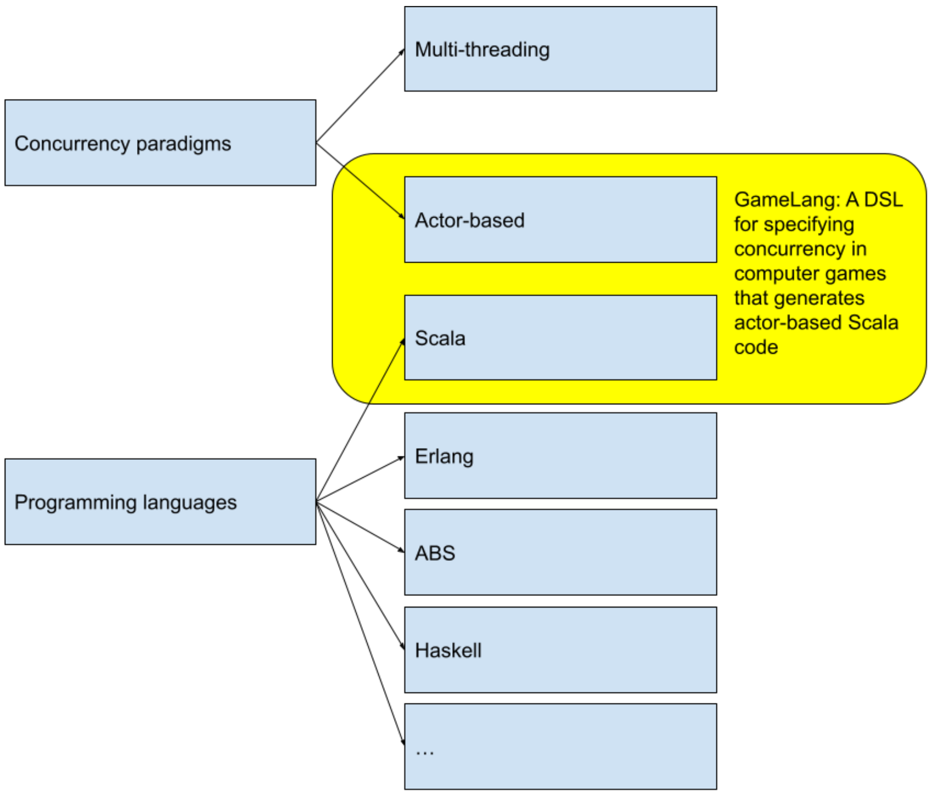
Concurrency paradigms
Multi-threading
Actor-based
Scala
GameLang: A DSL for specifying concurrency in computer games that generates actor-based Scala code
Programming languages
Erlang
ABS
Haskell
...

Figure 3.1: Visualization of how we choose our approach.

other and how the game functions. We aim to create a DSL that will generate Akka Scala programs from *GameLang* programs. This DSL assist the generation of the Scala-related concurrency syntax, which the game designers do not have to master.

### 3.1.1 The GameLang Syntax

In this section, we will introduce the syntax of *GameLang*. A game specification written in *GameLang* can be divided into three parts: the `player` specification, the `action` specification, and the `game` specification, which initializes the game.

```
// players specification
player <Player_Name> { ... }

// actions specifications
action <Action_Name> { ... }

// game initialization specifications
game <Game_Name> { ... }
```

In *GameLang*, a `player` specification corresponds to an actor in Scala, and can similarly have *fields* (of primitive types `integer` and `boolean`) and *functions* (whose bodies contain Scala code). A specification of a `player` additionally contains signatures of `message`s which can be received and sent by the player, as well as a concurrency specification stating which messages are "not compatible" with each other, i.e., which messages cannot be handled concurrently.

```
player <Player_Name> {
  // field declarations
  <Field_Name>: <Type_Name> = <Init_value>

  // function declarations
  func <Function_Name>(<Arguments>): <Return_type> {
    // <Scala_code>
  }

  // message signature declarations
  message <Message_Name>(<Arguments>)

  // behaviour specifications
  message <Message_Name> # message <Message_Name>
}
```

We introduce an operator $\nparallel$ in the concurrency specification in order to differentiate the messages that can not be run concurrently. By default, we assume all messages of an actor can be handled concurrently. The concurrency specification restricts it further. For instance, for two messages $M_1$ and $M_2$ on actor $A$ , an expression $M_1 \nparallel M_2$ means that the messages $M_1$ and $M_2$ cannot be run concurrently. Since all the messages belonging to one behaviour can be handled concurrently, the actor $A$ should have at least two behaviours, one for each message so that the requirement $M_1 \nparallel M_2$ can be fulfilled.

The syntax of the concurrency specification in *GameLang* is shown below. Note that we use notation `#` to denote the $\nparallel$ operator in *GameLang*.

```
receive <Message_Name_1> # receive <Message_Name_2>
```

The `action` specification defines the ordering of messages sent between players, i.e., the `first` player and the `second` player given in the argument and the implicit "`system`", which starts the game.

```
action <Action_Name> (
       first: <Player_Name>, second: <Player_Name>, <Arguments>){
```

```
  <Statements>
}
```

The body of an `action` specification may contain a composed `Statements`, which can either be a conditional, a player's internal method call, or a message passing statement. The general form of a message passing statement is as follows:

```
from A₁ to A₂: M { S }
```

Such a statement specifies an asynchronous message passing of message $M$ from an actor $A_1$ to an actor $A_2$. From actor $A_1$'s point of view, $A_1$ sends the message $M$ to $A_2$, whereas from actor $A_2$'s point of view, $A_2$ will execute statements $S$ after receiving the message $M$ from $A_1$. There are four kinds of *message passing* statements supported by *GameLang*:

