ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche

ScaFi: Integration and Performance Analysis with Scala Native

Tesi di laurea in
(Programmazione ad Oggetti)

Relatore
Prof. MIRKO VIROLI

Correlatore
Dr. GIANLUCA AGUZZI

Candidato
KEVIN MANCINI

Quarta Sessione di Laurea
Anno Accademico 2021-2022

Abstract

Aggregate Computing is an emerging paradigm for complex distributed systems where a vast number of distributed devices are involved in a global computation and must cooperate to produce a collective result. This situation is common in the Internet of Things, large-scale urban events, drone coordination and smart cities. Modern Aggregate Computing APIs are normally based on the Field Calculus that offers the basis for the global-to-local computation abstraction, providing Computational Fields. Moreover, these APIs also rely on abstraction layers that hide the complexity of the environment from the sight of the developer (complexity "hidden under the hood"), offering a simple and friendly way to develop this kind of applications. An Internal Domain-specific language that offers these features is Scala with Computational Fields (ScaFi), a Scala framework implementing aggregate programming mechanisms. A critical concept for these types of libraries is portability since their nature implies the possibility of being run over a wide range of different devices. The work shown in this thesis offers a solution to improve the portability and flexibility of ScaFi integrating Scala Native, a Scala ahead-of-time compiler that makes it possible to directly compile Scala code over devices that do not support the JVM (enabling the so-called Cross-compilation). Cross-compilation between different platforms is a very desirable feature for a programming language because it makes the language much more flexible. For this reason, it is often included in many modern languages such as Kotlin and Rust. To conclude, several tests are done to validate the stability and the performance of the integration and in order to prove that the implementation proposed can efficiently extend the number of devices on which ScaFi can be run.

“Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza”
Inferno, Canto XXVI (vv. 118-120)

Acknowledgements

I would like to acknowledge and thank my family, friends and relatives for supporting me during my studies.

I would not be where I am right now without Prof Rita Diodato, Comparetto and my high school teachers that encouraged my love for sciences and interest in humanistic subjects.

I would also like to thank my co-supervisor Dr Gianluca Aguzzi who guided me through this research and whose advices made this thesis possible.

I cannot not mention my supervisor Prof Mirko Viroli who offered me the possibility to pursue the thesis with him and that taught me so much during these three years.

Finally, to the Great Almighty, I express my sincere gratitude for guiding me every day through the difficulties.

Contents

List of Figures

Listings

Chapter 1

Introduction

Over the past decades, computer devices have become steadily more accessible and portable due to the technological progress. It is obvious that the traditional way of programming, which has always been based on an attempt of micro-managing each device individually, is not efficient in these scenarios. In fact, in complex distributed applications traditional solutions lack both modularity and reusability. Aggregate Computing is an emerging paradigm for complex distributed systems where a vast number of distributed devices are involved in a global computation to produce a result weakening the importance of the "local computation" over the single device. This situation is common in the Internet of Things, large-scale urban events, drone coordination and smart cities. Modern Aggregate Computing APIs are normally based on Field Calculus. Field Calculus (FC) gives a general model and properties useful to define the global and local relationship of devices in a network, similarly as Featherweight Java does for OOP. Additionally, these APIs also rely on an abstraction layer implementing resilient blocks that hide the complexity of the environment from the sight of the developer (approach: complexity "hidden under the hood"), offering a simple development environment and proving self-stability to the layers above. An Internal DSL that offers these features is Scala with Computational Fields (ScaFi), a Scala framework implementing aggregate programming mechanisms through computational fields. Internal DSLs are written using the syntax and semantics of the host language, which for ScaFi it is Scala. In fact, Scala providing first-class functions and a both rich and static type system is a perfect environment on which develop the Field Calculus system. ScaFi not only includes a complete API but also a simple simulator to run aggregate applications together with a simple GUI. A critical concept for these types of libraries is portability since their nature implies the possibility of being run over a wide range of different devices. The research shown in this thesis offers a solution to improve the portability and flexibility of ScaFi integrating Scala Native, a Scala ahead-of-time compiler that makes it possible to directly compile Scala code over devices that does not support the JVM. The document is organised in 7 chapters (excluding this one): *Scala* Chapter 2 where the main characteristics of Scala - needed to use ScaFi - are exhibited with a detailed explanation of the OOP features and the Scala type system. *Aggregate Computing* Chapter 3 where the evolution of this new type of computing technique is illustrated and particular attention will be given to the modern structure of these systems that will be described with a bottom-up approach (from the basis of the Field Calculus to the application code). A chapter *ScaFi* Chapter 4, about what it is, how it has been built and how it can be used. We continue with *Analysis* Chapter 5 where we state the requirements of Scala Native implementation and analyse the tools that can be used to do it. The *implementation* is shown in Chapter 6 and the testing phase is described in *Validation* Chapter 7. Finally, a *Conclusion* Chapter 8 includes final thoughts and some future improvements.

Chapter 2

Scala

In this chapter, it will be explained several features of the Scala language and - more specifically - Scala 2, essential to use ScaFi. We will give a general overview of the language (Section 2.1) and then a more exhaustive illustration of the main features (Section 2.2). This last section is inspired and based on the works: [28, 39, 18, 40]. In addition, we will describe Cross-Project (Section 2.3) - a Scala plugin - that is currently used by ScaFi. To conclude, we will expose the main innovations brought in with Scala 3 (Section 2.4).

2.1 Scala: Introduction

Scala is a multi-paradigm and general-purpose programming language. It conciliates functional and object-oriented programming becoming a high-level language suitable for developing high-performance systems. It can run on the JVM and JavaScript giving the developer flexibility and access to an enormous amount of libraries. Its name derives from "scalable" because the main purpose of this language is to serve as an intuitive workbench for the creation of systems subjected to growing complexity. In the following sections, several features and mechanisms of Scala will be introduced. However, this chapter is not trying to be a comprehensive guide for this language but only a reference and a preparation for the reader to better appreciate the next part of the paper where Scala code and libraries are being proposed. For this reason, some basics and expert skills won't be included in the overview. This amount of knowledge is the bare minimum to be able to use ScaFi and understand how it works. To be more specific, we need to introduce a system that categorises the knowledge of Scala into levels of expertise. Odersky Martin, the inventor of Scala, defines a Scala expertise levels system [37] differentiating between application programming and library designing. A table summarising the levels is shown below table 2.1.

Scala Levels Table		
Overall Level	Application Programmer	Library Designer
Beginner	Beginning (A1)	
Intermediate	Intermediate (A2)	Junior (L1)
Advanced	Expert (A3)	Senior (L2)
Expert		Expert (L3)

Table 2.1: Table: levels

The table shows how each expertise level corresponds to a certain level of knowledge in both application programming and library designing. In this document, we will mainly focus on the intermediate topics involved in application programming (levels A1, A2) because are essential to be able to write Scala code and interact with ScaFi. Of course, some of the junior (L1) and senior (L2) techniques will be discussed cause knowing them is necessary to proficiently use and - more importantly - understand a Scala library. It follows a list of the topics explained in this chapter classified by expertise level. This will be useful to have an overall view of the functionalities offered by the language and it can be used as a table of contents for the next section.

Application Programming **A1**, **A2**:

- OOP statements: classes, objects, fields, packages, imports;
- class construction;
- modifiers: access modifiers and others;
- type system and standard types.

Application Programming **L1**, **L2**:

- traits;
- generic programming;
- control abstractions;
- advanced type: existential types, structural types;
- variance annotations;
- implicit definitions.

2.2 Scala 2 Features

First, we are going to illustrate the main OOP features included in Scala such as Class, Field, Object, Package and Trait.

2.2.1 OOP Features

Classes

Scala classes can be seen as blueprints for creating objects. They may include values, variables, traits, methods, objects as well as nested classes. In the same way as in C++ or Java to utilise classes, you construct an object and then invoke its methods. Classes can contain primary (the class body) or auxiliary constructors (using `this`). If an object has been defined in the same file of a class with the same name, the object becomes the *companion object* of that class (these entities can then access each other's private fields and methods). Additionally, classes cannot include static methods, however, they can be included directly in the companion object. The keyword `case` can be used to define a class that models immutable data. In fact, in this case the word `new` is not needed to initialise the class and a already developed method *apply* takes care of the object construction. To conclude, a class must follow the single-class inheritance scheme, but it can implement more than one trait.

Fields

Fields can be created using the declarations `val` or `var`. On the one hand, fields constructed with *val* are immutable, on the other hand, those defined with *var* are mutable. If the keyword `private` is used during a field definition, the field can be accessed from outside the class. Additionally, primary construct parameters are private by default but using either `val` or `var` they can be accessed from other classes.

It follows listing 2.1 a Scala code snippet with several useful examples.

Objects

Objects are classes with only one instance (basically a *singleton*) that can inherit a class as well as multiple traits. Objects can have methods, inner classes and fields. In addition, they are created lazily when referenced. listing 2.3

Listing 2.1: Classes code example - Classes and Fields

```scala
// Class with Primary construct
class Point_1(var x: Int, var y: Int) {
 // Auxiliary constructor
 def this(x: Int) = {
  this(x, 0)
 }
 // Auxiliary constructor
 def this() = {
  this(0, 0)
 }
}
// Class with overridden setters
class Point_2 {
 private var _x = 0
 private var _y = 0
 private val min = 10
 def x = _x
 def x_=(n: Int): Unit = {
  if (n > min) _x = n else printMessage
 }
 def y = _y
 def y_=(n: Int): Unit = {
  if (n > min) _y = n else printMessage
 }
 private def printMessage = println("Low value")
}
// Single inheritance
class Point_3 extends Point_2 {
 override def toString: String = s"($x, $y)"
}
```

Listing 2.2: Classes code example - Object and Usage

```scala
// Companion object of Point_2
object Point_2 {
 def createZero(): Point_2 = {
  var p = new Point_2
  p._x = 0
  p._y = 0
  p
 }
}
// How to use
val p1_1 = new Point_1(2, 3)
var p1_2 = new Point_1(3)
var p2_1 = new Point_2
var p2_2 = Point_2.createZero()
var p3_1 = new Point_3
```

Listing 2.3: Object code example

```scala
object PizzaMargherita {
 val toppings = Array("tomato", "mozzarella", "basil")
 def description: String =
  this.getClass().getName().stripSuffix("$") + " is made with " + toppings
   .mkString(", ")
}

// How to use
println(PizzaMargherita.description)
// PizzaMargherita is made with tomato, mozzarella, basil
```

Packages

Packages are included in Scala to manage namespaces in structured programs. In C++ this is done using the keyword *namespaces*, in Java with *packages*. However, in Scala, packages are implemented differently. Firstly, a package may contain members such as classes, objects or traits. Moreover:

- Packages may have a *package object* that can be created naming the object with the same name as the package, in the same way as companion objects are created for classes. This object can be seen as a container shared through all the packages. It can have variables, methods and values definitions.

- Scala packages can be used without specifying the other packages at the top. Additionally, the same file can be part of more packages;

- In Scala it is possible to import members from other packages in different ways. Both the whole file can be included or a specific list of members can be selected (even renaming them). In addition, imports are admitted everywhere in the code.

These features make Scala's definition of packages far more flexible than Java's version. All these features can be better appreciated looking at the examples proposed in listing 2.4;

Traits

Traits are similar to Java *interfaces* and are used to share the same interfaces and fields over more classes. Traits cannot be instantiated and that is why they have no parameters. However, they can be extended by classes, objects and traits. Traits can be used in two ways:

Listing 2.4: Packages and Imports code example

```scala
// Imports
import pizza._ // Import everything from the package pizza
import pizza.Dough // Import the member Dough
import pizza.{Dough,Toppings} // Import the members Dough and Toppings
import pizza.{Toppings => NewName} // Import the member Toppings remaning it

// Package: pizza.Toppings
package Toppings
// Package object
package object Toppings {
 val Size = List(Big, Medium, Small)
 def showPrice(topping: Topping): Unit = {
  println(s"The price of ${topping.name} is ${topping.price}")
 }
}
```

- As interfaces: When a trait includes only abstract fields and methods. These traits can be implemented by concrete classes or objects only if all the trait's abstract members are implemented;
- As mixin: Here, a trait includes concrete methods and fields. The classes or object extending a mixin will acquire its concrete entities;

These two functionalities are shown in Listing 2.5

Class construction and linearisation

A class can inherit both mixin traits and another class but this must be done following a strict order:

- Super-class' construct;
- Traits' constructors (parents constructed first);
- Class constructor;

This order is fundamental to understand the *linearisation* of a class, namely the process that define the hierarchy of the class's parents. An illustration of this process is shown in Listing 2.6.

2.2.2 Modifiers

Modifiers are keywords that may precede member definitions modifying their accessibility or general usage. More than one modifier can be used in a single member definition and no specific order should be respected.

Listing 2.5: Traits code example

```scala
// Trait as interface
trait Good {
 var availableQuantity: Int
 var price: Double
 def calculatePriceWithTax(quantity: Int): Double
}
class Pen extends Good {
 var availableQuantity = 1000
 var price = 1.50
 val lowTaxPercentage = 0.02
 val highTaxPercentage = 0.04

 def calculatePriceWithTax(quantity: Int): Double = {
  if (quantity < 50)
   price * (1 + highTaxPercentage) * quantity
  else
   price * (1 + lowTaxPercentage) * quantity
 }
}

// Trait as mixin
trait SellableGood extends Good {
 def sell(quantity: Int): Unit = { availableQuantity -= quantity }
}

class SellablePen extends Pen with SellableGood

// How to use
var pen = new Pen
var spen = new SellablePen
println(pen.calculatePriceWithTax(20)) // 31.2
println(spen.calculatePriceWithTax(20)) // 31/2
spen.sell(10)
println(spen.availableQuantity) // 990
```

Listing 2.6: Class construction order

```scala
class A { print("A") }
trait B { print("B") }
class C extends A with B { print("C") }

new C // ABC
```

Access modifiers

Access modifiers are keywords used to restrict the definition of classes, packages or objects to a specific region of the code. This visibility is implemented differently than in Java. In fact, in Scala the keyword *public* does not exist because it is the default access level. On the contrary, in Java, the default modality is *package-private*. Follows a list of the access modifiers available in Scala:

- Public (not a keyword): This is the default access level, anyone can access the member;
- Protected: Members in this access level can be accessed by any subclass and object of these sub-classes. This keyword can be used with the following syntax: *private[x]*;
- Private: Members in this access level can be accessed only by the same class as well as by objects in the same class. This keyword can be used with the following syntax: *private[x]*;

2.2.3 Other modifiers

Override

This modifier must be included before any member definition which overrides another concrete member definition belonging to a parent class, in this case, the keyword *override* is mandatory. Moreover, it can be used (even if it is not strictly necessary) when implementing an abstract member;

Listing 2.7: Override code example

```scala
// Override
class getOne {
 def getValue(): Int = { 1 }
}
class getTwo extends getOne {
 override def getValue(): Int = { 2 }
}
```

