ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
Campus di Cesena

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

Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields

Tesi in
Ingegneria dei Sistemi Software Adattativi Complessi

Relatore:
Prof. MIRKO VIROLI

Presentata da:
ROBERTO CASADEI

Anno Accademico 2014-2015
Sessione III

Contents

List of Figures

Abstract (italiano)

La programmazione aggregata è un paradigma che supporta la programmazione di sistemi di dispositivi, adattativi ed eventualmente a larga scala, nel loro insieme – come aggregati. L'approccio prevalente in questo contesto è basato sul field calculus, un calcolo formale che consente di definire programmi aggregati attraverso la composizione funzionale di campi computazionali, creando i presupposti per la specifica di pattern di auto-organizzazione robusti.

La programmazione aggregata è attualmente supportata, in modo più o meno parziale e principalmente per la simulazione, da DSL dedicati (cf., Protelis), ma non esistono framework per linguaggi mainstream finalizzati allo sviluppo di applicazioni. Eppure, un simile supporto sarebbe auspicabile per ridurre tempi e sforzi d'adozione e per semplificare l'accesso al paradigma nella costruzione di sistemi reali, nonché per favorire la ricerca stessa nel campo.

Il presente lavoro consiste nello sviluppo, a partire da un prototipo della semantica operazionale del field calculus, di un framework per la programmazione aggregata in Scala.

La scelta di Scala come linguaggio host nasce da motivi tecnici e pratici. Scala è un linguaggio moderno, interoperabile con Java, che ben integra i paradigmi ad oggetti e funzionale, ha un sistema di tipi espressivo, e fornisce funzionalità avanzate per lo sviluppo di librerie e DSL. Inoltre, la possibilità di appoggiarsi, su Scala, ad un framework ad attori solido come Akka, costituisce un altro fattore trainante, data la necessità di colmare l'abstraction gap inerente allo sviluppo di un middleware distribuito.

Nell'elaborato di tesi si presenta un framework che raggiunge il triplice obiettivo: la costruzione di una libreria Scala che realizza la semantica del field calculus in modo corretto e completo, la realizzazione di una piattaforma distribuita Akka-based su cui sviluppare applicazioni, e l'esposizione di un'API generale e flessibile in grado di supportare diversi scenari.

Keywords – *aggregate programming, field calculus, distributed framework, self-organisation, actors, scala*

Abstract

Aggregate programming is a paradigm that addresses the development of large-scale, adaptative systems in their totality – as aggregates. One prominent aggregate programming approach is based on the field calculus, a formal calculus that allows the definition of aggregate programs by the functional composition of computational fields, enabling the specification of robust self-organisation patterns.

Currently, aggregate programming is supported, at varying degrees, by ad-hoc DSLs (cf., Protelis) which are mainly devoted to simulation, but there are no frameworks for mainstream languages aimed at the construction of real-world systems. Still, such a technology would be highly desirable in its role of promoting the use and adoption of the paradigm in practice as well as for boosting research in the field.

The work described in this thesis consists in the development of an aggregate programming framework in Scala, starting from an existing prototype implementing the operational semantics of the field calculus.

The choice of Scala as the host language is motivated by both technical and practical reasons. Scala is a modern language for the JVM which integrates the object-oriented and functional paradigms in a seamless way, has an expressive type system, and provides advanced features for the development of software libraries. Moreover, the possibility to employ a sound, Scala-based actor framework such as Akka is another leading factor for this commitment to Scala, given the critical abstraction gap inherent to the development of a distributed middleware.

In this dissertation, I present a framework achieving the threefold goal: i) the construction of a Scala library that implements the field calculus semantics in a correct and complete way, ii) the development of an Akka-based distributed platform for aggregate applications, and iii) the creation and exposition of an general, flexible API able to support various distributed computing scenarios.

Keywords – *aggregate programming, field calculus, distributed framework, self-organisation, actors, scala*

Introduction

Aggregate programming is a paradigm that supports the programming of large-scale, adaptative systems in their totality, by focussing on the behavior of the whole – an aggregate of devices – rather than on the behaviors of the parts. This programming model represents a depart from the traditional device-centric approaches and is intended to overcome challenges commonly found in the development of pervasive computing applications.

One prominent aggregate programming approach is based on the field calculus, a formal calculus that allows the definition of aggregate programs by the functional composition of computational fields , enabling the specification of robust self-organisation patterns. Founding the model on a minimal, formal calculus at the foundation allows to formally verify interesting properties (e.g., self-stabilisation) and guarantee that certain feed-forward compositions of operators maintain such properties. Given this premise, it is possible to envision a set of building blocks for the development of resilient, adaptative systems.

In the past few years, a number of computing models and languages falling more or less within the umbrella of aggregate programming have emerged. However, existing aggregate programming initiatives are often very specific – addressing a well-defined niche of applications –, built on ad-hoc Domain-Specific Languages or DSLs (cf., MIT Proto or Protelis), and mainly devoted to simulations. Still, a general-purpose framework for aggregate programming, aimed at the construction of real-world applications and possibly embedded in a mainstream language so as to lower the learning curve of the framework itself, would be highly desirable. In fact, such a technology would have a twofold impact: first, it would fill a gap in the pervasive computing technology landscape; secondly, it would be useful in order to promote the use and adoption of the paradigm as well as for boosting research in the field.

The work described in this thesis consists in the development of an aggregate programming framework in Scala, starting from an existing prototype implementing the operational semantics of the field calculus.

The choice of Scala as the host language is motivated by both technical and practical reasons. Scala is a modern language for the JVM which integrates the object-oriented and functional paradigms in a seamless way, has an expressive type system, and provides advanced features for the development of software libraries. Moreover, the possibility to employ a sound, Scala-based actor framework such as Akka is another leading factor for this commitment to Scala, given the critical abstraction gap inherent to the development of a distributed middleware.

In this dissertation, I present a framework achieving the threefold goal:

1. the construction of a Scala library that implements the field calculus semantics in a correct and complete way,
2. the development of an Akka-based distributed platform for aggregate applications, and
3. the creation and exposition of an general, flexible API able to support various distributed computing scenarios,

where it must also be noted that the initial design of the interpreter at the core of the proposed framework is authored by prof. Viroli.

This dissertation is organised into four parts. Part I, *Background: Scala for Library Development*, presents a subset of Scala features and techniques that are relevant to the construction of libraries, APIs, and DSLs. Chapter 1 addresses Scala as a language, focussing on four main portions: the use of traits, the type system, the support for generic programming, and the implicit system. On this foundation, Chapter 2 covers a set of techniques: the "Pimp my library" pattern for extending existing types; approaches to component-based design; dependency injection with the Cake pattern; the design of sets of mutually recursive types with family polymorphism; and basic techniques for internal DSL development.

Part II, *Background: Aggregate Programming*, is intended to provide a high-level description of the scientific foundation of aggregate programming. Chapter 3 provides an outline of the key issues arising from current trends in pervasive computing and introduces spatial computing as a promising response to these challenges. Then, a specific space-time programming approach based on the notion of the computational field is portrayed in Chapter 4, *Field Calculus*. Finally, Chapter 5 introduces the idea of programming aggregates of device as a whole and describes an aggregate programming stack based on field calculus and resilient coordination patterns.

After this background, Part III covers the main work of this thesis, the development of `scafi`, which is both a library for computational fields and an aggregate programming framework for distributed applications. Chapter 6, *Analysis*, illustrates the requirements, the problem, and *what* has to be done. Then, Chapter 7 describes the architecture and the key elements of the design of `scafi`, while a more detailed view of the solution is provided in Chapter 8, *Implementation*.

Finally, the evaluation of the work is carried out in Part IV. Chapter 9 provides an assessment of `scafi` with respect to the requirements, together with some general, retrospective considerations, whereas the conclusive thoughts are relegated to Chapter 10.

Part I

Background: Scala for Library Development

Chapter 1

Advanced Scala Features

This chapter describes a set of Scala features that are particularly relevant to library developers. General references for this chapter and Chapter 2 include [1], [2], [3], [4], [5], [6].

Outline:

1. OOP and traits: some recap, class linearisation, `super` resolution, member overriding, early definitions
2. Advanced types: structural types, compound types, existential types, self types, abstract types
3. Generic programming: type parameters, type bounds, F-bounded polymorphism, type variance
4. Implicits: implicit parameters, implicit conversions/views, implicit resolution

1.1 The Scala programming language

Scala is a general-purpose programming language with the following main characteristics:

- runs on the JVM and **integrates with the Java ecosystem** in a seamless way,

- is a **pure OOPL**, i.e., every value is an object and every operation is a method call,
- provides a **smooth integration of the object-oriented and functional paradigms**,
- is designed to be a **"scalable" language**, by keeping things simple while accomodating growing complexity,
- has a powerful, expressive static **type system** with type inference.

1.2 On object-oriented programming and traits

1.2.1 OOP in Scala: a quick tour of the basics

Let's start with an abstract example that summarises many features in one place:

```scala
package __root.__org
package parentpackage // org.parentpackage

package otherpkg1 { class SomeBaseClass }
package otherpkg2 { trait SomeTrait }

package mypackage { // org.parentpackage.mypackage
  import otherpkg1.{SomeBaseClass => MyBase} // Selective + rename
  import otherpkg2._  // Import all

  class A(val x: Int, private var y: Char, arg: Double)
    extends MyBase with Serializable with SomeTrait {
    private[mypackage] var _z = arg
    def z = _z
    def z_=(newZ: Double) { _z = newZ }

    private[this] var foo: Boolean = _

    @scala.beans.BeanProperty var bar: Int = _

    def this() { // Auxiliary constructor
      this(0, 'a', 0.0)
      this.bar = 2
      this.setBar(1)
    }

    def update(flip: Boolean, foo: Boolean) = this.foo = foo ^ flip
    override val toString = s"A($x,$y,$z,$foo)"

    class Inner { val myFoo = A.this.foo }
    var inner: A#Inner = new Inner
  }

  object A {
    def apply(c: Char) = new A(0, c, 0)

    def sortOfStaticMethod() = { true }
  }

  package object mypackage {
    val obj = new A()          // Use auxiliary constructor
    obj._z                     // Field '_z' is package private
    obj.z = 7.7                // Actually a method call: obj.z=(7.7)
    print(obj.toString)        // toString was overridden as a val!
    obj(true) = false          // Rewritten as: obj.update(true,false)
    val inner = new obj.Inner // Nested classes are object-specific
    val obj2 = A('z')          // Rewritten as: A.apply('z')
    A.sortOfStaticMethod       // Calls method on companion object
    obj2.inner = obj.inner     // Ok thanks to type projection
  }
}
```

Packages

- Packages can contain definitions for classes, objects, and traits.
- A package can also have an associated **package object** with the same name, where you can put function, value, and variable definitions.
- In Scala, packages can be used in a more flexible way than in Java. For example, in Scala you don't need to specify the package at the top of the file, and you could contribute to more than one package in the same file.
- Scala supports flexible forms of import (all, selective, aliased). Moreover, you can use `import` anywhere, even inside methods.

Objects Scala allows the definition of **singleton objects** with the `object` keyword. An object declaration is very similar to a class declaration, except that objects cannot obviously have abstract or undefined members. An object can extend a class and multiple traits, can have fields, methods, inner classes, and a primary constructor. Also, note that `objects` are constructed lazily:

```
var x = 0

object obj { x+=1 }

print(x)  // 0
obj       // Constructor is evaluated the 1st time 'obj' is dereferenced.
print(x)  // 1
```

Classes

- A class can inherit by at most one class (**single-class inheritance** scheme), but it can implement multiple traits.
- Each class has a **primary constructor** which consists of the class body itself; this means that the class body can contain also statements, not only definitions.
- The primary constructor can specify parameters. These parameters can be normal parameters as in function calls, or they can be marked by `val` or `var` (possibly with visibility modifiers) to make them become fields.
- A class can have **auxiliary constructors** which are named `this` and must start with a call to a previously defined auxiliary constructor or the primary constructor .
- Scala supports **fields** with automatically generated getters and setters.
- Each class can have a **companion object** with the same name as the class.
- For what concerns visibility, a class and its companion object can access each other's private entities.
- There are **no static methods** in Scala, but they can be implemented as methods on the class' companion object.
- Class declarations can include **nested classes**, but it is essential to note that an inner class is tied to the object, not to the class. However, you can express a type such as "a `Inner` of any `A`" via *type projection* `A#Inner` (see Section 1.3.1).

Fields and Visibility Fields are introduced with `val` and `var` declarations within the class body or as constructor parameters. The logic of method generation for fields is coherent with the concept of im/mutability; that is, a `val` field is only given a getter, while a `var` field is given a getter and a setter.

Access modifiers are more sophisticated as in Java. In particular, you can restrict visibility to a package, class, or object using the syntax `private[X]` or `protected[X]`. For example, you have the expressibility to state:

- `public`: public access – It is the default access-level (note it is different from Java's package-private default).
- `protected`: inheritance access – It means that any subclass can access the member, which can also be accessed from other objects of any subclass.
- `private`: class-private access – Tt means that the member can only be accessed from the same class as well as from other objects of the same class.
- `protected[package]`: package-private and inheritance access – It means the member is accessible everywhere in the package and from any subclass (possibly located in a different package).
- `private[package]`: package-private (without inheritance access).
- `protected[this]`, `private[this]`: object-protected/private access.

1.2.2 Traits

Traits are similar to abstract classes, in that they can include both abstract and concrete methods/fields. As a difference, traits can only have a parameterless primary constructor. Another difference is that, while traits can be used everywhere abstract classes can be used, only traits can be used as mixins.

Traits as interfaces By defining abstract fields and methods, traits work as **interfaces**. That is, all concrete classes (or objects) implementing the trait are required implement the trait's abstract entities.

```
trait Logger {
  def log(msg: String): Unit
}
```

```scala
class ConsoleLogger extends Logger {
  def log(msg: String) = println(msg)
}
```

Traits as mixins When a trait defines concrete fields and methods, it works as a **mixin**, meaning that its functionality can be mixed into other classes, object, or traits:

```scala
trait Comparable[T] {
  def compareTo(other: T): Int

  def >(other: T) = compareTo(other) > 0
  def <(other: T) = compareTo(other) < 0
  def ===(other: T) = compareTo(other) == 0
}

class Box[T <: Comparable[T]](val value: T) extends Comparable[Box[T]] {
  def compareTo(b2: Box[T]): Int = value.compareTo(b2.value)
}

class NumWrapper(val x: Int) extends Comparable[NumWrapper] {
  def compareTo(other: NumWrapper): Int =
    if(x==other.x) 0 else if(x > other.x) 1 else -1
}

val box1 = new Box(new NumWrapper(1))
val box2 = new Box(new NumWrapper(5))
box1 === box1  // true
box1 === box2  // false
box1 < box2    // true
box1 > box2    // false
```

This example makes use of some generic programming features that will be described later. For now, what should be noted is that the classes `NumWrapper` and `Box` acquire the concrete methods `<,>,===` of trait `Comparable`. The above example also shows that traits can effectively work as interfaces and mixins at the same time.

The interesting thing of using traits as mixins is that they *compose* (i.e., you can provide multiple mixins at the same time) and they are *stackable* (see the `Logger` example in Section 1.2.4). For the "compose" part:

```
trait X
trait Y
trait Z

class C extends X with Y with Z
```

1.2.3 Class construction and linearisation

In Scala, a class may inherit from a base class and mix-in multiple traits at the same time. A comprehension of the construction process is important to understand some interesting aspects of traits.

Class construction works by recursively calling constructors in the following order:

1. Superclass' constructor
2. Traits' constructors, from left to right, with parents constructed first (and not constructed a second time)
3. Class constructor

The process is exemplified in the next listing:

```
class A { print("A") }
trait R { print("R") }
trait S extends R { print("S") }
trait T { print("T") }
trait U extends T with R { print("U") }
class B extends A with U with S { print("B") }

new B  // A T R U S B
```

The working is intuitive: elements specified on the left of the “extension list” come first and thus are constructed first, and each element needs to construct its parents before itself.

The relevant fact is that the construction order is strictly related to the *linearisation of a class*, that is, the process that determines the linear hierarchy of parents of a class. For the example above, class `B` has the following linearisation:

$$\begin{aligned} lin(B) &= B \succ lin(S) \succ lin(U) \succ lin(A) \\ &= B \succ (S \succ \cancel{R}) \succ (U \succ R \succ T) \succ A \\ &= B \succ S \succ U \succ R \succ T \succ A \end{aligned}$$

Two things should be noted: i) the first occurrence of R is elided because the second occurrence wins (R must be constructed at a higher point in the hierarchy), and ii) the construction order is the reverse of the linearisation order. This is also intuitive: when defining a class, the elements of the extension list that are specified next (on the right) are down the hierarchy, thus they need to be constructed after the elements located at higher levels in the hierarchy and, as they come later, they can also override previous definitions.

1.2.4 Traits: member overriding and `super` resolution

With a basic understanding of class linearisation, it is worth to consider two interesting points about traits:

1. If multiple traits override the same member, the trait that wins is the one constructed last, i.e., the trait that is "closer" to the defining class/object in the class linearisation.
2. In a trait, the method calls on `super` are resolved depending on the order in which traits are added. That is, the method implementation to be dispatched is the one of the first subsequent trait (implementing the method) in the class linearisation order.

Let's exemplify these two aspects:

```
trait Logger {
  def log(msg: String)
}

trait ShortLogger extends Logger {
  val maxLength: Int
  val ellipsis = "..."

  abstract override def log(msg: String) {
    super.log(msg.take(maxLength)+ellipsis)
  }
}

trait WrapLogger extends Logger {
  val wrapStr = "|"
  abstract override def log(msg: String) {
    super.log(wrapStr + msg + wrapStr)
  }
}

trait ConsoleLogger extends Logger {
  override def log(msg: String) = println(msg)
}

val obj = new {
  val maxLength = 6
  override val ellipsis = ",,,"
} with ConsoleLogger with WrapLogger with ShortLogger

obj.log("linearisation") // |linear,,,|
```

Some points should be underlined:

- First, note how the different behaviors are composed at instantiation time.
- In `ShortLogger` and `WrapLogger`, the `log` method is decorated with `abstract override`, because you are – at the same time – overriding the method and calling *some* `super` implementation.
- Note how the mixin order is relevant to the final result: if you switch `WrapLogger` and `ShortLogger`, you'll have the last occurrence of the wrapping string stripped.
- Note how the chain of `log` calls flow from right to left in the trait list (or, equivalently, from bottom to top in the hierarchy as given by the linearisation).
- The `new {...}` part is an *early-definition* which is needed to concretise the `maxLength` abstract field before the `ShortLogger` can be constructed.

1.2.5 Trait instantiation, refinement, early definitions

Scala provides a mechanism to instantiate traits and abstract classes once they have no abstract members.

```
trait A
trait B

new A with B {
  println("This is a refinement")
  def foo: String = "bar"
}
```

The block following `A` is a *refinement* (empty, in this case). A refinement is a mechanism to provide a delta (i.e., overrides or additional members) to a type. This actually defines a structural type (see Section 1.3.2).

```
trait A {
  val x: Int
  require(x > 0)
}

val a = new A {
  val x = 10
} // java.lang.IllegalArgumentException: requirement failed
```

The problem here is that `A` is constructed *before* being refined. To solve this issue, we have to provide an early definition:

```
trait A {
  val x: Int
  require(x > 0)
}

val a = new { val x = 10 } with A
```

This works because, according to the rules for method overriding[4], an abstract member cannot override a concrete member.

1.3 The Scala type system

1.3.1 Some preparatory definitions

- A (static) **type** is a set of information hold by the compiler about program entities.

- A **binding** is a name used to refer to an entity.
- Types can be located at certain **paths**, where a path, according to the Scala Language Specification [1], is one of the following:
    1. The empty path ϵ.
    2. `C.this` – where C refers to a class.
    3. `C.super.x` – where x is a stable member of the superclass of the class referenced by C.
    4. `p.x` – where p is a path and x is a stable member of p; a stable member is a package or a member introduced by (non-volatile) object or value definitions.
- A **singleton type** has form `p.type` where path `p` points to a value conforming to `AnyRef`.
- A **type projection** `T#x` refers to the type member `x` of type `T`.
- A **type designator** refers to a named value type. A type designator can be:
    - *Qualified*: has form `p.t` where `p` is a path and `t` is a named type; it is equivalent to `p.type#t`.
    - *Unqualified and bound to a package or class or object `C`*: `t` is a shorthand for `C.this.type#t`.
    - *Unqualified and NOT bound to a package/class/object*: `t` is a shorthand for ϵ`.type#t`.

1.3.2 Advanced types

Parameterised types A *parameterised type* consists of a type designator `T` and n *type parameters* U_i:

```
T[U1,U2,...,Un]
```

Infix types Any type which accepts two type parameters can be used as an *infix type*. Consider the following example with the standard map `Map[K,V]`:

```
val m: String Map Int = Map("a" -> 1, "b" -> 2)  // m: Map[String,Int]
```

Structural types A *structural type* can be defined through refinement. In the refinement, you can add declarations or type definitions. You can also use refinement alone, without any explicit refined type; in that case, `{...}` is equivalent to `AnyRef{...}`.

```
def f(x: { def foo: Int }) = x.foo  // f: (x: AnyRef{def foo: Int})Int

f(new { def foo = 7 })  // Int = 7
```

Structural types enable a form of *duck typing*, where you can specify requirements on objects in terms of an exhaustive specification of methods and fields to be supported. However, this feature is implemented via reflection, so it comes with a cost.

Compound types A compound type is one of the form

```
T1 with T2 ... with Tn { R }
```

where the `T`s are types and `R` is an optional refinement. A compound type `C` without any refinement is equivalent to `C {}`. Consider the following:

```
trait A; trait B
new A // Error: trait A is abstract; cannot be instantiated
new A { }    // Ok
new A with B // Ok
```

Compound types are also known as *intersection types* because a value, in order to belong to the compound type, must belong to all the individual types.

Existential types An existential type has the form

```
T forSome { Q }
```

where Q is a sequence of type declarations (which may also be constrained, see Section 1.4.2).

The underscore _ has many uses in Scala. One of them consists in providing a syntactic sugar for existential types:

```
val m1: Map[_,_<:List[_]] =
  Map(1 -> List('a','b'), "k" -> List(true,true))

val m2: Map[A,B] forSome {
   type A;
   type B <: List[C] forSome { type C }
} = m1
```

Note that such a use of existential types is related to the notion of *variance* (see Section 1.4.3). The connection between generic type variance and existential types has been pointed out in [7].

Self-types *Self types* are used to constrain a trait or class to be used necessarily within a compound type that includes a type conforming to the self type. In other words, a trait or class can be used only when mixed in (together) with the self type.

```
trait A {
  def foo { }
}

trait B { self: A =>
  def bar = foo
}

new B { } // Error: illegal inheritance; self-type B
          //   does not conform to B's selftype B with A
new B with A // OK
new A with B // OK
```

One can read the self-type as a **dependency** ("B depends on A"), although it is more precise to say that "a concrete B will also be an A".

Abstract type members Just like it is possible to declare abstract fields and methods, a class or trait can declare *abstract types*.

```scala
trait Box {
  type TValue
  def peek: TValue
}

class StringBox(s: String) extends Box {
  override type TValue = String
  val peek = s
}

val sbox = new StringBox("xxx")
sbox.peek // String = xxx
```

Note that the previous example can be rewritten using type parameters in place of abstract type members.

1.4 Generic programming in Scala

Generic programming is all about defining generic entities and algorithms. In common sense, the term *generic* means "belonging to a large group of objects" (source: etymonline.com). Thus, we may say that an entity or algorithm is generic when it or its properties can be found in many other entities or algorithms; the other way round works as well, i.e., an entity or algorithm is generic if many other entities or algorithms "can be generated" from it.

Genericity involves some form of abstraction as a fully-specified entity is not generic by definition. The way to achieve genericity is by delaying the specification of certain details until a later time. When it comes to programming languages, three main forms of abstraction are given by:

1. types,
2. parameters,
3. abstract members.

A type abstracts from the specific values belonging to that type. A parameter allows to abstract from specific parts of an entity or algorithm. An abstract member formalises the promise that the member will be concretised in future.

Scala's abstract type members (see Section 1.3.2) and type parameters combine the abstraction provided by types with the abstraction provided by parameters and abstract members, respectively.

Next, I cover some basic and advanced aspects of generic programming in Scala. I'll be quick as many of these features are mainstream.

1.4.1 Type parameters

Classes, traits, methods, and type constructors can accept type parameters.

```
// Generic method
def headOption[T](lst: List[T]): Option[T] =
  lst match { case h :: _ => Some(h); case _ => None }
headOption(List[Int]())     // None
headOption((3 to 7).toList) // Some(3)

// Generic trait
trait TBox[+T] {
  def value: T
}

// Generic class
class Box[+T](val value: T) extends TBox[T]

// Type constructor
type Cellar[T] = Map[String, TBox[T]]

val secretbox = new Box("xxx")
val cellar: Cellar[Any] = Map("secretbox" -> secretbox)
cellar("secretbox").value // Any = "xxx"
```

Note that, thanks to type inference, often you will not need to specify the type parameter.

1.4.2 Type bounds (bounded quantification)

Scala allows you to specify constraints to type variables:

- *Upper bound*: `T<:UB`: `T` must be a subtype of `UB`.
- *Lower bound*: `T>:LB`: `T` must be a supertype of `LB`.

Other bounds exist but we'll see them when talking about implicits.