- from `system` to the `first` player:

  ```
  from system to first : <Message_Name>(<Arguments>){
     <Statements>
  }
  ```

- from `system` to the `second` player:

  ```
  from system to second : <Message_Name>(<Arguments>){
     <Statements>
  }
  ```

- from the `first` player to the `second` player:

  ```
  from first to second : <Message_Name>(<Arguments>){
     <Statements>
  }
  ```

- from the `second` player to the `first` player:

  ```
  from second to first : <Message_Name>(<Arguments>){
     <Statements>
  }
  ```

### 3.1.2 A GameLang Example

In this section, we present a small example to showcase how to use *GameLang* and what this DSL is capable of. The complete code can be found in Appendix A.1. The `player` block in this example contains the following fields: `score`, `shield`, `xCoordinates`, `yCoordinates`, and `range`. The types of the field are also given.

```
player Prisoner{
    //fields
    score: integer
    shield: boolean
    xCoordinates: integer
    yCoordinates: integer
    range: integer

    //functions
    func CheckCollision(xPos:integer, yPos:integer){
        //check if two players are in viscinity of each other
        ...
    }

    //messages
    message ActorInfo(x: Prisoner)
    message AskToFight(x: integer)
    message ChangeScore(x: integer)

    //concurrency specification
    receive AskToFight(x) # receive ChangeScore(x)
}
```

The communication between the players is specified in the `action` specification below. Both the `first` player and the `second` player are `Prisoners`. The argument `s` is an integer value, which defines the points used in the game.

```
action Fighting(first:Prisoner,second:Prisoner,s:integer){
  from system to first : ActorInfo(second){
    from first to second : AskToFight(s) {
      if(second.CheckIfColliding(first, s)){
        from second to first : ChangeScore(s){}
        second.changeScoreAndCheckShield(s)
        if(s < 0){ second.die() }
        else { from second to first : AskToFight(s) {} }
      }
      else{
        from second to first: Escape
        second.relocate()
      }
    }
  }
}
```

A message `ActorInfo` containing the identity of the `second` player is sent from the `system` to the `first` player. This is in essence a way to start the whole sequence of events that we have defined later. After the `first` player receives the message, it sends a message `AskToFight` to the `second` player.

The `second` player then check if it collides with the `first` player. In other words if they are within a certain distance of one another, i.e., range. If this condition is violated, i.e., the two players are not close enough, the `second` player will relocate its position. Our design principle is that the player who initiates the fight always gains points. So if the two players collide with each other, the `second` player will send a message `ChangeScore` to the `first` player, which allows the `first` player to gain some points, which is defined by the parameter `s`. After the message is asynchronously sent, the `second` player reduces its points by invoking its own function `changeScoreAndCheckShield`, which checks if there is a shield to protect itself or not. More points are lost if there is no shield. If the total points of the `second` player are still larger than zero after losing `s` points, then the `second` player will continue fighting by sending another message `AskToFight` to the `first` player. The fighting cycle then repeats again but this time in the opposite direction, i.e., from `second` to `first`. If the total points of the `second` player is unfortunately below zero after losing `s` points, the `second` player dies and the battle ends.

We design a game, in which an actor always gains points by initializing a fight and always loses points when defending. So if an actor $A$ fights back with an actor $C$ while initializing a fight with an actor $B$, it might die unnecessarily because the point gaining happens too late, although the fight with actor $B$ was started first. If this situation is left unchecked, it will lead to race conditions and points will not be calculated correctly. So we define the concurrency specification `receive AskToFight(x)` $\not\parallel$ `receive ChangeScore(x)` in the `player` block to avoid race conditions.

## 3.2 The Workflow

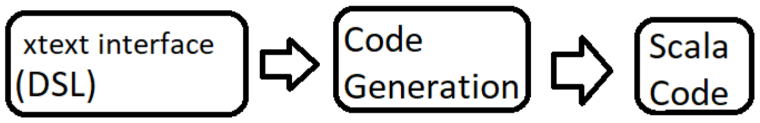

Figure 3.2: The workflow from the DSL to the Scala code

Figure 3.2 presents the workflow from our DSL to the Scala code. It starts with the users specifying a computer game in the DSL *GameLang*, which contains the `player` specifications, the `action` specifications, and the `game` specification. The `player` specifications and the `action` specifications together would contribute to the code generation of Akka actors. The `game` specification defines the initializetion of a game. The grammar of `GameLang` is written in Xtext. There are still some research problems relevant for this code generation to be solved. In the next chapter we discuss how the code generation would be implemented. The complete code generation is left for the future work.

# Chapter 4

# Implementation

In this chapter we will look at how the *GameLang* workflow is implemented, specifically how we map DSL constructs to Scala code. The source code of *GameLang* is available at GitHub[1].

## 4.1 Xtext Grammar for the GameLang Language

The grammar of the GameLang Language is implemented in Xtext. The `player` specification consists of `fields`, `functions`, `messages` and `concurrencySpecs`. We will explain them one by one in the following.

```
Player:
    "player" name=ID "{"
        fields += Field*
        functions += Function*
        messages += Message*
        concurrencySpecs += ConcurrencySpec*
    "}"
```

Every `player` has an ID. The `fields` defines variables, which can only be a number, a string text or a boolean variable.

```
Field:
  NumberField | TextField | BooleanField
;
```

[1]`https://github.com/metrolink/GameLang/`

The `func` defines functions. A function argument `FunctionArgument` is a parameter to the function being created or called. The function argument is optional and may contain values with types `number`, `string`, or `boolean`. A function may return a `number`, a `string`, a `boolean` value or nothing. Similar to Java, the return type of a function which returns nothing is `void`. The function body in `player` is one-to-one mapped to Scala.

```
Function:
  "func" name=ID "("
    (arguments += FunctionArgument ("," arguments += FunctionArgument)
   ↪ *)?
  ")" ":" returnType=("number" | "string" | "boolean" | "void")
  "{"
    scalaCode = ScalaCode
  "}"
;
```

```
FunctionArgument:
  name=ID ":" argType=("number" | "string" | "boolean")
;
```

