ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Design and implementation of a scalable domain specific language foundation for ScaFi with Scala 3

Tesi di laurea in:
PARADIGMI DI PROGRAMMAZIONE E SVILUPPO

Relatore
Prof. Mirko Viroli

Correlatore
Dott. Gianluca Aguzzi

Candidato
Luca Deluigi

IV Sessione di Laurea
Anno Accademico 2022-2023

Abstract

In the field of macroprogramming, one of the prominent engineering techniques is aggregate computing, which allows the definition of the overall behavior of a network of devices or agents through a single program, called the aggregate program. ScaFi is an aggregate programming framework that comprises an internal Domain Specific Language (DSL) written in Scala 2 with supporting components for simulation, visualization, and execution of aggregate systems, based on the Field Calculus formal language and computational model. Recently, a new formal language and computational model called Exchange Calculus has been proposed, extending the expressiveness of Field Calculus while simplifying its set of primitive constructs. In the meantime, Scala 2 has been succeeded by Scala 3, which introduces a relevant set of new features and improvements over its predecessor. These two novelties have promoted the development of a new ScaFi, called ScaFi-XC, entirely redesigned to be formally based on Exchange Calculus and to make the best use of its new host language, Scala 3, enhancing the original ScaFi with a more expressive and flexible DSL. This thesis presents the design and implementation of ScaFi-XC, focusing on the core DSL and associated components such as the engine and the simulator. In conclusion, ScaFi-XC has been successfully implemented and tested, delivering the expected results and fulfilling the requirements of the stakeholders, which are mainly the researchers and developers of the original ScaFi. Nevertheless, there is still work to be done to have a complete and stable version of the framework, such as porting all the support modules and improving the simulator.

To my past self who always looked forward to this achievement.

Acknowledgements

I would like to thank my supervisor, Professor Mirko Viroli, and his research team, in particular Gianluca Aguzzi and Roberto Casadei, for having supported me and having allowed me to work on this thesis. Thanks for all your time and guidance.

Next, I would like to thank my family, who supported my studies and my choices and accompanied me on this journey.

A special thanks goes to my beloved Giada, who has always been by my side in the past years, supporting and helping me to overcome the difficulties I encountered, and accompanying me in my new life in Rimini.

Finally, I would like to thank all my close friends, with whom I have shared great moments, studying with the *Spaceteam*, playing board games with *Zuga Rèmni*, living the university with *S.P.R.I.Te.*, and working with *EasyDesk*. These names represent the best groups of people I ever met and had the pleasure to pass my time with.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The importance of Collective Adaptive Systems (CASs) engineering is increasing along with the growing prevalence of large-scale cyber-physical systems, driven by the already pervasive Internet of Things (IoT) and edge computing trends [1, 2]. CASs are a particular subcategory of situated, distributed systems in which a collection of individuals, also called agents, exhibits a non-chaotic behavior characterized by *self-** properties, such as self-organization, self-healing, and self-configuration. Many of the self-* properties cannot be derived from an individual perspective confined to the behavior of a single agent, but rather they are *emergent* from the complex and dynamic network of interactions within the system and with the environment. As a consequence, obtaining such properties requires a holistic approach to the design and programming of CASs, which is not straightforward to achieve with traditional software engineering techniques.

Topologically, CASs can be considered a subset of Multi Agent Systems (MASs), and the programming of their behavior as a whole can be referred to as *Macro-programming* [3]. In the context of macroprogramming, a prominent paradigm is that of *aggregate computing*, where a single program, called the aggregate program, defines the overall behavior of a network of devices or agents [3]. Aggregate computing provides benefits to development productivity thanks to four main factors [4]: the *macro-level stance*, which abstracts over low-level details about the individual behavior of agents and their communication media, the *compositionality*, promoting the creation of complex behavior by combining simpler ones, the *formality*, allowing theoretical analyses and formal verification of its properties, and the *practicality*, thanks to the availability of tools supporting programming and simulation of CASs.

Among these tools, an in-depth analysis is dedicated to the *ScaFi*, an aggregate programming framework that comprises an internal Domain Specific Language (DSL) written in Scala 2 with supporting components for simulation, visualization, and execution of aggregate systems, based on the Field Calculus (FC) formal language and computational model [4]. Internal DSLs are distinguished from external DSLs by the fact that they are embedded in a host language, such as Scala, and that they exploit the flexibility of their host's syntax to resemble a different language even though they are implemented as libraries.

Recently, a new formal language and computational model called Exchange Calculus (XC) has been proposed, extending the expressiveness of FC while simplifying its set of primitive constructs [5]. In the meantime, Scala 2 has been succeeded by Scala 3, which introduces a relevant set of new features and improvements over its predecessor. These two novelties have promoted the development of a new ScaFi, entirely redesigned to be formally based on XC and to make the best use of its new host language, Scala 3. That project is the main subject of this thesis. Named "ScaFi-XC", it has the ambition to *scale*, meaning to grow with the demand of its user without losing its qualities, such as expressiveness, reusability, maintainability, readability, and performance.

Structure of the Thesis The thesis is structured as follows. In Chapter 2, the state of the art is reviewed, focusing on prominent frameworks and tools such as Protelis, FCPP, ScaFi, and an experimental implementation of XC in Scala. Chapter 3 discusses the background information necessary for understanding the thesis, including discussions on the Exchange Calculus and Scala 3, covering topics such as the type system, explicit nulls, and multiversal equality. Chapter 4 conducts a comprehensive analysis, beginning with a detailed examination of the requirements, followed by the analysis of the current state of the ScaFi framework, from which this project originates. Chapter 5 outlines the design process, starting with the study of four proposed prototypes, and culminating in the final design of the core DSL and associated components such as the engine and the simulator.

Chapter 6 focuses on the implementation details, covering aspects such as the implementation of the XC operational semantics, the build system, and integration with the Alchemist simulator. Chapter 7 presents the evaluation methodology and results, including unit tests, acceptance tests, continuous integration, and techniques employed for code quality and maintainability. Finally, Chapter 8 concludes the thesis and provides insights into future work and potential extensions or improvements.