Now let's see an example of upper and lower bounds:

```
trait A
trait B extends A
trait C extends B

class Pair[T1 >: B, T2 <: B](val _1: T1, val _2: T2)

new Pair(new A{}, new C{}) // Pair[A,C]
new Pair(new B{}, new B{}) // Pair[B,B]
new Pair(new C{}, new C{}) // Pair[B,C] !!!!
new Pair[C,B](new C{}, new C{}) // Error: do not conform with constraint
```

Note that, in Scala, all types have a maximum upper-bound (`Any`) and a minimum lower-bound (`Nothing`).

Self-recursive types and F-bounded polymorphism A self-recursive type is a type that is defined in terms of itself, for example:

```
case class Point(x: Double, y: Double) {
  // Positive recursion
  def move(x: Double, y: Double): Point

  // Negative recursion
  def isEqual(pt: Point): Boolean
}
```

Type parameters can be defined in a self-recursive way as well. In fact, Scala supports *F-bounded quantification*[8] (also known as *F-bounded polymorphism*), which means that a type parameter can be used in its own type constraint, such as in `T<:U[T]`.

1.4.3 Type variance

Type variance is a feature that integrates parametric and subtype polymorphism in OOPLs [9][7].

Given a type T with type components U_i (i.e., type parameters or type members), *variance* refers to the relation between the subtyping of T and the subtyping of its type components U_i. For example: is `List[Rect]` a subtype of `List[Shape]`? Or viceversa? Or are they unrelated?

Here are the possibilities:

- `T[A]`: T is invariant in A; i.e., T does not vary with respect to A.
  So, for example: given `List[T]`, if `A` and `B` are different types (possibly in a subtyping relationship), then `List[A]` and `List[B]` are unrelated.
- `T[+A]`: T is covariant in A; i.e., T varies in the same direction as the subtyping relationship on A.
  So, for example: given `List[+T]`, if `Rect` is a subtype of `Shape`, then `List[Rect]` is a subtype of `List[Shape]`.
- `T[-A]`: T is contravariant in A; i.e., T varies in the opposite direction to the subtyping relationship on A.
  So, for example: given `List[-T]`, if `Rect` is a subtype of `Shape`, then `List[Rect]` is a supertype of `List[Shape]`.

Now, let's consider a 1-ary function type `Function1[-T,+R]`: it is covariant in its return type and contravariant in its input type. Functions should conform to such a variance scheme to support safe substitutability. In fact, we ask: when is it safe to substitute a function `f:A=>B` with a function `g:C=>D`?

- `val a: A = ...; f(a)`
  The parameters provided by the users of `f` must be accepted by `g` as well, thus `C>:A` (contravariance as $C >: A \Rightarrow g <: f$).
- `val b: B = f(..)`
  The value returned to the users of `f` must support at least the interface of `B`, thus `D<:B` (covariance as $D <: B \Rightarrow g <: f$).

So, it is common to refer to function parameters as *contravariant positions* and to return types as *covariant positions*. The use of variance annotations allows the compiler to check that generic types are used in a manner which is consistent to these rules.

```scala
trait A[+T] { def f(t: T) }
// Error: covariant type T occurs in contravariant position
```

However, note that Scala, in method overriding, for some reason only allows covariant specialisation of the return types, while contravariant generalisation of method parameters is not allowed:

```scala
trait A { def a(a: A): A }

// Covariant specialisation of method return type: OK
class B extends A { def a(a: A): B = ??? }

// Contravariant generalisation of method parameters: ERROR
class C extends A { def a(a: Any): A = ??? }
```

Last but not least, it is important to remark that mutability makes covariance unsound. Let's assume that `scala.collection.mutable.ListBuffer` were covariant; in this case, the following listing shows a potential issue:

```scala
import scala.collection.mutable.ListBuffer
val lstrings = ListBuffer("a","b")  // Type: ListBuffer[String]
val lst: ListBuffer[Any] = lstring  // Would fail, but suppose it's OK
lst += 1      // Legal to add an Int to a ListBuffer[Any]
              // But 'lst' actually points to a list of strings!!!!!
```

1.4.4 Abstract types vs. type parameters

In many cases, code that uses type parameters can be rewritten with abstract types, and viceversa. This is another situation where object-oriented programming and functional programming merge nicely in Scala.

The encoding from type parameters to type members, in the case of a parameterised class C, is described in [4]. Here are a few points on the similarity and difference between abstract types and type parameters:

- Usually, type parameters are used when the concrete types are to be provided when the class is instantiated, and abstract types are used when the types are to be supplied in a subclass.
- The use of type parameters is not very scalable: if you have a lot of type parameters, code tends to clutter, while abstract types help to keep the code clean.
- The previous point is particularly true when type parameters are subject to (possibly verbose) type constraints.
- At the time of type instantiation, in the case of type parameters you do

not see the name of the instantiating parameter, thus you may lose some readability.

1.5 Implicits

The Scala *implicit system* is a static feature that allows programmers to write concise programs by leaving the compiler with the duty of inferring some missing information in code. This is achieved through the arrangement of code providing that missing data (according to a set of scoping rules) and a well-defined lookup mechanism.

An implicit lookup is triggered in two cases, when the compiler spots:

1. a *missing parameter list* in a method call or constructor (if that parameter list is declared as `implicit`), or
2. a *missing conversion* from a type to another which is *necessary* for the program to type-check – this happens automatically in three situations (unless an implicit conversion has already been performed):
   (a) when the type of an expression differs from the expected type,
   (b) in `obj.m` if member `m` does not exist,
   (c) when a function is invoked with parameters of the wrong type.

Using the `implicit` keyword, you can provide implicit data and implicit conversions. A note on terminology: an implicit conversion function `T=>U` from type `T` to type `U` is often called an *implicit view* (because it allows to *view* a `T` as a `U`).

The following listing exemplifies the different situations where the implicit mechanism triggers:

```scala
// A) MISSING PARAMETER LIST

def m(implicit s: String, i: Int) = s+i

m("a", 0)     // "a0"   (You can still explicitly provide them)
m             // Error: could not find implicit value for 's'
implicit val myString = "x"
m             // Error: could not find implicit value for 'i'
implicit val myInt = 7
```

```
m            // "x7"

// B) MISSING CONVERSION

// B1) Expression of unexpected type

case class IntWrapper(x: Int) {
   def ^^(p: Int): Int = math.pow(x,p).toInt
}
implicit def fromIntToIntWrapper(x: Int) = IntWrapper(x)
val iw: IntWrapper = 8 // iw: IntWrapper = IntWrapper(8)

// B2) Non-existing member access

2^^5       // 32 (there is no ^^ method in Int)

// B3) Function call with wrong param types

def pow(iw: IntWrapper, power: Int) = iw^^power

pow(3, 4)  // 81 (pow accepts an IntWrapper, not an Int)
```

1.5.1 Implicit scope

Scala defines well-defined rules for what concerns implicit lookup. First of all, ambiguity in implicit resolution results in a compilation error.

```
implicit def x = 'x'
implicit val y = 'y'

def f(implicit c: Char) { }

f // Error: ambiguous implicit values:
  //   both method x of type => Char
  //   and value y of type => Char
  //   match expected type Char
```

Secondly, when the compiler looks for `implicit` data or views from a certain lookup site, then:

1. It first looks if there is a conforming implicit entity among the unqualified bindings.
   E.g., if there is an object `o` in scope and that object has an implicit field member `o.i`, that value is *not* considered because the implicit entity must

be accessible as a single identifier.

2. Then, it looks:
   i) in case of implicit parameter lookup, at the implicit scope of the implicit parameter type, and
   ii) in case of an implicit conversion, at the implicit scope of the *target* type of the conversion.

The *implicit scope* of a type is the set of companion objects of its associated types. For a type, its associated types include the base classes of its parts. The parts of a type `T` are (according to the SLS[1]):

- if it is a parameterised type, its type parameters;
- if it is a compound type, its component types;
- if it is a singleton type p`.type`, the parts of p (e.g., the enclosing singleton object);
- if it is a type projection, i.e., `T = A#B`, the parts of `A` (e.g., the enclosing class or trait);
- otherwise, just `T`.

Let's verify these rules:

```scala
// A) Companion objects of the type parameters
trait A; object A { implicit def itoa(i: Int) = Map[A,A]() }

def f[X,Y](m: Map[X,Y]) = m

f[A,A](1)  // OK. Converts the Int value to a Map[A,A]

// B) Companion objects of types in an intersection type

trait B
trait C; object C {
  implicit def conv(i: Int) = new B with C { def foo { } }
}

def g(arg: C with B) = arg

g(1)  // OK. Converts the Int value to a B with C { def foo:Unit }

// C) Parts of the object of the singleton type

abstract class Provider[T](_x: T) { implicit val x = _x }
```

```
object P extends Provider(this)

def f(implicit p: P.type) = { p }
f(P) // OK.

// D) Parts of the type projecting from

object x {
  case class Y(i: Int)

  implicit val defaultY = Y(0)
}

implicitly[x.Y]      // x.Y = Y(0)
implicitly[x.type#Y] // x.Y = Y(0)
// In this case, the type projecting from is a singleton type

// Another example for D), with package objects

package a.b.c { class C }
package a.b {
  package object c {
    implicit val defaultC: C = new C
  }
}

implicitly[a.b.c.C] // OK
```

Where `implicitly[T](implicit e:T) = e` is defined in `scala.Predef`.

In general, it is best not to abuse the flexibility of the implicit scoping. As a rule of thumb, implicits should be put on the package object or in a singleton object with name `XxxImplicits`.

1.5.2 Implicit classes

Implicit classes[1] are classes declared with the `implicit` keyword. They must have a primary constructor that takes exactly one parameter. When an implicit class is in scope, its primary constructor is available for implicit conversions.

```
implicit class Y { } // ERROR: needs one primary constructor param

implicit class X(val n: Int) {
  def times(f: Int => Unit) = (1 to n).foreach(f(_))
```

[1]Reference: http://docs.scala-lang.org/overviews/core/implicit-classes.html

```
}

5 times { print(_) } // 12345
```

It is interesting to note that an implicit class can be generic in its primary constructor parameter:

```
implicit class Showable[T](v: T) { val show = v.toString }

Set(4,7) show  // String = Set(4, 7)
false show     // String = false
```

1.5.3 More on implicits

Context bound A type parameter `T` can have a *context bound* `T:M`, which requires the availability (at lookup site, not at the definition site) of an implicit value of type `M[T]`.

Let's consider an example. Scala provides a trait `scala.math.Ordering[T]` which has an abstract method `compare(x:T, y:T):Int` and provides a set of methods built on that method, such as `min`, `max`, `gt`, `lt` and so on. Moreover, implicit values for the most common data types are defined in the `Ordering` companion object. Then, we may want to define a function that requires to work on types for which some notion of ordering is defined. For example, let's define a function the returns the shortest and longest string in a collection:

```
implicitly[Ordering[String]] // Predefined ordering:
//  Ordering[String] = scala.math.Ordering$String$@65c5fae6
// Let's override it with a custom ordering for strings

implicit val strOrdering = new Ordering[String]{
  def compare(s1: String, s2: String) = {
    val size1 = s1.length;
    val size2 = s2.length;
    if(size1<size2) -1 else if(size1 > size2) 1 else 0
  }
}

def minAndMax[T : Ordering](lst: Iterable[T]) = (lst.min, lst.max)

minAndMax(List("hello","x","aaa")) // (String, String) = (x, hello)
minAndMax(List("hello","x","aaa"))(Ordering.String)
```

```
// (String, String) = (aaa,x)
```

Note that the `minAndMax` method can still be called with an explicit `Ordering[T]` instance. This reveals that context bounds are actually a syntactic sugar; in fact, `minAndMax` is rewritten as follows:

```
def minAndMax[T](lst: Iterable[T])
                (implicit ord: Ordering[T]) = (lst.min, lst.max)
```

Generalised type constraints *Generalised type constraints* are objects that provide evidence that a constraint hold for two types. As it is stated in the Scala API documentation [2]:

- `sealed abstract class =:=[-From,+To] extends (From)=>To`

  An instance of `A =:= B` witnesses that type `A` is equal to type `B`.
- `sealed abstract class <:<[-From,+To] extends (From)=>To`

  An instance of `A <:< B` witnesses that `A` is a subtype of `B`.

Note that the type constructor `=:=[A,B]` can be used with the infix notation `A=:=B`.

In practice, these constraints are used through an *implicit evidence parameter*. This allows, for example, to enable a method in a class only under certain circumstances:

```
case class Pair[T](val fst:T, val snd:T){
  def smaller(implicit ev: T <:< Ordered[T]) = if(fst < snd) fst else snd
}

class A

case class B(n: Int) extends Ordered[B] {
  def compare(b2: B) = this.n - b2.n
}

val pa = Pair(new A, new A)
pa.smaller // Error: Cannot prove that A <:< Ordered[A].

val pb = Pair(B(3), B(6))
pb.smaller // B = B(3)
```

In this case, the instance method `pair.smaller` can be invoked only for pairs of a type `A` that is a subtype of `Ordered[A]`. The implicit parameter is said to be an *evidence* parameter in the sense that the resolution of the implicit value represents a proof that the constraint is satisfied. These are also known as *reified type constraints* because the objects that are implicitly looked up represent reifications of the constraints.

`TypeTags` and reified types The Java compiler uses *type erasure* in the implementation of generics. This means that, at runtime, there is no information about the type parameters of generic classes. Scala also implements type erasure to ease integration with Java.

```
import scala.reflect.runtime.universe._

def f[T](lst: List[T]) = lst match {
  case _:List[Int] => "list of ints";
  case _:List[String] => "list of strs";
}
// warning: non-variable type argument Int in type pattern List[Int]
// (the underlying of List[Int]) is unchecked since it is eliminated
// by erasure:       case _:List[Int] => "list of ints";
//                           ^

f(List("a","b")) // "list of ints"
```

Note that the ability to work with generic types at runtime has been a subject of research for some time, also due to performance implications, as explained in [10].

To solve this issue, Scala provides `TypeTags` (which replaced `Manifests` in Scala 2.10), which are used together with the implicit mechanism to provide at runtime the type information that would otherwise be available only at compile-time.

```
import scala.reflect.runtime.universe.{TypeTag, typeOf}

def f[T : TypeTag](lst: List[T]) = lst match {
  case _ if typeOf[T] <:< typeOf[Int] => "list of ints"
  case _ if typeOf[T] <:< typeOf[String] => "list of strings"
}

f(List(1,2,3))   // "list of ints"
f(List("a","b")) // "list of strings"
```

Note the use of context bound. When an implicit value of type `TypeTag[T]` is required, the compiler provides it automatically.

Chapter 2

Advanced Scala Techniques

Chapter 1 provides an overview of some intermediate-level features of the Scala programming language. This chapter builds on such features and presents a handful of techniques that may be useful for library development. Many of these techniques have been extensively used for implementing `scafi`.

Outline:

- "Pimp my library" pattern
- Type classes
- Components and dependency injection
- Cake pattern
- Family polymorphism
- Development of internal domain-specific languages

2.1 "Pimp my library" pattern

The *Pimp my library* pattern is a common technique, based on implicits, aimed at extending existing types with additional methods and fields.

Let's consider an example in the Scala standard library. By default, you have access to a facility for generating a `Range` from an `Int`:

```
val range = 5 to 10 // Range.Inclusive = Range(5,6,7,8,9,10)
```

This works because object `Predef` (which is implicitly imported in all Scala compilation units) inherits from trait `LowPriorityImplicits`, which defines many implicit conversion methods and, in particular, a conversion method from `Int` to `RichInt`. Then, `RichInt` defines a few utility methods, including `to(end:Int):Inclusive` to produce an inclusive range.

The pattern is clear; when we need to extend some existing type, we can:

1. define a type with the "extension methods" that express the new intended behavior;
2. define an implicit conversion function from the original type to the newly defined type, together with some policy for the import of these implicit views.

This approach is particularly useful when the original type cannot be instantiated (e.g., because the class is `final` or `sealed`).

In summary, this simple idiom allows you to adapt (cf. Adapter design pattern), decorate (cf. Decorator design pattern), or extend existing classes in a transparent way (thanks to implicit views which are applied by the compiler at compile-time).

As an example, let's extend ints with a `times` method which repeats an action for the provided number of times.

```scala
implicit class MyRichInt(private val n: Int) extends AnyVal {
  def times(f: =>Unit) = new Range.Inclusive(1,n,1).foreach{_=>f}
}

5 times { print('a') } // aaaaa
```

The implicit class (see Section 1.5.2) `MyRichInt` has been defined as an implicit view from `Ints` to instances of itself. As a side note, it is also a *value class* (as it extends `AnyVal`), so it wins some efficiency by avoiding object allocation at runtime.

2.2 Type classes

Type classes are a feature popularised in the Haskell programming language. A *type class* provides an abstract interface for which it is possible to define many type-specific implementations.

In Scala, the type class idiom consists in[3]:

1. A trait that defines the abstract interface.
2. A companion object for the type class trait that provides a set of default implementations.
3. Methods using the typeclass, declared with a context bound.

As an example, let's try to implement the `Ordering[T]` type class (similar to the one in the Scala standard library):

```scala
// Type class
trait Ordering[T] {
  // Abstract interface
  def compare(t1: T, t2: T): Int

  // Concrete members
  def ===(t1: T, t2: T) = compare(t1,t2) == 0
  def <(t1: T, t2: T) = compare(t1,t2) < 0
  def >(t1: T, t2: T) = compare(t1,t2) > 0
  def <=(t1: T, t2: T) = <(t1,t2) || ===(t1,t2)
  def >=(t1: T, t2: T) = >(t1,t2) || ===(t1,t2)

  def max(t1: T, t2: T): T = if(>=(t1,t2)) t1 else t2
  def min(t1: T, t2: T): T = if(<=(t1,t2)) t1 else t2
}

// Companion object with implicit, default implementations
object Ordering {
  implicit val intOrdering = new Ordering[Int]{
    def compare(i1: Int, i2: Int) = i1-i2
  }
}

// Usage
def min[T : Ordering](lst: List[T]): T = {
  if(lst.isEmpty) throw new Exception("List is empty")

  val ord = implicitly[Ordering[T]]
  var minVal = lst.head
  for(e <- lst.tail) {
    if(ord.<(e,minVal)) minVal = e
  }
  minVal
}

min(List(5,1,4,7,-3))  // Int = -3
```

Note the use of the context bound constraint on the type parameter for `min` and how we need to perform an implicit lookup to to get the `Ordering[T]` instance on which we can invoke the methods of the typeclass interface.

It is clear that we could have achieved a similar result by using inheritance. However, type classes have some benefits:

- You can provide many implementations of the type class interface for the same type.
- Type classes separate the implementation of an abstract interface from the definition of a class. Thanks to this separation of concerns, you can easily adapt existing types to the type class interface.
- By playing on the scoping rules for implicits, you can override the default type class implementation.
- A type may implement multiple type classes without cluttering its class definition. Moreover, you can specify multiple type bounds on a type variable `T:CB1:CB2:...`, requiring type `T` to provide an implicit implementation object for multiple type classes.

2.3 Component modelling and implementation

In computer science, the notion of *component* and *component-oriented software development* have been interpreted in many different ways, i.e., according to many different (and possibly not formalised) component models. A *component model* defines:

1. *components* – telling things such as what is a component, how to specify a component, how to implement a component, what a component's runtime lifecycle consists in, and

2. *connectors* – i.e., mechanisms for component composition and component interaction.

Generally, for our purpose, a component can be abstractly defined as a *reusable* software entity with well-defined boundaries, as defined by a software *interface*. An interface is a means of specifying both *provided services* and *required services* (i.e., *dependencies*).

The paper [4] supports the idea that for building reusable components in a scalable way, three features or abstractions are key:

1. abstract type members (see Section 1.3.2),
2. explicit self-types (see Section 1.3.2),
3. modular mixin composition.

Thus, according to such a proposed service-oriented component model, the following mappings emerge:

- Concrete classes $\Longleftrightarrow$ components.
- Abstract classes or traits $\Longleftrightarrow$ component interfaces.
- Abstract members $\Longleftrightarrow$ required services.
- Concrete members $\Longleftrightarrow$ provided services.

In this context, we can interpret abstract member overriding as a mechanism for providing required services. As concrete members always override abstract member, we get *recursively pluggable components where component services do not have to be wired explicitly* [4]. In this sense, mixin composition turns out to be a flexible approach for the assembly of component-based systems.

And where do self-types fit into this frame? Well, self-types are a more concise alternative to abstract members (read "required services"), with some differences. Self-types can be seen as an effective way to specify *component dependencies* as they ensure that, when a component is being instantiated, it must be connected with the specified dependency. Note that while the self-type is one, thanks to compound types (see Section 1.3.2) you can provide for multiple dependencies.

Let's visualise these concepts with an example:

```scala
// *******************
// *** DATA MODELS ***
// *******************
class Account(val id: String, var total: Double = 0)

trait AbstractItem {
  def name: String
  def price: Double
}
case class Item(name: String, price: Double) extends AbstractItem

// ****************************
// *** COMPONENT: Inventory ***
// ****************************
trait Inventory {
  type TItem <: AbstractItem

  def availability(item: TItem): Int
  ...
}

trait InventoryImpl extends Inventory {
  private var _items = Map[TItem,Int]()
  ...
}

// ***********************
// *** COMPONENT: Cart ***
// ***********************
// Depends on the Inventory component
trait Cart { self: Inventory =>

  def changeCartItem(item: TItem, num: Int): Unit
  ...
}
trait CartImpl extends Cart { inv: Inventory =>
  private var _items = Map[TItem,Int]()
  ...
}

// ******************************
// *** COMPONENT: Application ***
// ******************************
// Depends on the Cart and Inventory components
trait ShoppingSystem { this: Cart with Inventory => }

// ********************************************
// *** Application with component instances ***
// ********************************************
object Shopping extends ShoppingSystem with CartImpl with InventoryImpl {
  type TItem = Item
}
```

This is a toy example, but it points out an approach to service-oriented component development. Note that the implementations of the components are `traits`; if they were classes, they could not be mixed in. Also, note how the concrete type of `TItem` is specified at the last moment when composing the application, `Shopping`. Then, it is interesting to see how the application logic component (`ShoppingSystem`) is defined as dependent on the other components (via compound self-type). As a result, the application façade object can be used as a cart or as an inventory in a inheritance-like (*is-a*) fashion. It is quite weird and not very effective for what concerns conceptual integrity: it would be better to have the application object *delegate* these functionalities to its components, rather than assimilating the application to a monolithic component object. We should apply the GOF's principle *favor object composition over class inheritance*[11]; by doing so, we come up with the Cake pattern.

2.4 Cake Pattern

The previous section pointed out that a better way to satisfy component dependencies is via composition (rather than inheritance). Let's adjust that example according to the *Cake pattern*. In this pattern [5]:

- You define components as traits, specifying dependencies via self-types.
- Then, each component includes:
  - a trait that defines the service interface,
  - an `abstract val` that will contain an instance of the service,
  - optionally, implementations of the service interface.

Thus:

```
// ****************************
// *** COMPONENT: Inventory ***
// ****************************
trait InventoryComponent {
  val inventory: Inventory
  type TItem <: AbstractItem

  trait Inventory { ... }

  trait InventoryImpl extends Inventory { ... }
}

// **********************
// *** COMPONENT: Cart ***
// **********************
trait CartComponent { invComp: InventoryComponent =>
  val cart: Cart

  trait Cart { ... }

  trait CartImpl extends Cart {
    val inv = invComp.inventory // NOTE: ACCESS TO ANOTHER COMPONENT'S IMPL
    private var _items = Map[TItem,Int]()

    def changeCartItem(it: TItem, n: Int) = {
       inv.changeItems(Map(it -> (-n)))
       _items += (it -> n)
    }
    ...
  }
}

// *****************************
// *** COMPONENT: Application ***
// *****************************
trait ShoppingComponent { this: CartComponent with InventoryComponent => }

// ********************************************
// *** Application with component instances ***
// ********************************************
object Shopping extends ShoppingComponent with
  InventoryComponent with CartComponent {
  type TItem = Item
  val inventory = new InventoryImpl { }
  val cart = new CartImpl { }
}

val inv = Shopping.inventory
val cart = Shopping.cart
```

Notes:

- The `Shopping` object centralises the **wiring** of components by implementing the components' abstract `val`s (`inventory` and `cart`).

- Note how the component implementations "receive" their dependencies (i.e., the instances of other components' implementations) through the (abstract) `val` fields.
- As the component implementation instances do not need dependency injection via constructor parameters, we can instantiate them directly (*without manual wiring*). In other words, we just have to choose an implementation and we do not need to do any wiring as these components have already been wired.
- The component instance `val`s can be declared `lazy` to deal with initialisation issues.

As design patterns should not be confused with their implementations, we note that the previous example just shows one particular realisation of some more general pattern which is a design response to the following issues (or *forces*):

- how to flexibly build systems out of modular components,
- how to declaratively specify the dependencies among components, and
- how to wire components together to satisfy the dependencies.

Thus, the Cake pattern can be described more precisely as a Scala-specific pattern for dependency injection and component composition.

We could play with Scala features to morph this pattern into multiple variations and possibly communicate better our intents:

```scala
trait ComponentA { val a: A; class A }

trait ComponentB { val b: B; class B }

object ComponentC { type Dependencies = ComponentB with ComponentC }
trait ComponentC { self: ComponentC.Dependencies =>
  val c: C

  class C { /* uses 'a' and 'b' internally */ }
}

trait ABCWiring1 extends ComponentA with ComponentB with ComponentC {
  lazy val a = new A; lazy val b = new B; lazy val c = new C
}

trait ApplicationWiring extends ABCWiring1
trait ApplicationComponents extends ComponentA with ComponentB with ComponentC

object Application extends ApplicationComponents with ApplicationWiring {
  println(s"$a $b $c")
}
```

The name of the ”Cake pattern” brings to mind some notion of *layering* which may refer to the way in which the components are *stacked* to compose a full application, or a notion of component stuffing where a component trait includes interfaces, implementations, and wiring plugs.