`Messages` are a collection of the messages that a `player` can receive.

```
Message:
  "message" name=ID "("
    (arguments += MessageArgument ("," arguments += MessageArgument)*)?
  ")"
;
```

A message is created with either none or multiple arguments, which can be of types `number`, `string`, `boolean`, or `player`.

```
MessageArgument:
  name=ID ":" (argType=("number" | "string" | "boolean") | argTypeRef=[
   ↪ Player])
;
```

The concurrency specification `concurrencySpecs` defines which two messages cannot be handled concurrently by a player. By default, all the messages can be handled concurrently by a player. However, this may not always be the case, as the game designers can restrict it by specifying those that cannot be handled concurrently in the `concurrencySpecs`. An Akka actor can have one behaviour at a time, so if we generate Scala code by assigning each one of the two messages in two separate behaviours, it is guaranteed that these two messages will not be handled concurrently.

```
ConcurrencySpec:
  "receive" msgLeft=[Message]
  isNotParallel?="#"
  "receive" msgRight=[Message]
;
```

The action describes a sequence of message sending and message receiving between the first player, the second player and the system.

```
Action:
    "action" name=ID "("
        "first" ":" firstPlayerType=[Player]
        ","
        "second" ":" secondPlayerType=[Player]
        ("," otherArgs += MessageArgument (","
            otherArgs += MessageArgument)*)?")"
        "{"statements += Statement*"}"
;
```

Inside the body of the action we can have communication statements, if-and-else statements or function call statements.

```
Statement:
  CommunicationStatement |
  IfStatement |
  FunctionCallStatement
;
```

A communication statement is a "from sender to receiver" statement. The feature sender can either be the system, the first player or the second player. The system is purely meant to be the very first sender and not utilized in subsequent from-to statements beyond the first statement. The feature receiver represents the receiving player that will receive the message, which is specified in the feature msgName. Note that it is not allowed to send messages to the system. The feature statements specifies the actions which will be performed by the receiver.

```
CommunicationStatement:
  "from" sender=("system"|"first"|"second") "to" receiver=("first"|"
   ↪ second")
  ":" msgName=[Message]"(" ")"
  "{"
    statements += Statement*
  "}"
;
```

The FunctionCallStatement specifies the internal function of a player. It should be the receiving player that calls the function.

```
FunctionCallStatement:
  ("first"|"second") "." functionName=[Function]
;
```

The if statement can be found in many other languages such as Java. If the condition is evaluated to true then the trueBranchStatements will be executed, if the condition is violated, the falseBranchStatements will be executed.

```
IfStatement:
  "if" "(" cond=Expr ")" "{"
    trueBranchStatements += Statement*
  "}"
  "else" "{"
    falseBranchStatements += Statement*
  "}"
;
```

Model is the top level view of what is in the game. It contains the implementation of players, the actions between the players, and the declaration of the game.

```
Model:
    players += Player*
    actions += Action*
    gameDeclaration = GameDeclaration
;
```

The GameDeclaration gives a name to the game and starts the game by initializes the first sequence of events.

```
GameDeclaration:
  "game" name=ID "{" "}"
;
```

As our DSL is based on Akka Scala, we also provide the opportunity to write Scala code directly in the DSL.

```
ScalaCode:
  "scala" "{"
    code=STRING
  "}"
;
```

## 4.2 Mapping DSL Constructs to Scala Syntax

In this section, we list our suggestion for the one-to-one mappings for code generation in Table 4.1. They are mappings from the *GameLang* DSL construct to the Scala syntax. An example of how the DSL can be mapped to a complete Scala code can be found in Appendix B.1.

Table 4.1: Mapping from the DSL to the Scala syntax

| **Constructs in the DSL** | **Corresponding Scala syntax** | **Explanation** |
|---|---|---|
| `player Player { ... }` | `object Player { ... }` `class Player { ... }` | A player specification corresponds to a Scala class and an object accompanying this class with the required messages. |
| `x: integer;` `b: boolean;` `w: string;` | `var x: Int;` `var b: Boolean;` `var w: String;` | Each field declaration corresponds to a Scala variable declaration. We do this by assigning a `:Type` modifier at the end of each type corresponding to the type in the DSL. It will always be `var` in the Scala code. |
| `message Message(` `arg1: integer,` `argN: string)` | `final case class` `msg_Message(` `arg1: Int,` `argN: String)` `extends msgType_T` | In the generated Scala code, every message is represented as a *case class* that extends sealed trait `msgType_T`. The generated case classes appear in the companion object of the corresponding player class. Arguments of the message specified in the DSL are converted to case class's arguments. |
| `func f(): integer {` `// code` `}` | `def f(): Int = {` `// code` `}` | The DSL supports `integer`, `boolean`, `string` and `void` as return types of functions; in the generated Scala code they correspond to `Int`, `Boolean`, `String` and `Unit`, respectively. |

| | | |
|---|---|---|
| `action ActionName(`<br>` first:Player1, second:Player2)`<br>`{`<br>`   from system to first: M(){`<br>`    ...`<br>`   from first to second: M_2(){`<br>`    ...`<br>`   }`<br>`  }`<br>`}` | The corresponding mapping is explained in the row below. | The `action` statement specifies a message passing chain between the players. The body of the actions consists of `from-to` statements and their corresponding body.<br>The order of the `from-to` statements corresponds to the order of events that will happen in the computer game. |