Abstract

Scala also offers the possibility to use abstract classes (similar to Java's definition of abstract class). Those are particularly similar to traits and for this reason, are rarely used. However, there are some situations when their usage is extremely useful:

1. When you need a trait with constructor arguments;

Listing 2.8: Abstract code example

```scala
// Abstract
// Syntax error
trait Concerts(code: String) {
 def generateTicket(): String
}
abstract class Concerts(code: String) {
 def generateTicket(): String
}
// Implementation with a class
class ConRoma(code: String) extends Concerts(code) {
 var ticketNumber = 0
 def generateTicket(): String = {
  ticketNumber += 1
  code + (ticketNumber - 1)
 }
}
```

2. When the Scala code will be called from a Java code (Java doesn't have the concept of traits);

Abstract override

These modifiers are used when a member is being partially overwritten. This means that even the new implementation of the member is abstract. Abstract override can only be used in members of traits. This strategy can be used to define "stackable traits", that are traits providing *stackable* modifications to underlying traits or classes [48];

Listing 2.9: Abstract Override code example

```scala
// Abstract override
abstract class superClass() {
 def me(): Unit
}
trait subTrait extends superClass {
 abstract override def me(): Unit = { super.me() }
}
```

Final

This modifier can be used with different members:

1. *final* Class member definitions: cannot be overwritten by any sub-classes;
2. *final* Class : cannot be inherited by any template;
3. *final* Objects: in this case the keyword final is redundant;

4. *final* Members of final classes: in this case the keyword final is redundant, however it is necessary case of *constant value definitions*;

Listing 2.10: Final code example

```
// Final
final class numbers {
 final val pi = 3.14
}
class operands {
 final def sum(a: Int, b: Int): Int = { a + b }
}
```

Sealed

A *sealed* class can only be inherited by other classes defined inside the same file. Nevertheless, any class can inherit a subclass of another sealed class;

Listing 2.11: Sealed code example

```
//Sealed
sealed abstract class Pizza
case class Margherita() extends Pizza
case class Diavola() extends Pizza
case class Capricciosa() extends Pizza
```

Lazy

The *lazy* modifier can be applied to value definitions. These are initialised only when the variable is being accessed for the first time (in case the program will try to access the value during its initialisation, a loop behaviour would follow);

Listing 2.12: Lazy code example

```
// Lazy
lazy val x = getOne()
```

2.2.4 Types and the Type system

Scala is characterised by a unique statically type system, nevertheless, still dynamic and flexible to use. There are several benefits deriving from the usage of a statically typed language:

- They help the IDE to provide a strong syntax support to the developer;
- They avoid many kinds of compile-time errors;
- They offer a more effective refactoring;

- They can provide a detailed documentation that hardly will be outdated;

We will quickly introduce the type categories offered by Scala.

Parameterised types

A parameterised type can be defined with the following syntax:

$$T[T_1, T_2, ..., T_n]$$

where T is the type designator, $T_1, T_2, ..., T_n$ are type parameters, $n \geq 1$ and T requires n parameters.

Each type parameter T_i may have a lower bound L_i and upper bound U_i. We say that the parameterised type T is well-formed if and only if every parameter is inside its bounds;

Listing 2.13: Types code example - Parameterised types

```scala
// Parameterised types
trait Iterator[T] {
  def hasNext(): Boolean
  def next(): T
}
```

Tuple types

This category of type corresponds to an alias for a Scala class. *Tuplen[*$T_1, ..., T_n$*]* with $n \geq 2$, its syntax is $(T_1, ...T_n)$.

These tuples are nothing more than classes with fields that can be accessed only using n selectors;

Listing 2.14: Types code example - Tuple types

```scala
// Tuple Type
val tuple = (1,2,3)
val (val1, val2, val3) = tuple
print(tuple._1) // 1
print(tuple._2) // 2
print(tuple._3) // 3
```

Compound types

A compound type is defined as $T_1 with...T_n R$ with $n \geq 2$. This entity is a collection of objects with members defined in the components $T_1, ..., T_n$ and a refinement R. The refinement consists of a set of type definitions and declarations that override the ones included in the components;

Listing 2.15: Types code example - Compound types

```
// Compound Type
trait Left
trait Right
  def turn( obj: Left with Right ): Unit = {
  // ...
  }
```

Infix types

An infix type is defined as $T_1 op T_2$ with op as an infix operator applied to T_1 and T_2 (two type operands). Another analogue syntax may be: $op[T_1, T_2]$;

Listing 2.16: Types code example - Infix types

```
// Infix Type
case class Person(name: String)
class Loves[A,B](val a: A, val b: B)
def printCouple(c: Person Loves Person) =
 print(c.a.name + " loves " + c.b.name)
```

Function types

This type is represented as $(T_1, ..., T_n) \Rightarrow U$ with $n \geq 2$. It can be seen as a set of function values

Listing 2.17: Types code example - Function types

```
// Function type
def map[B](func: A => B) = // ...
```

Annotated types

Here, n annotations are attached to a type T with the syntax: $T a_1, ..., a_n$ with $n >= 2$;

Listing 2.18: Types code example - Annotated types

```
// Annotation Type
@deprecated("Function deprecated")
  def dep(): Unit = {
  // ...
  }
```

Variance types

The variance is used to define the sub-typing of parameterised classes or traits. If we have the following sub-typing relationship: $A <: B$ we can define three types of variance: *Invariant* expressed as *X[T]*, that basically is a normal generic type and X[A] has not typing relationship with X[B]; *Covariant* expressed as *X[+T]*, where X[A] will have a sub-typing relationship with X[B]; *Contravariant* expressed as *X[-T]*, where X[A] will have a super-typing relationship with X[B];

Listing 2.19: Types code example - Variance types

```scala
// Variance type
trait Basket
trait AppleBasket extends Basket
trait PearBasket extends Basket
// If T = Basket , List [ T ] can include both AppleBasket and PearBasket
trait CovariantPicnic[+T] {
  def getFood(): List[T]
}
// If T = AppleBasket or PearBasket , T can return a Basket
trait ContravariantPicnic[-T] {
  def composeBasket(basket: T): Unit
}
// If T = Basket , getBasket will return a Basket
trait InvariantPicnic[T] {
  def getBasket(): T
}
```

Structural types

This category helps the developer in those situations where a *dot notation* would be needed (dot notation exists in dynamic contexts);

Listing 2.20: Types code example - Structural types

```scala
// Structural type
type Animal = { def call: String }
def getCall(a: Animal) = a.call
// Every object with a method 'm' implemented is accepted, such as
case class Cat() { def call: String = "miao" }
```

Self-types

Self-types are used to declare a trait mixed into another one even if it does not "extends" it. The syntax of this construct is shown in Listing 2.21. This type is used for dependency injection, a mechanism used to build dependencies among types. This normally done using the *Cake pattern* that builds layered components using traits and self-types;

2.2.5 Implicits

The mechanisms listed here are static features of Scala that permit the developer to shorten the code by giving the compiler the duty of deducing the missing data.

Listing 2.21: Types code example - Self types

```scala
// Self-type
trait A
class B { a: A =>
  def C(): Unit = {
  // ...
  }
}
```

Listing 2.22: Implicits code example

```scala
// Implicit member in the scope of Square
implicit var value: BigInt = 10

// Implicit parameter
def Square(implicit x: BigInt) = x*x

// Implicit conversion
Square(1)
// value autimatically converted from Int to BigInt
```

Using the keyword *implicit*, the compiler will provide the missing information according to scoping rules and lookup mechanisms.

This process is triggered in two cases:

- *implicit parameter list*: This type of implicit statement can be used in method calls or constructs by including the keyword *implicit* at the start of the list. Doing this, in case no parameters are passed, Scala will try to find and pass an implicit type and value automatically;
- *implicit conversion*: This type of implicit statement consists in an implicit conversion from a type A to a type B and can happen in three situations listing 2.22:
  - when a function is called with one or more wrong parameter types,
  - when the type of the expression is different than the expected one,
  - when having an object x and a member m, the statement $x.m$ is invoked even if it does not exist.

There are strict rules that define the behaviour of the compiler in the case of implicit notations. The first check that is done by the compiler is to see if any ambiguity in the implicit resolution is present, and in that case, throw a compilation error. If this does not happen, then:

1. Look for a conforming implicit entity accessible as a single identifier. If it finds something, it goes on, otherwise, it throws an error,

2. Here, the compiler behaviour can be: *i* when it is handling an implicit parameter, it will consider the implicit scope of the parameter type. *ii* When it is handling an implicit conversion, it will consider the implicit scope of the target type.

Implicit scope

It is a space where the compile looks for implicit entities including the *current scope* (nothing more than the local scope of the code and the one of the imports) and the *associated types*, that is a set of companion objects and type parameters.

It is generally not considered good practice to abuse the flexibility of these rules. It is normally suggested to merge all the implicit entities in one object or package object.

2.3 Cross-platform compilation

Cross-platform compilation consists in the possibility to run a certain application over different platforms with the same code base. It is a very desirable feature for a programming language because it makes the language much more flexible. For this reason, it is often included in many modern languages such as Kotlin [4] and Rust [5]. This feature is supported by Scala though the "sbt-crossproject" plugin [1]. Using this plugin, a Scala project can be executed over three different platforms:

- *JVM Platform*: Default and supported by Scala;

- *JavaScript Platform*: with Scala.js 0.6.23+ or 1.0.0+;

- *Scala Native Platform*: with Scala Native 0.3.7+.

By doing this, it is possible to build a web application with Scala and Scala.js, in the same way, it has been done with ScaFi-web. And it is possible to execute a Scala Application on devices that do not support Java and the JVM using Scala Native.

Now, we will illustrate these two new tools: Scala.js which has already been integrated in ScaFi, and Scala Native. For this, we will show a possible integration in the next chapter.

2.3.1 Scala.js (Integrated)

Scala.js [23] is a Scala compiler that can translate the code into JavaScript code, as a consequence, the application can run on a web browser or any other environment that supports JavaScript. In this case, we won't be limited anymore by only being able to run the software on the JVM.

2.3.2 Scala Native (Not yet integrated)

Scala Native [54, 55] is an under-development project implementing an optimising ahead-of-time compiler for Scala. As we have already said this makes us able to run Scala applications without the Java interpretation. But it is also important to list all the other advantages and features included with Scala Native:

- *Low-level primitives*: Including pointers and structs. They can be really useful to have more control over the application Listing 2.23;

Listing 2.23: Scala Native - Primitives code example

```scala
// Primitives
type vType = CStruct3[Int, Int, Int]
// C struct
val v = stackalloc[vType]()
v._1 = 3
v._2 = 2
v._3 = 1
length(v)
```

- *Interoperability with C code*: It is easy to call C code using “extern” objects and most importantly without run-time overhead Listing 2.24;

Listing 2.24: Scala Native - C code calling example

```scala
// Calling C code
import scala.scalanative.native._

@extern object clib {
 def malloc(size: CSize): Ptr[Byte] = extern
}
val p = clib.malloc(8)
```

- *Instant start-up time*: Since Scala Native is an ahead-of-time compiler, the start-up time is noticeably reduced. As a consequence, the application is immediately ready to be compiled (cause not compiled anymore by a just-in-time compiler but by the *LLVM*)

Scala Native offers a re-implementation of the JVM. The process of code compilation is illustrated in Figure 2.1. As we can see, the Scala code is first compiled into Native Intermediate Representation (NIR) through the Native Scala Compiler Plugin (nscplugin). The NIR is a high-level object-oriented representation that includes both LLVM instructions and primitives (necessary to compile Scala). Its purpose is to simplify the code optimisation process done by the Native Compiler that, after a second compilation, generates *.ll* files that can be read by the LLVM. The last phase of the compilation involves directly the LLVM that produces executable binaries.

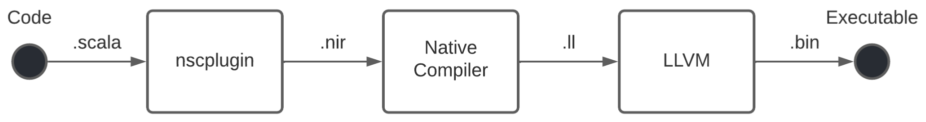


Figure 2.1: Scala Native compilation process

LLVM

LLVM [32] is nothing more but a collection of reusable and modular compiler technologies. The project tries to provide a modern SSA-based compilator able to support both static and dynamic compilation of programming languages. A sub-project of LLVM is *Clang* [31] that is a LLVM-native compiler for the languages: C, C++ and Objective-C offering great performances and detailed error and warning messages.

Scala Native Versions

Each Scala Native version has a list of compatible Scala versions, the latest is 0.4.3 (the one used in our integration). A table including all the Scala Native versions related to Scala versions is shown in Table 2.2. The reason behind our choice is that we wanted to maximise the number of Scala versions compatible with our integration (this can be appreciated looking at the table).

Compilation Modes

Scala Native 0.4.0+ offers three different modalities in which compile the native code:

- **default**: The default mode is optimised to run with the shortest compilation time. Not a lot of optimisations are applied to the code. For this reason, the run-time performances are poor in this scenario;

Scala Native Versions	Scala Versions
0.1.x	2.11.8
0.2.x	2.11.8, 2.11.11
0.3.0-0.3.3	2.11.8, 2.11.11
0.3.4+, 0.4.0-M1, 0.4.0-M2	2.11.8, 2.11.11, 2.11.12
0.4.0	2.11.12, 2.12.13, 2.13.4
0.4.1	2.11.12, 2.12.13, 2.13.4, 2.13.5
0.4.2	2.11.12, 2.12.13..15, 2.13.4..8
0.4.2-RC1, 0.4.3-RC2	2.11.12, 2.12.13..15, 2.13.4..8, 3.1.0
0.4.3	2.11.12, 2.12.13..15, 2.13.4..8, 3.1.0..1

Table 2.2: Scala Native versions

- **release-fast**: This mode is optimised for both the compilation time and run-time performance. This is done by adding a link-time optimisation. The code size is still relatively small;

- **release-full**: Here, some aggressive optimisations are included in the compilation (i.e. type-driven method duplication). With this modality, we can reach the best run-time performances. However, the code size and the compilation time increase noticeably.

Garbage Collectors

Scala Native offers three different garbage collectors and even the possibility to compile without it, here we compare these possibilities:

- **immix**: The default solution since v0.3.8 and it is a mostly-precise and mark-region tracing GC;

- **commix**: This GC has been introduced with v0.4.0 and consists in a parallel version and, in general, more performing version of immix;

- **bohem**: It is a conservative generational GC;

- **none**: The compilation can also be executed without any GC, this option is experimental and may be useful for short-running application or in those situations where pauses caused by the GC are not acceptable;

A complete comparison of these tools can be found in [45].