2.5 Family polymorphism

As it is a challenge to model families of types that vary together, share common code, and preserve type safety [5], *family polymorphism* has been proposed for OOPLs as a solution to supporting reusable yet type-safe mutually recursive classes [12].

To contextualise the problem, let's consider an example of graph modelling as in the original paper by Ernst [13]. We would like to implement classes for:

- a basic `Graph` with `Node`s and `Edge`s, and
- a `ColorWeightGraph` where `Node`s are colored (`ColoredNode`) and `Edge`s are weighted (`WeightedEdge`),

but we would like to do so in a way that:

- we can reuse base behaviors, and
- it is not possible to mix elements by different kinds of graphs.

Let's attempt a solution without family polymorphism:

```
// ABSTRACT GRAPH
trait Graph {
  var nodes: Set[Node] = Set()
  def addNode(n: Node) = nodes += n
}
trait Node
abstract class Edge(val from: Node, val to: Node)
```

```scala
// BASIC GRAPH
class BasicGraph extends Graph
class BasicNode extends Node
class BasicEdge(from:BasicNode, to:BasicNode) extends Edge(from,to)

// GRAPH WITH COLORED NODES AND WEIGHTED EDGES
class ColorWeightGraph extends Graph {
  override def addNode(n: Node) = n match {
    case cn: ColoredNode => nodes += n
    case _ => throw new Exception("Invalid")
  }
}
class ColoredNode extends Node
class WeightedEdge(from: ColoredNode,
     to: ColoredNode, val d: Double) extends Edge(from,to)

val bg = new BasicGraph
val cg = new ColorWeightGraph
val n = new BasicNode
val cn = new ColoredNode
// cg.addNode(n) // Exception at runtime
bg.addNode(cn)   // Ok (type-correct),
                 // but we didn't want ColoredNodes in a BasicGraph
```

There are two problems here:

- There is no static constraint that restricts users to not mix up the two families.
- In `ColorWeightGraph`, we cannot define `addNode` as accepting a `ColoredNode`, because covariant change of method parameter types is not allowed (it is a contravariant position).

These issues can be solved via family polymorphism:

```scala
trait Graph {
  type TNode <: Node; type TEdge <: Edge; type ThisType <: Graph

  trait Node { }

  trait Edge {
    var from: TNode = _; var to: TNode = _
    var fromWF: ThisType#TNode = _; var toWF: ThisType#TNode = _;
    def connect(n1: TNode, n2: TNode){ from = n1; to = n2 }
    def connectAcrossGraphs(n1: ThisType#TNode, n2: ThisType#TNode){
      fromWF = n1; toWF = n2
    }
  }
  def createNode: TNode;            def createEdge: TEdge
}

class BasicGraph extends Graph {
  override type TNode = BasicNode
  override type TEdge = BasicEdge
  override type ThisType = BasicGraph

  class BasicNode extends Node { }; class BasicEdge extends Edge { }

  def createNode = new BasicNode;   def createEdge = new BasicEdge
}

class ColorWeightGraph extends Graph {
  override type TNode = ColoredNode
  override type TEdge = WeighedEdge
  override type ThisType = ColorWeightGraph

  class ColoredNode(val color: String="BLACK") extends Node { }
  class WeighedEdge(val weight: Double=1.0) extends Edge { }

  def createNode = new ColoredNode;
  def createEdge = new WeighedEdge
}

val (g1, g2) = (new BasicGraph, new BasicGraph)
val (e1, n11, n12) = (g1.createEdge, g1.createNode, g1.createNode)
val (e2, n21, n22) = (g2.createEdge, g2.createNode, g2.createNode)

val cwg = new ColorWeightGraph
val (cwe, cwn1, cwn2) = (cwg.createEdge, cwg.createNode, cwg.createNode)

e1.connect(n11,n12)       // Ok, within same graph (of same family)
cwe.connect(cwn1, cwn2)   // Ok, within same graph (of same family)
//e.connect(n11,cwn2)     // ERROR!!! Cannot mix families

// e1.connect(n21,n22)
// ERROR: cannot connect an edge of a graph to nodes
//   of another graph, even if the graphs are of the same type

e1.connectAcrossGraphs(n11,n22)   // Ok. Within same family
                                  //   and across graph instances
// e.connectAcrossGraphs(n1,cwn1) // Of course, cannot mix families
```

Notes:

- `Graph` represents the *schema* of the family of graphs.
- `BasicGraph` and `ColorWeightGraph` extend the `Graph` trait and represent two distinct families of graphs.

- Families have type members introduced by `type` definitions.
- Remember that when a class is defined inside a class, a different class is reified for each different instance of the outer class. Then, note how type projection has beeen used to allow the mixing of graphs (within the same family).

In this case, the family traits also provide factory methods. An alternative approach would be to define `BasicGraph` and `ColorWeightGraph` as singleton objects, and then import their type members into the current scope.

```scala
object BasicGraph extends Graph {
    override type TNode = BasicNode
    override type TEdge = BasicEdge
    override type ThisType = BasicGraph

    class BasicNode extends Node { }
    class BasicEdge extends Edge { }
}

import BasicGraph._

val n = new BasicNode
```

2.6 Internal DSL development

The combination of Scala's features makes it a discrete tool for building (internal) *domain specific languages (DSLs)*. The features that support this kind of development include Scala's:

- implicits system,
- expressive type system,
- syntactic sugar,
- functional programming features (e.g., lambdas).

2.6.1 On syntactic sugar

Let's recap a few places where Scala provides syntactic sugar:

```
// Tuples
val tu1 = Tuple5('a', "s", 7, true, 8.8)
val tu2 = ('a', "s", 7, true, 8.8)

// 2-elements tuples
val t1 = Tuple2[String,Double]("xxx", 7.5)
val t2 = "xxx" -> 7.5

// Anonymous functions
val add1: (Int,Int)=>Int = (x,y) => x+y
val add2: (Int,Int)=>Int = _+_

// apply()
add2.apply(7,3)
add2(7,3)

// update()
val m = scala.collection.mutable.Map[Int,String]()
m.update(1, "xxx")
m(1, "aaa")

// Leaving out "." for member access
List(5,3,2) map { _%2 == 0 } reverse

// Leaving out () for parameterless methods
def f() { }
f

// Using braces { } for arg lists
def f(a: Int)(b: Char)(c: Boolean){ }
f { 7 } ( 'z' ) { false }

// Methods with name ending in ":" are right-associative
// A right-assoc binary op is a method of its 2nd arg
Nil.::(2).::(1)
1 :: 2 :: Nil

case class X(x: Int = 0) { def `set:`(y: Int) = X(y) }
4 `set:` 7 `set:` X(3)       // X = X(4)
(4 `set:` (7 `set:` (X(3)))) // X = X(4)

// Setters
m.+=(7 -> "a")
m += 7 -> "a"

// Varargs
def sum(xs: Int*) = xs.foldLeft(0)(_+_)
sum(1,4,3,2) // 10

// Call-by-name parameters
def f(s: => String) = println(s)
```

```
f { (1 to 9).foldRight("")(_+_) }
```

2.6.2 On associativity and precedence

This material is taken from the Scala Language Specification [1].

Prefix operations `op e` The prefix operator `op` must be one of the following: `+`, `-`, `!`, `~`. Prefix operations are equivalent to a postfix method call `e.unary_`*op*.

```
!false        // true
true.unary_! // false
4.unary_-    // -4

object a { def unary_~ = b }; object b { def unary_~ = a }
~(~(~a))  // b.type = b$@6c421123
```

Postfix operations `e op` These are equivalent to the method call `e.op`

Infix operations `e1 op e2` The first character of an infix operator determines the operator *precedence*. From lower to higher:

$$\text{(All letters)} \prec | \prec \hat{} \prec \& \prec < > \prec = ! \prec : \prec + - \prec * / \% \prec \text{(All others)}$$

Associativity depends on the operator's last character. All operators are left-associative except those with name ending in ':' that are right-associative.

Precedence and associativity determine how parts of an expression are grouped:

- Consecutive infix operators (which must have the same associativity) associate according to the operator's associativity.
- Postfix operators always have lower precedence than infix operators: `e1 op1 e2 op2 == (e1 op1 e2) op2`.

Infix operations are rewritten as method calls: a left associative binary operator `e1 op e2` is translated to `e1.op(e2)`, whereas if the operator has arity greater than 1, it must be used as `e1 op (e2,...,en)`, which is translated to `e1.op(e2,...,en)`.

Here are some examples:

```
obj m1 p1 m2 p2 m3 p3 == ((obj m1 p1) m2 p2) m3 p3)
                      == obj.m1(p1).m2(p2).m3(p3)
```

2.6.3 "Dynamic" features

Scala (since v2.9) has a feature similar to Ruby's `method_missing`:

```
class X
  def method_missing(name, *args)
    "you called '%s'" % [name.to_s]
  end
end

X.new.hello  //  => "you called 'hello'"
```

Scala provides a marker trait, `Dynamic`, that tells the compiler to rewrite accesses to non-existing members as calls to the following methods:

- `selectDynamic(fieldName)` – for field reading.
- `updateDynamic(fieldName)(args)` – for field writing.
- `applyDynamic(methodName)(args)` – for method calls.
- `applyDynamicNamed(methodName)(namedArgs)` – for method calls with named arguments.

To use this feature, you need to set the compiler option `-language:dynamics` or import `scala.language.dynamics`.

2.6.4 Examples

A DSL for writing URIs Source:

```
case class Uri(scheme: String = "http",
                  path: List[String] = List(),
                  querystring: Map[String,Any] = Map()) {
  def /(s: String) =
    this.copy(path = s :: path)

  def /?(t: (String,Any)) =
    this.copy(querystring = querystring + t)

  def &(t: (String,Any)) = this./?(t)

  override def toString = scheme + "://" +
    path.reverse.mkString("/") +
    (if(querystring.isEmpty) "" else "?" +
      querystring.keys.map(k => k+"="+querystring(k)).mkString("&"))
}

object UriDsl {
  def http  = Uri("http")
  def https = Uri("https")
  def ftp   = Uri("ftp")

  implicit def strToUrl(s: String): Uri = Uri(s)
}

import UriDsl._

http / "www.site.org" / "index.php" /? ("a"->7) & ("b"->true)
// Uri = http://www.site.org/index.php?a=7&b=true

"file" / "usr" / "bin" / "javac"
// Uri = file://usr/bin/javac
```

The idea behind the previous example is simple: methods in object `UriDsl` work as entry points by building an `Uri` instance, then we chain method calls by having methods return a new object of the same kind.

A DSL for math operations Source:

```
trait MathOperation

implicit class IntMathOperation(val n: Int) extends MathOperation {
  def \(d: Int) = Fraction(n,d)
}
trait MeanOperation extends MathOperation {
   def of(ns: Double*) = ns.foldLeft(0.0)(_+_) / ns.length
}
case class PowerOperation(base: Double) extends MathOperation {
  def by(exp: Double) = math.pow(base,exp)
}

case class Fraction(num: Int, den: Int) {
  def +(f2: Fraction) = {
    val m = Fraction.mcm(den, f2.den)
    Fraction((m/den)*num + (m/f2.den*f2.num), m)
  }
}

object Fraction {
  def simplify(f: Fraction): Fraction = {
     val d = gcd(f.num, f.den)
     Fraction(f.num/d, f.den/d)
  }

  def gcd(x: Int, y: Int): Int = // Euclid's algorithm
    if(x == y) x
    else if(x > y) gcd(x-y,y)
    else gcd(x, y-x)

  def mcm(x: Int, y: Int): Int = (x / gcd(x,y)) * y
}

object MathDsl {
  def mean = new MeanOperation { }
  def power(n: Double) = PowerOperation(n)
  def simplify(f: Fraction) = Fraction.simplify(f)
}
import MathDsl._

mean of (6, 10, 7, 9)         // Double = 8.0
power(2) by (mean of (4,6))  // Double = 32.0
1\2 + 4\3 + 1\6               // Fraction = Fraction(12,6)
simplify { 1\2 + 4\3 + 1\6 } // Fraction = Fraction(2,1)
```

Part II

Background: Aggregate Programming

Chapter 3

Space-Time Programming and Spatial Computing

This chapter discusses the main challenges of recent distributed computing scenarios and introduces space-time programming as a promising approach for building large-scale, decentralised, adaptative systems.

Outline:

- Why spatial computing – to understand the forces that have dictated the need for a new approach to distributed computing
- What is spatial computing – to understand the key concepts and characteristics defining spatial computing
- How is spatial computing in practice – approaches and technologies

3.1 Motivation: context and issues

The last century have witnessed tremendous technological advances with revolutionary repercussions. **Computers** have extended human's intelligence with precise, high-speed processing of symbols, supporting instantaneous calculations and algorithmic power. **Telecommunications** have opened the doors to low-latency, global communication by interconnecting distant places and allowing fast information flows between them.

More recently, the physical limits to Moore's law for what concerns processing rate have switched research into transistor scaling and multi-core architectures to keep the pace. Then, the decrease of the cost and size of hardware has allowed for a **mass production of miniaturised devices**, which is leading to the spread of computational abilities in our environments and a tighter interaction between artificial systems and natural systems.

Some advances not only increase the efficiency of what we already do, but also enable us to make new things, possibly things we did not even imagine. That is, innovations push forward our needs and imagination. The progressive decentralisation fostered by the low-cost production of computational devices and the embeddability resulting from the miniaturisation process make it possible to approach existing problems in novel ways (e.g., swarm robotics) and to think at new applications (e.g., ambient intelligence).

3.1.1 Distributed computing scenarios

These enabling conditions have led to the development of a variety of scientific and engineering trends sharing common ideas. Often these threads overlap and are referred to with different terms to account for peculiarities of applications or nuances in the interpretation or vision. A few examples:

- *Internet of Things (IoT)* – emphasises the interconnection between everyday life's objects.
- *Pervasive Computing, Ubiquitous Computing, Everyware* – refer to the diffusion, permeation of computation in all the places and aspects of life, fostering a vision where computational abilities are available everywhere they are needed.
- *Wireless Sensor Networks (WSN)* – deal with the engineering of networks of sensors that capture, move and possibly process environmental data; the term focuses on the technological infrastructure.
- *Smart things (cities, buildings, homes, dust), Ambient Intelligence* – refer to the embedding of "intelligence" in our environments as a set of context-

sensitive services; here, the emphasis is on the functionalisation of the plain old artificial environment with features perceived useful and somehow unexpectedly smart from traditionally idle structures

- *Swarm robotics* – considers systems composed of many simple robots that interact and coordinate on a local basis.

All these scenarios deal with a large number of computational devices that interact with one another and with their environment. Typically, applications exhibit common traits:

- Context-sensitiveness/awareness – services have to provide responses that are highly-sensitive to contextual information.
- Global-to-local correlation – services are expected to provide summarised information in specific places or to specific individuals.
- Collective behavior – tasks may not be practically feasible by only few system participants; in other words, a large number of components may be not only useful for precision or efficiency, but also functional to the service itself.

A tension between the global and local viewpoints is emerging.

3.1.2 Key issues and unsuitableness of traditional approaches

In general, the design and development of distributed systems is hard, as one have to deal with consistency and replication, failure and recovery, communication and synchronisation, as well as the problems resulting from autonomy, heterogeneity, openness, and many other aspects. In particular, coordination becomes key – the design effort usually turns from the computation dimension to the interaction dimension.

The aforementioned scenarios such as pervasive computing stress classical problems of distributed systems and also add new ones. In such cases, designers have to deal with the following issues:

- Unpredictability – the environmental dynamics may leave room only for few assumptions, requiring applications to deal with the inherent randomness and complexity of their surroundings.

- Network complexity – networks may consist of many nodes with possibly highly dynamic connections.
- Situatedness – the environment has to be considered in the design process.

Such scale and unpredictable dynamics make traditional distributed computing approaches based on men-in-the-loop or centralised control inadequate [14]. In fact, given the large number of elements composing such systems, centralisations can easily become bottlenecks, whereas for what concerns the unforeseeable of the environment and network dynamics, it may be advantageous to relax the corpus of assumptions and rely on some form of self-organisation.

Problems with centralisations Decentralisation is actually demanded in light of typical constraints and characteristics found in many pervasive computing scenarios:

- *Energy constraints and communication* – Often, the cost of communication compared with the cost of computation, combined with relevant energetic limitations, suggests to trade the former for the latter.
- *Communication latency* – As communication is costly and takes significant time, it is convenient to keep information close to its use (*locality principle*). In addition, in the case of network nodes equipped with actuators, it would be even more wasteful to have round-trip communications with a central server/coordinator.
- *Big Data* – The global data generated by networks may be impressive in regard to volume, velocity, and variety.
- *Network scalability* – The density of nodes in a network is related to the level of spatial detail provided by the network and may affect the quality of application results. However, a WSN application may be started small while at the same time keeping the ability to scale in order to increase the accuracy of spatial sampling in a second moment.

Problems with transparency Another traditional feature that does not work for these scenarios is distribution transparency. First of all, transparency poses limitations for what concerns scalability. In fact, location transparency (based on logical identifiers) requires a logically centralised naming service which, in general, has to deal with mobility as well. Secondly, it does not support the typical openness requirements where devices may frequently join and leave a network. Third, transparency does not account for situatedness.

The opposite approach to transparency is to make distribution explicit. In the so-called *network-aware computing models*, a node can communicate with another node in the network only if it knows that node's location (address) in the network. Here, the drawback lies in the abstraction gap: the model is too low-level and complex, and there is the need for environmental abstractions.

In [14], the distinction is clearly shown by expressing the different information requirements for communicative acts in the three cases:

- Transparent location models – *"I know who you are, but I don't know where you are."*
- Network-aware models – *"I know who you are and I know where you are."*
- Spatial-computing – *"I don't know who you are, but I know where you are."* (see Section 3.2)

Need for self-* As the complexity and scale of the systems grow, the open-loop structure where maintenance and recovery require human intervention becomes more and more costly and unsustainable. Additionally, it might be crucial to perform these activities without stopping the system operation, and doing so in a timely manner. This suggests to move towards self-adaptive systems[15] which are able, to some extent at least, to autonomously face the contingencies of an ever-changing context and self. In other words, the trend is about increasing the autonomy of artificial systems and the way seems to consist in endowing them with self-* properties such as self-configuration, self-healing, or – more generally – self-adaptativeness.

3.2 Spatial computing

Spatial Computing is, in a broad sense, an umbrella term for approaches to computation based on spatial features. In more precise terms, a spatial computer can be defined, as in [16], as a *"network of interacting devices such that the difficulty of moving information between devices is strongly correlated with the physical distance between them"*.

While there is no unanimous agreement on what spatial computing is exactly, it is important to understand the different contexts in which the term can be used.

3.2.1 Space-oriented computation

In [17], spatial computing is characterised as the paradigm of *computing somewhere*, namely, computing simultaneously *in* and *about* geographic space, and is distinguished from other space-related approaches by means of a taxonomy based on two axes:

1. *Information* – may be related or unrelated to location
2. *Communication constraints* – may be spatial or non-spatial

which results in four kinds of systems:

1. *"Computing somewhere" (location-related information, spatial constraints)*
   In these systems, the location strongly affects the stored information and, in turn, the computation based on such information, and communication depends on the spatial distribution of the system. Examples include WSNs and many pervasive computing applications.
2. *"Computing everywhere" (location-related information, non-spatial constraints)*
   This class of systems consists in the so-called Location-Based Services (LBS), e.g., applications showing nearby friends (possibly with notifications) or applications supporting navigation to Points Of Interest (POI).
3. *"Computing anywhere" (location-unrelated information, spatial constraints)*
   A paradigmatic example is given by Mobile Ad-Hoc Networks (MANET).

Here, communication is limited in space, but information may be totally location-unrelated.

4. *"Computing nowhere" (location-unrelated information, non-spatial constraints)*
   The systems belonging to this class – such as, for instance, the Internet – support remote interactions, and information is independent from the site where it is processed.

Another contribute to the categorisation of space-oriented approaches has been provided during the seminal Dagstuhl seminar on *Computing Media and Languages for Space-Oriented Computation*, where *"three thematic areas have been identified: **intensive computing** where space is used as a mean and as a resource, **computation embedded in space** where location is important for the problem and **space computation** where space is fundamental to the problem and is a result of a computation"*[1].

3.2.2 Defining spatial computing (as space-time programming)

In the context of this work, spatial computing is an approach to computing based on the use of spatial abstractions for system definition and coordination. The key ideas are delineated in [14].

A system, essentially a network of interacting devices situated in a physical environment, is logically represented as a *virtual space*. This correspondence between the actual situation and the logical representation creates a mapping between the physical space and the virtual one (cf., the amorphous medium in Section 3.2.4).

The system elements fill portions of the space and interact between one another according to a notion of temporal and spatial locality expressed by the concept of *neighbourhood*. As communication is driven by location in space, there is no need to know the name of the recipient.

The devices are *space-aware*: they can sense the surrounding, proximal envi-

[1]`http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=06361`

ronment through sensors and can change that by means of actuators.

Due to the variability of the term *spatial computing*, it may be better to use the more recent term *space-time programming* to better identify the acceptation described in this section.

3.2.3 Analytical framework

A survey of spatial computing DSLs have been presented in [18] together with an analytical framework, summarised next.

Generic Aggregate Programming Architecture A *Generic Aggregate Programming Architecture* is sketched to account for the different levels of abstractions that a spatial computing platform may have to deal with. It is comprised of five layers (from lower to upper):

1. *Physical platform*
2. *System management*
3. *Abstract device*
4. *Spatial computing*
5. *Users and applications*

General classes of operations By considering the space-time and informational duality of spatial computers, namely, the simultaneous situatedness in related physical and informational worlds, four classes of operations are derived:

1. *Physical Evolution* – refers to operations from space-time to space-time.
2. *Measure Space-Time* – refers to operations from space-time to information.
3. *Manipulate Space-Time* – refers to operations from information to space-time.
4. *Compute Pattern* – refers to operations from information to information.

3.2.4 Discrete vs. continuous space-time

Spatial computing is a matter of computation and coordination over space-time[19]. The devices are *situated* both in space and in time; the device state and position may change over time; and possible interactions depend on the network topology at a given instant. However, while real-world space-time is assumed to be continuous, spatial computers are composed of a discrete number of devices which work at discrete rounds as triggered by their clocks.

The amorphous abstraction[20] According to this abstraction, a system composed of a discrete set of interconnected devices (i.e., a network) can be seen as a discrete approximation of the continuous space in which it is distributed (see Figure 3.1). The more the density of devices increases, the more the network approximates the physical space. At the limit, the result is a *"continuous space filled with continuously computing devices and continuously propagating 'waves' of communication"* where *"the output of computation in each point can be viewed as a property of the space at that point"*[19].

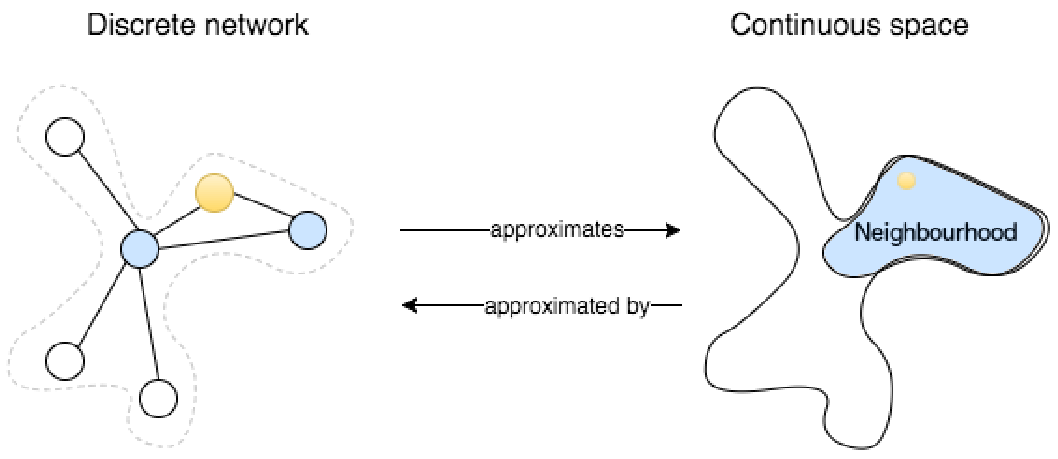

Figure 3.1: The amorphous medium abstraction creates a correspondence between a discrete network and a continuous space.

Chapter 4

Field Calculus

This chapter introduces a theoretical model for spatial computing (see Chapter 3) that provides a solid foundation for the aggregate programming approach described in Chapter 5.

Outline:

- The notion of computational field
- Field Calculus: basic constructs, operational semantics
- Higher-Order Field Calculus and code mobility
- Protelis

4.1 Computational fields

A computational field is a function that maps the points in some discrete or continuous space-time domain to some computational value [21][22]. The metaphor brings in computational terms the notions of scalar and vector fields found in math and physics. In the case of a network of devices, a field maps each device with a value. It should be stressed that fields are functions that may and typically will evolve over time.

Operations on fields are defined as functions taking input fields and producing output fields. Then, the key idea is to express the global behavior of a spatial computing system as a functional composition of operators that manipulate (evolve, combine, restrict) computational fields.