| | | |
|---|---|---|
| `from` $P_1$ `to` $P_2$ `: M(){`<br>`    // stmts`<br>`}` | `class` $P_1$ `{`<br>`  ...`<br>`  case ... => {`<br>`    ` $P_2$ `! M();`<br>`  ...`<br>`}`<br><br>`class` $P_2$ `{`<br>`   ...`<br>`   case M => {`<br>`      // stmts`<br>`   }`<br>`   ...`<br>`}` | The `from-to` statement specifies that a message `M` is being asynchronously sent from actor $P_1$ to actor $P_2$, and actor $P_2$ is receiving the message. In the generated Scala code, this manifests in classes corresponding to actors (i.e., players) $P_1$ and $P_2$. In the class corresponding to $P_1$, a statement $P_2$ `! M();` appears in one of the actor $P_1$'s message handling cases. In class corresponding to $P_2$, a new case for handling message `M` appears and its code is the one specified in the body of the `from-to` statement in the DSL. |
| `game GameName{`<br>` player P1();`<br>`}` | `object myGame{`<br>` playerName instanceOfPlayer();`<br>`}` | The game object lists all relevant objects that will be created at game start. This includes the starting actors of the game as well as their initial values. |

| | | |
|---|---|---|
| | `object SystemStart extends App`<br>`↪ {`<br>`val mainSystem :`<br>`↪ ActorSystem [ myGame .`<br>`↪ StartGame ] = ActorSystem`<br>`↪  ( system () ,`<br>`" AkkaQuickStart ")`<br><br>`mainSystem ! myGame .`<br>`↪ StartGame ()`<br>`}` | The Scala code that starts the game is the same for all the systems. So we hide it in the DSL. The code first creates the actor system and then sends the initial message to the game object. |

### 4.2.1 Self Loops

In an asynchronous setting, we cannot control when a message will be received and when it will be handled by an actor, except that the ordering of the messages coming from the same sender is preserved by the Scala language. A message may get lost if the actor is in a behaviour which does not define how to handle the message upon the message arrival. In order to prevent lost messages, our proposed solution to solve this for a future code generator would be to make sure that all behaviours of an actor define a case for each message. For the messages that should not be handled within a specific behaviour, the code generator would provide a *self-loop* to resend the message to itself again and store the resent message to its mailbox so that the message does not get lost and is postponed for a later use. This can be done by moving a message from the front of the mailbox of an actor to the back of the mailbox, essentially making it an at-least-once system.

### 4.2.2 Group Messages into Different Behaviours

In an asynchronous setting, we cannot control when a message will be received and when it will be handled by an actor, except that the ordering of the messages coming from the same sender is preserved by the Scala language. We can however use the not-concurrent operator, i.e., $\not\parallel$, in the `ConcurrencySpec` to constraint which two messages can not be executed concurrently, i.e., should not belong to the same behaviour. Initially, all the messages are grouped in the same behaviour and stored in a list. Then new behaviours containing new set of messages will be generated based on what is expressed in the `ConcurrencySpec`. The algorithm of how our code generator would group messages into different behaviours is shown in Listing 4.16 and explained below.

```
Set<ConcurrencySpec> C
Set<Messages> M

func behaviourList(M, C):
    LS = [M]
    for each M1 ∦ M2 in C{
        LS' = []
        for each B in LS{
            B1 = B
            B2 = B
            mark(M1, B1)
            mark(M2, B2)
            add(B1, B2, LS')
14          }
15          LS = LS'
16      }
17      return minimize(LS)
```

Listing 4.16: Algorithm for splitting the behaviours

The function `behaviourList` takes two input values, i.e., a set of messages `M` and a set of concurrency specification `C`. It returns a list of behaviours. Line 5 creates a list `LS`, which contains a set of messages `M`. Line 6 loops through each of the term in the concurrency specification, for instance, `M1` $\nparallel$ `M2`. Line 8 loops through each of the message sets (identified by behaviours). Each behaviour is represented by a set of marked or unmarked messages. The marking symbolizing self-loops. Initially, there is no markings over the messages. Lines 9 and 10 make two copies of the message set `B`. Lines 11 and 12 mark `M1` in one of the copies and mark `M2` in the other copy, respectively. This means either `M1` or `M2` has no progress other than being resent to itself. Consequently, we avoid `M1` and `M2` to be handled concurrently in either of the assigned behaviours and at the same time we avoid losing messages. Line 13 adds the two new message sets `B1` and `B2` to the behaviour list `LS'`. Note that when the nested for-loops terminates, some of the behaviours in the list `LS` might be supersets of the others. The subsets are redundant. For instance, we say a behaviour containing messages $\overline{\mathtt{A}}\ \overline{\mathtt{B}}\ \mathtt{C}\ \mathtt{D}$ is a superset of a behaviour containing messages $\overline{\mathtt{A}}\ \overline{\mathtt{B}}\ \overline{\mathtt{C}}\ \mathtt{D}$ because the former one can handle all the messages that the latter one handles, i.e., message `D`, but even more. Finally, the function `behaviourList` returns a minimized list of behaviours, in which none of the behaviours is a superset of another.

**Example.** As an example for how the algorithm defines messages in different behaviour groups, say we have four messages `A`, `B`, `C`, `D`, and two concurrency specifications `A` $\nparallel$ `C` and `B` $\nparallel$ `D`. In the beginning, `A`, `B`, `C`, and `D` belong to the same behaviour. We specify that the messages `A` and `C` cannot be in the same behaviour, and the messages `B` and `D` cannot be in the same behaviour.

```
message set:
A B C D

concurrency specification:
A ∦ C
B ∦ D
```

Note that a message may get lost if the actor is in a behaviour which does not define how to handle the message upon the message arrival. In order to prevent message lost, our proposed code generator makes sure that all behaviours of an actor define a case for each type of message. See the at-least-once mechanism defined in Section 1. We use overlines to mark which messages in a behaviour should be resent in a `self-loops`.