Chapter 2

State of the art

In the field of aggregate programming [6], multiple frameworks, tools, and experiments have been developed and made available as public resources to support a wide variety of use cases, using different languages and design approaches. Among the state-of-the-art tools in this field are *ScaFi*[1] [4], *Protelis*[2] [7], and *FCPP*[3] [8] which have been briefly described in their fundamental characteristics in this chapter, with a particular emphasis on ScaFi, of which ScaFi-XC is a redesign and reimplementation. Each of the mentioned libraries is backed by a robust, coherent theoretical foundation, that provides consistency and guarantees the emergence of global properties in derived CASs. The theoretical framework that serves as the basis for all cited implementations is the FC [9], specifically its higher-order version, the Higher-Order Field Calculus (HFC) [10]. FC, as well as its variants, is a type-safe, formal language for aggregate programming [9, 11] presented with its operational and denotational semantics, respectively describing the local and global interpretation of field expressions [11]. From a developer perspective, the key aspect of FC is the possibility of focusing on the denotational semantics of field constructs, abstracting away from the local interpretation of expressions and implementation of the constructs. In recent years, a new formal language called XC [5] has been developed, which is a promising evolution of FC. XC [5] has the potential to supersede FC entirely since it is a simpler yet more expressive language

[1]`https://github.com/scafi/scafi`
[2]`https://github.com/Protelis/Protelis`
[3]`https://github.com/fcpp/fcpp`

that can be used to implement all the FC constructs while retaining their original semantics. ScaFi-XC and FCPP in its current version are based on this newer formal language, further described in Chapter 3, where FC and XC are briefly compared. Some additional experiments with the implementation of XC already exist, such as *imperative-xc*[4] and *XC: Scala DSL Implementation*[5] [12], with the latter described in Section 2.4.

2.1 Protelis

Protelis is an external domain-specific language derived from the discontinued *Proto*, whose syntax resembles that of C or Java. However, Protelis is purely functional, albeit dynamically typed, and uses a virtual machine written in Java [7] for interpretation. As an external DSL, Protelis syntax is close to the FC language it implements, which distinguishes it from internal DSLs like ScaFi and FCPP. As a result, domain branching in Protelis is more transparent in its conditional control statements, such as the `if` statement, while, in internal DSLs, custom operators must be used to avoid conflicts with the host language's homonymous constructs. Further information on domain branching can be found in Section 3.1.4. Nevertheless, the Protelis environment comes with some costs, such as a lack of compiler support for type checking, as it uses duck typing. Additionally, IDE support is exclusively available for the Eclipse platform, as Protelis is based on the Xtext framework [13]. Moreover, external DSLs like Protelis cannot benefit from the community of developers and libraries of a general-purpose language such as Scala or C++, which are the host languages for ScaFi and FCPP, respectively.

An example of a gradient distance written using Protelis can be found in Listing 2.1.

2.2 FCPP

FCPP is an internal domain-specific language that is written in C++ and is designed for achieving high efficiency and performance for devices with limited re-

[4]`https://github.com/cric96/imperative-xc`
[5]`https://github.com/scafi/artifact-2021-ecoop-xc`

```
module org:protelis:tutorial:distanceTo

def myPosition() = self.getDevicePosition()

def nbrRange() = nbr(myPosition()).distanceTo(myPosition())

share (d <- POSITIVE_INFINITY) {
  // Must be executed outside the 'if' block!
  let shortestPathViaNeighborhood = foldMin(POSITIVE_INFINITY, d + nbrRange()
    )
  if (env.has("source")) { 0 } else { shortestPathViaNeighborhood }
}
```

Listing 2.1: Gradient distance from a source in Protelis.

Listing 2.2: Gradient distance from a source in FCPP.

```
//! @brief Computes the distance from a source with a custom metric through
    adaptive bellmann-ford.
template <typename node_t, typename G, typename = common::if_signature<G, field<
    real_t>()>>
real_t abf_distance(node_t& node, trace_t call_point, bool source, G&& metric) {
    internal::trace_call trace_caller(node.stack_trace, call_point);

    return nbr(node, 0, INF, [&] (field<real_t> d) {
        return min_hood(node, 0, d + metric(), source ? 0 : INF);
    });
}
```

sources like microcontrollers and embedded systems [8]. Although FCPP was originally based on FC, it has been updated to support XC. As stated in the paper, FCPP suffers more limitations than ScaFi when it comes to avoiding conflicts with the host language, resulting in a less “clean” syntax [8]. Additionally, it lacks integration with the Java environment, which is in turn natively supported by ScaFi, being written in Scala. Another critical difference between the design of FCPP and ScaFi is the presence of explicit `field` types, which are absent in ScaFi thanks to its design around `foldhood` operations, as described in Section 2.3.

Listing 2.2 represents an example of how gradient distance can be written using FCPP.

Listing 2.3: Gradient distance from a source in ScaFi.

```scala
def gradient(source: Boolean): Double =
    rep(Double.PositiveInfinity){ distance =>
        mux(source) { 0.0 } {
            foldhoodPlus(Double.PositiveInfinity)(Math.min)(
                nbr{distance} + nbrRange
            )
        }
    }
```

2.3 ScaFi

ScaFi, short for *Sca*la *Fi*elds, is a framework for aggregate programming featuring an internal DSL written in pure Scala 2 [4] and implementing a variant of the HFC. Besides the DSL, which represents the core of ScaFi, the framework offers additional components for the simulation, visualization, and deployment of aggregate programs. Scafi cross-compiles for Scala 2.11, 2.12, and 2.13, while its `core` and `simulator` packages are also cross-built for JavaScript (JS) using *Scala.js* [14].

The most notable aspect of its DSL is the *foldhood* semantics, which abstracts over the concept of *field* or *neighbouring value*, as shown in Listing 2.3. In that usage example, the `foldhoodPlus` operator invokes and collects the results of the passed expression for each neighbor, including itself, accumulating all of them into a single value using `Math.min`. With this approach, `nbr`, which is the main communication primitive of FC, can be utilized seamlessly in combination with local values, eliminating the requirement for a `lift` operator that would typically be necessary to operate with `field` types, such as in FCPP or in the "XC: Scala DSL Implementation" experiment, as described in Section 2.4. As a limitation of the approach, `nbr` cannot be used outside one of the `foldhood` variants.