4.2 Field calculus

4.2.1 Basis set of operators

Field Calculus is a minimal calculus for manipulating fields. The original set of constructs consists in:

1. *Built-in function application* – $(b \quad e_1\ e_2\ \ldots\ e_n)$
   The built-in function b is applied to the input fields e_1 to e_n. The output field is given by the point-wise evaluation of the operator to the input fields; in other words, each device is mapped to the result obtained by applying the operator to its local values of the input fields.
2. *Function definition* and *function call*
   $(\mathrm{def} \quad f(\overline{x}) \ \ e_B)$ – The function f is defined with a list of arguments $\overline{x}$ and a body consisting in the expression e_B.
   $(f \quad e_1\ \ldots\ e_n)$ – The function f is applied to the input expressions e_1 to e_n. The function call expression is equivalent to the function body e_B after the substitution of formal parameters x_i with their respective actual parameters e_i.
3. *Time evolution* – $(\mathrm{rep} \quad x \ \ w \ \ e)$
   The *rep* construct supports dynamically evolving fields by having an expression e depend on its previous value w (with x being the initial state).
4. *Interaction* – $(\mathrm{nbr} \quad e)$
   The *nbr* construct maps each device δ with a field consisting of the neighbours' most recent value resulting from the evaluation of e. Thus, it implies a communication from each device to its neighbors. The result is a field of fields.
5. *Domain restriction* – $(\mathrm{if} \quad e_0\ e_1\ e_2)$
   It ensures that the evaluation of e_1 occurs only in the network subset where e_0 evaluates to true, and that e_2 is evaluated only in the points (devices) where

the condition e_0 turns out to be false. Note it is different from first evaluating e_0, e_1, e_2 and then returning the appropriate value, which wouldn't perform any distributed branching.

Note that these constructs are subject to a twofold interpretation. Within the *local*, device-centric interpretation, any field calculus expression represents a locally computed value in a device at a given instant. Conversely, the same expression represents a computational field within the *global* interpretation. More on these different viewpoints can be found in Section 5.1

4.2.2 Higher-Order Field Calculus (HOFC)

The field falculus has been extended in a higher-order variant (HOFC) with first-class functions in [23]. The updated syntax follows:

$$\boxed{\begin{array}{lcl}
e & = & x \mid v \mid (\colorbox{lightgray}{e}\ \overline{e}) \mid (f\ \overline{e}) \mid (rep\ x\ w\ e) \mid (nbr\ e) \mid (if\ e\ e\ e) \\
v & = & \ell \mid \phi \\
\ell & = & b \mid n \mid \langle \ell, \ell \rangle \colorbox{lightgray}{$\mid o \mid f \mid (fun\ (\overline{x})\ e)$} \\
w & = & x \mid \ell \\
F & = & (def\ f(\overline{x})\ e) \\
P & = & \overline{F}\ e
\end{array}}$$

Differences from basic field calculus syntax are highlighted in grey. Essentially, the syntactic extension accounts for the possibility to refer to built-in (o) and user-defined functions (f) as local values (ℓ) as well as defining new anonymous functions on-the-fly (lambdas) with the syntax $(\text{fun}\ \ args\ \ body)$. Then, these values ($v$), which could also be referred to by variables x (these may be arguments in higher-order functions, or the evolving value in a rep), can be used in function position for application.

4.2.3 Operational semantics

The following description is based on the big-step operational semantics presented in [23].

Preliminary definitions and notation.

- Devices – represented by δ.
- Fields – represented by $\phi ::= \overline{\delta} \mapsto \overline{\ell} = \delta_1 \mapsto \theta_1, ..., \delta_n \mapsto \theta_n$.
- Value tree – ordered tree of values represented by $\theta ::= v(\overline{\theta})$.
  - Root of the value tree: $\rho(v(\overline{\theta})) = v$.
  - k-th subtree of the value tree: $\pi(v(\theta_1, ..., \theta_n)) = \theta_k \quad 1 \leq k \leq n$.
- Value tree environment – represented by $\Theta ::= \overline{\delta} \mapsto \overline{\theta}$.

Rule [E-LOC] – Evaluation of local values.

$$\frac{}{\delta; \Theta \vdash \ell \Downarrow \ell()}$$

A local value evaluates to a value tree with the value itself as the root and an empty subtree.

Rule [E-FLD] – Evaluation of field values.

$$\frac{\phi' = \phi \mid_{\mathbf{dom}(\Theta)\cup\{\delta\}}}{\delta; \Theta \vdash \phi \Downarrow \phi'()}$$

A field value evaluates to a value tree with the field value itself as the root, adequately restricted to take into account *domain alignment*. Namely, the domain of a field value is given by the current device (on which the computation is running) and all the aligned neighbours, i.e., the neighbours whose most recent value tree is structurally compatible with the value tree being evaluated.

Rule [E-B-APP] – Built-in function application.

$$\frac{\begin{array}{ccc} \delta; \pi_1(\Theta) \vdash e_1 \Downarrow \theta_1 & \dots & \delta; \pi_{n+1}(\Theta) \vdash e_{n+1} \Downarrow \theta_{n+1} \\ \rho(\theta_{n+1}) = o & & v = \epsilon^{o}_{\delta;\Theta}(\rho(\theta_1), ..., \rho(\theta_n)) \end{array}}{\delta; \Theta \vdash e_{n+1}(e_1, ..., e_n) \Downarrow v(\theta_1, ..., \theta_{n+1})}$$

where the auxiliary function $\epsilon^{o}_{\delta;\Theta}(\overline{v})$ represents the application of the built-in function o to input values $\overline{v}$ in the environment Θ on device δ.

The inference rule is read as follows:

- The expressions e_i of the function application $e_{n+1}(e_1, ..., e_n)$ evaluate to value trees θ_i.
- The built-in function o to be executed is given by the root of the value tree θ_{n+1}, while the function arguments correspond to the root of value trees θ_1 to θ_n.
- The application of o to its arguments on the device δ with tree environment Θ results in a local value v.

The built-in function application expression evaluates, in a device δ with environment Θ, to a value tree of the form $v(\theta_1, ..., \theta_{n+1})$.

Rule [E-D-APP] – Application of user-defined or anonymous functions.

$$\frac{\begin{array}{c}\delta;\pi_1(\Theta)\vdash e_1 \Downarrow \theta_1 \qquad \dots \qquad \delta;\pi_{n+1}(\Theta)\vdash e_{n+1} \Downarrow \theta_{n+1}\\ \rho(\theta_{n+1}) = \ell \qquad args(\ell) = x_1, ..., x_n \qquad body(\ell) = e\\ \delta;\pi^{\ell,n}(\Theta)\vdash e[x_1 := \rho(\theta_1)\ \dots\ x_n := \rho(\theta_n)] \Downarrow \theta_{n+2} \qquad \mathbf{v} = \rho(\theta_{n+2})\end{array}}{\delta;\Theta\vdash e_{n+1}(e_1, ..., e_n) \Downarrow \mathbf{v}(\theta_1, ..., \theta_{n+2})}$$

where the auxiliary function $\pi^{\ell,n}(\theta)$ extracts the $(n+2)$-th subtree of θ if the root of θ_{n+1} equals to ℓ; the same function is defined to work with value tree environments as well, by mapping each element $(\delta_i \mapsto \theta_i)$ to $(\delta_i \mapsto \pi^{\ell,n}(\theta_i))$, leaving out the mappings of non-aligned neighbours. The auxiliary functions $args$ and $body$, as the name implies, extract the arguments and the body from the function definition (which may be a user-defined function or an anonymous function).

This rule is similar to the previous one, but the resulting value tree has an additional subtree corresponding to the evaluation of the function body with respect to an environment – repetita iuvant – containing only the value trees of the neighbours executing the same function ℓ. Figure 4.1 provides a visual representation of the rule.

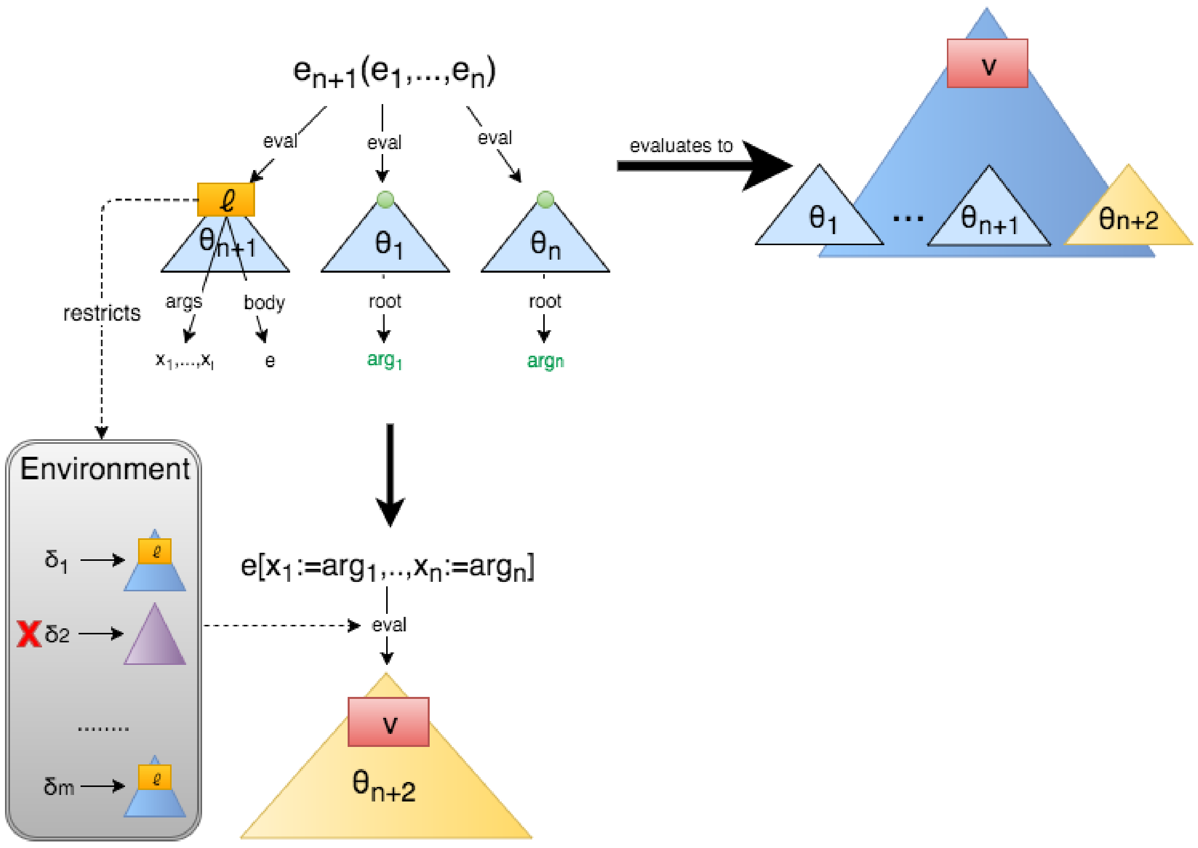

Figure 4.1: Visualisation of the evaluation of a function application according to rule [E-D-APP].

Rule [E-REP] – `rep` **construct.**

$$\ell_0 = \begin{cases} \rho(\Theta(\delta)) & if\ \Theta \neq \emptyset \\ \ell & otherwise \end{cases}$$

$$\frac{\delta; \pi_1(\Theta) \vdash e[x := \ell_0] \Downarrow \theta_1 \qquad \ell_1 = \rho(\theta_1)}{\delta; \Theta \vdash (rep\ \ x\ \ell\ e) \Downarrow \ell_1(\theta_1)}$$

As described previously, the `rep` construct models evolution over time, where an expression e is evaluated with reference to a state variable x, which is initially set to ℓ and then updated by the result of the `rep` expression. In this rule, particularly relevant is the part devoted to the reuse of the previously computed state (if any). In the `rep` expression, the state variable x is substituted by ℓ_0, which is set to the previous state by accessing to the root of the value tree for the current device δ in the device itself's environment, or the initial state ℓ if no previous state is found in the value tree environment.

Given the body expression for the `rep` evaluates to a value tree θ_1, the value tree resulting from the evaluation of the entire `rep` expression consists in a root value equals to the root of θ_1 and a single subtree equals to θ_1 itself.

Rule [E-NBR] – `nbr` construct.

$$\frac{\Theta_1 = \pi_1(\Theta) \qquad \delta; \Theta_1 \vdash e \Downarrow \theta_1 \qquad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]}{\delta; \Theta \vdash (nbr \ \ e) \Downarrow \phi(\theta_1)}$$

where the auxiliary function $\pi_n(\theta)$, also extended to operate on value tree environments, extracts the n-th subtree of the value tree θ. By "entering" the expression e, according to the expected value tree format for `nbr`, the auxiliary function pi models structural alignment.

As can be seen from the inference rule, the `nbr` body expression, evaluated with respect to an environment Θ_1 of aligned devices, result into a value tree θ_1. Then, the result of the whole `nbr` expression is given by a field mapping each device in the environment with the corresponding root value for θ_1, where for the current device δ any previous value is overwritten by the newly computed value; then, the single subtree is given by θ_1 itself. Figure 4.2 shows the rule in graphical form.

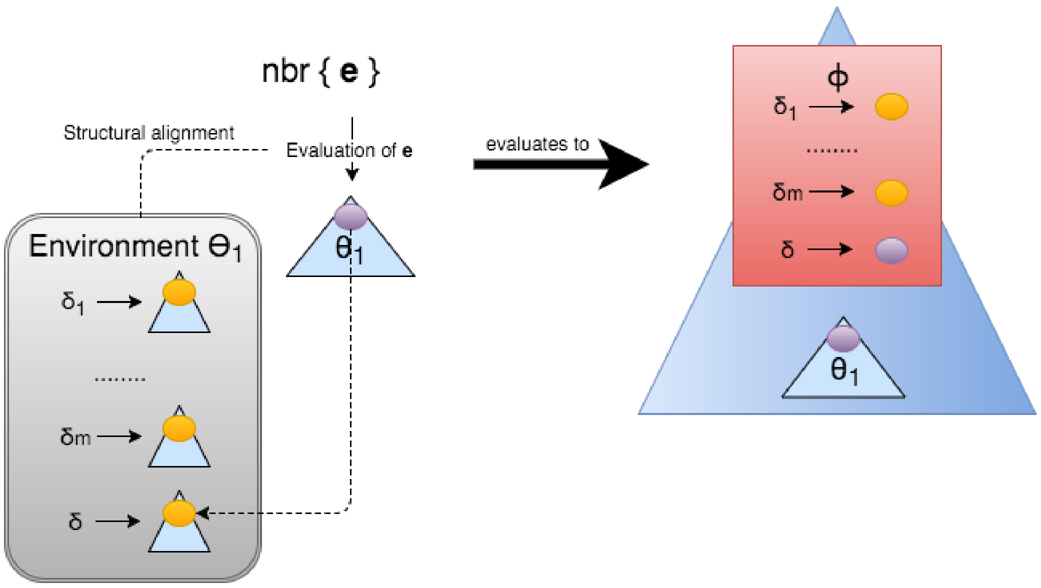

Figure 4.2: Visualisation of the evaluation of a nbr expression according to rule [E-NBR].

Rules [E-THEN] and [E-ELSE] – `if` branching construct.

$$\frac{\delta;\pi_1(\Theta)\vdash e\Downarrow\theta_1 \qquad \rho(\theta_1)=true \\ \delta;\pi^{true,0}(\Theta)\vdash e'\Downarrow\theta_2 \qquad \ell=\rho(\theta_2)}{\delta;\Theta\vdash(if\ \ e\ \ e'\ \ e'')\Downarrow\ell(\theta_1,\theta_2)}$$

$$\frac{\delta;\pi_1(\Theta)\vdash e\Downarrow\theta_1 \qquad \rho(\theta_1)=false \\ \delta;\pi^{false,0}(\Theta)\vdash e''\Downarrow\theta_2 \qquad \ell=\rho(\theta_2)}{\delta;\Theta\vdash(if\ \ e\ \ e'\ \ e'')\Downarrow\ell(\theta_1,\theta_2)}$$

Here, the crucial part is domain restriction: $\pi^{true,0}(\Theta)$ restricts the environment to the pairs $d_i\mapsto\ell_i(\theta_{i1},\theta_{i2})$ where $\rho(\theta_{i1})$ is equal to $true$ and maps them to $d_i\mapsto\theta_{i2}$.

The whole `if` expression is evaluated to a value tree with the local result as the root value and two subtrees: one is the value tree for the condition, and the other is the value tree for the *then* or *else* part (depending whether the local condition result $\rho(\theta_1)$ is $true$ or $false$, respectively).

4.2.4 Case study: Protelis

Protelis[24] is a functional language for expressing field-based computations. More precisely, it is an external DSL developed with the Xtext language workbench. Protelis is inspired by Proto but, with respect to Proto, provides the following improvements:

- novel, Java-like syntax (rather than LISP-like),
- integration with the Java ecosystem – Protelis is hosted in Java, and Protelis programs can import and use Java code, and
- support for code mobility via first-class distributed functions (see about higher-order field calculus in Section 4.2.2).

In Protelis, a program consists of three parts:

1. a set of Java imports,
2. a set of function definitions, and
3. a set of statements – which can be variable declarations, assignments, expressions.

and the program result is given by the last statement.

As an example, consider the hop-gradient:

```
import com.example.Node.sourceSensor

def hopGradient(source) {
  rep(hops <- Infinity){
    mux(source) { 0 }
    else { 1 + minHood(nbr{hops}) }
  }
}

let isSrc = sourceSensor.read();
hopGradient(isSrc)
```

Two prominent uses of Protelis include the development of a Protelis module for the Alchemist simulator[25], and the implementation of a distributed management system for the automation of the recovery of a set of interdependent enterprise services[26].

Chapter 5

Aggregate Programming

This chapter introduces the aggregate programming paradigm as a way of programming *aggregates* of devices in a top-down way, and presents a multi-layered architecture based on field calculus (see Chapter 4) and reusable building blocks.

Outline:

- From single-device-view to aggregate-view
- The aggregate programming stack
- Composable self-organisation via self-stabilising building block operators

5.1 From device to aggregate view

Aggregate Programming is a paradigm which has to do with the programming of aggregates, where an *aggregate* can be generally defined as a moltitude of elements. This paradigm represents a departure from traditional *device-centric* approaches where the single device is considered the programmable unit and the different system elements have to be designed so that they produce, by interaction, some desired global behavior.

Software systems have been so far designed with a bottom-up approach to system behavior specification. This reflects the natural view in which the system behavior emerges out of the behaviors of the parts.

However, sometimes it may be easier or more effective to specify the global system behavior in a top-down way. Of course, the system elements remain the ultimate vehicle for the whole, and they have to be programmed, so it is not possible to completely exclude the details of how the dynamics unfolds from the bottom-up. So, a balance of bottom-up and top-down reasoning is important to raise the abstraction level and seems essential to instill self-organising properties to systems.

Therefore, the key idea of aggregate programming is to let local behaviors be deduced from the high-level, global specification. This requires the presence of a transformation logic that we may refer to as a *global-to-local mapping.*

Such an attempt to capture emergent abstractions also means that, when programming aggregates, two distinct points of view may be embraced by programmers:

1. *Local viewpoint (device-centric view)* – refers to the interpretation of the aggregate computation when executed by a single-device; this is the traditional view, where the programmer reasons about the local computations performed by a device and about how the device interacts with the other elements of the system.
2. *Global viewpoint (aggregate view)* – refers to the overall computation performed by the system as a whole; in this view, the programmer is more concerned about *what* the system should perform rather than about *how* to program the system elements to achieve the desired behavior.

For example, a `rep` expression in the field calculus may be interpreted according to the first or second viewpoint: locally, a device repeatedly builds a new state depending on the current state; globally, the computation may be seen as a computational field dynamically evolving over time.

5.2 The aggregate programming stack

A practical aggregate programming platform is one that reduces the abstraction gap by providing developers with user-friendly APIs. This raising of the abstraction level is likely to require multiple layers so as to manage complexity and foster a logically-incremental architecture.

An aggregate programming stack built upon field calculus and aimed at the development of Internet of Things applications have been depicted in [27]. Figure 5.1 shows the layers in such an architecture.

The significance of founding the aggregate programming support on a minimal calculus is that it allows for formal proving of properties of interest. Thus, on top of it, it is possible to define building block operators with provable characteristics and, more importantly, ensure that any or some compositions of these building blocks retain the same properties.

5.2.1 Composable self-organisation

Self-stabilisation A system is said to be *self-stabilising* if, independently of the current state, it is able to reach a stable state within finite time. In other words, such a system is guaranteed to recover from perturbations, more or less promptly.

The paper [28], by considering a restriction of the field calculus, shows that self-stabilisation is *"proved for all fields inductively obtained by functional composition of fixed fields (sensors, values) and by a gradient-inspired spreading process."*

Classes of building blocks Upon this foundation, a set of general, self-stabilising building block operators is presented in [29], together with a related categorisation (reported in Table 5.2.1). The classes of functions in the taxonomy have been extracted by considering the key "moves" performed in a general aggregate computation cycle:

1. *Sensing* – This input phase consists in the acquisition of information, mostly associated to the space-time structure of the system. The building blocks related to *structure* can be modelled as functions that query environamental sensors for values (computational fields).
2. *Detection of situations of interest* – The collected information has to be analysed in order to determine the context and thus be able to set the system goals. This typically involves the production of summary information by means of *aggregation.*

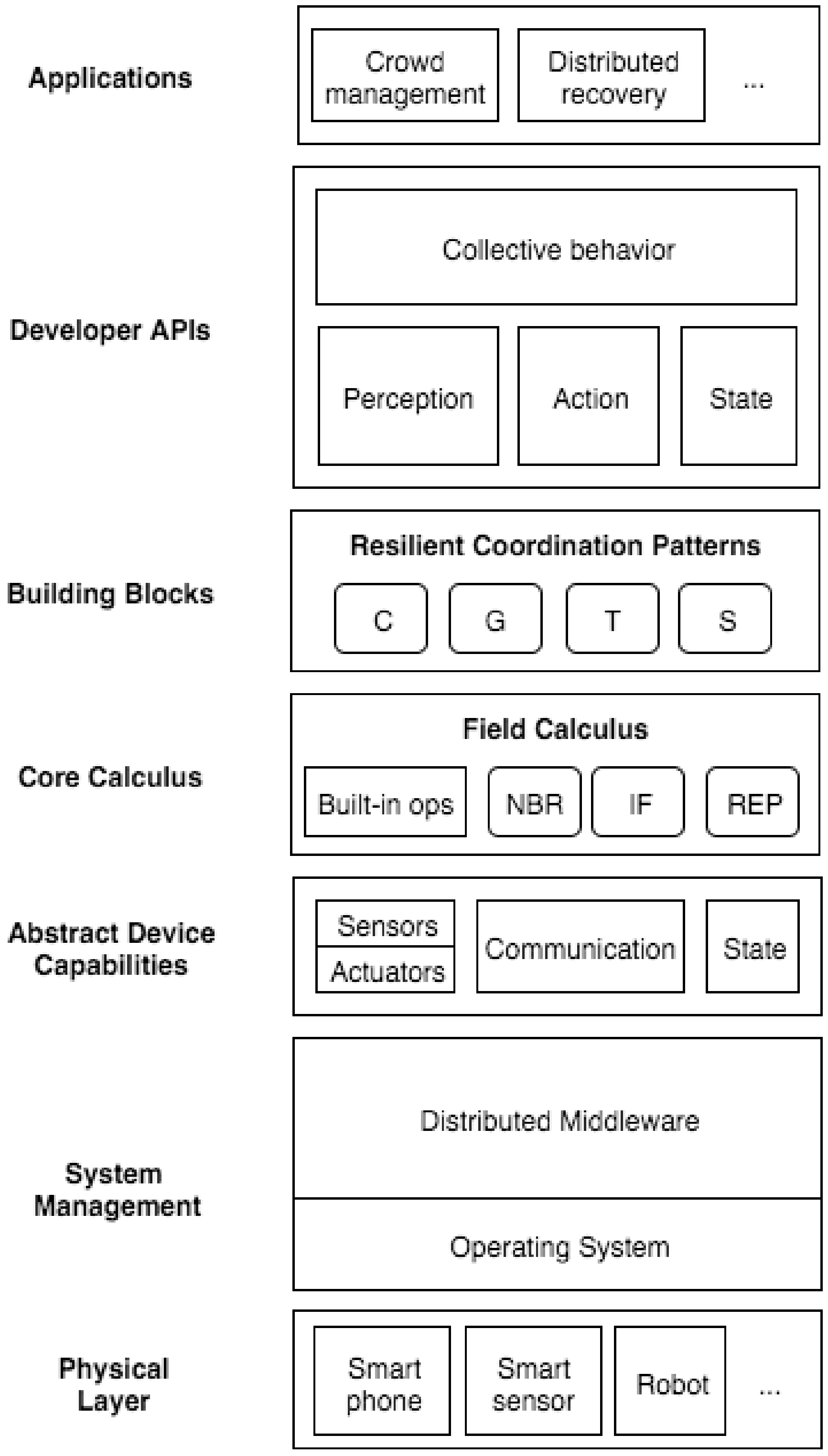

Figure 5.1: Aggregate Programming Stack

3. *Moving information where needed* – The information has to be moved or *spread* to the elements responsible for action. This may involve the identification of a sparse subset of devices (*simmetry breaking*) so as to to limit the information flow (and thus the action) to specific portions of the system.
4. *Acting based on context information* – In this output phase, the system generates a response to the realised situation. In particular, the response will typically depend on the result of some *computations*, which may be *restricted* to specific locations of the space-time fabric.