The two behaviours below are the result of applying the concurrency specification $\mathtt{A} \nparallel \mathtt{C}$ to messages `ABCD`.

```
message sets:
A̅ B C D
A B C̅ D

concurrency specification:
B ∦ D
```

Finally, the four final behaviours below are the result of applying the concurrency specification $\mathtt{B} \nparallel \mathtt{D}$ to the message sets $\overline{\mathtt{A}}\mathtt{BCD}$ and $\mathtt{AB}\overline{\mathtt{C}}\mathtt{D}$.

```
message sets:
A̅ B̅ C D
A̅ B C D̅
A B̅ C̅ D
A B C̅ D̅
```

### 4.2.3 Behaviour Switching

In Akka Scala, an actor is the basic building block of concurrent computation. The behaviour of an actor defines the set of messages an actor can handle and react to. In this thesis work, we develop a DSL to express which pairs of messages cannot be executed concurrently in *GameLang*, based on which the code generator will separate messages into different behaviours as discussed in Section 4.2.2. This also means an actor may need to switch its behaviours in order to receive the next expected message.

As discussed in Section 2, the concept and the construct of Scala behaviours is not relevant for the game design and should be abstracted away in our DSL and be taken care of by the code generator. The possible approaches to generate Scala code for behaviour switching from our DSL is discussed as follows. First, the code generator needs to identify which message the actor should receive next. This can be done by parsing the `action` specifications in the DSL code and apply static analysis [37]. Next, the code generator should determine a behaviour which contains the identified message. However, a message may be supported by more than one behaviours, as the example shown in Section 4.2.2. So, how does the code generator choose between several possible behaviours for behaviour switching? A rough guideline is given below:

- In case that the current behaviour also contains the next expected message, the actor will not switch the behaviour. Otherwise, the following cases describe how the behaviour is switched.
- Assume among the set $\{B_1, \ldots, B_k\}$ of behaviours, one of the behaviours $B_i$ is deemed clearly better than the others[2]. In that case, the algorithm will switch to $B_i$.
- Assume that all behaviours $\{B_1, \ldots, B_k\}$ have an equal amount of non-self-loop messages. In this case, all the behaviours are valid choices to switch to, and the algorithm makes a randomized choice.

Finally, a statement stating the next behaviour is added by the code generator into the current case of the message handling.

In a game that there are multiple actors trying to communicate with the same actor, the receiving actor needs a way to process the receiving messages in a manner that do not create concurrency issues. Although each of the `action` block only specifies the interaction between two players, the number of players can scale up. There can be naturally several players in a game which communicating with each other. Which also means, there can be interleaving of messages. The concurrency specification using the $\nparallel$ operator in this case constrains the concurrency behaviour of an actor.

### 4.2.4 Code Generation in Xtext

The language workbench Xtext provides built-in code generation capabilities via model-to-text transformations [41]. Based on the grammar of the language, Xtext generates an object model that represents the language constructs, which is populated during parsing. To specify code generation in Xtext, one needs to define a polymorphic function (such

[2]A possible scenario where one of the behaviours is clearly superior, is when $B_1$ has four non-self-loop messages, $B_2$ has two non-self-loop messages, and $B_3$ has three non-self-loop messages. In this case $B_1$ is superior as it is less risky for the actor having to switch behaviour again in the near future.

as `compile`) for every language construct[3]; this function should return a string that represents the generated code for the particular language construct. The bodies of the `compile` functions follow the description presented in Table 4.1.

[3]That is, for every node type in the syntax tree of a program written in the language.

# Chapter 5

# Related Work

**Actor-based concurrency** The actor-based concurrency is very different from the thread-based concurrency with locks. Isolated states and asynchronous message passing yield another programming pattern than what the traditional multithreading do. Isolated mutable states and immutable messages together guarantees implicit synchronization. However, the concept of asynchronous message passing and no global state challenges the coordination. An application may require consensus or a concerted view of state between multiple actors. When multiple actors must be strictly orchestrated in order to provide a distinct application function, correct messaging can become very demanding. Thus, many implementations provide higher-level abstractions that implement low-level coordination protocols based on complex message flows, but hide the internal complexity from the developer. For Erlang, OTP is a standard library that contains a rich set of abstractions, generic protocol implementations and behaviors [16]. The Erlang design philosophy is to spawn a new process for every event so that the program structure directly reflects the concurrency of multiple users exchanging messages. So with lightweight processes, it is not unusual to have hundreds of thousands, even millions, of processes running in parallel, often with a small memory footprint. The ability of the runtime system to scale concurrency to these levels directly affects the way programs are developed, differentiating Erlang from other concurrent programming languages [12].

**ABS** The work by Kamburjan et. al. [33] proposed a framework to statically verify communication correctness in a actor-based concurrency model of ABS using futures. The framework provided a type discipline based on session types, which gives a high-level abstraction for structured interactions. By using it the users of the framework can statically verify if the local implementations in ABS comply with the communication correctness. In this work, the programmers need to implement the ABS concurrent programs themselves. No code generation was involved.

**Session types** We are not aware of works on session types that perform complete code generation to Scala. However, there have been several works [13, 32, 39, 38] in generating Scala APIs from session types. With the use of the generated API, the Scala programs written by the programmers can be type checked at the compile time against the communication protocol defined as session types.

Chor [3] is a programming framework for choreographic programming [14, 36]. Chor provides an IDE for programming with choreographies, equipped with a type checker for verifying that choreographies respect protocol specifications given as session types. Programs in Chor can be compiled to executable endpoint implementation in the Jolie programming language, a general-purpose language for distributed computing, which is extended to support the development of multiparty asynchronous sessions.