A more thorough examination of ScaFi can be found in Section 4.2.

2.4 XC: Scala DSL Implementation

The first implementation of XC in Scala is based on ScaFi and presented in the XC papers [5] [12]. This implementation uses Scala 2. Although ScaFi hides the `field` abstraction from the user, *NValue*s had to be explicitly implemented in this experiment, given the new semantics they provide. The Scala 2 implicit conversions

Listing 2.4: Gradient distance from a source in XC Scala 2 DSL.

```scala
def distanceTo(source: Boolean, metric: NValue[Double]): Double =
    exchange(Double.PositiveInfinity)(n =>
      mux(source) {
        0.0
      } {
        (n + metric).withoutSelf.fold(Double.PositiveInfinity)(Math.min)
      }
    )
```

allowed for the implementation of an automatic conversion from local values to NValues, as explained in the XC paper and Section 3.1.2. In the experiment, publicly available on GitHub[6] under the Apache 2.0 License and on Zenodo [15], the FC constructs have been implemented using `exchange`, the only communication primitive of XC, suggesting a new syntax for a pure Scala XC, later taken as inspiration for ScaFi-XC.

An example of a gradient distance written with this DSL can be found in Listing 2.4.

[6]`https://github.com/scafi/artifact-2021-ecoop-xc`

Chapter 3

Background

This chapter discusses the two main driving factors that led to a major re-design of ScaFi, which are the introduction of the XC and the release of Scala 3. On one hand, XC opens the way for considerable design improvements given the simpler set of foundational constructs it requires, consisting of the single primitive `exchange`, from which the entire language takes its name. Additionally, it provides new opportunities for aggregate program developers, enabled by the expressiveness of XC, regarding, in particular, the possibility of sending differentiated messages to neighbors using the `exchange` primitive [5]. On the other hand, Scala 3 introduces significant language changes and improvements from Scala 2 while maintaining binary retro-compatibility. This promotes the rewriting of Scala 2 libraries to leverage new language features while providing cross-builds to Scala 2 through the Scala 3 compiler "Dotty"[1].

3.1 The Exchange Calculus

XC is a language that formalizes a tiny set of key mechanisms, sufficient to express the overall behavior of a distributed collective adaptive systems in a declarative fashion [5]. This language provides two types of semantics, operational and denotational. The operational semantics defines the local interpretation of XC mechanisms on each device, while the denotational semantics abstracts away the

[1]`https://github.com/lampepfl/dotty`

operational semantics details and provides an interpretation of these mechanisms at the network level. Therefore, operational semantics guides the implementation of XC as a framework, while denotational semantics is the sole knowledge base required when programming a CAS using this language. Finally, the XC language generalizes over FC and is derived from the typed lambda calculus [5].

XC is built upon the following fundamental components:

- the basic system model and its assumptions;
- the data type for neighboring values, *NValues*;
- the only communication primitive, `exchange`, which allows sending differentiated messages to neighbors;
- the concept of *alignment*, which enables *functional composition of distribute behavior* [5].

3.1.1 System model

Similarly to FC, XC targets a system modeled as a collection of devices generally equipped with sensors and/or actuators. These devices repeatedly compute execution *rounds* of the **same program** and exchange asynchronous *messages* with their respective neighbors [5]. In this environment, devices can experience failures, reboots, network outages, and dynamic neighborhood changes. At each execution round, a device independently gathers a local context, consisting of inbound messages from neighbors, sensors data, and memory of its previous round of execution, if any, and then it *atomically executes* the XC program acting on its local context [5]. The program can result in an output, that comprises side effects such as actuation, as well as, implicitly, the messages to send to neighbors for coordination [5]. At the end of each round, a device begins waiting for an arbitrary time lapse, during which the device is considered "sleeping". Once the sleep time is over, the device "wakes up" and starts the next execution round [5]. During sleep, a device must still collect inbound messages and apply two policies: *last-message buffering* and *last-message dropping.*

Last-message buffering means that every message received by a device is collected in a buffer and kept until some established criterion determines its *expiration*, even across multiple execution rounds [5]. As a result, the message expiration is also the minimum time that a device takes to realize that a neighbor has disappeared, either because of a failure or a neighbor network change.

Last-message dropping means that every message received by a device supersedes the last message, still in the buffer, coming from the same device. [5] This implies a notion of identity of devices, which is a way to recognize a neighbor's identity to discard obsolete messages coming from them.

Communication between devices defined in XC is agnostic of the message exchange medium, channel, network topology, or discovery mechanisms. Messages in such a model are subject to traditional distributed systems communication properties, such as unpredictable delays and drops [5]. In addition, for XC and its implementations, a device memory of its previous execution round result can be treated as a self-message effectively. In this perspective, a device reboot, which in practice consists of a complete loss of memory, is modeled as a self-message drop, thereby simplifying the communication model in the operational semantics.

3.1.2 NValues

XC features two kinds of values: *local values* and *neighboring values* (NValues or nvalues). Local values l refer to all the traditional types `A` like integer, float, list, and so on. NValues, instead, are a map $\underline{\mathbf{w}}$ from device identifiers δ_i to local values l_i, where the default local value l is denoted with $l[\delta_1 \mapsto l_1, ..., \delta_n \mapsto l_n]$ [5].

NValues refer to values coming from neighbors, which, in highly decoupled distributed systems, almost always consist of a subset of all devices. This scenario may occur when other devices are out of reach in a spacial-dependent neighboring relationship. The default value is used when evaluating a NValue $\underline{\mathbf{w}} = l[\delta_1 \mapsto l_1, ..., \delta_n \mapsto l_n]$ for a given δ_i with δ_i not present in $\underline{\mathbf{w}}$. The notation above can thus be read as "the nvalue $\underline{\mathbf{w}}$ is l everywhere (i.e. for all neighbors) except for devices $\delta_1, ..., \delta_n$ with values $l_1, ..., l_n$, respectively" [5].