Listing 2.25: Scala Native - JUnit Test

```scala
import org.junit.Test
import org.junit.Assert._

class Tests {
 @Test def Test(): Unit = {
  assertTrue("Assertion message", true)
 }
}
```

Link-Time Optimisation

Link-time optimisation or LTO aims to increase the run-time performance optimising the binaries that are generated. Here, three alternatives are proposed by Scala Native:

- **none**: The default mode, it does not optimise calls between Scala and C, only calls Scala-Scala are still optimised;
- **full**: Applies in-lines between Scala and C through FullLTO offered by LLVM.
- **thin**: Applies in-lines between Scala and C through ThinLTO offered by LLVM. This mode does not slow down too much the compilation but increases noticeably the run-time performance (even more than the *full* alternative);

Testing and Profiling

Scala Native also offers some testing and profiling tools. Firstly, it supports JUnit, and JUnit tests can be written like in any other java project as shown in Listing 2.25.

On the other hand, the profiling is supported with the command *time* helpful to measure the execution time (to test our implementation we will use a more advanced benchmark). In the documentation it is also suggested to use *Flamegraphs* [27] to have a graphical representation of the CPU usage.

2.4 Scala 3

With Scala 3, the language has been heavily revolutionised by minor cleanups, big new implementations and several modifications of the API. In the following, we list a brief overview of the main innovations, more details can be found in [38]:

- **New Syntax**: The syntax has been revisited and the following changes have been done:

    1. A new "quiet" Control syntax for structures (while, if, for);
    2. The keyword "new" is now optional;
    3. Supporting distraction-free with optional braces;
    4. Indentation sensitive programming;
    5. Implicits revisited and heavily modified;

- **Type System Improvements**: The type system has been changed to support a better type inference and several new features:

    1. Enumerations;
    2. Opaque Types;
    3. Intersection and union types;
    4. Dependent function type;
    5. Polymorphic function type;
    6. Type lambdas;
    7. Match types;

- **Meta-programming**: New powerful tools for meta-programming are Inline (reduce methods and values at compile time), Compile-time operations (functionalities implemented in the package *scala.compiletime*), Quoted code blocks (quasi-quotation tools) and Reflection API (tools to generate and analyse program trees).

Chapter 3

Aggregate Computing

In this chapter, we analyse the concept of aggregate computing and describe each one of the abstraction layers on which modern APIs for aggregate programming are based using a bottom-up approach. We start outlining what space-time and aggregate programming are and why they are so important today (Section 3.1), then we give an overview of the development that brought traditional models to the ones used today (Section 3.2). In addition, we review the theoretical foundations and properties of field calculus (Section 3.3), then we study the resilient building blocks existing between Field Calculus constructs and developer APIs showing a possible implementation (Section 3.4). To conclude, we discuss the main problems of these approaches and future research direction (Section 3.5).

3.1 Introduction

This section illustrates the definitions of space-time programming, aggregate processes, aggregate programming - with a quick overview on how it is structured - and the context in which the necessity of these techniques has emerged.

3.1.1 Context

Over the past decades, computer devices have become steadily more accessible and portable due to the technological progress. The decreasing cost and size of electronic components has resulted in a massive increase in the number of wireless networked devices, ranging from smartphones and point-of-service terminals and tablets to smart lighting. Additionally, the Internet of Things (IoT) has noticeably increased the number of sensors and embedded systems with which we interact daily. All these phenomena are leading to an exponential growth of the number and variety of objects connected to the network.

Fox example, with swarm robotics, multiple robots have to cooperate as a unified system to produce the expected collective behaviour. The result of a single computation is the result of the communication of a high number of devices located in a physical space and not anymore a local computation on a single machine.

In a similar way, modern biology and material science are also starting to use hardware solutions composed by an enormous number of smaller and error-prone devices. Each of them able to interact only with its adjacent neighbours. In this scenario the system have to act harmoniously as a group to efficiently carry on the computation.

It is obvious that the traditional way of programming, that has always been based on an attempt of micro-managing each device individually, is neither efficient nor effective. In fact, in complex distributed applications traditional solutions lack both *modularity* and *reusability*.

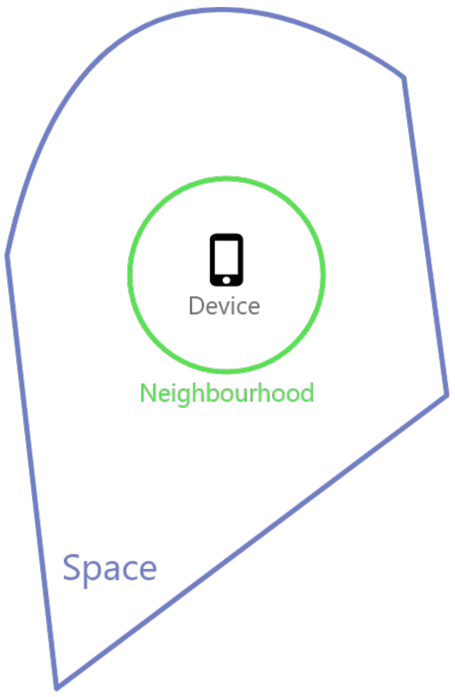
3.1.2 Space-time programming

We define **space-time programming** as an umbrella term for programming techniques based on the use of spatial abstractions. This space, also called *virtual space*, consists of a network of interacting devices existing in a real environment. This type of **spatial computation** is executed by an abstract entity defined as *spatial computer*, that in more concrete term refers to a whole physical network of interacting devices able to cooperate to do certain computations. This approach is defined in [24] also as "*computing somewhere*" - just one of the four classes in which all space-related approaches can be categorised:

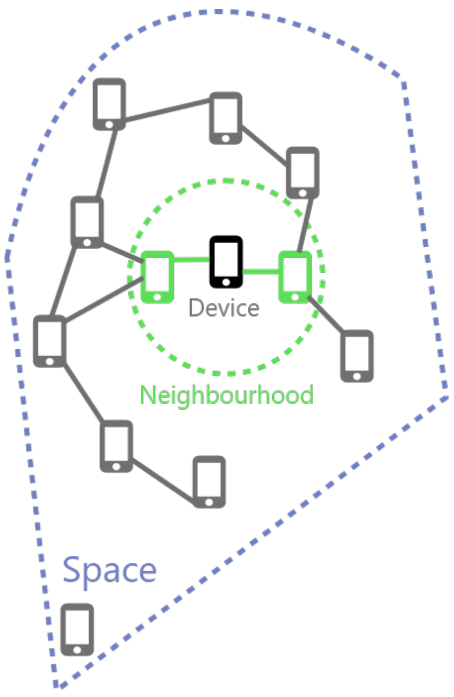
- *Computing somewhere*: location-related information and spatial constraints;
- *Computing everywhere*: location-related information and non-spatial constraints;
- *Computing anywhere*: location-unrelated information and spatial constraints;
- *Computing nowhere*: location-unrelated information and non-spatial constraints;

3.1.3 The amorphous abstraction

It is correct to specify the difference between the continuous and discrete network representation. The first one refers to the *amorphous medium abstraction* [16] - a view of the network including a infinite number of devices, the second one consists in an approximation of the continuous version where only a limited number of interacting devices exist. This comparison can better be appreciated and understood analysing the Figure 3.1. The reason why both these representations are important in distributed algorithms is that programs able to properly operate in both these environments normally have less problems in terms of sensitivity and scalability.



(a) Continuous network space



(b) Discrete network space

Figure 3.1: Comparison of amorphous medium abstraction

3.1.4 Aggregate programming

For the above-mentioned reasons, a new approach - offering a programming strategy both scalable and easy to use in these applications for the developers - is needed today. **Aggregate programming** [14] provides an innovative solution simplifying the development of this type of system, keeping the focus on collections of devices rather than the single entity. Its goal is to simplify the creation and design of these complex systems. This large-scale programming approach relies on three main points:

- subsystems and modules must have a transparent composition;

- programmers are not required to interact with the coordination mechanisms, that are hidden "under the hood";
- specific coordination rules are needed in subsystems located over different regions and times.

Those three key points are ensured by the abstraction layers that compose aggregate programming APIs (Figure 3.2). The lowest layer is nothing more than the devices sensors, actuators and all that is directly supported by the hardware. Then the three middle layers consist of software libraries and finally the top one includes the code that is written by the developer using an aggregate programming API. Each of those layers will be discussed and examined in the following sections.

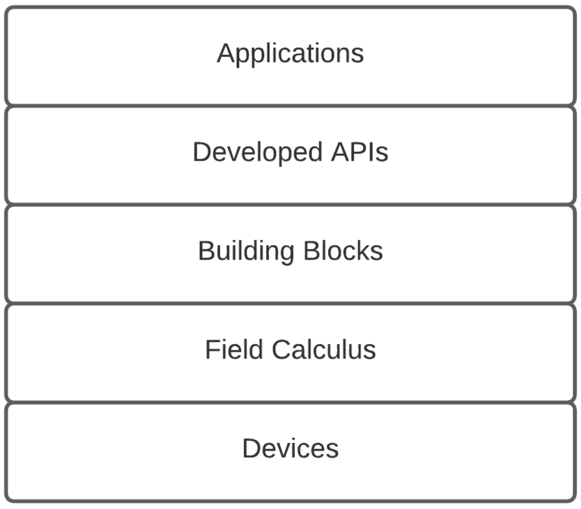


Figure 3.2: Aggregate programming abstraction layers

3.1.5 Aggregate processes

An *Aggregate process* [22] can be called "computational bubble" and it is an abstraction that identify a set of devices that perform a specific work. These groups are dynamic and context-driven collection of computational devices that can stretch or shrink over time. This concept is essential to describe and implement areas of the network where groups of devices have to behave differently.

3.2 Evolution of coordination models

Aggregate computing is just a modern solution for distribute computing. First we will analyse how older models evolved through the years [50]. This will be fundamental to appreciate the importance of field calculus at the base of aggregate computing.

3.2.1 Early coordination models

- **Generative communication**: These models are based on the notion that interaction between various, autonomous software systems - also called *agents* - can be possible using a shared data space. Numerous coordination models have historically described this space as a whiteboard for parallel computing systems, where agents are able to write and read data, producing the so called *generative communication*. One example of this method is Linda [26], where processes - rooted on a centralised tuple-space - communicate writing and reading (using queries to represent partially the structure matching the piece of data needed) heterogeneous chunks. These actions are implemented using a suspensive semantic that freezes the process firing the query until its conclusion.

- **Programmable Coordination Rules**: The above-mentioned tuple-based coordination model rooted on a shared data space to make agents communicate can be improved using a logic tuple-space model, namely giving the possibility to agents to program tuples through first-order logic rules. This improvement removes the limit of the previous approach where only data can be saved in a tuple. A well-known framework that uses this technique is Shared Prolog [8], but also several others exist like MARS [17], which is based on a Linda-like model integrated with a sort of intelligence, able to manipulate the data stored in the shared space.

- **Distribution**: This branch of techniques is not focused on distributed system coordination but instead on centralised local components that are spread through the physical environment. As a result, there is no need to use a shared space and tuples can finally be distributed. This progress enables us to use distributed settings, event-based interactions and coordination abstractions. JavaSpaces [25] offers a good example of this model. Here, a step further is taken towards the pervasive computing scenario.

- **Self-organising Coordination**: These models are directly inspired by scientific scenarios and solve the problem of openness, large-scale and intrinsic adaptiveness proposing a *self-organising coordination*. It refers to those mechanisms where coordination abstractions (logical rules) locally manage interactions between devices to guarantee that the global coordination works properly. Further reading on this topic can be found in [52], where a framework for implementing and modelling self-organising coordination is exhaustively illustrated.

- **Field-based coordination**: The most relevant progress made during the evolution of coordination models has been the introduction of *coordination fields*. First, the notion of field [56] belongs to sciences (electromagnetic or gravitational field) and is basically a way to manipulate distributed data. Then, the definition of coordination field or co-field has been introduced in [33] trying to adapt this mathematical concept to support self-organisation models. The final goal was to simulate the environment through the co-field structure and - by singularly storing it in every device - to let agents interpret and analyse the current state of the system at any time.

- **Spatial computing approaches**: This is the last intermediate phase that precedes the modern way how aggregate computing is structured. With these approaches the general level of abstraction of the spatial system increased, leading to a simplification of the programming process. Additionally, spatial patterns, tools to stream data over space and time as well as space-time models are introduced in order to successfully manipulate data that is spreading over the space and evolving over the time.

In the following section we will discuss the Field Calculus that is the modern basis of aggregate computing.

3.3 Field Calculus

Field Calculus [53] is the theoretical foundation and mathematical core of aggregate programming. Field calculus or “FC” gives a general model and properties useful to define the global and local relationship of devices in a network, similarly as Featherweight Java [29] does for OOP. FC has been built using a semantic and syntax inspired by Proto [51] - a general purpose space-time programming language - but simplified. The result is a compact and minimal model able to cover all Proto’s functionalities and more. This section is going to discuss the computational model and syntax of the Field Calculus introducing the concept of computational field. Then, it will be explained what higher-order field calculus is and its advantages. Moreover, we will talk about properties of FC models such as self-stabilisation and space-time universality and conclude with an overview of Protelis - a DSL for field calculus.

3.3.1 Computational Model

In the computational model a program **P** is run by a network of devices δ including dynamic neighbouring relationships representing logical or physical proximity. This scenario is expressed in Figure 3.3 where each point corresponds to a device and each line to a neighbouring relation.

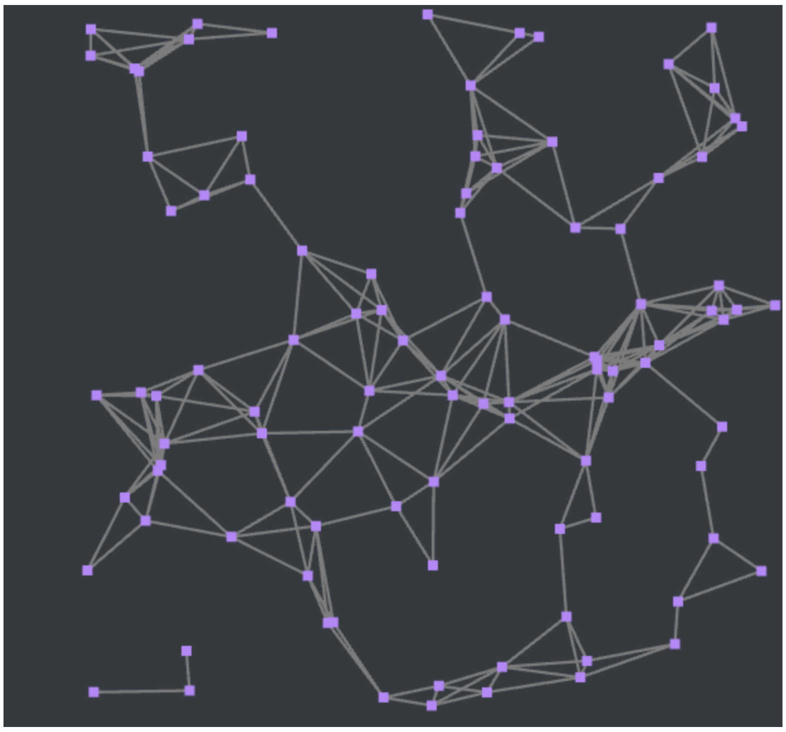


Figure 3.3: Random network generated using ScaFi web [7]

In addition, the computational model defines the concept of computational field ϕ, that maps each device (at a given time) to a certain value produced by a round-based computation executed on the device. Therefore, we consider the computation from two opposed viewpoints, first from a local point of view, where computations are nothing more than round-based schemes executed in single devices.