Function	**Space**	**Time**
Structure	nbr-range, ...	dt, ...
Aggregation	C	T
Spreading	G	
Symmetry breaking	S	random
Restriction	if	
Compute	local functions, random	

Table 5.1: Taxonomy for some building block operators.

Building block operators The building blocks identified in [29] are, for all intents and purposes, generalised coordination operators that could be used to cover several coordination patterns. These operators also include the `if` construct of the field calculus, which is effectively able to restrict computations in space within particular regions, thus being extremely valuable for composition. The other four operators are:

1. ***G**radient-cast*: `G(source, init, metric, accumulate)`.
   It simultaneously performs two tasks: builds a distance-gradient from the `source` according to `metric`, and builds `accumulate`d values along the gradient starting from `init` at the `source`.
2. ***C**onverge-cast*: `C(potential, accumulate, local, null)`.
   It allows to **c**ollect information distributed across space by accumulating values down the gradient of a `potential` field. In a sense, `C` is the dual of `G`.

3. ***T**ime-decay*: `T(initial, decay)`.
   This operator can be used to summarise information across time by decreasing the `initial` field according to a `decay` function.
4. ***S**parse-choice*: `S(grain, metric)`.
   This operator can be useful for creating partitions and for selecting sparse subsets of devices in space.

These can be composed into new operators that will maintain the same properties of robustness. For example, `C` and `G` can be combined to originate a self-stabilising `summarise` operator that first collects information across the space and then propagates back the computed summary.

Part III

`scafi`: Development

Chapter 6

Analysis

This chapter is intended to provide a summary of the results of the *analysis* phase in the `scafi` development cycle. Despite of the organisation of this part, it should be noted that the development of `scafi` has been carried out in an iterative, incremental process where even the requirements have been discovered and refined progressively.

Outline:

1. Requirements
2. Requirement analysis
3. Problem analysis
4. Evaluation of the abstraction gap

6.1 Requirements

`scafi` has no explicitly stated requirements, and many requirements are also not well-defined or immutable. However, as engineering is nothing without requirements, I try to make a subset of them explicit, so that analysis has material to work on and evaluation can be carried out against something.

6.1.1 Field calculus

1. `scafi` must provide a language for field calculus (`scafi` DSL) that allows for the specification of aggregate computations.
   (a) That language must be typed and embedded within Scala – i.e., it must be an internal DSL (and not an external DSL).
   (b) The DSL should be concise, easy-to-use, and modular.
   (c) The DSL must be *complete* – i.e., it must expose all the basic field-calculus constructs.
   (d) The DSL should support the higher-order version of field calculus.
2. `scafi` must provide a Virtual Machine (VM) for the `scafi` DSL.
   (a) The semantics of the `scafi` DSL must be *correct* – i.e., it must implement the Field Calculus semantics in a proper way, without any errors.
   (b) The `scafi` VM should be *reasonably performant*.
3. The `scafi` VM must be *tested* – i.e., it is required to provide:
   (a) *unit tests*, at least for the most critical parts of the implementation;
   (b) *functional tests*, ensuring that both individual constructs and aggregate programs work as expected.
4. `scafi` must also come with a *basic simulator* that allows for aggregate computations to be executed and controlled locally.

6.1.2 Aggregate programming platform

1. `scafi` must provide a platform which supports both the definition and the execution of *distributed systems* that implement aggregate computing applications.
2. Due to the variety of scenarios that aggregate computing potentially targets, the platform should be quite *general*, providing *flexibility* for what concerns system design, deployment, and execution.

3. The platform should come with some *pre-defined configurations* to unburden the users from the hassle of defining setups for the most common scenarios.
4. The platform should provide support for *code mobility*, i.e., the ability of shipping code from a node to another.
   (a) Code mobility should be supported in a user-transparent way, at the infrastructure-level.
5. The platform should support a form of spatial computing by providing a set of spatial abstractions.

6.2 Requirement analysis

6.2.1 Field calculus – DSL and VM

A (programming) language consists of syntax and semantics. Moreover, the execution of programs written in a language requires a virtual machine implementing the semantics by mapping program elements to some underlying platform.

`scafi` must provide an internal domain-specific language for the field calculus and a virtual machine for the execution of DSL programs. The platform underlying the `scafi` VM is represented by Scala itself. Another consequence of having Scala as the host language, is that the `scafi` DSL somehow inherits the typing from Scala.

The field calculus constructs have been described in Section 4.2.1 together with their operational semantics in Section 4.2.3. This defines *what* has to be provided to `scafi` users.

The requirement of *completeness* for the DSL states that all the basic constructs of the field calculus must be provided, namely:

- built-in operator call,
- function definition and function call,
- interaction (`nbr`),
- time evolution (`rep`),

- domain restriction (`if`).

Moreover, the field calculus version to be implemented should be the higher-order one (see Section 4.2.2). Note that, in such a case, the `if` construct would not be primitive.

6.2.2 Aggregate programming systems

Let's analyse a general aggregate computing system by following a top-down approach, with the goal of exploring the conceptual space.

By a *structural* point of view, the system consists of a network of interacting *devices* (or nodes), immersed in some kind of *environment.* Its *behavior* is given by the global *aggregate computation* running over the network of devices. For what concerns the dimension of *interaction* the system *boundary* may be closed or open (which is not an on-off property, but a degree of closure); while in the former case the system does not interact with its environment, in the latter case it may allow some information to flow inside-out or outside-in (porosity).

Some entities are emerging from this prose:

- Network
- Device
- Global aggregate computation
- Environment
- System boundary

Now let's focus on a single device:

- Structure – The device may have sensors and actuators.
- Behavior – The device runs its (current) local aggregate computation by rounds, with a certain (possibly varying) frequency.
- Interaction
  - The device broadcasts its state to its neighbourhood and is suitable to receive, in turn, messages from its neighbours.
  - The device can sense and act on its surrounding environment.

Note that such a modelling is conceptual and may be implemented in different ways to accomodate real-world scenarios: for example, the device may not run its local computation itself, for it may delegate its computation to (some part of) the system. We may say that a device essentially represents, at a given time instant, solely a context for the execution of a local-view aggregate computation.

The concepts arising from such a zooming on the device include:

- Sensor, sensing
- Actuator, acting
- (Surrounding) environment
- Broadcast, message
- Neighbourhood, neighbour
- Local aggregate computation
- Round, frequency of operation

Field calculus The aggregate programming approach in `scafi` is founded on the field calculus. This reduces the conceptual and design spaces; nevertheless, generalisations are welcome.

Device firing Typically, the devices of a network undergo computation in asynchronous rounds. Actually, this is just a general problem of scheduling and different strategies may be chosen.

Communication Communication can be conceptually represented as a message broadcast from a device to the devices belonging to its neighbourhood.

Neighbours discovery A device does not necessarily need to know its neighbours, for example because the broadcast mechanism might rely on another entity.

However, if it is the case, how can a device know its neighbourhood? In general, such information can be obtained in three ways:

1. The information is innate (genetic) or internally inferred.

2. The information comes from the outside, in two ways:
   (a) (Push-based) The device is externally given the information by somebody.
   (b) (Pull-based) The device asks somebody for that information.

The neighbourhood may also change over time, so the knowledge has to be updated, possibly with low latency.

Communication in spatial computing In the spatial computing approach, according to [14], it is not important to know *who* the neighbors are as far as communication is based on location (*"I don't know who you are, but I know where you are"*).

In this case, the problem of communication could be solved via spatial awareness, in two distinct ways:

- the communication mechanism is a spatial broadcast, or
- the device can discover its neighbours via spatial perception (which could be modelled as a sensor providing a set of locations).

Sensors A sensor is a value provider. The value to be provided may be any transformation of analogical or digital quantities, according to a logic that is internal to the sensor itself.

A sensor can work according to two different models:

1. Pull-based: the device asks a value (possibly the most recent one) from the sensor.
2. Push-based: the sensor notifies newly produced values to the device (with some possibly varying frequency that may be independent of the actual source sampling).

Actuators An actuator is any mechanism performing side-effects based on the state of the device in which it is deployed.

6.3 Problem analysis

6.3.1 Field calculus DSL: embedding within Scala

Implementing the field calculus language as an internal DSL means that it is constrained by the host language, which is Scala (by requirement).

Scala is an object-oriented language with extensive functional-programming support. Note that, though Scala does provide first-class functions at the language level, these are ultimately implemented as method calls on objects. Scala is also well-known for its rich type system. These characteristics and the expressive power that has been investigated in Part I constitute promising premises for the endeavour of embedding the field calculus into Scala.

In particular, two prominent issues can be glimpsed:

1. *Implementation of field calculus constructs as method calls* – How can operators such as `rep` or `nbr` be implemented as method calls?
2. *Integration with the Scala type system* – How can we conciliate the Scala type system so as to support the manipulation of computational fields?

From operational semantics to method call Let's consider a Protelis program for the hop-gradient:

```
rep(hops <- Infinity){
  mux(source) { 0 }
  else { 1 + minHood(nbr{hops}) }
}
```

The same program might be represented in Scala as follows:

```
rep(Double.PositiveInfinity){
   hops => { mux(source) { 0.0 } { 1 + minHood(nbr{ hops }) } }
}
```

where the `rep` body has been represented as a 1-ary function from the current state to the new state. Now, let's suppose that `rep`, `mux`, `minHood`, and `nbr` are method calls. The above syntax is valid because Scala allows methods to be defined as accepting multiple parameter lists, which can be enclosed by parenthesis or braces at the call site.

At this point, two key questions arise:

1. How would these methods be implemented internally?
2. How should the evaluation of the program proceed?

For what regards the second question, let's consider the `IF` construct for domain restriction (capitalised as `if` is a reserved keyword in Scala):

```
IF(cond){ then_body } { else_body }
```

Depending on whether the condition evaluates to true or false, either `then_body` or `else_body` must be evaluated: this means that the evaluation of these two expressions must be delayed. For such a need, Scala supports a convenient syntax – call-by name parameters – which spares programmers from wrapping expressions into functions (which would result in a syntax a bit clumsy).

Integration with the Scala type system Here, the goal is to "reuse" the Scala type system to retain the advantages of static type checking for the aggregate programs expressed in `scafi`. Therefore, the methods implementing the field calculus constructs should be generic. For example, a `rep` expression may return an `Int` or a `String`; we also know that the type of the initial value and the type of the state-transforming function must be coherent.

Another issue concerns the `nbr` construct which, according to its denotational semantics, would return a field of fields, which in turn would appear as a field of local values in the context of a certain executing device. The problem is that the introduction of a type `Field[T]` would require the same type to implement all the operators supported by its element type `T`, with a point-wise evaluation semantics, or to lift existing types to work with fields:

```
minHood(1 + nbr { sense("") }) }
```

Before a `*hood` operation is used to turn a field to a local value, the operations on the result of `nbr` would be field operations.

6.4 Abstraction gap

6.4.1 Distributed Platform

Building a distributed platform from scratch is hard. Many issues have to be considered and many pitfalls are to be avoided. The big themes to be faced include but are not limited to:

- Communication – It should be supported by reliable, high-level, flexible mechanisms.
- Fault-tolerance – Systems should be resilient (i.e., by reducing the impact of failure) and able to recovery from failure. That is, they should deal with partial failure.
- Naming – It may be useful to associate names with computational entities and resolve. names to addresses. However, how to deal with openness, mobility, failure?
- Concurrency – Concurrent activities must be properly coordinated.

As `scafi` is Scala-based, this need of raising the abstraction level and reusing existing, robust solutions leads to consider the Akka actor framework[1] as the basis for the development of the aggregate distributed platform.

[1] http://akka.io/

Chapter 7

Design

The design phase is responsible of a first elaboration of the *what* into the *how*. At this point, the information produced during analysis has to be used to envisage a solution, in order to reduce the gap between concepts and implementation.

This chapter is intended to describe the main elements of the design of `scafi`, which consists of two parts: the core library implementing the field calculus and the actor-based platform addressing the development of distributed "aggregate applications."

Outline:

- Architecture of the core library
- Architecture of the distributed platform
- Distributed platform design
- API design

It must also be noted that the initial design of the interpreter at the core of the proposed framework is authored by prof. Viroli.

7.1 Design architecture

7.1.1 `scafi` DSL and VM

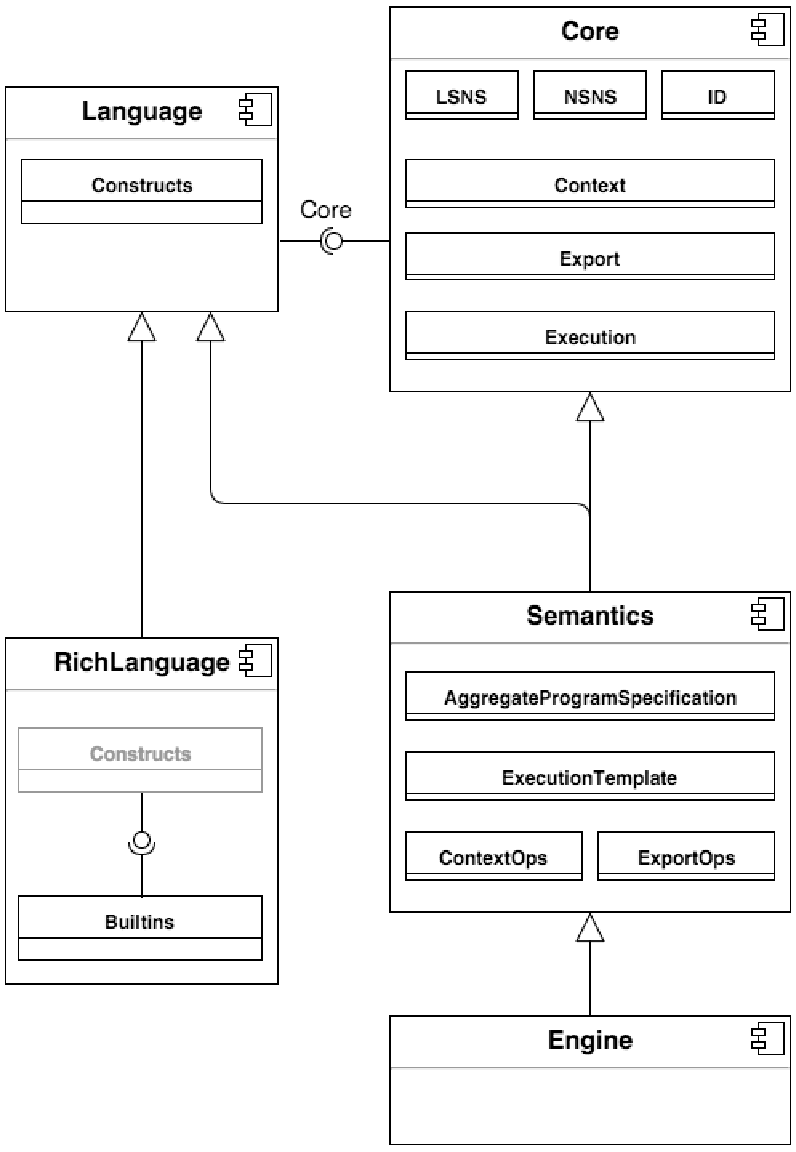

Figure 7.1: Design architecture for the language and virtual machine.

Figure 7.1 represents the key components of the `scafi` core library. The `Core` component defines the basic abstractions and architectural elements, which are to be refined by child components. The `Language` component, based on the abstractions defined in `Core`, defines the fundamental `Constructs` of the DSL. Upon these primitives, derivate operators (`Builtins`) can be provided to make the language more expressive (`RichLanguage`). The `Semantics` component extends the (syntactical and structural part of) `Language`, refines core abstractions and provides a semantics for the language `Constructs`, which is then made executable by `Engine`.

7.1.2 Spatial abstraction

The idea of the architecture depicted by Figure 7.2 is to model a *space* and the notions of *neighbouring relation* and *situation* in such a space by means of a `SpatialAbstraction` component. A space is characterised by a position type `P`, whereas the situation of elements of type `E` in the space is achieved via a spatial container `Space[E]`, which also must also define a notion of neighbourhood.

Now, it may be useful to consider a graph (ad-hoc network) as a particular case of spatial abstraction where each node of the network is located at a different position in the space and the neighbouring relation is a function from a position to an arbitrary set of positions.

Another fundamental type of spatial abstraction is given by metric spaces. The `MetricSpatialAbstraction` component introduces a distance type `D` and a way of expressing how distances between positions are calculated (`DistanceStrategy`). Finally, a `BasicSpatialAbstraction` is defined as a 3-dimensional space with the Euclidean metric.

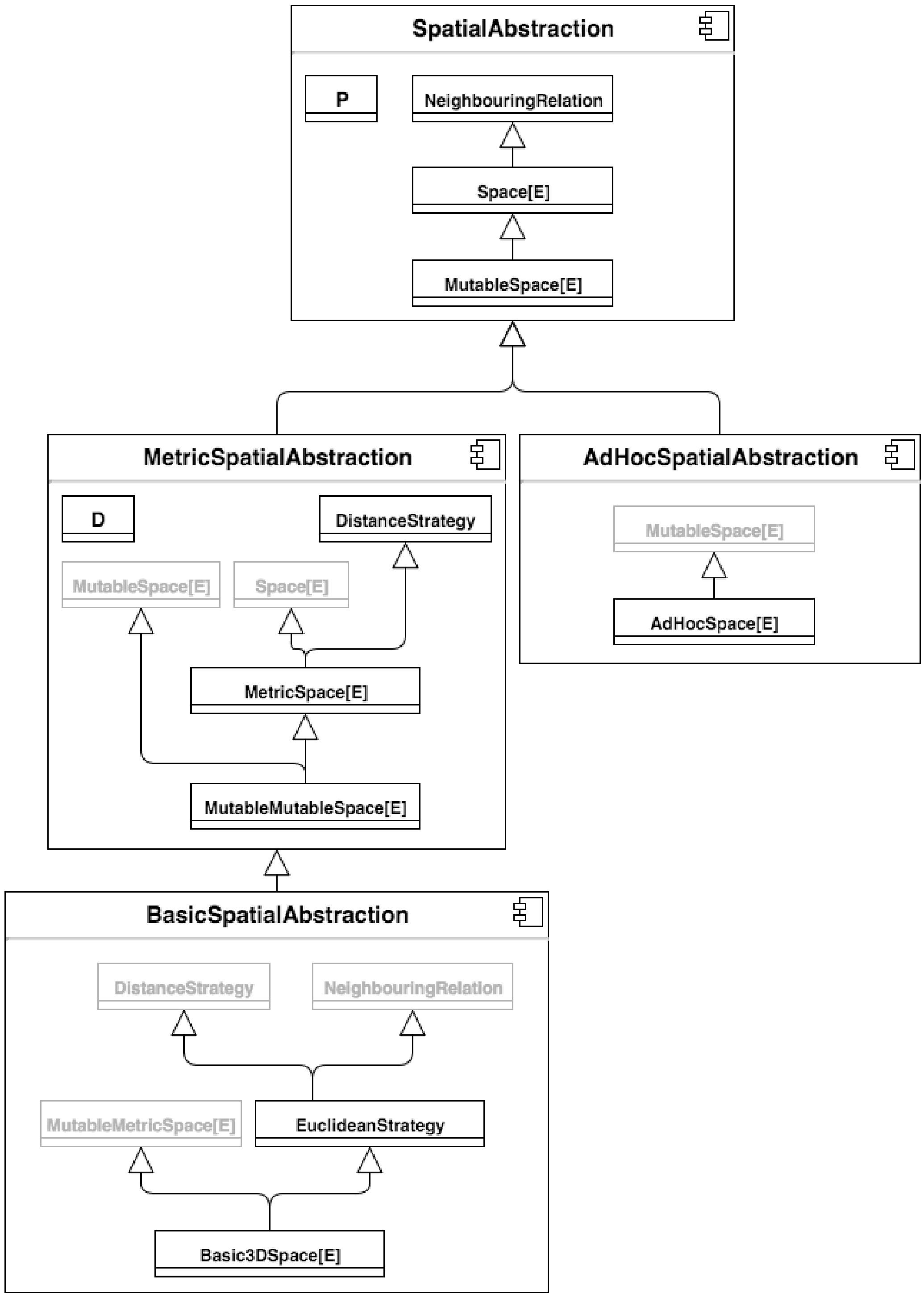

Figure 7.2: Design architecture for the spatial abstraction.

7.1.3 Simulator

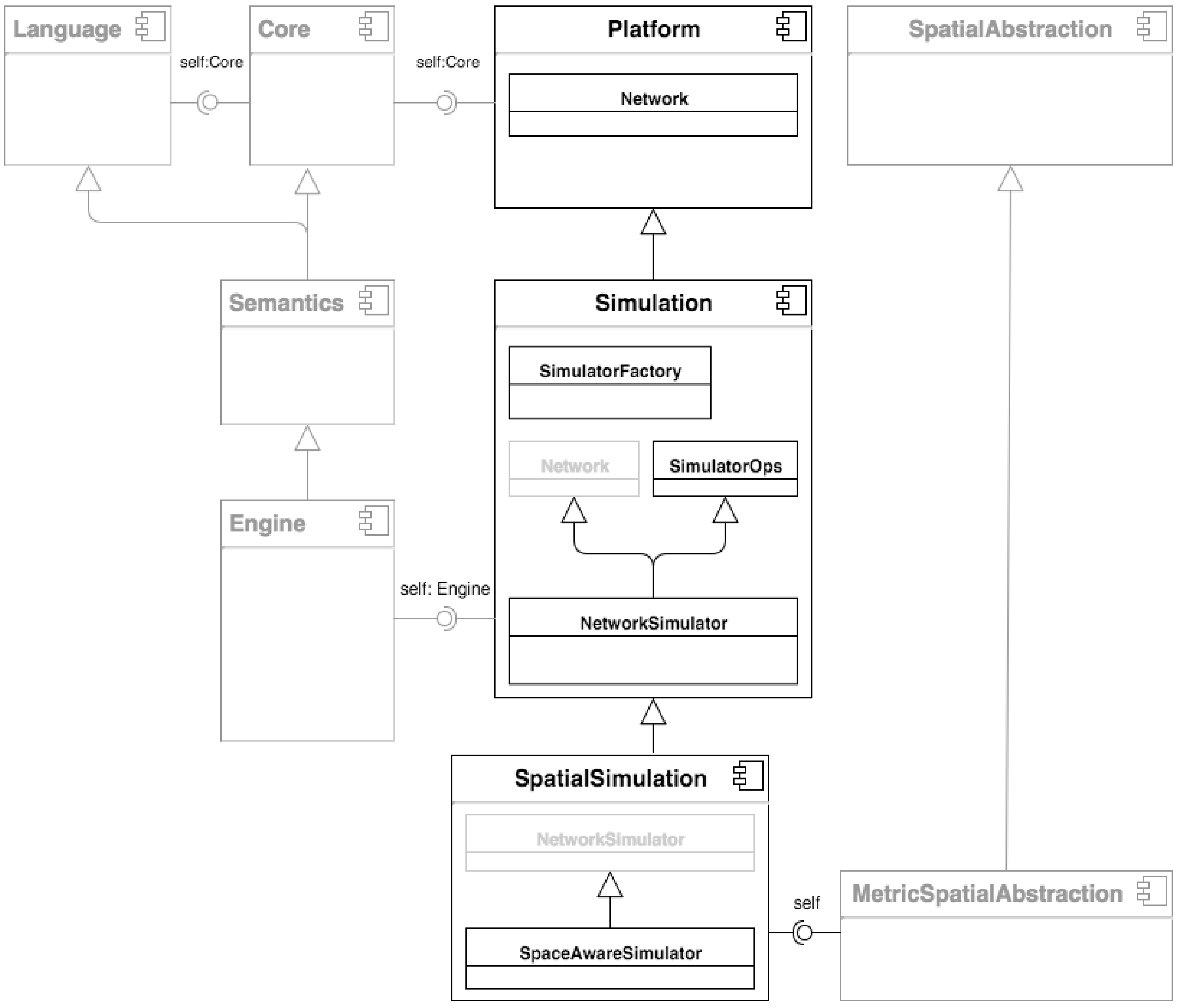

Figure 7.3: Design architecture for what concerns simulation.

A structural representation of how the simulators relate with the core architecture is given by Figure 7.3. A `Simulation` extends a `Platform`, as it should provide a platform-view of the running system, and requires the interface provided by `Engine` in order to put aggregate programs into execution. A simulator must also provide the means (`SimulatorFactory`) for building the systems to be simulated (which, in this case, are networks of devices). In particular, two kinds of systems are supported: ad-hoc networks (graphs) and spatial networks (`SpatialSimulation`).

7.1.4 Distributed platform

The distributed platform, as other parts of `scafi`, is defined by means of progressive refinements and extensions of more basic components.

At this level, the components are split into multiple subcomponents. In fact, building a distributed platform requires to handle multiple concerns and to manage significant complexity.

By the point of view of design, the actor-based platform is a specific kind of distributed platform. This level of indirection is intended to favor reuse and leave the opportunity to create additional platform designs. In turn, the actor-based platform splits into a dichotomy:

1. actor-based, peer-to-peer platform (decentralised),
2. actor-based, server-based platform (with a centralisation point for system coordination).

Then, the server-based platform has been specialised into a `SpatialPlatform` which depends on a `MetricSpatialAbstraction` in order to provide a space-aware device management service.