For example, in Figure 3.1, the device δ_2 wakes up for computation ϵ_2^4 and processes a nvalue $\underline{\mathbf{w}} = 0[\delta_1 \mapsto 5, \delta_3 \mapsto 4, \delta_4 \mapsto 2]$, which corresponds to the messages carrying the scalar values 5, 4, and 2 sent by devices δ_1, δ_3, and δ_4, respectively, some of which while device δ_2 was asleep. For all other devices, the entry in $\underline{\mathbf{w}}$ evaluates to 0. After the computation, δ_2 sends out the messages represented by $\underline{\mathbf{w'}} = 0[\delta_1 \mapsto 7, \delta_4 \mapsto 1]$. For instance, 7 is sent to δ_1, 1 to δ_4, and 0 to all other neighbors, such as δ_3. Evaluation of a nvalue for a given δ' can be noted as $\underline{\mathbf{w}}(\delta')$ and its result is the local value l' if $\delta' \mapsto l'$ is in $\underline{\mathbf{w}}$, or the default value l of $\underline{\mathbf{w}}$. For $\underline{\mathbf{w'}}$, $\underline{\mathbf{w'}}(\delta_1)$ is 7, and $\underline{\mathbf{w'}}(\delta_3)$ is 0. Another notation used in the paper is $\underline{A}$, to indicate the type of a nvalue $\underline{\mathbf{w}} = l[\delta_1 \mapsto l_1, ..., \delta_n \mapsto l_n]$ where $l_1, ..., l_n$ are of type A [5].

NValues generalize local values, in the sense that a local value l with type A can be automatically converted to a nvalue $l[]$ with type $\underline{A}$, with l as the default value for every device [5]. This approach simplifies the formalization of XC, where local values and nvalues are treated uniformly [5]. The same principle can be applied to functions, whereby they can be implicitly "lifted" to operate on NValues. This can be achieved by applying said functions pointwise on the content of the maps, using the default values where required [5]. For example, given $\underline{\mathbf{w_1}} = 1[\delta_1 \mapsto 2, \delta_3 \mapsto 4]$ and $\underline{\mathbf{w_2}} = 3[\delta_1 \mapsto 5, \delta_2 \mapsto 6]$, $\underline{\mathbf{w_3}} = \underline{\mathbf{w_1}} + \underline{\mathbf{w_2}} = 4[\delta_1 \mapsto 7, \delta_2 \mapsto 7, \delta_3 \mapsto 7]$. Another example is $\underline{\mathbf{w_4}} = \underline{\mathbf{w_1}} + 1 = 2[\delta_1 \mapsto 3, \delta_3 \mapsto 5]$, which uses the automatic promotion of 1 to $1[]$.

NValues can be folded over, using the built-in function $nfold(f : (A, B) \to A, \underline{\mathbf{w}} : \underline{B}, l : A) : B$, which takes an accumulator function f repeatedly applied to neighbors' values in a nvalue, excluding the value for the *self* device, starting from a base local value l, and using the default value of $\underline{\mathbf{w}}$ for neighbors not present in the map [5]. For example, given a device δ_1 performing a $nfold$ operation on $\underline{\mathbf{w}} = 3[\delta_1 \mapsto 10, \delta_2 \mapsto 1, \delta_3 \mapsto 2]$ while the current set of its neighbors is $\{\delta_3, \delta_4\}$, then $nfold(+, \underline{\mathbf{w}}, 1) = 6$. Given that nvalues are agnostic to the ordering of elements, i.e., the ordering of device identifiers in the map, f is assumed to be associative and commutative [5].

Additional built-in operations on nvalues are $self(\underline{\mathbf{w}} : \underline{A}) : A$ which returns the local value $\underline{\mathbf{w}}(\delta)$ for the self device δ, and $updateSelf(\underline{\mathbf{w}} : \underline{A}, l : A) : \underline{A}$ which

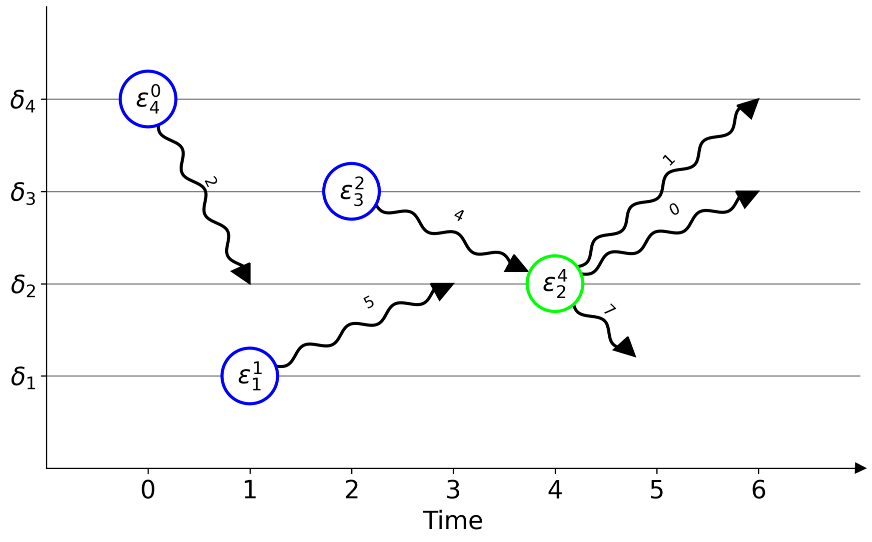


Figure 3.1: XC system model, from the point of view of the wake-up event ϵ_2^4 pictured in green.

returns a new nvalue with the same content of $\underline{\mathbf{w}}$ but with the value for the self device replaced by l [5]. Given the built-in function *uid* that returns the device identifier of the self device, the following property holds: $self(\underline{\mathbf{w}}) = \underline{\mathbf{w}}(uid)$. The complete syntax for XC is available in the original paper [5, p. 4].