In each round [13], a node:

1. *Wakes up*;

2. *Gathers* information (sent by neighbours during the sleeping time) generating neighbouring fields ϕ - structures mapping a unique set **D** of devices δ to values v;

3. *Detects* information from internal sensors or other internal sources of data;
4. *Finds* information stored in the local memory during the previous round;
5. *Evaluates* the program **P** using the data collected during steps: 2, 3, 4;
6. *Stores* the result in the local memory and *sends* a message to all the neighbours;
7. *Goes* to sleep again for a certain amount of time;

From now on, we will say "device ϕ fires" referring to a device ϕ that executes a whole computational round.

From the global or aggregate perspective a single spatial computing machine handles a data abstraction defined as a space-time field evolution ϕ, that can be seen as a map of events ϵ - moments in the space-time domain when devices fire - to the respective results. In this scenario, each computation takes fields evolution as input and returns fields evolution as output.

3.3.2 Syntax and semantic

As already mentioned above, the syntax and semantic [9] of the field calculus have been built with a minimalist approach and kept simple.

program $\boxed{P\ ::=\ \overline{F}\ e}$

function declaration $\boxed{F\ ::=\ def\ d(\overline{x})\ e}$

expression $\boxed{e\ ::=\ x \mid v \mid let\, x\ =\ e\, in\, e \mid f(\overline{e})}$

$\boxed{rep\ (e)\{\ (x) => \ e\} \mid nbr\ \{e\} \mid if\ (e)\ \{e\}\ \{e\}}$

value $\boxed{v\ ::=\ l \mid \phi}$

local value $\boxed{l\ ::=\ c(\bar{l})}$

neighbouring field value $\boxed{\phi\ ::=\ \overline{\delta}\ \longmapsto\ \bar{l}}$

function name $\boxed{f\ ::=\ d \mid b}$

The program P is defined as a list of function declarations and an expression e. A function declaration F defines a function with name d, a list of parameters $\overline{x}$ and an expression e for the body. Expressions can be:

- a variable x;
- a value v, that can be both a local value l - result of a construct c - and a neighbouring field value which we have already defined in the previous subsections;
- a *let* that is calculated applying the value v_0 of e_0 and recursively apply the results;
- a function f that can be declared d or build-in b, for example logical operators;
- a branch *if* (domain restriction) that splits the system into two groups of devices (depending on how each one evaluates the condition). The operations carried on in different domains do not interfere with each other;
- a *nbr* defining a function producing a neighbouring field value ϕ , that is a result of e from the neighbours;
- a *rep* defining a function which evaluates an expression e substituting the variable x.

3.3.3 Higher-Order Field Calculus

Higher-order field calculus or HFC is an expansion of the standard field calculus. It includes all the functionalities existing in the FC and implements a way for the developers to use functions as any other value. Doing this, through HFC is possible to dynamically add code to the network or modify the existing one. Precisely, in HFC functions can:

- Take other functions as arguments and, in the same way, return functions rather than only values;
- Be created "on the fly". As a result, it is possible to inject new code into the system;
- Be moved from a device to another one via constructs such as *nbr* or *rep*;
- Compose a field and possibly be shared through the network.

Even if the potential of this model is exponentially higher then the standard field calculus, its syntax [13] is still simple and laconic. As can be observed, all the definitions are noticeably similar to the original versions.

program	$\boxed{P \ ::= \ \overline{F}\ e}$
function declaration	$\boxed{F \ ::= \ def\ d(\overline{x})\ e}$
expression	$\boxed{e \ ::= \ x \mid (\overline{x}) \ =^{\tau}> \ e \mid v \mid let\, x \ = \ e\, in\, e \mid f(\overline{e})}$
	$\boxed{rep\,(e)\{\ (x) => \ e\} \mid nbr\ \{e\} \mid if\ (e)\ \{e\}\ \{e\}}$
value	$\boxed{v \ ::= \ l \mid \phi}$
local value	$\boxed{l \ ::= \ f \mid c(\bar{l})}$
neighbouring field value	$\boxed{\phi \ ::= \ \overline{\delta} \ \longmapsto \ \bar{l}}$
function value	$\boxed{f \ ::= \ d \mid b \mid (\overline{x}) \ =^{\tau}> \ e}$

3.3.4 Properties of field calculus models

Now we dig into the properties required in field calculus models. Those are global-local coherence, self-stabilisation, space-time universality and eventual consistency.

Global-local coherence

This property is guaranteed by the field calculus keeping aligned the *nbr* operations through the distributed network. Nevertheless, this is not trivial due to the fact that *nbr* operations can be requested multiple times and that the function's execution could proceed with different speed depending on the device.

If global-local coherence is not achieved, at a certain time a subset of the network's devices may not be properly synchronised with the whole network.

Self-stabilisation

Self-stabilisation [49] prevents the system from assuming incorrect states. This property plays an important role in distributed systems because it ensures that firstly, a program with an unvarying input converges to a defined value in a discrete time and that secondly, this is always true for every transitory input values happened before the execution of the program. In other words, the output of an algorithm defined as self-stable will be completely independent by the past values assumed by the system.

Space-Time Universality

Space-time universality implies that the field calculus is Turing-complete [46, 47] for every local computation as well as able to gather values from certain past events. A detailed proof of this property is illustrated in [9].

Eventual consistency

The system can be defined as eventual consistent if the state on which it converges depends only on the continuous environment and not on the way how the devices are distributed. This definition expands the one of self-stabilisation. A rigorous definition of this property and explanation can be found in [15].

3.3.5 Protelis

To properly be able to use HFC in programming are also needed several other tools such as an interpreter, a language and all the other tools responsible for handling run-time aspects.

Protelis [44, 42] is a DSL that solves this problem implementing:

- A HFC semantic;
- An interpreter;
- A virtual machine;
- A device interface abstraction and API;
- A communication interface abstraction and API;

Protelis source code first have to be converted into a valid HFC code, this is done by the Protelis parser, then this new code together with the execution context is given to the virtual machine that runs the interpreter at regular intervals fig. 3.4. The structure of Protelis assures an easily-portability into both simulated (i.e Alchemist [43]) and real world scenarios.

Protelis has been developed using Java, thus it runs on the JVM. This guarantees a high portability and the possibility to use Java's libraries. Protelis syntax looks similar both to C and Java, making it fast to learn. Moreover, it is a pure functional language also inspired by Proto.

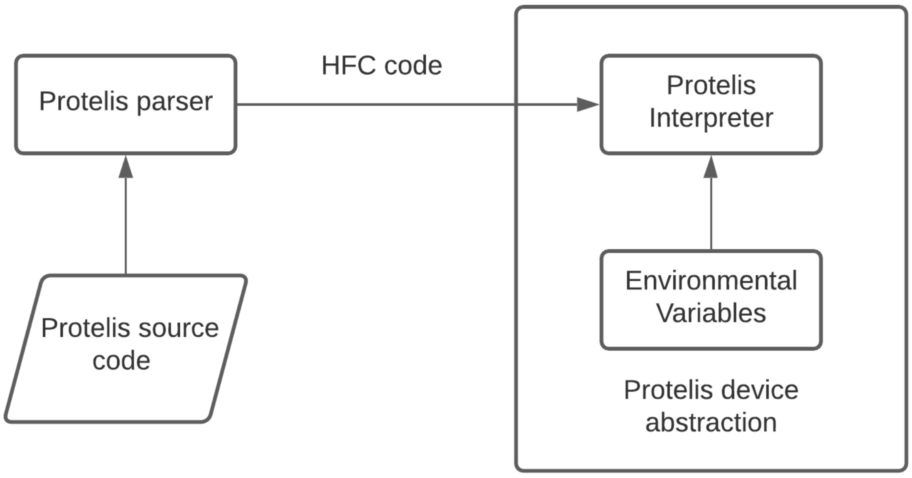


Figure 3.4: Protelis environment structure

3.4 From Building blocks to Applications

As we said in the first section, one of the goals of aggregate programming is to hide all the complex coordination mechanisms that are implemented with field calculus. To guarantee this, other two abstraction layers exist between the real application code and the field calculus layer. The first one is composed by the resilient coordination operators, the last remaining is the developer API layer, both will be discussed in this section.

Resilient layer

As already said before, this layer is composed by resilient operators, these are respectively: **if** with the same meaning as the one found in field calculus, **G** that basically spreads information through the network, **C** does the opposite collecting information, **T** tracks values over time and **S** being responsible for partitioning the network into smaller zones with the same radius (this is done using a leader-electing mechanism). Now let us take a closer look on how they work [16]:

- **G**(source, initial, metric, accumulate): This operator spreads information starting from *initial* for a distance equal at *metric* and following the path defined in *source*. In addition, the value that is being spread changes every time according to the function *accumulate*;
- **C**(potential, accumulate, local, null): This operator gathers the data *local*

from a field of devices *potential* starting with the value *null* and merging values using the function *accumulate*;

- **T**(initial, decay): This operator keeps track of the time, starting with an initial value *initial* and decreasing it according to the function *accumulate*;
- **S**(grain, metric): This operator elects a set of leaders according to three rules: (*i*) every device must not be distant from a leader more than *grain*, (*ii*) every two leader must be distant at least $\frac{1}{2}$ *grain*, (*iii*) every distance must be measured according to the function *metric*. Then the network is partitioned depending on the selected leaders;

Developer API layer

The last level before the application code is the APIs layer and it includes the libraries made to create a user-friendly interface for developers. Moreover, it also improves the underlying levels adding:

- *Reusability*: using generic components;
- *Productivity*: implementing specific components for certain application contexts;
- *Declarativity*: offering high-level functionalities and programming patterns;
- *Flexibility*: using low-level functions;
- *Efficiency*: implementing a coherent substitution semantic;

Additionally, these APIs also inherit the properties of below layers such as being resilient and self-stable.

The abstract structure on which aggregate computing is founded makes it possible to build API functions easily and concisely. For example only few lines of code are needed to implement functions to measure device-to-device distance or to execute a broadcast (Listing 3.1).

Application Code

Here, we just want to show how powerful aggregate programming is. It is illustrated a simple crowd detecting system (fig. 3.5, Listing 3.2) written using ScaFi [18] - API for aggregate programming based on Scala [39]. The program simply counts the number of neighbours and evaluates the crowd risk using a mathematical formula. This program is simulated using ScaFi-web [7].

Listing 3.1: Pseudo-code examples

```
procedure broadcast(nbrRange, value)
  G(source, value, nbrRange,identity)
end procedure

procedure crowdDetector(limit)
  nbrs <- C(1,sum,1,0)
  if nbrs>limit
    Crowd Warning
  end if
end procedure
```

Basically, the program evaluates the crowd situation of each node counting the number of its neighbours *nbrs*, using the following formula:

$$nbrs = foldhoodPlus(0)((a, b) => a + b)(nbr(1))$$

foldhoodPlus folds over the node's neighbourhood starting with the value *zero* and summing the results of the expressions *nbr(1)* executed by the nodes. The word "Plus" in *foldhoodPlus* indicates that the node calling the function will be excluded from the computation.

Then the software fits the result in the range 0 - *maxNbrs*. Then, the code turns on the led of the node with the colour resulting from the computation:

$$0.5 + (nbrs * 10)/colors.toDouble$$

Where *color* is calculated as:

$$colorGap * maxNbrs * 2$$

These formulas guarantee that the colours will be within a certain range, with a specific padding between each admissible values and that all the possible results correspond in different *hsl* colours.

This is just a quick overview these tools and they will be described in a more deeply way in Chapter 4.

3.5 Problematic and Research directions

Aggregate computing has evolved over the last decade from a disparate collection of ideas and technologies to a solid core calculus and a consistent layered framework. While several researches are trying to further refine the underlying layers, a sizeable percentage is devoted to solving challenges afflicting the higher levels of the stack. Now, we are going to list the major directions on which these researches are leading, Listing 3.2: ScaFi code implementing a simple crowd detecting software

```scala
class MyProgram extends AggregateProgram
with Actuation {
  override def main(): Any = {
    val maxNbrs = 15
    var colorGap = 10
    val colors = colorGap * maxNbrs * 2
    var nbrs = foldhoodPlus(0)((a, b) => a + b)(nbr(1))
    if (nbrs > maxNbrs) nbrs = maxNbrs
    ledAll to hsl(0.5 + (nbrs * 10) / colors.toDouble, 0.5, 0.5)
  }
}
val program = new MyProgram
```

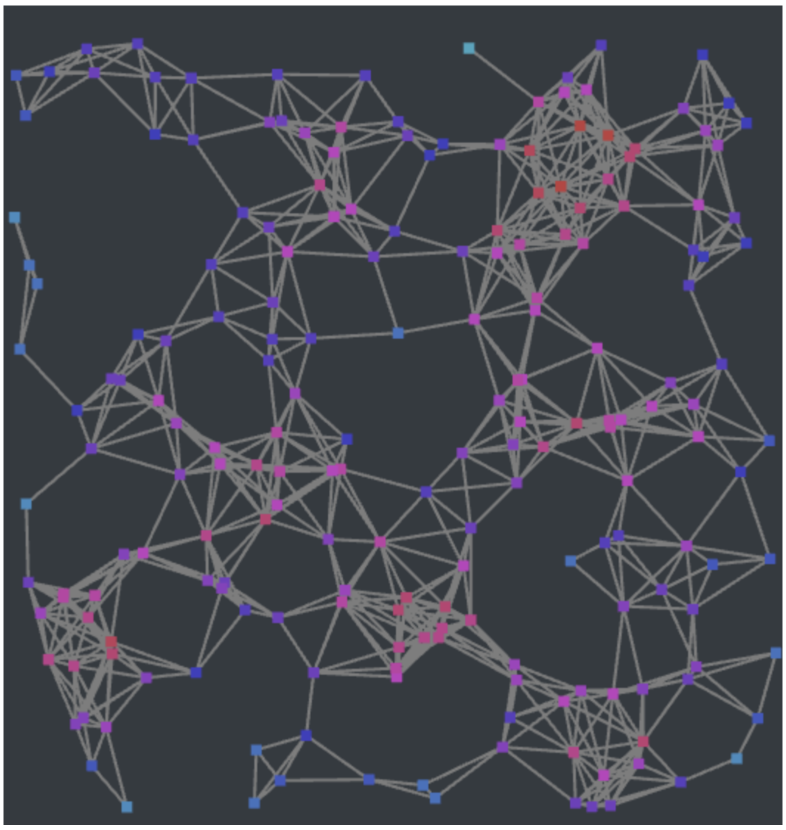


Figure 3.5: Simulation of crowd detecting software with ScaFi-web

included the one studied in this paper. Those are the development of libraries, the understanding and controlling dynamics, mobility of devices and processes, security and limits of software platforms.