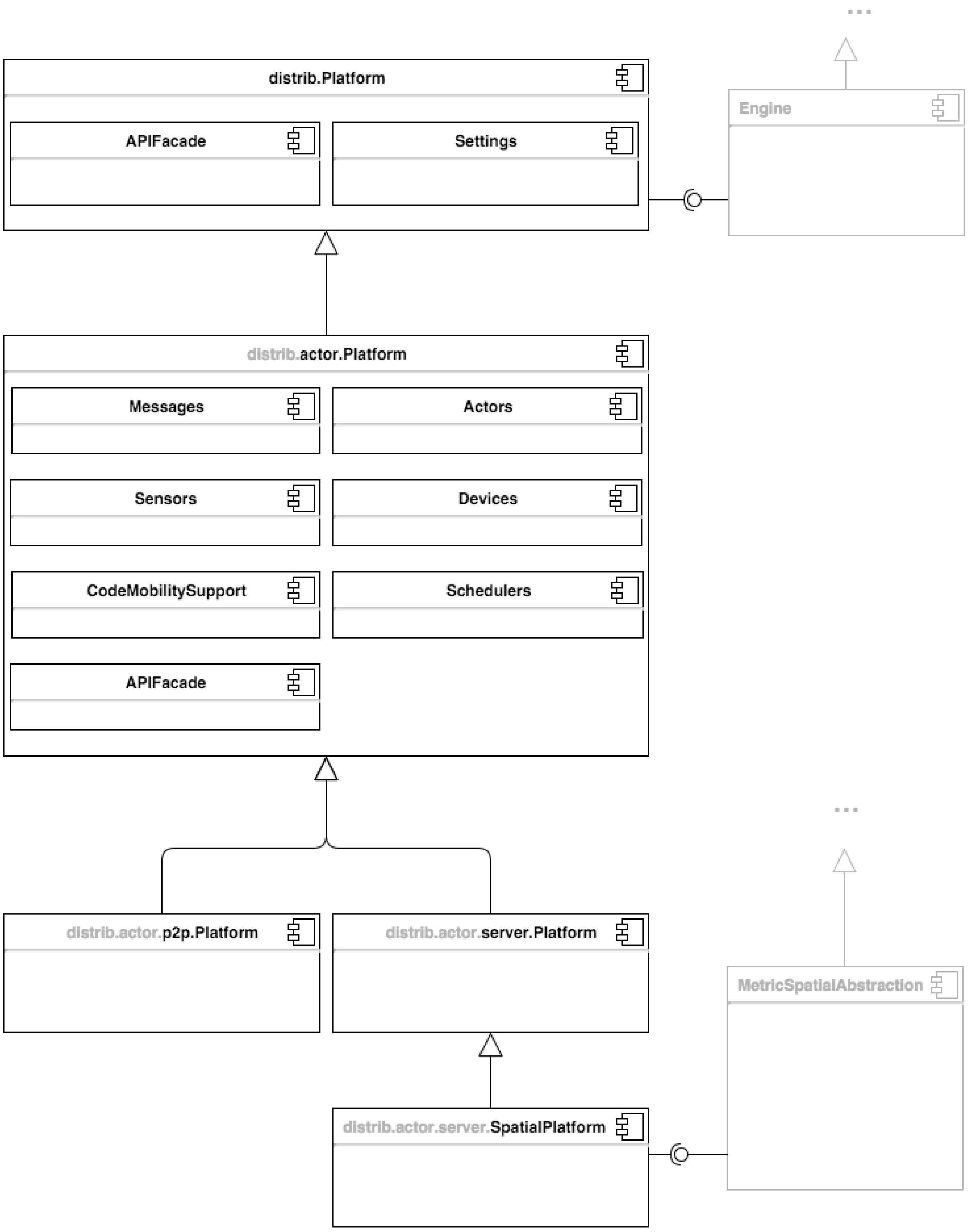

Figure 7.4: Design architecture for the actor-based platform.

7.2 Actor platform design

7.2.1 System design

A simplified view of the elements participating in an actor-based aggregate computing application is provided by Figure 7.5.

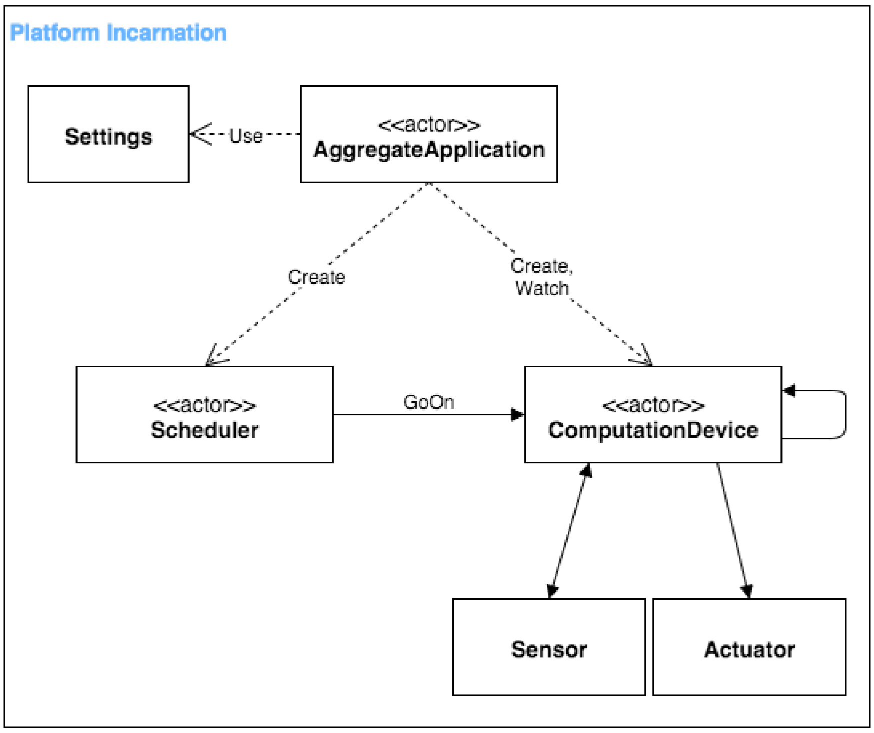

Figure 7.5: Structure diagram of the main entities of an aggregate computing system.

Essentially, the key types of elements are:

- `AggregateApplication` – It represents, in any subsystem, a particular aggregate application, as specified by some `Settings`. Also, it works as a supervisor for all the other application-specific actors.
- `Scheduler` – Optionally, a scheduler may be used to centralise system execution at a system- or subsystem-level.

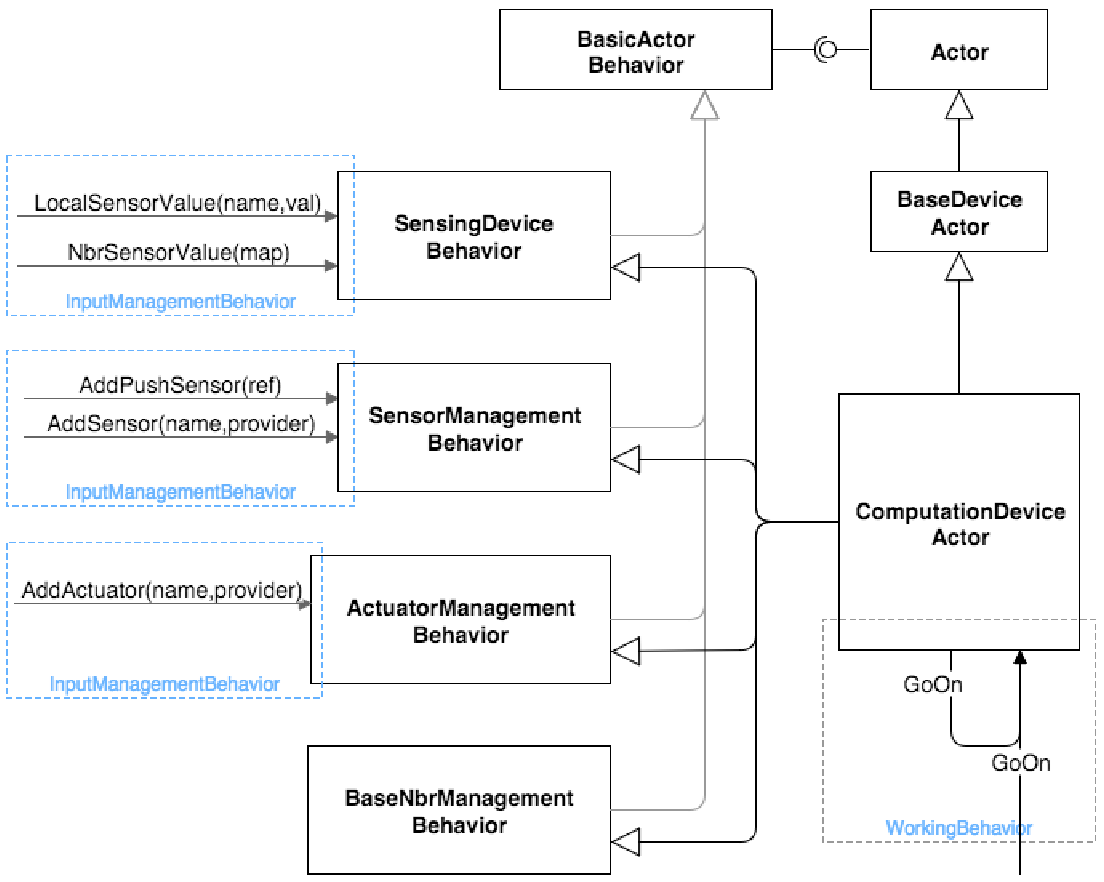

Figure 7.6: Structure and interface of device actors.

- `ComputationDevice` – It is a device which is able to carry out some local computation. It communicates with other devices and interacts with `Sensors` and `Actuators` (which may be actors as well or not).

Also, note how all these entities are specific to a particular platform incarnation.

Devices Figure 7.6 shows how devices are modelled. A first key distinction is between actors and behaviors. In fact, one design goal is to split a big, articulated behavior into many small, reusable, composable behaviors.

The convention in the diagram is to express message-based interfaces by means of incoming and outcoming messages which are represented as arrows with a filled arrowhead.

By a conceptual point of view, a device must, at minimum, manage its sensors and actuators. Then, in the context of aggregate programming, a device must also interact with its neighbours (`BaseNbrManagementBehavior`); such interaction has not been detailed yet, as it may be somehow different in the p2p and server-based cases. Also, a computation device executes some program with a certain frequency (here represented by a tick message called `GoOn`).

7.2.2 Server-based actor platform

This particular kind of platform follows the client/server architectural style. The devices are clients of a central server that owns the information about the topology and is responsable for the propagation of the states of the devices.

Figure 7.7 statically describes the message interfaces of device and server:

- Each device registers itself with the server at startup (`Registration`).
- After a computation, a device communicates its newly computed state to the server (`Export`).
- Each device asks the server (`GetNeighbourhoodExports`) for the most recent states of its neighbours (`NeighbourhoodExports`), with some frequency.

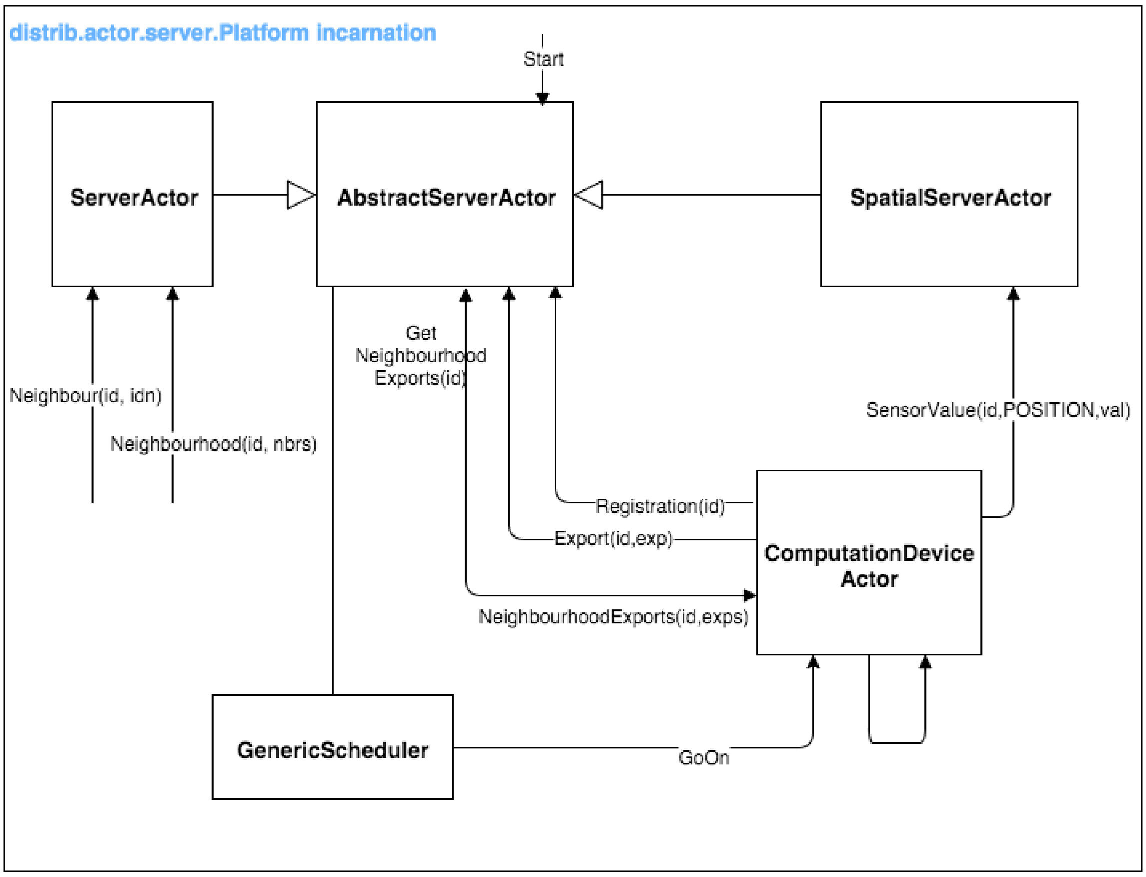

Figure 7.7: Key elements and relationships in a server-based actor platform.

7.2.3 Peer-to-peer actor platform

This platform follows a peer-to-peer architectural style. Each device, at the end of each computation, propagates its newly computed state (`MsgExport`) directly to all its neighbours. Here, the critical point concerns how a device gets acquainted with its neighbours; for now, let's just suppose that a device is able to receive information about a neighbour (`NbrInfo`).

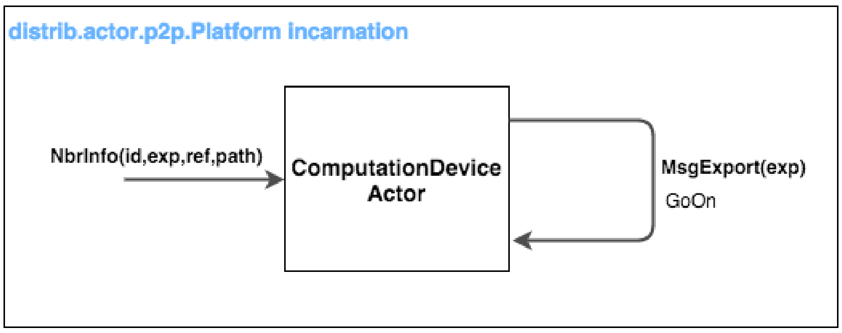

Figure 7.8: Key elements and relationships in a peer-to-peer actor platform.

7.2.4 API design

The general platform API that can be used to create distributed applications is visualised in Figure 7.9.

A `PlatformConfigurator` works as the entry point for the process of construction of a system. It is used to setup a `PlatformFacade`, which in turn allows to create one or more aggregate applications. The user can control an aggregate application via the corresponding `SystemFacade`, which supports operations such as the creation of devices, the specification of neighbouring relations, or the actual start of the system (e.g., by activating a scheduler). Then, the control interface for devices is given by their `DeviceManagers`, which can be used to attach sensors or actuators.

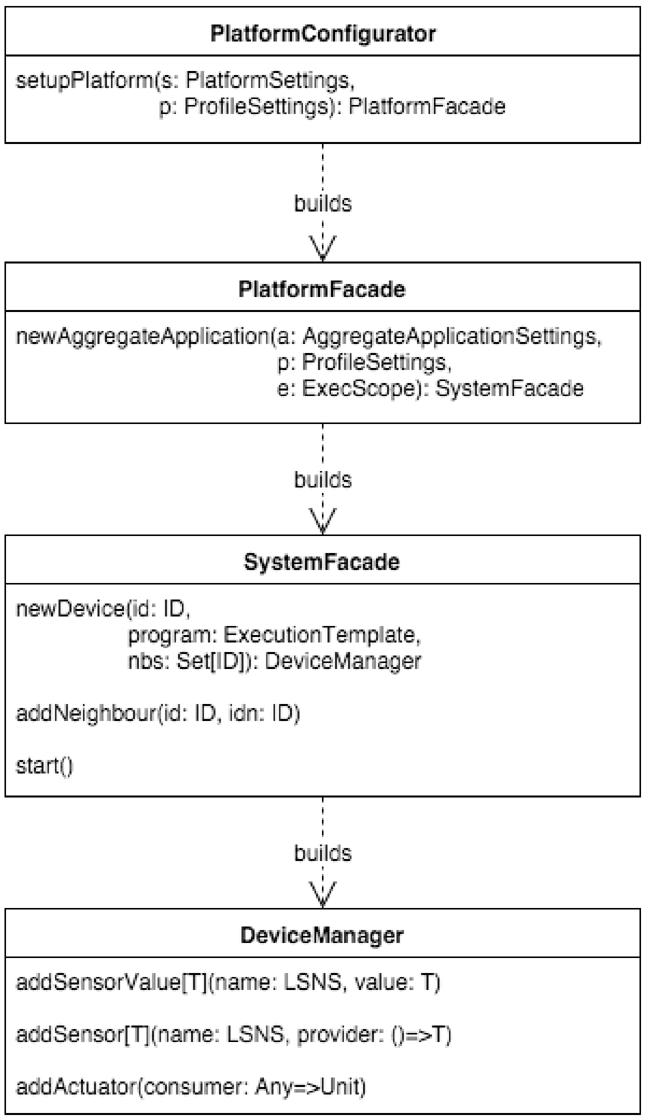
PlatformConfigurator
setupPlatform(s: PlatformSettings,
p: ProfileSettings): PlatformFacade
builds
PlatformFacade
newAggregateApplication(a: AggregateApplicationSettings,
p: ProfileSettings,
e: ExecScope): SystemFacade
builds
SystemFacade
newDevice(id: ID,
program: ExecutionTemplate,
nbs: Set[ID]): DeviceManager
addNeighbour(id: ID, idn: ID)
start()
builds
DeviceManager
addSensorValue[T](name: LSNS, value: T)
addSensor[T](name: LSNS, provider: ()=>T)
addActuator(consumer: Any=>Unit)

Figure 7.9: API façades for the distributed platform.

Chapter 8

Implementation and testing

Based on the architectural invariants and the design models delineated in Chapter 7, this chapter provides a more in-depth tour of the implementation of the `scafi` DSL and platform. As is the source code the ultimate representative of an implementation, here the focus will be on the key insights and the rationale.

Outline:

- Project organisation
- Architecture implementation
- Field calculus implementation
- Distributed platform implementation
- Testing

8.1 Project organisation

The project has been split into multiple subprojects for better management:

- `scafi-core` – implements the core functionality, namely, the field calculus language and virtual machine, a basic simulator, and spatial abstractions.
- `scafi-tests` – includes unit and acceptance tests for `scafi-core`
- `scafi-distributed` – implements the distributed platform.
- `scafi-demos` – provides examples and demonstration programs.

- `scafi-docs` – includes the latex files for the tutorial and reference manual.

Each subproject is versioned and has its own dependencies and build configuration.

More technically, the project is versioned with Git and uses sbt (the Scala Build Tool) for project and build automation.

8.2 `scafi` DSL and VM implementation

The implementation of the component-based architecture presented in Chapter 7 has been implemented in Scala using a few features and techniques described in Chapter 1 and 2.

Briefly, components are represented by traits defining (abstract) types, traits, classes, objects and so on. This allows to create families of mutually recursive types (cf., family polymorphism in Section 2.5) that can be refined incrementally.

For what concerns the implementation of the `scafi` DSL and virtual machine, Figure 8.1 highlights the key classes and traits:

- The `Constructs` trait exposes the field calculus primitives as methods.
- The `Context` trait is used to represent an execution context for an aggregate computation round.
- An `Export` is a data structure for the result of a (local) aggregate computation; a typical operation on an export is the extraction of the root value.
- `ExecutionTemplate` is where the semantics is actually implemented. Internally, the state of the computation is traced by a `Status` object, which works as an immutable stack and keeps track of the branches in the computation tree.

8.2.1 Operational semantics of the field calculus

The operational semantics of the field calculus (briefly described in Section 4.2.3) has been implemented in the class `ExecutionTemplate` within the component `Semantics`.

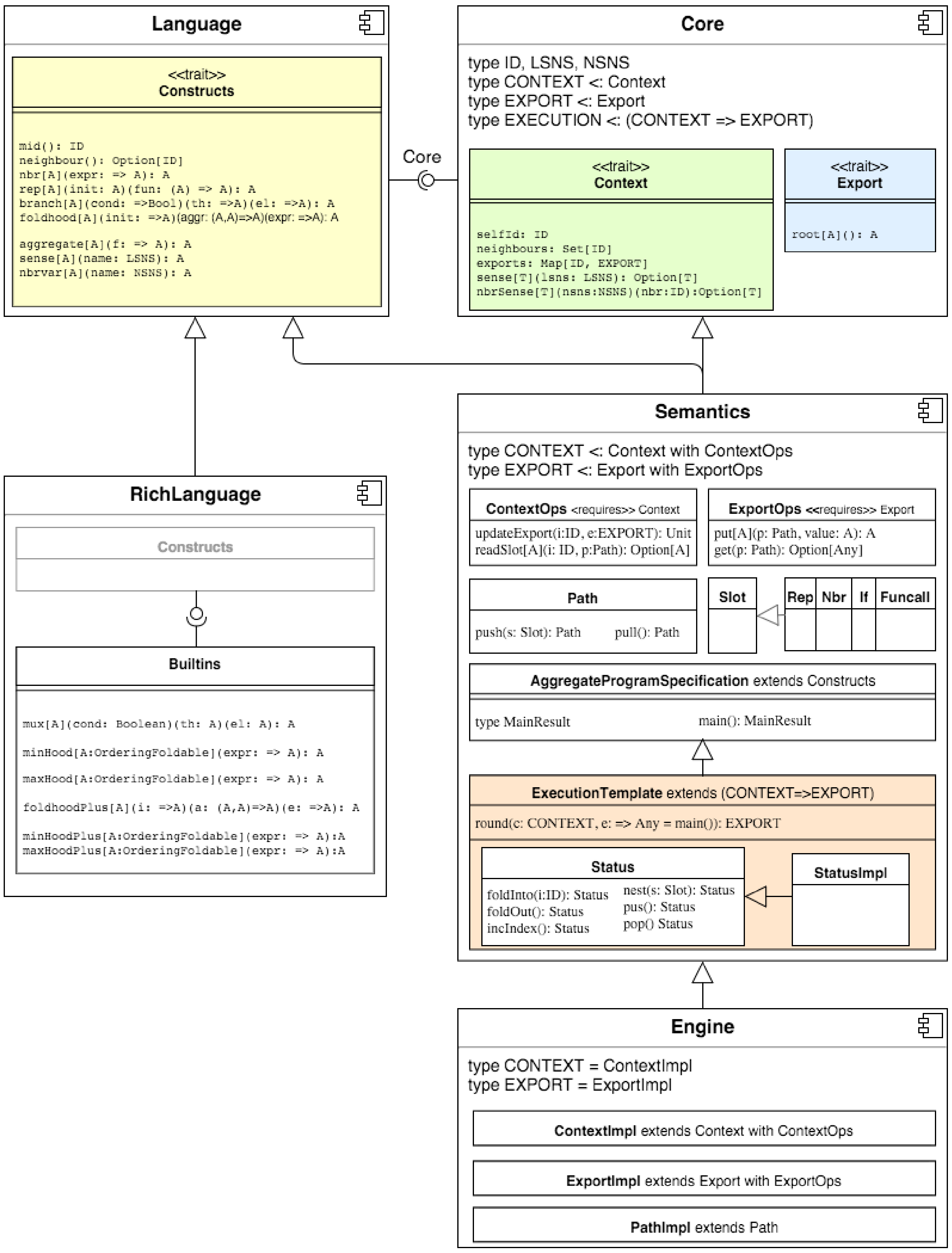
Language
<<trait>>
Constructs
mid(): ID
neighbour(): Option[ID]
nbr[A](expr: => A): A
rep[A](init: A)(fun: (A) => A): A
branch[A](cond: =>Bool)(th: =>A)(el: =>A): A
foldhood[A](init: =>A)(aggr: (A,A)=>A)(expr: =>A): A
aggregate[A](f: => A): A
sense[A](name: LSNS): A
nbrvar[A](name: NSNS): A
Core
Core
type ID, LSNS, NSNS
type CONTEXT <: Context
type EXPORT <: Export
type EXECUTION <: (CONTEXT => EXPORT)
<<trait>>
Context
selfId: ID
neighbours: Set[ID]
exports: Map[ID, EXPORT]
sense[T](lsns: LSNS): Option[T]
nbrSense[T](nsns:NSNS)(nbr:ID):Option[T]
<<trait>>
Export
root[A](): A
RichLanguage
Constructs
Builtins
mux[A](cond: Boolean)(th: A)(el: A): A
minHood[A:OrderingFoldable](expr: => A): A
maxHood[A:OrderingFoldable](expr: => A): A
foldhoodPlus[A](i: =>A)(a: (A,A)=>A)(e: =>A): A
minHoodPlus[A:OrderingFoldable](expr: => A):A
maxHoodPlus[A:OrderingFoldable](expr: => A):A
Semantics
type CONTEXT <: Context with ContextOps
type EXPORT <: Export with ExportOps
ContextOps <<requires>> Context
updateExport(i:ID, e:EXPORT): Unit
readSlot[A](i: ID, p:Path): Option[A]
ExportOps <<requires>> Export
put[A](p: Path, value: A): A
get(p: Path): Option[Any]
Path
push(s: Slot): Path
pull(): Path
Slot
Rep
Nbr
If
Funcall
AggregateProgramSpecification extends Constructs
type MainResult
main(): MainResult
ExecutionTemplate extends (CONTEXT=>EXPORT)
round(c: CONTEXT, e: => Any = main()): EXPORT
Status
foldInto(i:ID): Status
foldOut(): Status
incIndex(): Status
nest(s: Slot): Status
pus(): Status
pop() Status
StatusImpl
Engine
type CONTEXT = ContextImpl
type EXPORT = ExportImpl
ContextImpl extends Context with ContextOps
ExportImpl extends Export with ExportOps
PathImpl extends Path

Figure 8.1: Structure diagram for the DSL and VM.