3.1.3 The "exchange" primitive

The following description is based on the original paper for XC [5]. The only communication primitive present in XC is the function

$$exchange(e_i, (\underline{\mathbf{n}}) \Rightarrow \text{return}\, e_r \text{ send}\, e_s)$$

which is defined using syntactic sugar and translates to

$$exchange(e_i, (\underline{\mathbf{n}}) \Rightarrow (e_r, e_s))$$

The evaluation of the primitive follows three steps:

1. the device evaluates the expression e_i to obtain the *initial* local value l_i;

2. $\underline{\mathbf{n}}$ is substituted with the nvalue $\underline{\mathbf{w}}$ of messages received from neighbors for this exchange, using l_i as the default value for $\underline{\mathbf{w}}$, and the device evaluates the expression e_r to the value v_r to be returned;

3. the device evaluates the expression e_s to obtain a nvalue $\underline{\mathbf{w_s}}$ to be sent to neighbors such as δ', that will use their corresponding value $\underline{\mathbf{w_s}}(\delta')$ in their next execution round.

As a shorthand,

$$exchange(e_i, (\underline{\mathbf{n}}) \Rightarrow \mathrm{return}\, e \,\mathrm{send}\, e)$$

can be written as

$$exchange(e_i, (\underline{\mathbf{n}}) \Rightarrow \mathrm{retsend}\, e)$$

according to the XC paper [5].

Two examples of reusable functions written in XC can be seen in Listing 3.1. There, $mux(cond, e_1, e_2)$ is a conditional expression that first evaluates $cond$, e_1, and e_2, and then returns the value of e_1 if $cond$ is true, or the value of e_2 otherwise. `mux` is useful to avoid breaking the alignment of the network by using conditionals, as explained in Section 3.1.4. In the examples provided, $\underline{senseDist}$ is a network-based sensor that returns the nvalue containing the distances to neighbors, abstracting over the way the device obtains the measurements, with $Infinity$ as its default value used for all other devices. `distanceEstimate` computes the minimum distance from a source using the distance sensor and the neighbors estimate $\underline{\mathbf{n}}$ of their minimum distance from the same source. `distanceTo` computes the minimum distance from a source determined by a boolean expression src, which is represented as a *gradient* with value:

- 0 in all the devices where src is true;
- the distance from the closest source in all the devices where src is false that are connected to at least one source;
- $Infinity$ otherwise.

Listing 3.1: Implementation of a network-wide gradient, called `distanceTo`, using the XC language.

```
def distanceEstimate( n : num): num {
    nfold(min,  n  + senseDist, Infinity)
}


def distanceTo(src: bool): num {
    exchange(Infinity, ( n ) => retsend mux(src, 0, distanceEstimate( n )))
}
```

3.1.4 Alignment

A program can execute multiple exchanges in a single round and XC ensures that messages are dispatched to corresponding exchange expressions using the concept of *alignment*. The corresponding exchange expressions are those that are located in the same position within the Abstract Syntax Tree (AST) and same stack frame, thus ensuring correct alignment in case of branches, function calls, and recursion [5]. As a consequence, the evaluation of an aggregate program implicitly builds a tree representation, called *value tree*. All the aligned devices replicate and then exchange the value tree with each other, including the device itself in the next rounds.

Conditionals such as $if\ (cond)\ e_1\ else\ e_2$ interfere with alignment because only the `exchange` operations in the same position within the AST and stack frame align [5]. As a consequence, `exchange` only aligns across devices that take the same branch of all the conditionals that are the parent or the ancestor of the `exchange` operation in the AST.

Alignment controls the evaluation of sub-expressions, in particular the evaluation of expressions involving nvalues, because in such expressions only aligned neighbors are considered. As a result, every `if` expression splits the network into two non-communicating sub-networks, with each device evaluating a different branch based on the condition [5]. These isolated sub-networks are also referred to as *sub-domains* or simply *domains*.

3.1.5 Formalization of XC

XC is formalized in the paper [5] in its syntax, operational semantics, and denotational semantics. The language takes inspiration from Meta Language (ML) and is a standard functional language with a classic Hindley-Milner type system. Its formalization makes XC type sound and deterministic once extended with *value-tree* typing and *configuration* typing [5].

3.1.6 Implementing FC primitives with exchange

Since FC primitives and expressiveness can be implemented through XC, XC inherits all the findings reported in the literature that hold for FC. These include eventual recovery and stabilization after transient changes [16], independence from the density of devices [17], real-time error tolerance, and convergence [18]. Furthermore, the list of benefits extends beyond these [5]. Additionally, XC opens the possibilities for writing programs not expressible with FC, thanks to the expressiveness of the `exchange` primitive which allows sending differentiated messages to neighbors. In FC, the concept of *field*, also called *neighboring value*, is defined as a neighbor-dependent value consisting of a map $\phi = \overline{\delta} \mapsto \overline{l}$ from neighbors to local values, which can be promoted to nvalue with any valid default value l. This preserves the behavior of programs written in FC but interpreted within XC [5].

FC presents three main primitives, all implementable through XC:

- `nbr`, used to access the neighbors' values [11];
- `rep`, used to compute a new value of an expression based on the result of the same expression in the previous round [11];
- `share`, used to efficiently access neighbors' values while computing a new value from the previous result with a single primitive [19].

The `nbr` primitive can be implemented as:

$$nbr(e : A) : \underline{A} = exchange(e, (\underline{n}) \Rightarrow \text{return } \underline{n} \text{ send } e)$$

The `rep` primitive can be implemented as:

$$rep(e_i : A)\{(x) \Rightarrow e_n\} : A = exchange(e_i, (\underline{x}) \text{ => retsend } e_n[x := self(\underline{x})])$$

The `share` primitive can be implemented as:

$$share(e_i : A)\{(\underline{x}) \Rightarrow e_n\} : A = self(exchange(e_i, (\underline{x}) \text{ => retsend } e_n))$$

3.2 Scala 3

Scala is a statically typed, general-purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. Scala 3 is the latest major release of the Scala language, a high-level programming language that combines object-oriented and functional programming paradigms. It is compiled using the *Dotty* compiler[2], which is based on the Dependent Object Types (DOT) calculus [20], while Scala 2 is compiled using the *scalac* compiler[3]. This section provides an overview of the main features of Scala 3, that are relevant for the re-design of ScaFi while also highlighting the differences between Scala 2 and Scala 3.