Multiparty asynchronous session types [28] extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. It introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual peers.

**Computer games** Compared to other software applications where concurrency is easier to predict and handle such as in Listing 2.1, computer games differ by providing gameplay, which has been described in number of different ways though it is most commonly described as the way the player handles the game and the rules. These game rules exist in all computer games, and they could as such serve as unifying element and provide a common ground for all computer games [35]. Through this work it is possible to identify the main properties of a computer game, and develop concurrency on top of this system.

**Game engines** Through understanding the game engine architecture it is possible to identify the main properties of a computer game, and develop concurrency on top of this system. The state-of-the-art game engine *Unreal* [22] is implemented in *C++*. Since the programming language used in most modern game engines is *C++*[24], we should briefly consider whether *C++*'s native start-up and shut-down semantics can be leveraged in order to start up and shut down the engine's subsystems. In *C++*, global and static objects are constructed before the program's entry point is called. However, these constructors are called in a totally unpredictable order. The destructors of global and static class instances are called after the entry point returns, and once again they are called in an unpredictable order. This behavior is not desirable for initializing and shutting down the subsystems of a game engine, or indeed any software system that has interdependencies between its global objects such as Akka actors. This is problematic for systems that could utilize concurrency, because a common design pattern for implementing major subsystems such as the ones that make up a game engine is to define a singleton class (often called a manager) for each subsystem. If *C++* gave us more control over the order in which global and static class instances were constructed and destroyed, we could define our singleton instances as globals, without the need for dynamic memory allocation. If this was the case, then identifying and creating concurrency in computer games would be a lot simpler.

**Paradigms used in game development** A common programming paradigm in computer game design is object-oriented design (OOD), though data-oriented design (DOD) is receiving traction as games seek to be more optimized. Research has shown that OOD and DOD are well used in games. Both have their advantages and disadvantages in any part, such as performance, maintainability, entry-level and other aspects that architects and managers usually face when choosing a technology stack, design principles, and team building [20]. While OOD was found to be much easier to understand for juniors, the DOD approach was found to require knowledge of computer science to understand, in particular how memory and the OS work. DOD allows for the ability to design and develop software with effective use of multicore processors due to the uniform distribution of load across multiple threads.

**Language workbenches** Language workbenches are a generic term that refers to the tools that implement the idea of language-oriented programming. That in turn is the general style of development which operates about the idea of building software around a set of domain-specific languages [21]. This thesis seeks to utilize this technology in conjunction with concurrency and game engines to develop a new tool that helps game designers.

Language workbenches exist in many different flavors with a common goal to facilitating the development of (domain-specific) languages [17]. Common features of language workbenches include notation, semantics, validation, testing and composability as well as an editor. A language workbench must support notation, semantics, and an editor for the defined languages and its models. It may support validation of models, testing and debugging of models and the language definition, as well as composition of different aspects of multiple defined languages. Xtext, which is our language workbench of choice, is characterized by the following features [18]:

- textual notation
- interpretative as well as model2text and model2model semantics
- structural, type and programmatic validation
- DSL testing and debugging
- free-form editing mode
- syntax highlighting, outline, folding, syntactic completion, diff and autoformatting
- semantic reference resolution, semantic completion, error marking, quickfixes, origin tracking and live translation

# Chapter 6

# Conclusion and Future Work

In this thesis, I have studied how actor-based concurrency in Akka Scala can be used to implement a game genre of computer games. I used a genre known as point chasers to simulate a simple game. In this example, each player is implemented as an actor and the players communicate with each other through asynchronous message passing. As point chasers are interacting with each other and at the same time influencing each others scores, we found this to be a good example to show how actor-based concurrency can be used to express similar types of computer games.

We designed a DSL which is called *GameLang* to help game designers generate actor-based concurrent computer games without having to master all the technical details of the Akka library. We have abstracted the concepts of behaviours and states in Akka as well as simplified the terminology to closer resemble terms in a computer game. In our DSL we have explored a way to describe the order of events in the computer games, as well as an approach to express concurrent communication. These are expressed via the `action` blocks, which use `from-to` statements to specify message sending from one actor to another, as well as the concurrency operator $\not\parallel$, respectively. In order to find the best implementation of the concurrency specification we have created an algorithm that will generate behaviours. These behaviours will contain groups of messages that best fits the action.

## Future work

We have identified the following directions of future work.

- As behaviours of an actor are not relevant for the game design, we want to apply static analysis in the code generation to identify the necessary behaviour switching and generate the corresponding Scala code in the future. If the designers have the technical knowledge, it is possible to further inspect the generated Scala code and make further adjustments.
- We want to evaluate our DSL by conducting user studies, mainly aimed at game designers and beginner programmers who just start their studies, as this group has little to no game developing experience. We believe that this group is the most accurate representation of our target user group.
- We want to extend our approach to support other game genres. Though it is already possible to specify arbitrary games within our system, the tools to do so are somewhat limited. Other genres of interest would generally still be 2D games with text-based interfaces and minimal graphical interfaces. Given more DSL-level support, it will be possible to specify turn-based games, such as chess or Pong[1].
- We would like to investigate how our approach can be applied to other actor-based languages, such as Erlang [9], ABS [31] and Haskell [34]. As the concepts of actor-based programming are universal and the concepts of utilizing actors as players in games is not a unique concept bound to Akka, it should be possible to create various translation layers from our DSL to other actor-based languages.
- Finally, we plan to implement extensive IDE support for the DSL.

[1]`https://en.wikipedia.org/wiki/Pong`

# Bibliography

[1] Message Delivery Reliaibility.
**URL:** https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html.