The execution of an aggregate program begins in method `round`, which accepts a context (instance of type `CONTEXT`), the expression to be evaluated, and returns an export (instance of type `EXPORT`).

The export is equivalent to the notion of *value tree* that was introduced when presenting the semantics for the HOFC, i.e., a tree-like data structure with a root value and an ordered-list of subtrees. The current implementation is as follows:

- The export is implemented (`ExportImpl`) as a map from `Path`s to `Any` values.
- A `Path` is a data structure that keeps track of the nodes of the value tree, implemented (`PathImpl`) as a list of `Slot`s.
- A `Slot` is a generic element that can be a node in the value tree.
- There are four concrete slots: `Nbr, Rep, If, FunCall`. All these case classes accept an index because there may be multiple uses of the same construct at the same level of the tree.

During the computation, the interpreter must keep track of the current position in the value tree so as to implement structural alignment. For the purpose, a stack-like data structure has been defined (trait `Status`); more precisely, it is a stack of triples of the form `(p:Path, index:Int, nbr: Option[ID])`, where the 3-rd component is used for the `foldhood`, which will be described in a while.

The execution context must provide access to the following information:

- The device (identifier) that is performing the computation.
- The set of neighbours and their respective exports, i.e., the *value tree environment.*
- Local sensors.
- Environmental sensors, which return fields mapping the device itself and its neighbours to values.

Specification and execution of aggregate programs As previously said, the field calculus logics is implemented by class `ExecutionTemplate`:

```
trait AggregateProgramSpecification extends Constructs {
    type MainResult
    def main(): MainResult
}

trait ExecutionTemplate extends (CONTEXT=>EXPORT) with
    AggregateProgramSpecification {
    // ... impl of FC operational semantics ...
}
```

In order to use the field calculus operators, expressions must refer to the methods declared in `Constructs` and, ultimately, these method calls will resolve to the implementations defined in `ExecutionTemplate`, which essentially works as an interpreter. This means that any executable aggregate program is necessarily an instance of `ExecutionTemplate`. However, it is possible to separate the specification of a program from its executable embodiment:

```
trait MyProgram extends AggregateProgramSpecification with Builtins {
  type MainResult = Double

  def hopGradient(source: Boolean): Double = {
    rep(Double.PositiveInfinity){
      hops => { mux(source) { 0.0 } { 1+minHood(nbr{ hops }) } }
    }
  }

  def main() = hopGradient(sense("source"))
}

object MyExecutableProgram extends ExecutionTemplate with MyProgram

val ctx: Context = _ // some context
val result = MyExecutableProgram(ctx)
```

Note how the object `MyExecutableProgram` is invoked in function notation with the context argument.

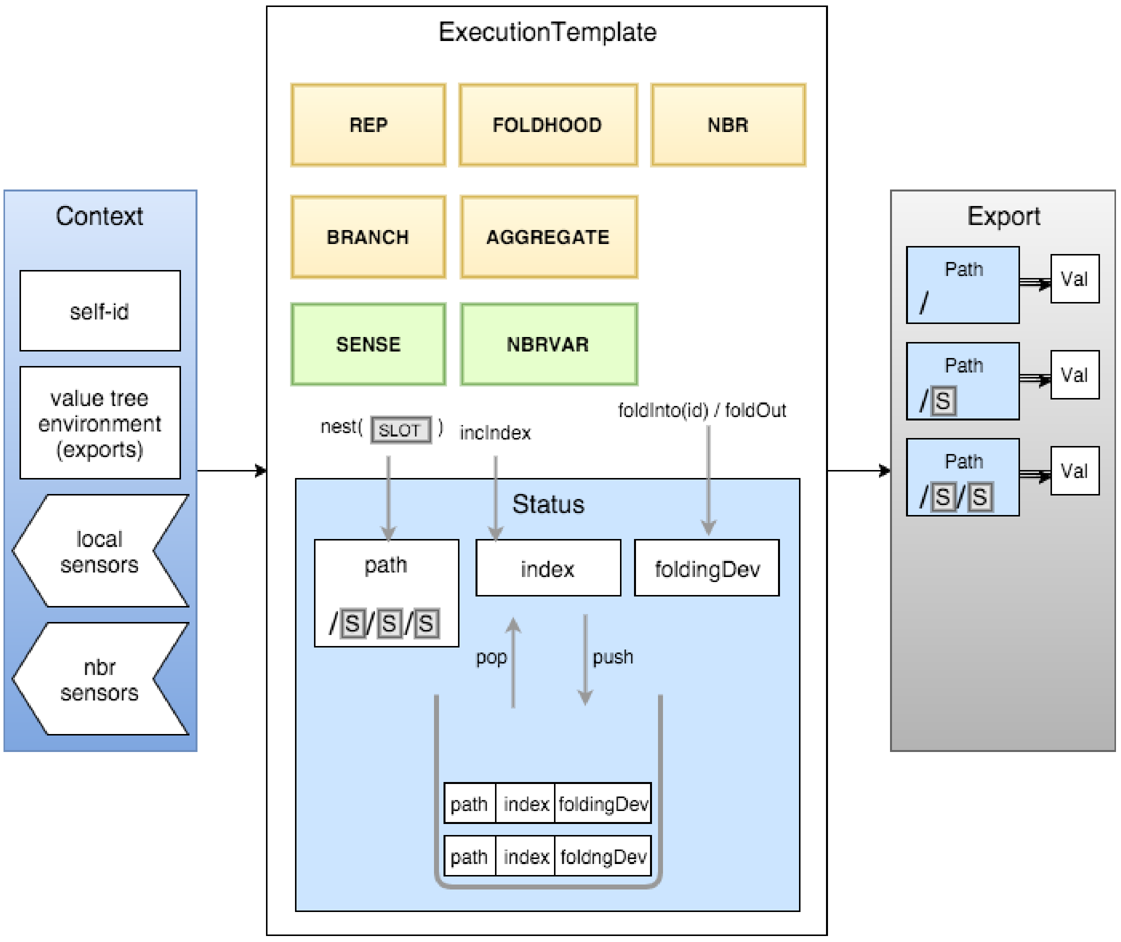

Figure 8.2: Informal diagram of the elements involved in the execution of aggregate programs in `scafi`.

The beginning of an aggregate computation The `round` method is the entry point for the evaluation of aggregate expressions:

```
trait ExecutionTemplate extends (CONTEXT=>EXPORT) with
    AggregateProgramSpecification {
    private var ctx: CONTEXT = _
    private var exp: EXPORT = _
    private var status: Status = _

    def apply(c: CONTEXT): EXPORT = {
      round(c,main())
    }

    def round(c: CONTEXT, e: => Any = main()): EXPORT = {
        ctx = c
        exp = factory.emptyExport
        status = Status()
        exp.put(factory.emptyPath, e) // Reference to 'e' triggers the
    evaluation.
        this.exp
    }
```

Given a `CONTEXT` instance and an expression *e* to evaluate (note that *e* is a call-by-name argument, i.e., it is passed *unevaluated*), the method `round` returns one export value. The root of this export will be equal to the result of the evaluation of the expression itself. Potential subtrees may be added to the export as long as the evaluation of *e* proceeds.

Before starting the evaluation of the program expression, the machine state is initialised: the current execution context is set to the input context, the current export is initialised to an empty export, and the auxiliary status object is constructed anew.

Construct: `rep` The `rep` construct accepts an initial value (of a generic type `A`) and a state-transforming function (endofunction). The implementation code for the operator follows:

```
def rep[A](init: A)(fun: (A) => A): A = {
   ensure(status.neighbour.isEmpty, "can't nest rep into fold")

   nest(Rep[A](status.index)) {                                   // 1.
     val in = ctx.readSlot(ctx.selfId, status.path).getOrElse(init)   // 2.
     fun(in)                                                      // 3.
   }
}
```

The evaluation consists in three steps:

1. The `nest` function is called with a slot (a `Rep` at the current index, which initially is zero) and an unevaluated block of code. Effectively, `nest` wraps the expression around state-management code in order to progressively "navigate", back and forth, the value tree.

```
def nest[A](slot: Slot)(expr: => A): A = {
    try {
      status = status.push().nest(slot)  // i) Prepare nested call
      exp.put(status.path, expr)         // ii) Function return value is
    result of expr
    } finally {
      status = status.pop().incIndex();  // iii) Restore the status
    }
}
```

First, the current status triple (`path,index,nbrOpt`) is pushed on the stack, and a new status is built by extending the path with `slot`. Secondly, `expr` is evaluated, the result is returned after being used to add an export for the current path. Thirdly, before leaving the method, the original status is restored (the popped triple becomes the current status), and the index is increased to take into account the possibility of having multiple occurrences of the same slot at the same level.

2. The state value for the `rep` is set to either the export of the current device at the current value tree path (if available) or the initial value.

3. The body of the `rep` is invoked with the state value, continuing the descent of this branch of the value tree.

Constructs: `foldhood` and `nbr` The following listing reports the implementation code for these two constructs:

```
def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A = {
    ensure(status.neighbour.isEmpty, "can't nest fold constructs")

    try {
      val v = aligned()
      val res = v.map { i =>
        handling(classOf[OutOfDomainException]) by (_ => init) apply {
          frozen { status = status.foldInto(i); expr }
        }
      }
      res.fold(init)(aggr)
    } finally {
      status = status.foldOut()
      status = status.incIndex()
    }
}

def nbr[A](expr: => A): A = {
    ensure(status.isFolding, "nbr should be nested into fold")
    nest(Nbr[A](status.index)) {
      if (status.neighbour.get == ctx.selfId){
        status = status.foldOut(); expr
      } else {
        ctx.readSlot[A](status.neighbour.get, status.path)
          .getOrElse(throw new OutOfDomainException(
             ctx.selfId, status.neighbour.get, status.path))
      }
    }
}
```

Note that `foldhood` is not one of the original constructs of the field calculus (see Section 4.2.1). In fact, it has been introduced for a technical reason which I am about to explain.

According to natural semantics of the field calculus, the `nbr` construct evaluates to a field of fields, where each device of the system is mapped to a field which in turn maps the device's neighbours to some computational objects. Thus, in the context of a device (local viewpoint), a `nbr` would produce a neighbourhood field. Now, the issue is that as long as such a field is not be condensed to a local value (e.g., by means of `*-hood` operators such as `minhood`), operations are performed on entire fields and this would require to lift any operator to work with fields. However, given the constraints of the Scala type system, the feasibility and effectiveness of such a lifting are questionable.

In `scafi`, the solution to this problem consists in the introduction of a construct, `foldhood`, which is responsible to the evaluation of an `nbr` expression against all the aligned neighbours of the currently executing device.

More in detail, the implementation of `foldhood` works by retrieving the aligned neighbours and mapping them to their local expression `expr` (which will typically include a `nbr` or `nbrvar` expression), unevaluated. Then, the resulting structure if folded (reduced) with the given `aggr`egation function and `init`ial accumulation value. The `aligned` function returns all the neighbours for which the context contains an export at the current value tree node (path), and appends the current device at the end of the list (this is necessary in this implementation because of how `nest` works).

For what concerns the implementation of `nbr`, note that the operations depends on the currently folding device: if it is the device that is performing the aggregate computation, then the body expression for the `nbr` is evaluated; otherwise, this involves a neighbour and the value to be returned is the neighbour's export value at the current path.

Construct: `if` This construct, renamed as `branch` because the `if` keyword is reserved in Scala, implements domain restriction:

```
def branch[A](cond: => Boolean)(th: => A)(el: => A): A = { // 1
    val b = cond                        // 2a
    nest(If[A](status.index, b)){   // 2b
      if (b) th else el             // 3
    }
}
```

The important details are the following ones:

1. The *condition*, *then*, and *else* expressions are passed unevaluated (as *thunks*).
2. The condition is evaluated first, and the result is used for the nesting in the value tree – this means that the devices for which the condition is `true` are in a different domain with respect to those for which the condition turns out to be `false`.
3. Based on the result of the condition, either the `then` or the `else` expression is evaluated, so that domain restriction is respected.

Construct: `aggregate` This construct has been added to support first-class aggregate functions (see Section 4.2.2 about Higher-Order Field Calculus). The key aspect in the implementation is the nesting based on the function identifier which accomplishes domain restriction via alignment:

```
def aggregate[T](f: => T): T = {
    var funId = Thread.currentThread().getStackTrace()(3)

    nest(FunCall[T](status.index, funId)) { f }
}
```

Sensors Access to local sensors and environmental sensors is provided by the execution context:

```
def sense[A](name: LSNS): A =
    ctx.sense[A](name).getOrElse(throw new SensorUnknownException(ctx.selfId,
    name))

def nbrvar[A](name: NSNS): A = {
    val nbr = status.neighbour.get
    ctx.nbrSense(name)(nbr)
      .getOrElse(throw new NbrSensorUnknownException(ctx.selfId, name, nbr))
}
```

Some words of explanation about `nbrvar` are needed. An environmental sensor conceptually returns a field mapping devices to values. As a consequence, it is subject to the same technical issues as the `nbr` operator and the solution is the same: `nbrvar` must be nested in a `foldhood`, so that it is possible to query the sensor for the device of the current “fold”.

8.2.2 Spatial abstraction implementation

As shown in Figure 8.3, any `SpatialAbstraction` has a type `P` for representing positions in the space and a type `SPACE[E]` that abstractly defines a spatial container of elements of type `E`. Basically, a spatial container contains some elements and relates them to positions, thus realising a notion of *situatedness*; moreover, it implements a `NeighbouringRelation` to specify when two positions are considered nearby.

An `AdHocSpatialAbstraction` constrains spatial containers to be ad-hoc spaces, i.e., spaces where elements are in a one-to-one relationship with the positions, and where the notion of neighbourhood is specific to each element.

By contrast, a `MetricSpatialAbstraction` introduces a type `D` for distances and requires spatial containers to implement some `DistanceStrategy` that, given two positions, returns the distance between them (`getDistance` method).

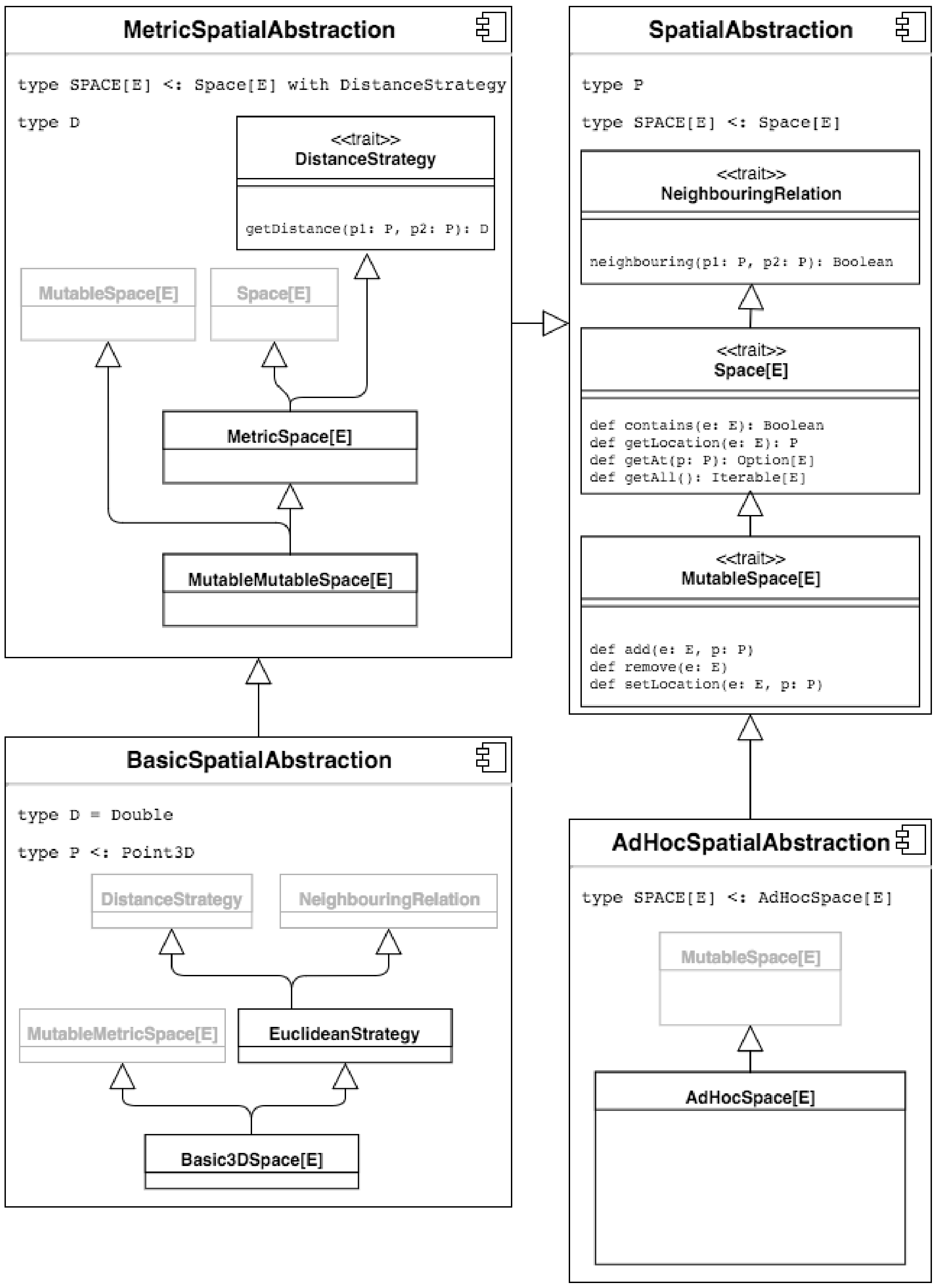

Figure 8.3: Structure diagram for the spatial abstraction.

8.3 Distributed platform implementation

The high-level design architecture for the distributed middleware and a more detailed tour of the design of the actor-based platform are provided in Sections 7.1.4 and 7.2, respectively.

The implementation of the platform components follows the same pattern as in the core library, where family polymorphism is used to define a set of related types that can be refined incrementally in more specialised components. Moreover, for a more effective organisation of code, the self-type feature has been used to split a big component into multiple sub-components located at different source code files.

8.3.1 Actors and reactive behavior

The actor platform has been implemented using the Akka framework. In Akka, actors are defined by extending the `akka.actor.Actor` trait and implementing the `receive` method, of type `Receive=PartialFunction[Any,Unit]`, that associates reactions to incoming messages.

An interesting implication of having (reactive) behaviors expressed by `PartialFunctions` is that they compose. This composability feature has been extensively used to promote separation of concerns. For example, the device behavior related to the management of sensors can be kept separated from the behavior aimed at handling actuators:

```
def SensorManagementBehavior: Receive = {
  case MsgAddPushSensor(ref) => { ref ! MsgAddObserver(self); ref ! GoOn }
  case MsgAddSensor(name, provider) => setLocalSensor(name, provider)
}

def ActuatorManagementBehavior: Receive = {
  case MsgAddActuator(name, consumer) => setActuator(name, consumer)
}

def CompositeBehavior: Receive =
  SensorManagementBehavior
  .orElse(ActuatorManagementBehavior)
```

Moreover, it is also possible to leverage on trait stacking to automatically extend some behavior by mixing in behavior traits:

```
trait BasicActorBehavior { selfActor: Actor =>

  def receive: Receive =
    workingBehavior
      .orElse(inputManagementBehavior)
      .orElse(queryManagementBehavior)
      .orElse(commandManagementBehavior)

  def inputManagementBehavior: Receive = Map.empty
  def queryManagementBehavior: Receive = Map.empty
  def commandManagementBehavior: Receive = Map.empty
  def workingBehavior: Receive = Map.empty
}

trait SensorManagementBehavior extends BasicActorBehavior { selfActor: Actor =>
  def SensorManagementBehavior: Receive = { ... }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior.orElse(SensorManagementBehavior)

  // ...
}

trait ActuatorManagementBehavior extends BasicActorBehavior { selfActor: Actor
    =>
  def ActuatorManagementBehavior: Receive = { ... }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior.orElse(ActuatorManagementBehavior)

  // ...
}

class DeviceActor extends Actor
  with SensorManagementBehavior
  with ActuatorManagementBehavior { ... }
```

8.3.2 Code mobility: proof-of-concept

Section 4.2.2 has presented the higher-order field calculus, an extension to field calculus with distributed first-class functions.

In the context of `scafi`, the "code" to be shipped is represented by functions (which can be *closures*) or, more generally, by objects. Note that, in Scala, functions are represented by objects of one of the function traits `Function0[R]` to `Function22[-T1,-T2,...,-T22,+R]`, and a lambda expressions is nothing but syntactic sugar over an anonymous class definition refining a function class.

```
val f1 = () => "foo"
val f2 = new Function0[String] { def apply = "foo" }
```

The problem inherent in transferring objects, by an implementation point of view, lies in the fact that the destination machine may not have the class for the incoming object. In fact, a class may be local to a specific subsystem.

When a communication across the network is initiated, the message (object) undergoes a serialisation (also known as marshalling) process on the sending side; that is, the object is converted into a stream of bytes for delivery through the wire. On the receiving side, the stream of bytes is deserialised to reconstruct the message.

However, if the message contains objects of classes which are not available at the recipient site, the unmarshalling cannot be performed; when using Java Serialization, the result is typically an exception of type `ClassNotFoundException` or `NoClassDefFound`.

How to deal with such a situation? The idea is to prevent or recover from the failure and let the receiver load the missing classes. This could be implemented in different ways. What is important is to handle such scenarios at the infrastructure-level.

The currently implemented solution, which is perfectible, works by intercepting deserialisation exceptions due to missing classes. This is achieved with a custom Akka (de)serialiser configured to be used for specific kinds of messages. When an error is caught, the deserialisation process returns a special message with the responsible class name to the defined recipient. At this point, the receiver B can ask the original sender A for the missing classes.

When A receives a request for a class c, it has to obtain the code of c as well as its dependencies. For the purpose, frameworks such as Apache ByteCode Engineering Library (BCEL) or OW2 ASM can be used; these usually provide visitors for navigating through classes. The result of such a lookup procedure is something like a map from class names to the corresponding `Array[Byte]` objects. This bundle has to be sent to side *B* which will operate to register the provided classes to the local classloader.

This process essentially works (as a proof-of-concept), though there is some urgent work to do:

- *Decontamination* – The current major flaw is that the high-level actors are aware of the class loading, while it would be far better to keep the process confined at the infrastructure level.
- *Soundness* – The protocol should be tested for correctness.
- *Efficiency* – The protocol should avoid the retrieval of unnecessary dependencies and the transmission of unnecessary classes.
- **Object and closure cleaning** – According to SI-1419[1], "named inner classes always contain a reference to their enclosing object, regardless of whether it is use or not"; it is important to remove unnecessary pointers[2] both for performance and to avoid references to non-serializable objects.
- *Security* – The entire code mobility process should be evaluated with respect to security.

8.4 Testing

It is well-known that testing represents an important activity in software engineering. For what concerns this first stage of development of `scafi`, the testing of the field calculus VM has been considered an essential requirement. In fact, the virtual machine is the core of the library, what ultimately executes aggregate computations; so, it must work properly and once it works, just as importantly, it should continue to operate in a correct way after refactoring activities and new developments. For this reason, a good-coverage safety net of **automated regression tests** have been prepared.

[1] https://issues.scala-lang.org/browse/SI-1419

[2] See Spark's `ClosureCleaner`: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/util/ClosureCleaner.scala

When I refer to tests, I will use the following terminology:

- A *unit test* checks the correctness of an small, individual software unit, which typically corresponds to a class.
- A *functional test* verifies a software component against a specification.
- An *integration test* checks the correctness of a set of interacting software components where the emphasis is on the mutual interactions.
- An *acceptance test* validates a software component against user requirements.

Concerning the `scafi` VM, the implementations of the following interfaces have been unit tested: `Status`, `Context`, `Export`, `Path`.

Functional tests are defined at two distinct levels:

1. Round-level – A round of an aggregate program is executed with a controlled context, and the resulting export is checked against an expected form.
2. Network-level – By means of a simulator, an aggregate program is executed on an input network and, after a certain number of rounds, the output network's values are checked against expected values.

Note that the network-level test represents a test for the simulator as well. Moreover, it is also an integration test as it verifies that the simulator and the interpret properly work together.