3.2.1 General considerations on Scala 3

Even though Scala 3 introduces breaking changes in the syntax from Scala 2, most of the code written in Scala 2 can be compiled with Dotty and is also binary compatible with Scala 2. The Scala community has exploited this possibility to avoid reimplementing a new standard library for Scala 3, using the Scala 2 standard library instead, rewritten to be cross-compiled for both Scala 2 and Scala 3.

Natively, Scala 2 and Scala 3 compile to Java Virtual Machine (JVM) byte-code, but Scala 3 also supports the JS and *LLVM* backends, which are used to compile Scala code to JavaScript and native code, respectively. These compiler

[2] `https://github.com/lampepfl/dotty`
[3] `https://github.com/scala/scala`

backends exist thanks to community-driven projects[4][5] [14]. The provided support for cross-platform distribution, together with the popularity of Scala for distributed systems development [21] and the flexibility of the upgraded language features for advanced internal DSL design, makes Scala 3 a valuable choice for a new ScaFi implementation based on XC.

3.2.2 Values in Scala

In Scala, every value has a type, which follows the type hierarchy explained in Section 3.2.10. When declaring a new variable or field as a container of a value, the type can either be explicitly declared or inferred by the compiler. Every declaration of a variable or field must be preceded by the `val` keyword for an immutable value or the `var` keyword for a mutable value. Immutable values cannot be reassigned, while mutable values can be reassigned, but their type cannot be changed. It is worth noting that `val` does not guarantee that the value itself is immutable, but only that the reference to the value cannot be changed.

3.2.3 New control syntax and significant indentation

Scala 3 introduces a new syntax for control expressions, as well as new rules that allow the indentation alone to replace the use of curly braces. Both these changes are aimed at making the code more readable and concise, sometimes noticeably closer to the natural language. For instance, in Listing 3.2, the `if` expression is written with the traditional syntax, while in Listing 3.3 the same expression is written using the new syntax. The same is true for the `for` expression, the `while` loop, and the `match` expression.

In cases where a code block consists of many lines that make it difficult to follow the indentation, the new syntax can be augmented with `end` statements, as demonstrated in Listing 3.4.

[4]Scala.js at `https://scala-js.org`
[5]Scala Native at `https://scala-native.org`

Listing 3.2: Examples of syntax using braces in Scala 2.

```scala
if (x < 0) {
  println("negative")
} else if (x == 0) {
  println("zero")
} else {
  println("positive")
}

val y = if (a < b) { a } else { b }


val ints = List(1, 2, 3, 4, 5)
for (i <- ints) println(i)

val doubles = for (i <- ints) yield i * 2


var z = 1
while (z < 3) {
  println(z)
  z += 1
}


val result = i match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}


class Person(var firstName: String, var lastName: String) {
  def printFullName() = {
    println(s"First name: $firstName")
    println(s"Last name: $lastName")
  }
}

val p = new Person("John", "Stephens")
```

Listing 3.3: Examples of syntax avoiding braces in Scala 3.

```scala
if x < 0 then
  println("negative")
else if x == 0 then
  println("zero")
else
  println("positive")

val y = if a < b then a else b


val ints = List(1, 2, 3, 4, 5)
for i <- ints do println(i)

val doubles = for i <- ints yield i * 2


var z = 1
while
  z < 3
do
  println(z)
  z += 1


val result = i match
  case 1 => "one"
  case 2 => "two"
  case _ => "other"


class Person(var firstName: String, var lastName: String):
  def printFullName() =
    println(s"First name: $firstName")
    println(s"Last name: $lastName")

val p = Person("John", "Stephens")
```

Listing 3.4: Example of syntax using `end` statements in Scala 3.

```scala
object test:
  val x = 0
  val i = 1

  if x < 0 then
    println("negative")
  end if

  val ints = List(1, 2, 3, 4, 5)
  for i <- ints do
    println(i)
  end for

  var z = 1
  while z < 3 do
    println(z)
    z += 1
  end while

  val result = i match
    case 1 => "one"
    case 2 => "two"
    case _ => "other"
  end result

  class Person(var firstName: String, var lastName: String):
    def printFullName() =
      println(s"First name: $firstName")
      println(s"Last name: $lastName")
    end printFullName
  end Person
end test
```

3.2.4 Traits and classes

Traits are a powerful feature that replaces Java's interfaces and abstract classes and were first introduced as a mechanism to organize behavior into small, modular units [22]. In Scala, traits can be used to define interfaces and to provide partial implementations and can be composed and mixed into other traits or classes. With Scala 3, traits are now able to have parameters like classes do, enhancing their expressiveness. Furthermore, Scala 3 introduces new rules for the instantiation of classes that enable the avoidance of the `new` keyword, as shown in Listing 3.3. This feature is called *universal apply methods* and is implemented by the generation of `apply` methods for classes when the user does not provide one, during the compilation. Another enabling factor is the special role given to methods named `apply`, which in Scala can be invoked without the need to specify the method name. When combined with companion objects, described in Section 3.2.6, it is often possible to replace auxiliary constructors in classes with multiple `apply` methods in the companion object, which is a typical pattern in Scala. Anonymous classes can also be used to instantiate a trait or abstract class by providing a concrete implementation of abstract methods on the fly. In addition, traits, classes, and singleton objects can be nested, and they can access each other's private members, just like in Java.

If the definition of something depends on another from a different package, it is possible to use the `import` keyword to import the needed definitions. `import` statements can be put at the beginning of a file or inside a block and can be used to import a single definition, a group of definitions, or all definitions from a package. Additionally, Scala supports import aliases to avoid name clashes. Since Scala 3, import aliases have a dedicated syntax, following the pattern `import A as B`, where `A` is the fully qualified original name and `B` is the alias.