[2] Multithreading in Java - Everything You Must Know.
**URL:** https://www.digitalocean.com/community/tutorials/multithreading-in-java/.

[3] Chor.
**URL:** http://www.chor-lang.org/.

[4] Erlang.
**URL:** https://www.erlang.org/.

[5] Haskell.
**URL:** https://www.haskell.org/.

[6] Scala.
**URL:** https://www.scala-lang.org/.

[7] What is a thread?
**URL:** https://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/definition.html.

[8] 7Geordi. How do you handle concurrency in your games?
**URL:** https://www.reddit.com/r/gamedev/comments/108fjg/how_do_you_handle_concurrency_in_your_games/c6bhvff/.

[9] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.
**URL:** https://nbn-resolving.org/urn:nbn:se:kth:diva-3658.

[10] Joe Armstrong. Erlang - Software for a Concurrent World. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, page 1. Springer, 2007.

[11] Joe Armstrong. A History of Erlang. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–26. ACM, 2007.

[12] Francesca Cesarini and Simon Thompson. *Erlang Programming - A Concurrent Approach to Software Development.* O'Reilly, 2009. ISBN 978-0-596-51818-9.
**URL:** `http://www.oreilly.de/catalog/9780596518189/index.html`.

[13] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[14] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A Formal Theory of Choreographic Programming. *CoRR*, 2022.

[15] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A Survey of Active Object Languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017. doi: 10.1145/3122848.
**URL:** `https://doi.org/10.1145/3122848`.

[16] Benjamin Erb. Concurrent Programming for Scalable Web Architectures. In *Informatiktage 2012 - Fachwissenschaftlicher Informatik-Kongress 23. und 24. März 2012, B-IT Bonn-Aachen International Center for Information Technology in Bonn*, volume S-11 of *LNI*, pages 139–142. GI, 2012.

[17] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language*

*Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi: 10.1007/978-3-319-02654-1_11.
**URL:** `https://doi.org/10.1007/978-3-319-02654-1_11`.

[18] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Comput. Lang. Syst. Struct.*, 44:24–47, 2015. doi: 10.1016/j.cl.2015.08.007.
**URL:** `https://doi.org/10.1016/j.cl.2015.08.007`.

[19] Eleonora Fanouraki. Did you know that 60% of game developers use game engines?
**URL:** `https://www.slashdata.co/blog/did-you-know-that-60-of-game-developers-use-game-engines`.

[20] Kirill Fedoseev, Nursultan Askarbekuly, Ekaterina Uzbekova, and Manuel Mazzara. A Case Study on Object-Oriented and Data-Oriented Design Paradigms in Game Development. 07 2020. doi: 10.13140/RG.2.2.16657.66405.

[21] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?
**URL:** `https://www.martinfowler.com/articles/languageWorkbench.html`.

[22] Epic Games, 2023.
**URL:** `https://www.unrealengine.com/en-US/`.

[23] Ian Gordon. *Foundation of scalable systems.* O'Reilly Media, Inc., 2022. ISBN 9781098106065.

[24] Jason Gregory. *Game Engine Architecture.* CRC Press, 2018. ISBN 978-1-1380-3545-4.

[25] Reiner Hähnle. The Abstract Behavioral Specification Language: A Tutorial Introduction. In Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2012.

[26] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[27] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.
**URL:** `http://ijcai.org/Proceedings/73/Papers/027B.pdf`.

[28] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[29] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional language. *ACM SIGPLAN Notices*, 27(5):1, 1992.

[30] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy With Class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–55. ACM, 2007.

[31] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

[32] Sung-Shik Jongmans and José Proença. ST4MP: A Blueprint of Multiparty Session Typing for Multilingual Programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, volume 13701 of *Lecture Notes in Computer Science*, pages 460–478. Springer, 2022.

[33] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-Based Compositional Analysis for Actor-Based Languages Using Futures. In Kazuhiro Ogata, Mark Lawford, and Shaoying Liu, editors, *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016.

[34] Konrad Kleczkowski. Actor Model in Haskell.
**URL:** `https://kleczkow.ski/actor-model-in-haskell/`.

[35] Lars V Magnusson. Game Mechanics Engine. 1 2011. doi: 10.13140/RG.2.2.16657.66405.

[36] Fabrizio Montesi. Choreographic Programming (Ph.D. thesis). 2013.

[37] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN 978-3-540-65410-0. doi: 10.1007/978-3-662-03811-6.
**URL:** `https://doi.org/10.1007/978-3-662-03811-6%7D`.

[38] Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[39] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[40] SciencetechEasy. Life Cycle of Thread in Java — Thread State, 2020.
**URL:** `https://www.scientecheasy.com/2020/08/life-cycle-of-thread-in-java.html/`.

[41] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and GH Wachsmuth. *DSL engineering-designing, implementing and using domain-specific languages*. 2013.

[42] Derek Wyatt. *Akka Concurrency*. Artima Inc, 2013. ISBN 0981531660.