8.4.1 An excerpt of functional tests

The test cases are implemented with ScalaTest[3], a popular testing framework for Scala.

Alignment semantics. The check consists in verifying that operations build on a neighbourhood field take into account aligned neighbours only.

```
Alignment("should support interaction only between structurally compatible devices") {
  // ARRANGE
  val ctx1 = ctx(selfId = 0, nbs = Set(0,1,2))
  // ACT + ASSERT (no neighbor is aligned)
  round(ctx1, { rep(0)(foldhood(_)(_+_)(1)) }).root[Int]() shouldBe 1
```

[3]http://www.scalatest.org/

```
    // ARRANGE
    val exp = Map(1 -> export(path(Rep(0)) -> 1))
    val ctx2 = ctx(selfId = 0, nbs = Set(0,1,2), exports = exp)
    // ACT + ASSERT (one neighbor is aligned)
    round(ctx2, { rep(0)(foldhood(_)(_+_)(1)) }).root[Int]() shouldBe 2
  }
```

`rep` semantics. For the `rep` construct, three aspects should be tested:

1. The use of the initial state at the first execution.
2. The registration of a new state after the computation.
3. The reuse of an existing state.

```
  REP("should support dynamic evolution of fields") {
    // ARRANGE
    val ctx1 = ctx(selfId = 0, nbs = Set(0,1,2))
    // ACT
    val exp1 = round(ctx1, { rep(9)(_*2) })
    // ASSERT (use initial value)
    exp1.root[Int]() shouldBe 18
    exp1.get(path(Rep(0))) shouldBe Some(18)

    // ARRANGE
    val exp = Map(0 -> export(path(Rep(0)) -> 7))
    val ctx2 = ctx(selfId = 0, nbs = Set(0,1,2), exports = exp)
    // ACT
    val exp2 = round(ctx2, { rep(9)(_*2) })
    // ASSERT (build upon previous state)
    exp2.root[Int]() shouldBe 14
    exp2.get(path(Rep(0))) shouldBe Some(14)
  }
```

Sensors. The context comprehends the sensor values. Thus, the result of reading the value of a sensor should be coherent with the context.

```
  SENSE("should simply evaluate to the last value read by sensor") {
    // ARRANGE
    val ctx1 = ctx(0, Set(0), Map(), Map("a" -> 7, "b" -> "high"))
    // ACT + ASSERT (failure as no sensor 'c' is found)
    round(ctx1, { sense[Any]("a") }).root[Int]() shouldBe 7
    round(ctx1, { sense[Any]("b") }).root[String]() shouldBe "high"
  }

  SENSE("should fail if the sensor is not available") {
    // ARRANGE
    val ctx1 = ctx(0, Set(0), Map(), Map("a" -> 1, "b" -> 2))
    // ACT + ASSERT (failure as no sensor 'c' is found)
    intercept[Exception] { round(ctx1, { sense[Any]("c") }) }
    // ACT + ASSERT (failure if an existing sensor does not provide desired kind of data)
    intercept[Exception] { round(ctx1, { sense[Boolean]("a") }) }
  }
```

`aggregate` construct. It has to be verified that the construct supports domain restriction, i.e., when a device executes an aggregate function, its neighbourhood should only consists of aligned devices executing the same function.

```
class TestFunctionCall extends FlatSpec with Matchers {
  import ScafiAssertions._
  import ScafiTestUtils._

  val AggregateFunctionCall = new ItWord

  private[this] trait SimulationContextFixture {
    implicit val node = new Node
    val net: Network with SimulatorOps =
      simulatorFactory.gridLike(n = 6, m = 6, stepx = 1, stepy = 1, eps = 0, rng = 1.1)
    net.addSensor(name = "source", value = false)
    net.chgSensorValue(name = "source", ids = Set(2), value = true)
    net.addSensor(name = "obstacle", value = false)
    net.chgSensorValue(name = "obstacle", ids = Set(21,22,27,28,33), value = true)
  }
  // NETWORK (devices by their ids)
  //  0  1  2  3  4  5
  //  6  7  8  9 10 11
  // 12 13 14 15 16 17
  // 18 19 20 21 22 23
  // 24 25 26 27 28 29
  // 30 31 32 33 34 35
  // For each device, its neighbors are the direct devices at the top/bottom/left/right

  private[this] class Node extends Execution {
    def isObstacle = sense[Boolean]("obstacle")
    def isSource = sense[Boolean]("source")

    def hopGradient(source: Boolean): Int = {
      rep(Double.PositiveInfinity){
        hops => {
          mux(source){ 0.0 } { 1+minHood(nbr{ hops }) }
        }
      }.toInt
      // NOTE 1: Double.PositiveInfinity + 1 = Double.PositiveInfinity
      // NOTE 2: Double.PositiveInfinity.toInt = Int.MaxValue
    }

    def numOfNeighbors: Int = foldhood(0)(_+_)(nbr { 1 })
  }

  AggregateFunctionCall should "support restriction, e.g., while counting neighbors" in new
     SimulationContextFixture {
    // ARRANGE
    import node._
    // ACT
    implicit val endNet = runProgram({
      mux(isObstacle)(() => aggregate { -numOfNeighbors } )(() => aggregate { numOfNeighbors })()
    }, ntimes = 1000)(net)
    // ASSERT
    assertNetworkValues((0 to 35).zip(List(
      3, 4, 4,  4,  4, 3,
      4, 5, 5,  5,  5, 4,
      4, 5, 5,  4,  4, 4,
      4, 5, 4, -3, -3, 3,
      4, 5, 4, -4, -3, 3,
      3, 4, 3, -2,  2, 3
    )).toMap)
    // NOTE how the number of neighbors for "obstacle" devices are restricted
  }

  AggregateFunctionCall should "work, e.g., when calculating hop gradient" in new
     SimulationContextFixture {
    // ARRANGE
    import node._
    val max = Int.MaxValue
    // ACT
    implicit val endNet = runProgram({
      mux(isObstacle)(() => aggregate { max } )(() => aggregate { hopGradient(isSource) })()
    }, ntimes = 1000)(net)
    // ASSERT
    assertNetworkValues((0 to 35).zip(List(
      2, 1, 0,   1,   2, 3,
      3, 2, 1,   2,   3, 4,
      4, 3, 2,   3,   4, 5,
      5, 4, 3, max, max, 6,
      6, 5, 4, max, max, 7,
      7, 6, 5, max,   9, 8
    )).toMap)
  }
}
```

Part IV

Evaluation

Chapter 9

Evaluation

This chapter is intended to correlate the initial goals with the final results. Does `scafi` fulfill its original expectations?

Outline

1. Requirement verification
2. Demonstrative programs
3. Evaluation results

9.1 Requirement verification

A set of requirements has been defined in Section 6.1.

9.1.1 Field calculus DSL and VM

`scafi` provides a language for field calculus, embedded within Scala, i.e., as an internal DSL. The language is typed: its typing leverages on the typing of Scala as operations resolve to generic method invocations. By a syntactic point of view, it feels less awkward than MIT Proto[30], and no additional syntactic structures are introduced other than those available in Scala (cf., external DSLs such as Protelis).

Complete field calculus support All the basic constructs (see Section 6.2.1) are supported.

Correctness and testing Functional tests have been defined to check the correctness of the semantics of the field calculus primitives and built-in operators. Tests have been written for both single-rounds and network simulations.

Higher-order field calculus support `scafi` supports distributed first-class functions. However, as there is no direct way to transparently lift Scala functions in order to adapt them to the desired domain-restriction semantics, it has been necessary to introduce a new explicit construct (`aggregate`).

9.1.2 Aggregate programming platform

Distributed system specification and execution As described in Section 7.2.4, `scafi` provides an API for the setup and execution of distributed aggregate systems.

Configurability and support for different profiles In `scafi`, there are two levels of system configuration. The first level is given by incarnations, where the programmer creates a platform embodiment by choosing a certain kind of architecture (*profile*) and defining the set of types to work with. `scafi` provides a set of predefined platforms:

- Local simulation
- Local, spatial simulation
- Peer-to-peer, actor-based platform
- Actor-based platform with central server
- Actor-based platform with spatial central server

The second level of configuration is based on a structure of settings, which comes with defaults, so that the programmer is required to set up only a subset of the settings.

Code mobility A basic support for code mobility in distributed aggregate systems has been implemented as described in Section 8.3.2.

Nevertheless, the current implementation should be tested and improved. In particular, it would be important to encapsulate the code shipping facility within a well-defined, testable, infrastructure-level service.

Spatial computing A `SpatialAbstraction` component has been defined to support the definition of spaces (see Sections 7.1.2 and 8.2.2). Such spatial features can be used both in simulations (cf., `SpatialSimulation` component) and in the costruction of distributed systems (cf., `actor.server.SpatialPlatform`).

Non-functional requirements The next section, by means of demonstrative programs, will show how the non-functional requirements such as simplicity of use and flexibility can be considered satisfied.

9.2 Demos: the framework in action

A few demonstrative programs have been written in order to show how the framework can be used to develop distributed applications. Here, I am going to present them as an evidence for the accomplishment of the goals of this thesis.

In this presentation, the following dimensions of variability are considered:

- Architectural style
  - Peer-to-peer (P2P)
  - Server-based
- Neighbourhood management
  - Ad-hoc network (graph)
  - Use of spatial abstractions, with devices situated in a metric space
- Mobility
  - Fixed network
  - Mobile network
- Use of sensors

- Kind of configuration
  - Programmatic
  - File-based
  - Command-line
- Use of graphical user-interfaces (GUI)

9.2.1 Demo 0 – Peer-to-peer, ad-hoc network

In the peer-to-peer architectural style, there is no centralisation point, and the devices directly interact with one another. An ad-hoc network is a graph of devices and is specified by providing the nodes and the links (edges) of the network, explicitly.

A. Programmatic configuration Source code:

```scala
import scala.concurrent.duration.DurationInt

// STEP 1: CHOOSE INCARNATION
import it.unibo.scafi.incarnations.{ BasicActorP2P => Platform }

object Demo0A_Inputs {
  // STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
  trait Demo0A_AggregateProgram extends Platform.AggregateProgram {
    override def main(): Any = foldhood(0){_ + _}(1)
  }

  // STEP 3: DEFINE SETTINGS
  val aggregateAppSettings = Platform.AggregateApplicationSettings(
    name = "demo0A",
    program = () => Some(new Demo0A_AggregateProgram {})
  )
  val deploymentSubsys1 =
    Platform.DeploymentSettings(host = "127.0.0.1", port = 9000)
  val deploymentSubsys2 =
    Platform.DeploymentSettings(host = "127.0.0.1", port = 9500)

  val settings1 = Platform.settingsFactory.defaultSettings().copy(
    aggregate = aggregateAppSettings,
    platform = Platform.PlatformSettings(
      subsystemDeployment = deploymentSubsys1,
      otherSubsystems = Set(Platform.SubsystemSettings(
        subsystemDeployment = deploymentSubsys2,
        ids = Set(4,5)
      ))
    ),
    deviceConfig = Platform.DeviceConfigurationSettings(
      ids = Set(1,2,3),
      nbs = Map(1 -> Set(2,4), 2 -> Set(), 3 -> Set())
    )
  )

  val settings2 = settings1.copy(
    platform = Platform.PlatformSettings(
      subsystemDeployment = deploymentSubsys2
    ),
    deviceConfig = Platform.DeviceConfigurationSettings(
      ids = Set(4,5), nbs = Map()
    )
  )
}

// STEP 4: DEFINE MAIN PROGRAMS
object Demo0A_MainProgram_Subsys1
  extends Platform.BasicMain(Demo0A_Inputs.settings1)

object Demo0A_MainProgram_Subsys2
  extends Platform.BasicMain(Demo0A_Inputs.settings2)
```

Note how the incarnation – which in this case is `BasicActorP2P`, aliased to `Platform` – works as a container for types and classes.

With this configuration approach, the great majority of the code is devoted to the programmatic construction of the settings. In any case, the key steps are clear:

1. Choice of a platform incarnation
2. Definition of the aggregate computation program
3. System configuration
4. Definition of the "main" programs for launching subsystems

In this example, the system consists of two subsystems: the first with devices 1, 2, 3; the second with devices 4, 5. Also, the neighbourhood of device 1 is set to include devices 2 and 4. The aggregate program simply computes, in each device, the number of neighbours (plus the device itself).

B. File-based configuration Source code:

```scala
// STEP 1: CHOOSE INCARNATION
import it.unibo.scafi.incarnations.{ BasicActorP2P => Platform }

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Demo0B_AggregateProgram extends Platform.AggregateProgram {
  override def main(): Any = foldhood(0){_ + _}(1)
}

// STEP 3: DEFINE MAIN PROGRAM
object Demo0B_MainProgram_Subsys1 extends
  Platform.FileMain("demos/src/main/scala/demos/Demo0B_Subsys1.conf")
object Demo0B_MainProgram_Subsys2 extends
  Platform.FileMain("demos/src/main/scala/demos/Demo0B_Subsys2.conf")
```

With this configuration approach, the system source code is not cluttered with configuration code, which is relegated to files. The following listing reports the content of Demo0B_Subsys1.conf:

```
aggregate {
  application.name = "demo0B"
  application.program-class = "demos.Demo0B_AggregateProgram"
  deployment {
    host = "127.0.0.1"
    port = 9000
  }
  subsystems = [${subsys2}]
  execution {
    scope {
      type = "device" // Alternatives: device, global, subsystem
      strategy = "delayed"
      initial-delay = 1000
      interval = 1000
    }
  }
  devices {
    ids = [1,2,3]
    nbrs = {1:[2,4]}
  }
}

subsys2 {
  deployment {
    host = "127.0.0.1"
    port = 9500
  }
  ids = [4,5]
}
```

C. Command-line configuration Source code:

```
// STEP 1: CHOOSE INCARNATION
import it.unibo.scafi.incarnations.{ BasicActorP2P => Platform }

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Demo0C_AggregateProgram extends Platform.AggregateProgram {
  override def main(): Any = foldhood(0){_ + _}(1)
}

// STEP 3: DEFINE MAIN PROGRAM
object Demo0C_MainProgram extends Platform.CmdLineMain
```

In this case, there is no need to differentiate the main programs as the subsystems are configured by command-line arguments.

Two run configurations might be:

1. `--program "demos.Demo0C_AggregateProgram" -h 127.0.0.1 -p 9000 -e 1:2,4;2;3 --subsystems 127.0.0.1:9500:4:5`
2. `--program "demos.Demo0C_AggregateProgram" -h 127.0.0.1 -p 9500 -e 4;5`

9.2.2 Demo 1 – Server-based, ad-hoc network

In the server-based architectural style there is a central server that is responsible of the propagation of the device exports according to the implemented notion of neighbourhood.

```
// STEP 1: CHOOSE INCARNATION
import it.unibo.scafi.incarnations.{ BasicActorServerBased => Platform }

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Demo1_AggregateProgram extends Platform.AggregateProgram {
  override def main(): Any = foldhood(0){_ + _}(1)
}

// STEP 3: DEFINE MAIN PROGRAM
```

```
object Demo1_MainProgram extends Platform.CmdLineMain

object Server_MainProgram extends Platform.ServerCmdLineMain
```

Note that, with respect to Demo 0, the only differences are i) the choice of the platform incarnation, and ii) the definition of a separate main program for the server (actually, this is just for clarity).

9.2.3 Demo 2 – Server-based, spatial network

In this case, spatial abstractions are used: the server holds the information about the location of the devices in a 3-dimensional space (situation), and the neighbouring relation is based on the euclidean distance and a threshold value. The devices retrieve their spatial position by a sensor and notify the server with each new value (in this demo, the position is set once, so the network is fixed).

```
// STEP 1a: CHOOSE BASE INCARNATION
import it.unibo.scafi.distrib.actor.server.{SpatialPlatform =>
    SpatialServerBasedActorPlatform}

// STEP 1b: REFINE AND INSTANTIATE INCARNATION
object Demo2_Platform
  extends BasicAbstractDistributedIncarnation
  with SpatialServerBasedActorPlatform
  with BasicSpatialAbstraction with Serializable {
  override val LocationSensorName: String = "LOCATION_SENSOR"
  override type P = Point2D

  override def buildNewSpace[E](elems: Iterable[(E,P)]): SPACE[E] =
    new Basic3DSpace(elems.toMap) {
      override val proximityThreshold = 2.5
    }
}

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Demo2_AggregateProgram extends Demo2_Platform.AggregateProgram {
  override def main(): Any = foldhood(0){_ + _}(1)
}

// STEP 3: DEFINE MAIN PROGRAM
object Demo2_MainProgram extends Demo2_Platform.CmdLineMain {
  override def onDeviceStarted(dm: Demo2_Platform.DeviceManager,
                               sys: Demo2_Platform.SystemFacade) = {
    dm.addSensorValue(Demo2_Platform.LocationSensorName,
                      Point2D(dm.selfId,0))
  }
}

object Demo2_Server extends Demo2_Platform.ServerCmdLineMain
```

Here, the crucial points are the refinement of the spatial incarnation (in particular, the overriding of the factory method for the spatial container, and the definition of the position type `P` to `Point2D`) and the specification of the device location via `addSensorValue`.

9.2.4 Demo 3 – Server-based, mobile spatial network

In this demo, the devices are equipped with a location sensor that is let to randomly vary its value (for some time). The aggregate program computes the hop-distance from a "source" device.

```
import it.unibo.scafi.incarnations.BasicAbstractActorIncarnation
import it.unibo.scafi.space.{Point2D, BasicSpatialAbstraction}

// STEP 1a: CHOOSE BASE INCARNATION
import it.unibo.scafi.distrib.actor.server.{SpatialPlatform =>
    SpatialServerBasedActorPlatform}

// STEP 1b: REFINE AND INSTANTIATE INCARNATION
object Demo3_Platform extends BasicAbstractActorIncarnation
  with SpatialServerBasedActorPlatform
  with BasicSpatialAbstraction with Serializable {
  override val LocationSensorName: String = "LOCATION_SENSOR"
  override type P = Point2D

  override def buildNewSpace[E](elems: Iterable[(E,P)]): SPACE[E] =
    new Basic3DSpace(elems.toMap) {
      override val proximityThreshold = 1.5
    }
}

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Demo3_AggregateProgram extends Demo3_Platform.AggregateProgram {
  def hopGradient(source: Boolean): Double = {
    rep(Double.PositiveInfinity){
      hops => { mux(source) { 0.0 } { 1+minHood(nbr{ hops }) } }
    }
  }

  def main() = hopGradient(sense("source"))
}
```

```
// STEP 3: DEFINE MAIN PROGRAMS
object Demo3_MainProgram extends Demo3_Platform.CmdLineMain {
  override def onDeviceStarted(dm: Demo3_Platform.DeviceManager,
                               sys: Demo3_Platform.SystemFacade) = {
    val random = new scala.util.Random(System.currentTimeMillis())
    var k = 0
    var positions = (1 to 5).map(_ => random.nextInt(10))
    dm.addSensor(Demo3_Platform.LocationSensorName, () => {
      k += 1
      Point2D(if(k>=positions.size) positions.last else positions(k), 0)
    })
    dm.addSensorValue("source", dm.selfId==4)
  }
}

object Demo3_ServerMain extends Demo3_Platform.ServerCmdLineMain
```

9.3 Evaluation results

A concise exposure of how functional requirements for `scafi` can be considered verified is provided in Section 9.1. The demo programs presented in Section 9.2 are intended to provide some evidence also for what concerns non-functional requirements, whereas some insights on the internal quality of the project can be found in Chapters 7 and 8.

In particular, the demo programs clearly show how distributed, aggregate systems can be set up in `scafi` with just a few lines of code. This is coherent with the framework's goal of ease of use.

The goal of flexibility is also important, because of the variability of scenarios that may take advantage of aggregate programming techniques. For example, as shown in [26], even ad-hoc networks consisting of a small number of nodes may employ aggregate programming techniques to implement distributed adaptative algorithms. As shown in the demos, `scafi` supports both peer-to-peer and server-based architectural styles, and allows for the definition of both ad-hoc networks and networks overlaying some space-time fabric.

9.3.1 On internal quality: guidelines for improvement

`scafi` has been developed in limited time and, in part, as an explorative endeavour; so, in general, analysis and design activities should continue, accompanied by extensive refactoring.

To be more precise, the following elements should require some attention.

Implementation of the component architecture The current implementation based on family-polymorphism should be investigated in detail in order to gain a better understanding on the limitations of the approach. Real-world usage of the framework may provide some advice in this direction. In particular:

- How must different incarnations be allowed to interact? For example, presently, actors use instance-level message types – this means that a device actor built from an incarnation `i1` is unable to interact with a devide actor built from an incarnation `i2`. If this is not the intended behavior, it is possible to implement a different semantics by avoiding extractors in pattern matching and using type projection instead.
- The incarnation instances appear as huge objects containing a lot of members. However, when exposing an incarnation to users, it would be better to fine-tune the visibility of members so that only significant types and objects (from the user-perspective) are visible.

Code mobility implementation As pointed out at the end of Section 8.3.2, the implementation of the code shipping facility has room for improvement.

Serialisation of inner classess According to [31], the serialisation of instances of inner classes is error prone and should be avoided. Unfortunately, the component-based approach used in `scafi` makes use of component traits that work as containers for many inner classes. This creates significant problems for the implementation of the actor-based distributed platform, where serialisation is used extensively. Currently, the problem has been naively solved by making inner members of these components `Serializable`. A better solution would be to (transparently) remove, from instances of inner classes, the reference to the object of the containing class (when this is not used).

Testing Currently, the actor-based platform is totally untested. Given the complexity of this module, some extent of automated tests would be very valuable. Akka provides a dedicated module, `akka-testkit`, for testing actor systems at different levels.

Chapter 10

Conclusion and Perspectives

This chapter includes a few brief, general considerations about this thesis' work, as well as some references to future developments.

10.1 Final thoughts

The aggregate programming approach synthesised in Part II and implemented in `scafi`, based on computational fields and building blocks for robust coordination as described in [27], seems promising. However, this potential should be tested against the development of (real-world) applications, so as to let application forces stress the approach to its limits and make practice reinforce current understanding. For this reason, a framework such as `scafi`, which is aimed at simplifying the development of aggregate applications, may be particularly valuable.

10.2 Agenda

`scafi` is by no means complete. In particular, the following aspects would be very important for making `scafi` desirable in practice:

- Integration with the Alchemist simulator [25] – `scafi` comes with a very basic simulator which can be used to setup simple networks of devices and run aggregate computations on them. However, the goal was not to build a fully-featured simulator, and it would not be savvy to reinvent the wheel, unless strong motivations urged to do so.

- Android integration – The possibility of running `scafi` on Android smartphones would be greatly valuable for the development and testing of aggregate applications.
- Cloud services – In some scenarios, it may be highly beneficial to take advantage of the cloud for executing aggregate computations. In these cases, devices would be mainly devoted to sensing and acting on the environment, with limited in-site processing.

In parallel to this thesis, activities working in these directions have already been started (for example, see [32]), with preliminary results supporting the idea that a complete toolchain might be available in reasonable time.

Bibliography

[1] The Scala language specification – version 2.9. http://www.scala-lang.org/docu/files/ScalaReference.pdf. Accessed: 2015-11-29.

[2] The scala api documentation – version 2.11.0. http://www.scala-lang.org/files/archive/api/2.11.0/. Accessed: 2015-11-29.

[3] Joshua D. Suereth. *Scala in Depth*. Manning Publications Co., Greenwich, CT, USA, 2012.

[4] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41, October 2005.

[5] Cay S. Horstmann. *Scala for the Impatient*. Addison-Wesley Professional, 1st edition, 2012.

[6] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[7] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 441–469, London, UK, UK, 2002. Springer-Verlag.

[8] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM.

[9] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, September 2006.

[10] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. *SIGPLAN Not.*, 35(10):146–165, October 2000.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] Chiari Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18:285–331, 2008.

[13] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[14] Franco Zambonelli and Marco Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. In *Autonomic communication*, pages 44–57. Springer, 2005.

[15] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[16] Jacob Beal, Olivier Michel, and Ulrik Pagh Schultz. Spatial computing: Distributed systems that take advantage of our geometric world. *ACM Trans. Auton. Adapt. Syst.*, 6(2):11:1–11:3, June 2011.

[17] Matt Duckham. *Decentralized Spatial Computing - Foundations of Geosensor Networks*. Springer, 2013.

[18] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the Aggregate: Languages for Spatial Computing. page 60, feb 2012.

[19] Jacob Beal and Mirko Viroli. Space–Time Programming. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[20] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F Knight Jr, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[21] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.

[22] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In Carlos Canal and Massimo Villari, editors, *ESOCC Workshops*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer, 2013.

[23] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.

[24] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1846–1853, New York, NY, USA, 2015. ACM.

[25] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.

[26] Shane S. Clark, Jacob Beal, and Partha Pal. Distributed recovery for enterprise services. In *SASO*, pages 111–120. IEEE Computer Society, 2015.

[27] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate Programming for the Internet of Things. *IEEE Computer*, 48(9):22–30, 2015.

[28] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In eva Kühn and Rosario Pugliese, editors, *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, June 2014. Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June.

[29] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *Proceedings of the 2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '14, pages 8–13, Washington, DC, USA, 2014. IEEE Computer Society.

[30] MIT Proto. Mit Proto. *Software available at http://proto. bbn. com*, 2012.

[31] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle secure coding standard for Java.* The SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ, 2012.

[32] Simone Costanzi. Integrazione di piattaforme d'esecuzione e simulazione in una toolchain Scala per aggregate programming.