An advanced example of trait usage can be found in Listing 3.5[6], which shows how to use traits in a service-oriented way, a design pattern that promotes the use of traits to define services and their dependencies and to compose services into a single class [23]. In the mentioned example, some advanced features of Scala are presented, such as abstract type members, nested traits, and *self-type annotations.*

[6]`https://docs.scala-lang.org/scala3/book/domain-modeling-oop.html`

The *self-type annotation* is a way to declare that a trait must be mixed into a class that extends another trait, and it is used to express dependencies between traits without resorting to inheritance. Self-type annotations allow the composition of traits to be more flexible and less coupled and delay the choice of trait order of linearization to the class that mixes them.

Object-oriented programming in Scala 3

In Scala, methods for classes, traits, enums, and objects can be defined using the `def` keyword, as shown in Listing 3.3. Abstract methods do not have a body and must be overridden by concrete subclasses, while concrete methods have a body and can be overridden by subclasses. Moreover, Scala allows a method with no arguments to be overridden by a field with the same name. Additionally, methods can serve as operators using infix notation, enabling invocation with a single argument without using the dot or parentheses, as demonstrated in Listing 5.12. This feature provides flexibility in defining custom operators and developing DSLs. Since Scala 3, the `infix` keyword must precede `def` to explicitly denote the intent of defining custom operators.

Even though method overloading is still possible, the Java generic type erasure rules apply in Scala too, and must be taken into account when defining overloaded methods. In Scala, it is often possible to avoid method overloading by using default arguments, which are arguments that are automatically assigned a value if no value is provided by the caller, as shown in Listing 3.8. Scala 3 improves the default way to handle most method invocation ambiguities with the `@targetName` annotation, allowing the specification of a unique name for the method once compiled to JVM bytecode.

Thanks to *automatic eta expansion*, methods can be used in place of function values, and its implementation has been improved in Scala 3 to be almost completely seamless.

One of the most important features of Scala is the ability to define *type members*, which are members of a class or trait that define types, and can be used as types in the same way as classes or traits, as shown in Section 3.2.9. Abstract type members prevent Scala traits' and classes' sources from growing in size both

Listing 3.5: Advanced example of usage of traits, know as service-oriented design. The code was taken from the Scala 3 book.

```scala
trait SubjectObserver:

  type S <: Subject
  type O <: Observer

  trait Subject:
    self: S =>
      private var observers: List[O] = List()
      def subscribe(obs: O): Unit =
        observers = obs :: observers
      def publish() =
        for obs <- observers do obs.notify(this)

  trait Observer:
    def notify(sub: S): Unit


object SensorReader extends SubjectObserver:
  type S = Sensor
  type O = Display

  class Sensor(val label: String) extends Subject:
    private var currentValue = 0.0
    def value = currentValue
    def changeValue(v: Double) =
      currentValue = v
      publish()

  class Display extends Observer:
    def notify(sub: Sensor) =
      println(s"${sub.label} has value ${sub.value}")



import SensorReader.*

// setting up a network
val s1 = Sensor("sensor1")
val s2 = Sensor("sensor2")
val d1 = Display()
val d2 = Display()
s1.subscribe(d1)
s1.subscribe(d2)
s2.subscribe(d1)

// propagating updates through the network
s1.changeValue(2)
s2.changeValue(3)
```

vertically and horizontally, as they can often replace generic type parameters with all the benefits of member inheritance, such as overriding and composition. Type parameters, as well as type members, can be constrained with *upper bounds* and *lower bounds*, using the operators `<:` and `>:`, respectively, such as in `Type <: Supertype` and `Type >: Subtype`. Additionally, generic types also support type variance control, which is used to specify how the subtyping relationship between two generic types is related to the subtyping relationship between their type arguments as explained in depth in the dedicated section below.

3.2.5 Algebraic Data Types

The union of the object-oriented and the functional programming paradigms within Scala has promoted the use of algebraic data types, a kind of composite type that resembles mathematical operations between types. Examples of such types are *sum types*, *product types*, and *intersection types*. In Scala 2 these can be implemented using *sealed traits*, *case classes*, and inheritance, respectively. However, since Scala 3, the syntax for defining these types has been simplified and clarified, as elaborated in the following paragraphs.

Sum types

Since Scala 2, sum types were enabled by the `sealed` modifier on traits or abstract classes, which prevents inheritance outside the file where the sum type is defined. This way, it is possible to define a finite set of subtypes to check for in pattern matching. Moreover, the ability to define a singleton instance with the `object` keyword inheriting from the sum type enables the use of sum types for implementing enumerations. With the introduction of Scala 3, the `enum` keyword simplifies the syntax in both the mentioned use cases, as shown in Listing 3.6. Additionally, with Scala 3, the `|` operator can be used for sum type options in type definitions, for matching any pattern of a list in pattern matching, and for defining discriminated union types of the form `A | B`.

Listing 3.6: Example of sum types in Scala 3.

```
// Scala 3 enum sum type
enum Option[+T]:
  case Some(x: T)
  case None

// Scala 2 sum type
sealed trait Option[+T]
case class Some[+T](x: T) extends Option[T]
case object None extends Option[Nothing]
```

Product types

Product types, supported since Scala 2, are enabled by the features provided by *case classes*, which are Scala's implementation of the concept of *records* in functional programming. Case classes are equipped with compiler-generated methods for pattern matching, equality, copying, and printing. In case classes, constructor parameters are public immutable fields by default, and the `copy` method can be used to create a new instance with specified fields altered. An example of a case class can be found in Listing 3.6.

Intersection types

Since Scala 2, it is possible to define a type as the intersection of other types using the `with` keyword, which is also used for mixin composition. Mixin composition is the practice of combining multiple traits into a class, resembling multiple inheritance, but with the compromise of avoiding the diamond problem through linearization [24]. Mixin composition is employed in ScaFi to declare dependencies of programs, as shown in Listing 4.1.