# Appendix A

## The Complete GameLang Example

```
game myGame{
    Prisoner P1
    Prisoner P2
    Prisoner P3

    action fighting(P1,P2,1000)
    action fighting(P1,P3,500)
}
player Prisoner{
    //fields
    score: integer
    shield: boolean
    xCoordinates: integer
    yCoordinates: integer
    range: integer

    //functions
    func CheckCollision(xPos:integer, yPos:integer){
        //check if two players are in viscinity of each other
        ...
    }

    //messages
    message ActorInfo(x: Prisoner)
    message AskToFight(x: integer)
    message ChangeScore(x: integer)

    //concurrency specification
    receive AskToFight(x) # receive ChangeScore(x)
}

```

```
action Fighting(first:Prisoner,second:Prisoner,s:integer){
  from system to first : ActorInfo(second){
    from first to second : AskToFight(s) {
      if(second.CheckIfColliding(first, s)){
        from second to first : ChangeScore(s){}
        second.changeScoreAndCheckShield(s)
        if(s < 0){ second.die() }
        else {
           from second to first : AskToFight(s) {}
        }
      }
      else{
        from second to first: Escape
        second.relocate()
      }
    }
  }
}
```

Listing A.1: The complete GameLang Example in Section 3.1.2

# Appendix B

## The Complete Scala Code Mapped from GameLang

```
//full example of the Scala code

import akka.actor.typed.{ActorRef, ActorSystem, Behavior}
import akka.actor.typed.scaladsl.{ActorContext, Behaviors}
import akka.util.Timeout
import com.typesafe.config.ConfigFactory

import scala.concurrent.duration._
import scala.language.postfixOps
import scala.util.Random


object Prisoner{

  sealed trait msgType_T

  final case class msg_AskToFight(replyTo: ActorRef[msgType_T],xPos:Int
   ↪ , yPos:Int, point: Int) extends msgType_T

  final case class msg_ActorInfo(name: ActorRef[msgType_T], point: Int)
   ↪  extends msgType_T

  final case class msg_ChangeTheScore(point: Int) extends msgType_T

  def apply(): Behavior[msgType_T] = {
    Behaviors.setup(context => new Prisoner(context).behaviour_B2())
  }
}

class Prisoner(context: ActorContext[Prisoner.msgType_T]) {

```

```
import Prisoner._

val rand = new scala.util.Random
var score :Int = 2000
var shield :Boolean= true
var position = Array.ofDim[Int](2) //(x,y)
val positionRange :Int= 5
xCoordinate :Int= rand.between(1, 10) //x cordinates
yCoordinate :Int= rand.between(1, 10) //y cordinates

def relocate(): Unit = {
  xCoordinate = rand.between(1, 10)
  yCoordinate = rand.between(1, 10)
}

def ChangeScoreAndCheckShield(point:Int): Unit = {
  if (shield) {
    score -= point/2
    shield = false
  }
  else {
    score -= point * 2
    println(context.self.toString + " lost points and now have: " +
↪ score)
  }
}

def CheckIfColliding(xPosition: Int, yPosition: Int): Boolean = {
  if (xPosition <= xCoordinate + positionRange && xPosition >=
↪ xCoordinate - positionRange //check if players are close
    && yPosition <= yCoordinate + positionRange && yPosition >=
↪ yCoordinate - positionRange){
    return true
  }
  else{
    return false
  }
}

def behaviour_B1(): Behavior[msgType_T] = {
  Behaviors.receiveMessagePartial {
    case msg_ChangeTheScore(point) =>
      score += point
      println(context.self.toString + "now has " + score)
      behaviour_B2 //Change behavior

    case msg_AskToFight(replyTo, xPos, yPos, point) =>
      xCoordinate = rand.between(1, 10)
      yCoordinate = rand.between(1, 10)
      //Pushes the current message to the back of the mailbox.
      context.self ! msg_AskToFight(replyTo, xPos, yPos, point)
      behaviour_B1 //Behavior.same

    case msg_ActorInfo(name, point) =>
      name ! msg_AskToFight(context.self, xCoordinate, yCoordinate,
 ↪ point)
      behaviour_B1 //Behavior.same
  }
}

def behaviour_B2(): Behavior[msgType_T] = {
  Behaviors.receiveMessagePartial {
    case msg_AskToFight(replyTo, xPos, yPos, point) =>
      if (CheckIfColliding(xPos,yPos)) {
        replyTo ! msg_ChangeTheScore(point)

          ChangeScoreAndCheckShield(point)


        if (score < 0) {
          println(context.self.toString + " stopped")
          Behaviors.stopped
        }
        else {
          replyTo ! msg_AskToFight(context.self, xCoordinate,
 ↪ yCoordinate, point)

          behaviour_B1
        }
      } else {
        replyTo ! msg_ChangeTheScore(0)
        relocate()
        behaviour_B2
      }


    case msg_ChangeTheScore(point) =>
      //Pushes the current message to the back of the mailbox queue.
      context.self ! msg_ChangeTheScore(point)
      behaviour_B2 //Behavior.same

      case msg_ActorInfo(name, point) =>
        name ! msg_AskToFight(context.self, xCoordinate, yCoordinate,
  ↪ point)
        behaviour_B1 //Change the behavior
    }
  }
}

object Prison {

  final case class StartGame()
  val change_points :Int= 500
  def apply(): Behavior[StartGame] =
    Behaviors.setup { context =>
      //create-actors
      val prisoner = context.spawn(Prisoner(), "P1")
      val prisoner2 = context.spawn(Prisoner(), "P2")
      val prisoner3 = context.spawn(Prisoner(), "P3")

      Behaviors.receiveMessage { message =>
        //Send messages to P2 and P3
        prisoner ! Prisoner.msg_ActorInfo(prisoner2, change_points)
        prisoner ! Prisoner.msg_ActorInfo(prisoner3, change_points)
        Behaviors.same
      }
    }
}

//main-class
object AkkaQuickstart extends App {

  val customConf = ConfigFactory.parseString(
    """
    akka.log-dead-letters = OFF
    akka.log-dead-letters-during-shutdown = false
    """)
  // create the actor-system
  val prisonMain: ActorSystem[Prison.StartGame] = ActorSystem(Prison(),
  ↪  "AkkaQuickStart", ConfigFactory.load(customConf))

  prisonMain ! Prison.StartGame()
}
```

Listing B.1: The corresponding Scala code for the prisoner example in Section 3.1.2