3.5.1 Library development

This direction may be the most obvious and consists in expanding the existing library collection in order to create a universally more applicable and easier-to-use interface at the top of the stack. The majority of these improvements and modifications is based on the creation of alternative implementations of resilient building blocks like in [11], others are intended to capture popular design patterns and functionalities specific to particular application domains.

3.5.2 Dynamics controlling techniques

The theoretical research over the properties of these distributed systems - such as eventual consistency and self-stabilisation - have been resulting in working solutions. However, all those modern solutions presuppose that the system is frequently in a quasi-stable state. This term refers to a network where both the environment and the connections between the devices do not change for an extended period of time. Nevertheless, this is normally not applicable for large scale systems and this quasi-stable states are scarce and too transient due to the frequent perturbations in the system. In addition, these theories are not even applicable with systems implementing feedback between building blocks, excluding so their usefulness for these situations.

A possible solution could be using techniques to control or predict dynamics [30], some of these approaches have already produced well-developed solutions working with systems that require feedback. Therefore, these frameworks cannot yet be used in aggregate computing and work is needed to adapt them to be compatible.

3.5.3 Mobility of Devices and Processes

Another critical aspect - where aggregate computing requires improvements - is the tolerance of the network to handle mobility of processes and devices. This research field is related to the previous section because environments with mobile devices and processes are subject to perturbations and thus they rarely reach a quasi-stable state - required to achieve self-stabilisation. The main research direction here is studying how to predict the effects of this instability proposing alternative building block implementations to support applications involving mobility [12, 10].

3.5.4 Security

Security issues are a critical concern for every open environment, especially in pervasive computing and IoT systems where individuals or organisations - unaware of weaknesses in the network security - administrate the whole environment. But not all these problems are related to coordination, code mobility may be a concern in some systems or also code execution. In addition, *confidentiality* on the propagated or collected data in distributed systems should be better understood together with *availability*, also *authenticity* and *integrity* issues determine frailties of these type of systems. Attacks based on epidemic deviation are a concern due to the fact that aggregate computing is based on cooperation between devices, some researches are studying *trust* mechanisms to avoid this [19].

3.5.5 Software Platforms limits

Aggregate computing can be useful in a plethora of applications and software platforms for pervasive programming should be compatible with a wide range of systems - supporting *heterogeneity* - and easy to use. Many modern distributed applications need to manage devices with different architectures and operative systems. For this reason, the software platform and the compiler should impose as few as possible software or hardware limitations that would decrease the number of devices on which the application can run. A solution to manage aggregate computations in heterogeneous networks is shown in [20], where - using a "pulverisation approach" - the application logic becomes deployment-independent. This paper will follow this direction showing how it has been possible to integrate Scala Native [54, 55] with ScaFi to finally run a ScaFi program on a physical machine without the intermediary of the JVM.

Chapter 4

ScaFi

This chapter will introduce Scala with Computational Fields (ScaFi) [18] - a Domain-specific language based on Scala implementing aggregate programming mechanisms. It will be given a general introduction of the software in Section 4.1, then a detailed illustration of the project architecture in Section 4.2 and standard library in Section 4.3. We will conclude by describing two other relevant projects useful to integrate the usage of ScaFi in Section 4.4.

4.1 Introduction

ScaFi is a library implemented using Scala. It offers a set of tools to develop and execute aggregate programming applications such as:

1. A Domain-specific language (DSL) implementing a Field Calculus on which aggregate programs are based. The DSL is defined as:
   - Internal: cause the language is both founded and typed using Scala;
   - Modular: as the solution is kept easy, concise and reusable;
   - Complete: because it includes all the field calculus constructs;
   - It supports a high-order field calculus;
2. A Virtual Machine (VM) to interpret the DSL;
3. A Simulator and a Graphic User Interface (GUI) to execute aggregate applications.

With these tools, ScaFi can both define and execute distributed systems implementing aggregate computing applications. Since aggregate computing may target a wide range of devices (with different hardware and software specifications), the system has been built to provide *flexibility* and *portability*.

The DSL is defined as *complete* because it provides implementations for:

- function definition and call;
- build-in operator call;
- *nbr*, interaction;
- *rep*, time evolution;
- *if*, domain restriction.

In addition, other several entities must be defined to recreate the whole system such as network, environment, system boundary. Another important entity in ScaFi is the device, which includes:

- A structure: sensors and actuators;
- A behaviour: the local aggregate computation with a defined frequency;
- An interaction:
  - broadcast and receive devices state;
  - sense the environment;

ScaFi, as we have already said, is an Being an Internal DSL. This implies that its implementation is based on the host language syntax and semantics, that in this case is Scala. Scala providing first-class functions and a both rich and static type system is the perfect environment on which to develop the field calculus system.

4.2 ScaFi Architecture

The ScaFi architecture can be distinguished into different modules:

- *scafi-commons*: including basic entities such as spatial and temporal abstractions;
- *scafi-core*: including the DSL implementation and standard library;
- *scafi-simulator*: including functionalities to simulate the execution of aggregate systems;

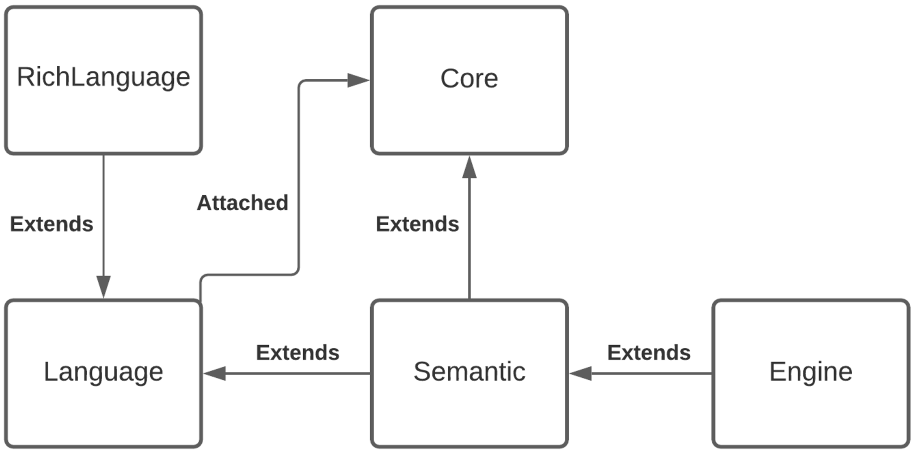


Figure 4.1: Scheme of Core module

- *scafi-simulator-gui*: including a GUI to visualise the simulation of the aggregate system;
- *spala*: including the actor-based platform;
- *scafi-distributed*: including the ScaFi integration for the module *spala*.

In this chapter, we will discuss more in detail the first three modules of ScaFi because they will be subject of the main research of the paper.

4.2.1 ScaFi Core

The Core module is composed of several components. Here, we describe its structure showing the relationship between the main components. First, a *Core* component is used to define the basic elements. Then, a *Language* entity is based on the core definitions and implements the constructs of the DSL. This last component is also extended by *RichLanguage*. The *Semantic* extends both the core and language to provide the semantics. This component is then made executable with the *Engine* that extends it. A simplified scheme of the Core module is shown in Figure 4.1.

4.2.2 ScaFi Commons

In the Commons module, a *SpatialAbstraction* component defines the notion of *neighbouring relationship* modelling the space where the elements are positioned.

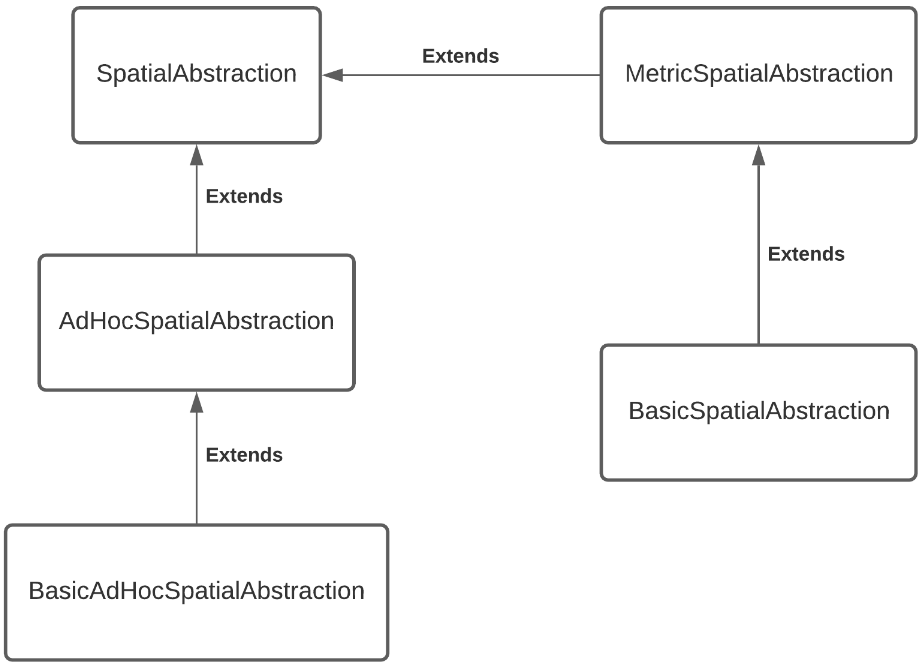


Figure 4.2: Scheme of Common module

Then, this component is extended by *AdHodSpatialAbstraction*, that models a specific situation of the network where: every node is located differently and the neighbouring relation is a function on the domain $\overline{P} \rightarrow P$, with $\overline{P}$ being the position of the element and P a set of possible positions. Another component extending SpatialAbstraction is *MetricSpatialAbstraction* that deals with those situations where a distance is used to calculate the position of the nodes. Additionally, both *BasicAdHocSpatialAbstraction* and *BasicSpatialAbstraction* are used to merge the Euclidean metric with the space abstraction generating a three-dimensional space. A simplified scheme of the Commons module is shown in Figure 4.2

4.2.3 ScaFi Simulator

In the Simulation module, a *MetaActionManager* component is used to manage the action requests send to the simulator. Then, the *Simulation* component defines the platform-view of the running system extending *SimulationPlatform* (to access to the core module). The *SpatialSimulation* component simulates spatial networks and extends *SpaceAwarePlatform* and *Simulation*. A simplified scheme of the Simulator module is shown in Figure 4.3

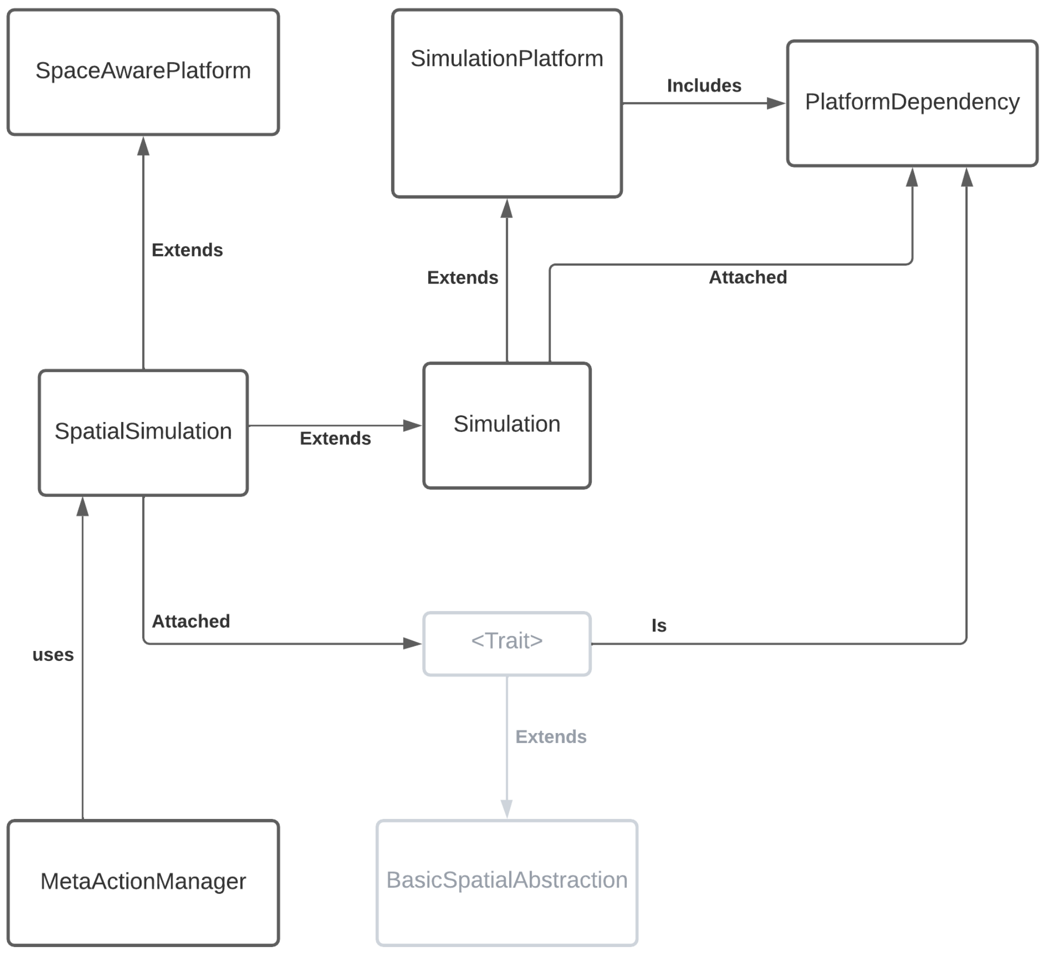


Figure 4.3: Scheme of Simulator module

4.3 ScaFi Standard Library

In this section, we analyse the ScaFi standard library describing some of its modules and giving an overview of the functions included. The library is composed of a basic semantic and several additional modules. Here, we explain the main packages that have been used in the following chapters. Those are FieldUtils, Gradients, BlockG, BlockC, BlockS, TimeUtil. A more in-depth exposition can be found in [21].

4.3.1 Basic semantic and syntax

The basic syntax and semantics of ScaFi is composed by the following definitions:

- `mid()`: Gets the ID of the node calling the function;
- `rep(init)(func)`: This is the implementation of the *rep* concept already explained in field calculus section 3.3;
- `nbr(e)`: This is the implementation of the *nbr* concept already explained in field calculus section 3.3;
- `aggregate(func)`: Creates sub-domains of nodes working on the same function *func*;
- `foldHood(init)(acc)(e)`: This function executes the expression *e* in every neighbour and collapses the evaluations obtained starting with an initial value *init* and merging the results according to the function *acc*;
- `branch(cond)(a)(b)`: This function implements a domain restriction based on the condition *cond*. If a node evaluates the condition as true it will execute the code block *a*, otherwise the block *b*;
- `mux(cond)ab`: Multiplexer, that depending on the Boolean value of the condition *cond* returns the expression *a* (if true) or *b* (if false);
- `sense(name)`: Retrieves the value of the sensor *name*;
- `nbrvar(name)`: Similar to *nbr* but works locally as a “environmental probe”.

4.3.2 Field-operation utilities (FieldUtils)

This package includes all those functionalities based on the aggregation of neighbours' values. These functions can be accessed by two objects: *includingSelf* and *excludingSelf* that respectively include or exclude the node calling the function from the computation.

Now we list and describe some of these functions:

- `sumHood(e)`: This function takes an expression *e*, executes it over the neighbourhood and returns the sum of all the evaluations;
- `unionHoodSet(e)`: This function takes a sequence of expressions *e*, executes them over the neighbourhood and returns the union of all the evaluations;

- `unionHood(e)`: This function takes an expression *e*, executes a unionHoodSet on it;

- `mergeHood(e)(overwritePolicy)`: This function works similarly to unionHood but takes a map (from expressions to values) called *e* and a function *overwritePolicy* used to collapse two values in one;

- `anyHood(e)`: This function takes an expression *e*, executes it over the neighbourhood and returns *true* if at least one node evaluates the expression to be true, otherwise it returns *false*;

- `everyHood(e)`: This function takes an expression *e*, executes it over the neighbourhood and returns *true* if all the nodes evaluate the expression as true, otherwise it returns *false*;

4.3.3 Gradients

In this package, the function *classicGradient* is defined as follows: **classicGradient(source, metric)**. It calculates the gradient, a computational field whose values describe the distance from the sources defined in the Boolean field *source*. And a *metric*, namely a 0-ary function, describing the distance between two nodes.

4.3.4 Gradient-cast (BlockG)

Here are collected functions that use gradients to deliver the result.
Some of them are:

- `G(source, field, acc, metric)`: This function offers a generalised gradient with parameterers: *source*, *field* (values spread through the network), *acc* (function evolving the values) and *metric*;

- `distanceTo(src, metric)`: This function calculate the distance of every node from a source *src* and with a distance *metric*;

- `broadcast(src, field, metric)`: This function broadcast a *field* from a node *src* to every one within a distance of *metric*;

- `channel(src, target, width)`: This function define a path connecting a starting node *src* and a ending node *target* with a specified *width*;

4.3.5 Collect-cast (BlockC)

This package implements the function *C(potential, acc, local, Null)*. And, as already said in chapter 3, it does the opposite of *G* by collecting the value *local* in a *potential* field and collapsing the values according to the function *acc*.

4.3.6 Leader Election (BlockS)

This package implements the function *S(grain, metric)*. And, as already said in chapter 3, it elects a set of leaders with *grain* the mean distance between two and with a definition of distance named *metric*.

4.3.7 Time Utilities (TimeUtils)

Here, we have implemented all the time-related functions. The most important one is *T(initial, floor, decay)* what simply create the time-abstraction decreasing the *initial* value according to a function *decay* until it reaches the *floor* value. All the other functions in this package are variants of *T*.

4.4 Other relevant projects

We now want to introduce some other projects that are based on aggregate computing and are tightly connected with ScaFi.

4.4.1 ScaFi-web

ScaFi-web [7] is an under-development project that can be used to quickly develop ScaFi applications and simulate them using customised networks.

This web app also allows us to use already developed codes to promptly observe in action the ScaFi standard library. Moreover, the *advanced* mode can be disabled to see the pure ScaFi code or enabled to simplify the code helping the learning process.

As already said it is possible to customise the network on which the program runs, this can be done by choosing: the displacement (grid or random), rows and columns numbers, step size over the axis X and Y, tolerance, radius, LED's colour and size, nodes' sensors (name and value).

To conclude also the executing speed can be chosen between three different modalities.

4.4.2 Alchemist simulator

Alchemist [43, 41] is a simulator for chemical-oriented computational systems that can easily be used to simulate ScaFi distributed computations. It can be seen as more advanced alternative to the built-in simulator that can be found in ScaFi (modules: scafi-simulator, scafi-simulator-gui).

Chapter 5

Analysis

In this chapter, we start introducing the core focus of this research (Section 5.1) describing why portability is such an important topic in ScaFi. The, we will state the requirements and set the goals for our implementation, which aims to improve ScaFi's portability using Cross-project, and more specifically Scala Native (Section 5.2).

5.1 ScaFi portability

In ScaFi, and every other framework for the development of Collective Adaptive Systems (CAS), the *portability* is a fundamental property. It provides to the library the capability of being used over a wide range of devices with different physical characteristics (*Hardware-independent*) and with few software constraints (*Software-independent*).

Let us define these two terms:

- **Hardware Independent**: When a software can successfully run on every equipment. Some possible limitations to this concepts can be:
  - Memory requirements;
  - Processing power requirements;
  - Architecture limitations (not executable on different types of hardware architectures such as x64 or arm64).
- **Software Independent**: When the software can successfully run on every piece of equipment, without depending on the software already present in the machine. There can be several possible limitations and here we list some of them:

  - Operative System requirements;
  - Intermediate program required;
  - Translator program required.

These two concepts are utopias and can never be completely achieved. Every software implementation must have some hardware or software limitations. However, they represent two possible directions to which improve the flexibility and portability of a system.

A solution to improve *Software independence* in ScaFi is by integrating Cross-compilation to successfully execute ScaFi applications in environments that do not provide a JVM. As we explain in the previous chapters, this has already been partially done through the plugin "sbt-crossproject" and with Scala.js. However, we want to continue that work integrating Scala Native.

5.2 Requirements

We now define the requirements of the ScaFi implementation illustrated in the next chapter:

1. *ScaFi* will be integrated with Scala Native:
   - over the following modules: *Core*, *Commons*, *Simulator* and *Tests.*
   - offering customised settings to improve the performances with:
     - Garbage-collectors;
     - Link-time optimisations;
     - Compilation modes.
   - generating light-weight binaries;
   - offering relatively good run-time performances.
   - with the least impact over the whole project: avoiding numerous changes of the standard code base.
2. The integration will be tested and analysed:
   - with the standard test routine to test the integration with all versions Scala supported by ScaFi;
   - with personalised and complete tests to evaluate the run-time performances and average binary size:
     - testing the basic semantic;

  - testing the basic code modules;
  - testing high level patterns (i.e SCR and Channel);
  - testing single and multi processes;
- comparing the results obtained between different Scala versions, compilation modes and with the performances of JVM and Scala.js.

Chapter 6

Implementation

In this chapter, we show how Scala Native has been integrated in ScaFi. Starting with how we added and enabled the plugin (Section 6.1). Continuing with the core of the integration (Section 6.2), inspired by the one already present of Scala.js. Concluding listing the conflicts caused by unimplemented Java libraries in the Native compiler (Section 6.3).

6.1 Plugins

To successfully use Scala Native plugin in a cross-platform project only a few lines of code are needed. First you need to add the plugin dependencies of Scala Native (sbt-scala-native) and Scala portable for native (sbt-scala-native-crossproject) in the *plugins.sbt* (Listing 6.1). We decided to use respectively the versions 0.4.3 and 1.1.0 that are the latest stable releases.

Listing 6.1: Scala Native Plugins - Source: project/plugins.sbt

```
// Scala Native plugins

addSbtPlugin("org.scala-native" % "sbt-scala-native" % "0.4.3")

addSbtPlugin("org.portable-scala" % "sbt-scala-native-crossproject" % "1.1.0")
```

It is also necessary to enable the plugin in the file "build.sbt" for all the modules that will support the Native compilation. In the analysis phase, we decided to apply Scala Native to commonsCross, coreCross, simulatorCross and TestsCross (in the same way that it has been done with Scala.js).

This can be easily done with three steps:

- Import the build package (Listing 6.2): We need to import all the utilities to enable the plugin. Later we will use other tools from this package to set the settings for the compilation;

Listing 6.2: Scala Native Import - Source: project/build.sbt

```scala
// Imports
import scala.scalanative.build._
```

- Update project lazy val Listing 6.3: We also have to add between the files in the val *scafi* all those that will be generating use Scala Native such as coreCross.native, commonsCross.native, simulatorCross.native, testsCross.native (in the same way we did for Scala.js);

Listing 6.3: Scala Native Files - Source: project/build.sbt

```scala
// Update files
lazy val scafi = project.in(file(".")).aggregate(
 core, commons, spala, distributed, simulator, `simulator-gui`, `renderer-3d`,
 `stdlib-ext`, `tests`, `demos`, `simulator-gui-new`, `demos-new`,
 `demos-distributed`, coreCross.js, commonsCross.js, simulatorCross.js, testsCross.js,
 coreCross.native, commonsCross.native, simulatorCross.native, testsCross.native)
 //...
```

- Enable the plugin in the modules Listing 6.4: Doing this is essential if we want to have running Scala Native in a module and we have to do it for every single one that will have to compile in the native language.

Listing 6.4: Scala Native Enable Plugin - Source: project/build.sbt

```scala
// Add Native Platform
lazy val commonsCross =
 crossProject(JSPlatform, JVMPlatform, NativePlatform).in(file("commons"))
lazy val coreCross =
 crossProject(JSPlatform, JVMPlatform, NativePlatform).in(file("core"))
lazy val simulatorCross =
 crossProject(JSPlatform, JVMPlatform, NativePlatform).in(file("simulator"))
lazy val testsCross =
 crossProject(JSPlatform, JVMPlatform, NativePlatform).in(file("tests"))
```

Once done these three steps, it is possible to continue the integration of Scala Native with the module's code.

6.2 Integration

To do the integration we first disable the linking errors caused by stubs, because we want to be sure that every function is implemented. To do it is enough to set the variable to false Listing 6.5.

Listing 6.5: Scala Native disable stub errors - Source: project/build.sbt

```
// Set to false or remove if you want to show stubs as linking errors
nativeLinkStubs := false
```

We also decided to customise the native compilator setting using the commix garbage collector to increase the run-time performances Listing 6.6. Of course, these configurations only determine the type of native compilation of the ScaFi library, every project that makes use of it can choose different compilation modes and settings.

Listing 6.6: Scala Native settings - Source: project/build.sbt

```
lazy val commonNativeSettings = Seq(
 nativeConfig ~= {
  _.withLTO(LTO.none)
   .withMode(Mode.default)
   .withGC(GC.commix)
 }
)

lazy val commonsCross = //..
 .nativeSettings(commonNativeSettings: _*)
lazy val coreCross = //...
 .nativeSettings(commonNativeSettings: _*)
lazy val simulatorCross = //...
 .nativeSettings(commonNativeSettings: _*)
lazy val testsCross = //...
 .nativeSettings(commonNativeSettings: _*)
```

Since different platforms manage the stack differently, we had to define an object *PlatformDependentConstants* including two different variables (used to access the stack trace) that will be defined with different values depending on the platform. This is what has been done with Scala.js and the same has been done to integrate Scala Native (Listing 6.7). We assigned the value "5" to the val *CallerClassPosition*, that is used to select the caller class, and "6" to *StackTracePosition*, used to access the stack trace. We found these values trying different combinations and analysing the errors obtained running the module testsCross.

Listing 6.7: Scala Native constants - Source: PlatformDependentConstants.scala

```
package it.unibo.scafi

object PlatformDependentConstants {
 val CallerClassPosition = 6
 val StackTracePosition = 5
}
```

6.3 Conflicts with Java Libraries

In this section, we list the library conflicts caused by the usage of Scala Native and we propose our solutions. This happened because the native compiler compiles without the JVM and so without all the Java libraries. Some of them have been implemented even in Scala Native, however, for the majority of them, an alternative solution must be found.

6.3.1 Java.lang.Thread

The current version of ScaFi utilises the library Java.lang.Thread to access get the stack trace Listing 6.8.

Listing 6.8: Scala Native stacktrace (Old) - Source: Semantics.scala

```scala
// Get stacktrace using Java.lang.Thread - Not implemented in SN
override def elicitAggregateFunctionTag(): Any = Thread
 .currentThread()
 .getStackTrace()(PlatformDependentConstants.StackTracePosition)
```

To solve this problem we found another way to access the stack trace but using libraries implemented in Scala Native. In fact, an equivalent way to access the stack trace is through the object *Throwable* using the method *getStackTrace* Listing 6.9.

Listing 6.9: Scala Native stacktrace (New) - Source: Semantics.scala

```scala
// Get stacktrace using Java.lang.Throable - Implemented in SN
override def elicitAggregateFunctionTag(): Any =
 new Throwable().getStackTrace()(PlatformDependentConstants.StackTracePosition)
```

Even if Throwable offers an access to the stack trace with the same performance, its implementation is slightly different. For this reason, we had to modify the PlatformDependentConstants of the JVM decreasing them by 1. Their new values are listed below:

- **v2.11**: CallerClassPosition = 3, StackTracePosition = 3;
- **v2.12**: CallerClassPosition = 5, StackTracePosition = 4;
- **v2.13**: CallerClassPosition = 5, StackTracePosition = 4;

6.3.2 Java.time

ScaFi uses Java.time both in the modules commons and simulator to use the following classes: Instant, ChronoField, ChronoUnit and TemporalField. These classes are not yet implemented in Scala Native and no direct alternative exists. For these reasons we decided to use a external plugin that implements the package *time* using Scala (scala-java-time [2]) Listing 6.10.

Listing 6.10: Scala Native Time library - Source: project/build.sbt

```scala
// Used library as alternative to Java.time (Not implemented in SN)
```

```
lazy val commonsCross = //...
 .nativeSettings(
  libraryDependencies += "io.github.cquiroz" %%% "scala-java-time" % "2.4.0-M1"
  )

lazy val simulatorCross = //...
 .nativeSettings(
  libraryDependencies += "io.github.cquiroz" %%% "scala-java-time" % "2.4.0-M1"
  )
```

6.3.3 Plugin scoverage

Another problem has been encountered with the CI during the testing phase. The CI is built to do a code coverage after the build testing using the plugin *scoverage*. However, it cannot be used with Scala Native at the moment. We did not find a quick alternative for this problem and we decided to disable this check only for the native part of the application. This is done changing the value of the variable *coverageEnabled* inside *nativeSettings* Listing 6.11.

Listing 6.11: Scala Native Coverage - Source: project/build.sbt

```
// To disable coverage in native settings
coverageEnabled := false
```

Chapter 7

Validation

In this chapter, we describe how the integration of Scala Native illustrated in the previous chapter has been tested. To do it, we use a benchmark for native tests modifying it to include: our tests, Scala configurations and test configurations. Here, we first show how the benchmark "scala-native-benchmark" works (Section 7.1), then how it has been adapted to support our testing criteria and the tests used (Section 7.2). We conclude by analysing the results obtained (Section 7.3).

7.1 Benchmark for Scala Native

A valid benchmark for Scala Native applications can be found in the GitHub repository [3] called "scala-native-benchmark". This code-base has been useful to create the platform used to validate our implementation. For this reason, we will first illustrate how this code is structured (this will be useful to understand how we modified it in the next section) and how it can be used.

7.1.1 General Structure

This benchmark consists of a collection of Python scripts that can test a group of Scala files following certain configurations.

Configurations: ***confs***

This folder includes all the possible configurations which with the Scala code can be compiled and run. Here, it is possible to include compiling settings. and switch from the JVM to the Scala Native compiler.

A single configuration may normally include the following files:

- *build.properties*: Where it is possible to define the "sbt-version";

Listing 7.1: Test example

```scala
abstract class Benchmark {
 def run(input: String): Any

 def main(args: Array[String]): Unit = {
  //...
 }

 // Other methods...
}

object MyTest extends communitybench.Benchmark {

 def run(input: String): Int = {
  // Test code...
 }
 // To use the imput args
 override def main(args: Array[String]): Unit =
  super.main(args)
}
```

- *compile*: Where it have to be specified the command to use to compile the program:
  - JVM: compile;
  - JS: fullLinkJS;
  - Scala Native: nativeLink.
- *build.sbt*: Here, as in every build.sbt file we should specify the name of the project, the version of Scala that has to be used, the dependencies and the settings of the plugins used in the project;
- *plugins.sbt*: Here, we list the dependencies of the plugins;
- *run*: Where we have to specify the command to execute to run the program following the format: "target/scala-2.%version%/%build-name%-out" (for Scala native compilations).

Source code: *src/main/scala*

In this folder, the tests' code are included. It is suggested to use a different package for every test. In addition inside the package *communitybench* it is possible to find the definition of the abstract class *Benchmark* that have to be inherited to properly define a test. It must be implemented the method *run* that will contain the code of the test that have to be executed (Listing 7.1).

Input and Output: ***input*,*output***

The folder *input* is used to pass the arguments to the projects that have to be run. It is necessary to create a file for every different test and its name must follow the pattern: “%package-name%.%file-name%”. Inside it is possible to specify the parameters that have to be passed during the test.

The folder *output* is used to specify the output that a test should produce and are fundamental to understand whether or not a test is successful. In the same way, as we did for the input, here every file must be named following the pattern: “%package-name%.%file-name%”.

Scripts: ***scripts***

This folder contains the Python scripts that will make this benchmark work and several constants such as:

configuration.py

default_runs: Defining the number of times the test must be executed (default: 2000);
default_batches: Defining the number of times that the code must be tested during a single run (default: 20).

benchmarks.py

benchmarks: Array defining the tests and the order which with they will be executed.

cmdline.py

latest: Name of the configuration to be used if others are not specified;
stable: Name of the configuration to be used if others nor *stable* are not specified.

comparison.py

default_warmup: Number of iterations to skip before calculating percentiles (default: 500);

7.1.2 How to use

To properly use the benchmark it is necessary to first create and fill the following files:

- ./build.sbt;
- ./project/plugins.sbt (optional);

Then, it will be possible to use the following command:

python3 script/run.py

To generate the following folders

- binaries: Containing all the binaries produced (one for each test case);
- results: Containing all the statistics generated by the runs (the execution time of every run and batch) and the compilation times;

Another useful command is:

python3 script/summary.py

That will generate a folder *reports* including a comparison of the configurations that have been tested using the previous command.

python3 script/run.py

Finally, the file *notebook.ipynb* defines *numpy* statements for generating graphs using the data inside the folder *results*.

7.2 Benchmark for ScaFi

We modelled this benchmark to support our tests [6, 36] and modified some constants, to make it easy to fire the tests using the same commands that are shown in the previous section. Now, we explain the modifies that has be done in detail:

- configurations: we implemented a configuration folder for Scala Native with Scala 2.11.x, 2.12.x and 2.13.x (one folder of each situation). There are three configurations with the version 2.12.x that define different compilation modes (releaseFull, releaseFast, default);
- libraries: we added the .jar files of *scafi-commons*, *scafi-core* and *scafi-simulator* compiled using Scala Native or the JVM.
- output: we set every output file to the value "()", because every test will simply simulate a simple aggregate program and will return just a *Unit* value;

- scripts: changes constants to reduce the number of batches and runs, and to change the predefined benchmarks and configuration.

Then, for each test we created an object implementing the abstract class *benchmark*, inside its method *run* we defined the test code and the settings for the simulation Listing 7.2.

Listing 7.2: ScaFi Tests definition

```
// Definition of single test
object BuildingBlocksBundleBenchmark extends communitybench.Benchmark {

 def run(input: String): Unit = {
  val howMany = 5
  val range = 2
  val ticks = 10000
  val simulator = simulatorFactory.gridLike(
   GridSettings(howMany, howMany, range, range),
   range
  )
  (0 to ticks) foreach { _ => simulator.exec(new TestClass) }
 }

 override def main(args: Array[String]): Unit = super.main(args)
}
```

We now list (in alphabetical order) the tests that have been used, together with a brief description:

1. **BuildingBlocksBundleCheck**: used to verify the bundle size using all the building blocks;
2. **CCheck**: includes an usage of the function *C*, *hopDistance* and *mid*;
3. **ChannelCheck**: execute a high-level pattern *channel* between the node *0* and *10* with size *1*;
4. **FewProcessCheck**: generate few processes using the function *sspawn2*;
5. **FoldhoodCheck**: implements a basic usage of the function *foldHood*;
6. **FoldhoodAndNbrCheck**: implements a usage of the function *foldHood* calling *mid* and *nbr*;
7. **GCheck**: includes a possible implementation of the function *G* (to test the building G);
8. **GradientCheck**: tests *classicGradient*;
9. **ManyProcessCheck**: generate many processes using the function *sspawn2*;

10. **RepCheck**: implements a simple code using *rep*;

11. **SCheck**: includes a possible implementation of the function *S* (to test the building S);

12. **SCRPattern**: execute a high-level pattern *SCR* using the functions: *S*, *C*, *G*, *classicGradient*;

13. **taticFieldCheck**: implement a constant main with value *10*;

14. **TCheck**: includes a possible implementation of the function *T* with time *100* (to test the building T).

7.3 Analysis

This section analyses the results obtained by running these tests. In some situations, we could refer to the tests mentioning their number (according to the list shown in the previous section). We will separately analyse the performances observing the compilation times, binary sizes, start-up and execution times.

7.3.1 Compilation times

These tests have been done compiling the project with Scala Native using three different versions of Scala (2.11, 2.12, 2.13) - specifically the ones supported by ScaFi - and all the three compilation modes available. We have used the Scala Native compiler with *LTO.thin* and *immix* as the GC. We also added the results obtained compiling with the JVM to give a more wide perspective. The results obtained are shown in Figure 7.1.

Observing the chart it is possible to see how better is the compilation using versions 2.12 and 2.13. On the other hand, Scala 2.13 scored poorly in every test. Additionally, we can also appreciate how the compilation time is just noticeably superior in those tests where a high number of Bundles are extended, this means that, independently of the code written, the performance of the computation using Scala Native is quite stable. The reason behind this behaviour is the optimisation offered by the native compiler. Only the bundle's code that is actually used is also compiled and included in the final binary file. This can be noticed looking at the compilation times of *FewProcessCheck* and *ManyProcessCheck*.

If we look at the results using other compilation modes, we will see an important increase of the performance. This is due to the reduction of optimisation that will also generate a smaller binary but resulting in poorer execution performances.

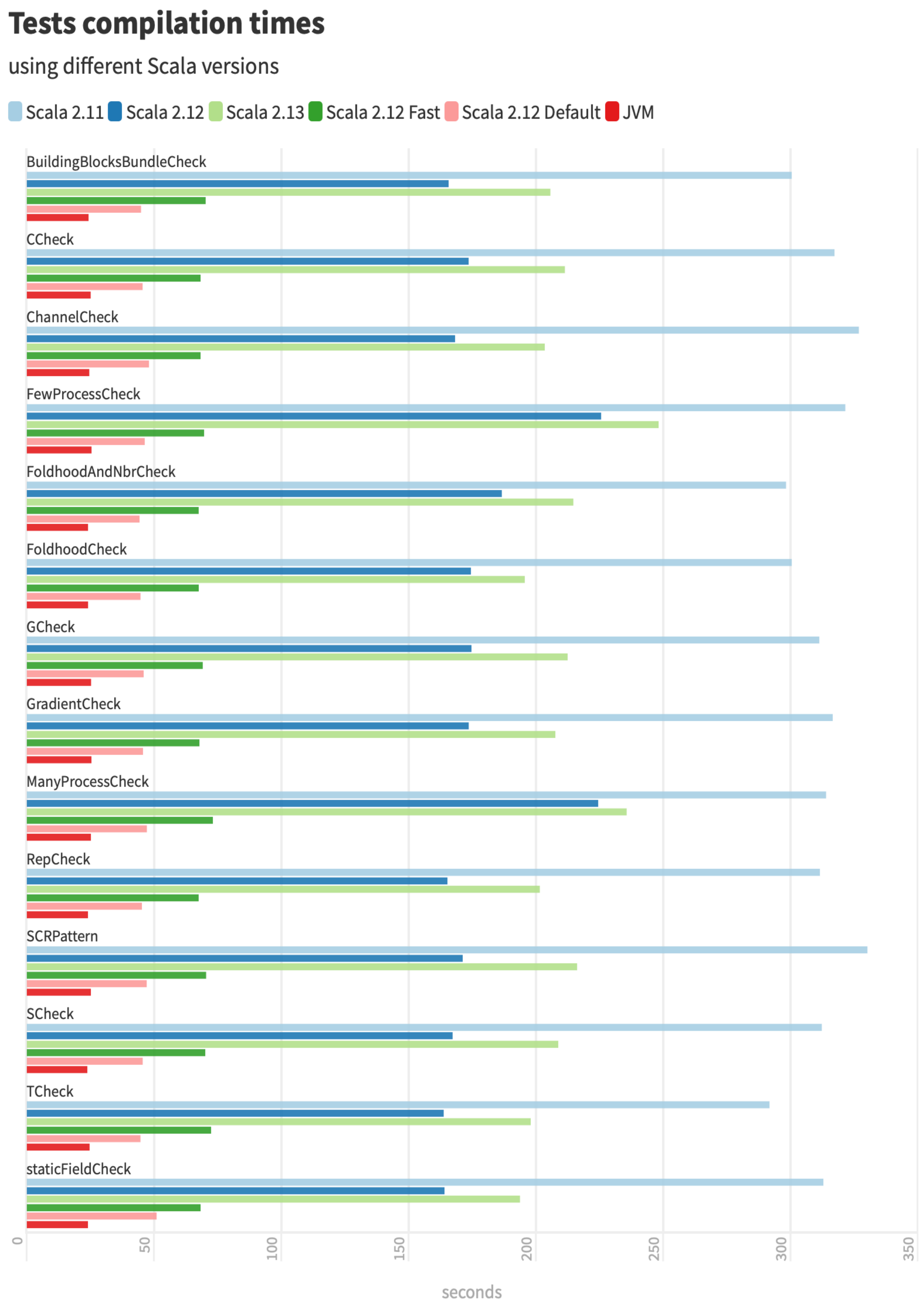


Figure 7.1: Compilation times - Bar char [35]

7.3.2 Binary sizes

Here, we compare the binary sizes of the compiled tests. First, we vary the Scala version then the compilation mode. The results obtained are shown in Figure 7.2. Three main analyses can be done over the bar plot proposed.

Comparison between Scala version

Firstly, it is easy to appreciate how the smallest binaries are generated using a Scala version 2.12. On the other hand, switching to 2.11 causes a considerate increase of the binary sizes. With 2.13 the result shows only an augment of a fraction of the original value.

Comparison between Compilation modes

Secondly, comparing the compilation modes we can see how the compilation is lighter using *ReleaseFast* or *Debug*. However, the run-time performance will be much worse.

Comparison between tests of a certain version

Lastly, if we see the binary size difference between the tests, it is possible to understand how those tests that extend a high number of Bundles are likely to produce bigger binaries. However, this difference is just a fraction and we can consider it irrelevant and that the optimisation made by the compiler is adequate.

7.3.3 Start-up times

Using Scala Native it is possible to generate binaries and to execute them instantly start-up, really useful in those scenarios where it is not acceptable to waste time waiting for the JVM.

7.3.4 Execution times

We then analysed the execution times of every test observing that, in general, every box plot can be well approximated with an hyperbole (due to the warm-up effect). This means, that the software is showing great stability. Some tests showed more stable results, others (in particular the most complexes such as the high-level patterns) showed more irregularities. Here, we include two antipode cases (Figure 7.3), the first (Test 4) represents the normal case where we have a stable execution from the start to the end (the number of irregularities reduces

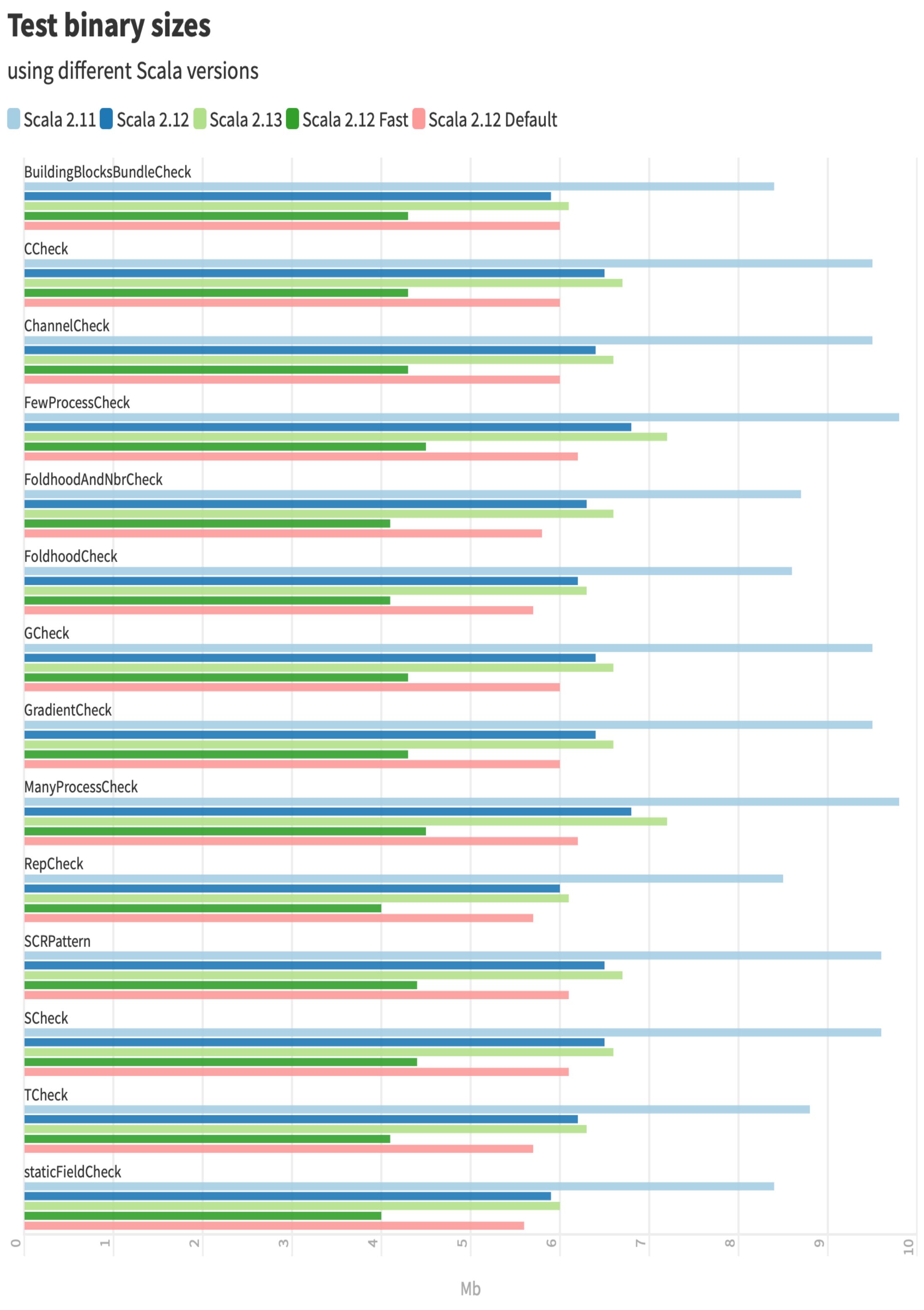
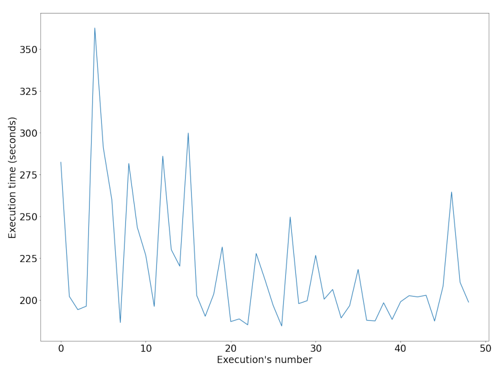
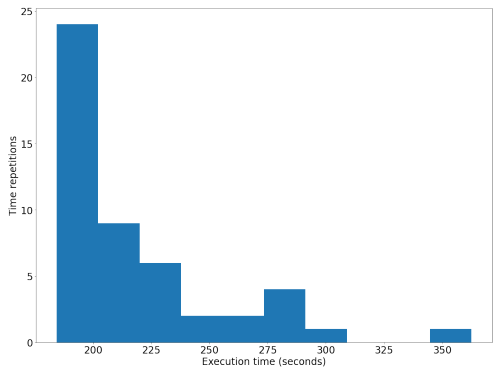


Figure 7.2: Binary sizes - Bar char [34]

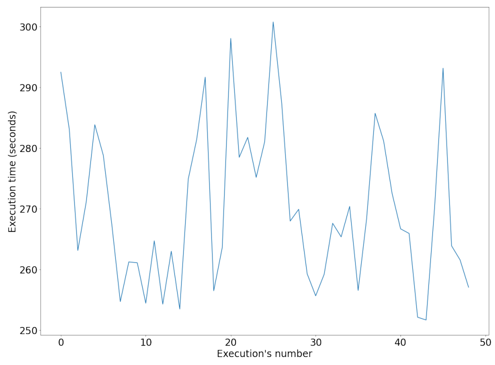
over time due to a warm-up effect), the second (Test 3) shows the worst result obtained with much more peaks from start to end.



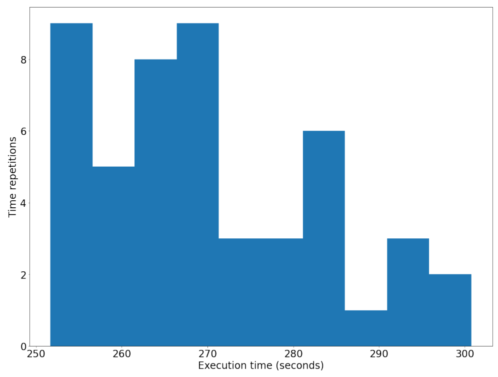
(a) Test 4 - Line Plot



(b) Test 4 - Box Plot



(c) Test 3 - Line Plot



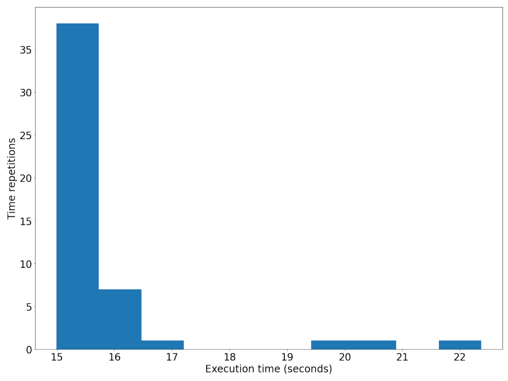
(d) Test 3 - Line Plot

Figure 7.3: Tests - execution times

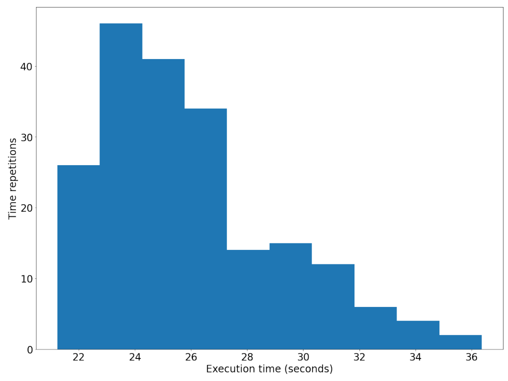
Another interesting comparison can be done between the results obtained running the same test with different native compilation modes and using the JVM (Figure 7.4). Here, as expected, the execution time increases in using fastRelease or default. We can also observe that the performance gap between fullRelease and the JVM is acceptable, with an execution time of the native compilation of just a few seconds more. To appreciate the difference we include a table that shows the percentiles and mean of each mode (Table 7.1).

7.3.5 Considerations

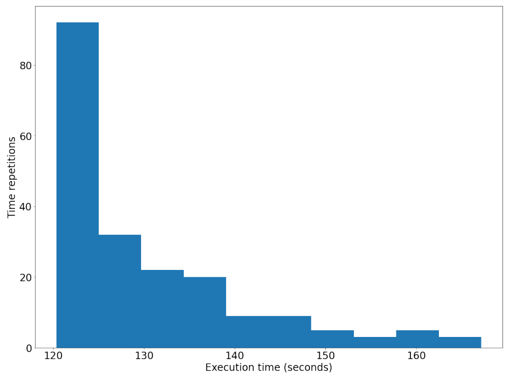
After these analyses, we can affirm that Scala Native has been successfully integrated with ScaFi. In fact, both the compilation time and execution time are regular and stable in a wide range of different tests and with different compilation



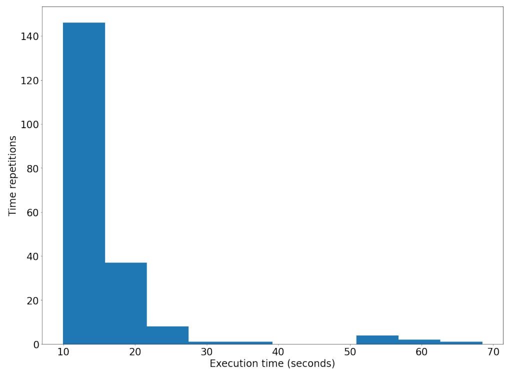
(a) fullRelease



(b) fastRelease



(c) default



(d) jvm

Figure 7.4: Test 10 - Execution with different compilation modes

	FullRelease	FastRelease	Default	JVM
5th percentile	15.181	22.465	120.066	10.068
50th percentile	15.476	25.497	122.889	10.586
95th percentile	18.360	32.131	155.379	20.776
Mean	15.842	26.131	128.317	12.474

Table 7.1: Test 10 - Calculation over different compilation modes fig. 7.4

settings. Other considerations about the final result of this project and possible future improvements are included in the next and last chapter.

Chapter 8

Conclusions

This chapter contains general observations about the work presented in this thesis as well as some references to potential future advances Section 8.1.

The work illustrated in this document managed to successfully integrate Scala Native in ScaFi. Additionally, we validated the implementation of the common patterns doing analyses of the binaries, execution times and compilation times. We proved that the integration is stable and produces competitive performances. As a consequence, it is now possible to execute ScaFi codes over devices that are supporting neither the JVM nor JavaScript. This has been a step toward** for the ScaFi project and a notable improvement for its portability and flexibility, very desirable features of every framework based on aggregate or distributed computing. With this integration, we widened the possible applications of ScaFi opening its way toward the world of IoT, robotic swarms and embedded systems. All those other scenarios where the device due to a lack of memory or other hardware limitations cannot support the JVM.

This strategy, of offering cross-platform solutions, is a modern programming approach. Several other modern programming languages have implemented this feature, such as Rust and Kotlin. The reason is that with the current technological progress we need to provide software solutions that are, as much as possible, both hardware and software independent, without having to implement different codebases.

8.1 Agenda

We proved the effectiveness of this integration. However, other improvements must be done to consider it complete. Future works may do research toward the following directions:

- Enable "releaseFull" or "releaseFast" as compilation modes in the ScaFi project. It would increase the code optimisations and, as a consequence, the run-time performances of the library's modules;

- Compare the performances between the different platforms supported, such as Scala.js and Scala Native.

- Execute a comprehensive test of ScaFi with embedded systems and analyse the results. Our tests have been done on a Windows machine, however, testing the integration directly on devices with different architectures and smaller memory spaces could show interesting insights.

Bibliography

[1] Github repository - crossproject: Cross-platform compilation support for sbt. `https://github.com/portable-scala/sbt-crossproject`.

[2] Github repository - scala-java-time: Repository - implementation of the java.time package. `https://github.com/cquiroz/scala-java-time`.

[3] Github repository - scala-native-benchmark: work-in-progress modernization of scala native benchmarks. `https://github.com/scala-native/scala-native-benchmarks`.

[4] Online documentation - kotlin: a cross-platform, statically typed, general-purpose programming language. `https://kotlinlang.org/docs/home.html`.

[5] Online documentation - rust: a multi-paradigm, general-purpose programming language. `https://www.rust-lang.org`.

[6] Gianluca Aguzzi. Github repository - scafi tests for modules: Commons, core and simulator. `https://github.com/cric96/scafi-benchmark`.

[7] Gianluca Aguzzi, Roberto Casadei, Niccolò Maltoni, Danilo Pianini, and Mirko Viroli. Scafi-web: A web-based application for field-based coordination programming. In *International Conference on Coordination Languages and Models*, pages 285–299. Springer, 2021.

[8] Vincenzo Ambriola, Paolo Ciancarini, and Marco Danelutto. Design and distributed implementation of the parallel logic language shared prolog. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 40–49, 1990.

[9] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. Space-time universality of field calculus. In *International Conference on Coordination Languages and Models*, pages 1–20. Springer, 2018.

[10] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 91–100, 2017.

[11] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. Optimally-self-healing distributed gradient structures through bounded information speed. In *International Conference on Coordination Languages and Models*, pages 59–77. Springer, 2017.

[12] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. Optimally-self-healing distributed gradient structures through bounded information speed. In Jean-Marie Jacquet and Mieke Massink, editors, *Coordination Models and Languages*, pages 59–77, Cham, 2017. Springer International Publishing.

[13] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic (TOCL)*, 20(1):1–55, 2019.

[14] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[15] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the internet of things. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(3):1–29, 2017.

[16] Viroli Mirko Beal Jacob. Space–time programming. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences A.373:20140220, 2015.

[17] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[18] Roberto Casadei. Thesis - aggregate programming in scala: a core library and actor-based platform for distributed computational fields. 2014.

[19] Roberto Casadei, Alessandro Aldini, and Mirko Viroli. Towards attack-resistant aggregate computing using trust mechanisms. *Science of Computer Programming*, 167:114–137, 2018.

[20] Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. *Future Internet*, 12(11), 2020.

[21] Roberto Casadei and Mirko Viroli. Online documentation - scafi: Scala-based library and framework for aggregate programming. `https://scafi.github.io`.

[22] Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Engineering Applications of Artificial Intelligence*, 97:104081, 2021.

[23] Sébastien Doeraene. Scala.js: Type-directed interoperability with dynamically typed languages. Technical report, 2013.

[24] Matt Duckham. Decentralized spatial computing: Foundations of geosensor networks. Springer International Publishing, 2013.

[25] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley Professional, 1999.

[26] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[27] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[28] Cay S Horstmann. *Scala for the Impatient.* Pearson Education, 2012.

[29] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[30] Amy Kumar, Jacob Beal, Soura Dasgupta, and Raghu Mudumbai. Toward predicting distributed systems dynamics. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 68–73, 2015.

[31] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[32] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[33] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In *International Workshop on Engineering Societies in the Agents World*, pages 68–81. Springer, 2002.

[34] Kevin Mancini. Flourish studio online - interactive bar plot for binary sizes. `https://public.flourish.studio/visualisation/8758750/`.

[35] Kevin Mancini. Flourish studio online - interactive bar plot for compilation times. `https://public.flourish.studio/visualisation/8757579/`.

[36] Kevin Mancini. Github repository - scafi testing benchmark. `https://github.com/KevinManciniHull/scala-native-benchmarks`.

[37] Martin Odersky. Online documentation - scala levels: beginner to expert, application programmer to library designer. `https://www.scala-lang.org/old/node/8610`.

[38] Martin Odersky. Online documentation - what is new in scala 3. `https://docs.scala-lang.org/scala3/new-in-scala3.html`.

[39] Martin Odersky et al. Online documentation - the scala programming language. *http://www.scala-lang.org*, 15, 2008.

[40] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[41] D Pianini, S Montagna, and M Viroli. Online documentation - alchemist: Chemical-oriented simulation of computational systems. `https://alchemistsimulator.github.io`.

[42] Danilo Pianini and Matteo Francia. Github repository - the protelis project: Language and tools for practical aggregate programming. `https://github.com/Protelis`.

[43] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.

[44] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1846–1853, 2015.

[45] Denys Shabalin. Just-in-time performance without warm-up. Technical report, EPFL, 2020.

[46] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

[47] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[48] Bill Venners. Scala's stackable trait pattern. `https://www.artima.com/articles/scalas-stackable-trait-pattern`.

[49] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(2):1–28, 2018.

[50] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From field-based coordination to aggregate computing. In *International Conference on Coordination Languages and Models*, pages 252–279. Springer, 2018.

[51] Mirko Viroli, Jacob Beal, and Kyle Usbeck. Operational semantics of proto. *Science of Computer Programming*, 78(6):633–656, 2013.

[52] Mirko Viroli, Matteo Casadei, and Andrea Omicini. A framework for modelling and implementing self-organising coordination. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1353–1360, 2009.

[53] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *European Conference on Service-Oriented and Cloud Computing*, pages 114–128. Springer, 2013.

[54] Richard Whaling. Online documentation - scala native: an optimizing ahead-of-time compiler and lightweight managed runtime for scala. `https://scala-native.readthedocs.io/en/latest/`.

[55] Richard Whaling. *Modern Systems Programming with Scala Native: Write Lean, High-Performance Code without the JVM.* Pragmatic Bookshelf; 1st edition, 2020.

[56] William L. Hosch. Britannica, the editors of encyclopaedia. field. encyclopedia britannica. `https://www.britannica.com/science/field-physics`, 2020.