Starting with Scala 3, the `with` keyword is deprecated for type definitions, such as in method arguments, in favor of the `&` operator. This operator is also used for pattern matching, as in Listing 5.11, within the `distanceTo` signature. Although deprecated for type definitions, the `with` keyword is still used for mixin composition in the definition of traits and classes.

Listing 3.7: Example of a singleton object in Scala 3.

```scala
import scala.math.*

class Circle(radius: Double):
  import Circle.*
  def area: Double = calculateArea(radius) // can access the private method

object Circle:
  private def calculateArea(radius: Double): Double =
    Pi * pow(radius, 2.0)

val circle1 = Circle(5.0)
circle1.area   // Double = 78.53981633974483
```

3.2.6 Singleton objects

In Scala, the `object` keyword is used to define singleton objects which are classes with only one instance. Singleton objects serve as a replacement for Java's static methods and fields and can be used to define utility methods and constants, as well as to implement the *companion object* pattern. This pattern promotes the use of a singleton object to hold methods and fields that are not specific to any instance of a class but are still related to the class itself. A class with a companion object can access the private members of the companion object, and vice versa, provided that they share the same name and are defined in the same file. Moreover, singleton objects can be used to implement traits to create *modules*, such as factories for collections. An example of a singleton object used as a companion of a class can be found in Listing 3.7.

3.2.7 Functional programming with Scala 3

Scala offers many features typical of Functional Programming (FP) languages, including *lambdas*, higher-order functions (i.e., functions that accept or return other functions), *currying*, algebraic data types, and a standard library of immutable collections. Lambdas, also known as anonymous functions, can be treated as any other value, thus passed as arguments, or returned as results. In Scala, lambdas are defined using the `=>` operator, which is also used for defining function types such as `A => B`, where `A` denotes the input type and `B` denotes the output type. In some cases, it is possible to write lambdas with a more concise syntax, using

Listing 3.8: Example of extension methods usage in Scala 3.

```scala
case class Circle(x: Double, y: Double, radius: Double = 1.0)

extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2

val circle = Circle(0, 0, 5)
println(circle.circumference) // 31.41592653589793

val circle2 = Circle(0, 0) // radius is 1.0, the default value
println(circle2.circumference) // 6.283185307179586
```

the _ placeholder for the input argument, as shown in Listing 6.2.

Scala 3 introduces new varieties of function types:

- *dependent function types*, where the result type can depend on the function's parameters, such as type members (an example can be found in Listing 5.11);
- *polymorphic function types*, that accept type parameters, explained in Section 3.2.9;
- *context function types*, that accept only context parameters, explained in Section 3.2.8.

Following the principles of FP, domain models should be immutable and deprived of behavior. More specifically, the behavior should be defined in terms of pure functions implemented in modules as extension methods, which are methods that can be added to existing types without modifying their source code. Starting with Scala 3, extension methods have obtained a dedicated syntax that avoids the cumbersome use of implicit classes, as shown in Listing 3.8. However, some additional attention is needed when extension method names overlap with class methods, as the compiler prioritizes class methods, potentially shadowing conflicting extension methods. In such cases, it may be necessary to invoke the extension explicitly from the instance that provides it, passing the extended instance as the first argument.

Currying is the process of transforming a function that accepts multiple arguments into a sequence of functions, each taking a subset of the arguments, thus

Listing 3.9: Example of currying and partial application in Scala 3.

```scala
// Currying a function
def add(x: Int)(y: Int): Int = x + y

// Partial application of a curried function
val addTwo = add(2) _

// Usage
val result = addTwo(3) // result = 5
```

simplifying the function's usage with partial applications. An example of currying and partial application of a function is present in Listing 3.9.

3.2.8 Contextual Abstractions

In Scala, contextual abstractions derive from the core concept of *term inference*, which is the ability of the compiler to synthesize a "canonical" term for a given type. Examples of features enabled by term inference are *extension methods*, *type classes*, *context parameters*, and *context bounds*. While in Scala 2 almost every feature related to contextual abstractions was enabled by the `implicit` keyword, Scala 3 contextual abstractions have been redesigned to be more explicit on the intent of their usage, with the introduction of new ad-hoc keywords for each use case: `given`, `using`, and `extension`. The `given` keyword is used to define a *given instance*, which is a value to be passed as a *context parameter* in methods or constructors that require one. The `using` keyword defines such *context parameters*, which are method parameters requiring a *given instance* available in scope and correspond to implicit parameters of Scala 2. If given instances are not passed explicitly as context parameters, the compiler performs *term inference* to search for *given instances* in scope to fulfill *context parameters* at every method invocation. Starting with Scala 3, context parameters can be anonymous, and given instances can be abstract. The `extension` keyword is now necessary to define *extension methods*, enabling the addition of methods to existing types without altering their source code, as explained in Section 3.2.7. To retrieve the value of an anonymous context parameter of type `T` Scala 3 provides the `summon[T]: T` method, replacing Scala 2's `implicitly[T]` method.

Context bounds in the form `T: Type` are syntactic sugar for context parameters with a more concise syntax, that gets desugared by the compiler to a context parameter of type `Type[T]`. For example, in Listing 5.11, `N:Numeric:UpperBounded` is desugared to two new context parameters in the form `(using Numeric[N], UpperBounded[N])`. An abstract class like `Type[T]` designed to add behavior to any closed data type without sub-typing is called a *type class*.

Given instances can be imported and exported using the `import` and `export` keywords. However, in Scala 3, *given imports* and *given exports* require the `given` keyword, even where a wildcard is used, enhancing clarity regarding their origin within the current scope. Moreover, Scala 3 allows for anonymous concrete given instances, for which the compiler synthesizes a name automatically. This feature is useful to avoid polluting the namespace with names that are not meant to be used directly by the user, because often given instances are imported by their type instead of by their name.

Starting with Scala 3, implicit conversions are defined by providing given instances of the type `Conversion[From, To]` and must be enabled with a compiler flag to prevent warnings when conversion is silently applied before passing an argument to a function call. By default, the Scala compiler provides implicit conversions for primitive types, such as `Int` to `Long`.

3.2.9 The Scala 3 type system

Scala, being a statically typed language, has all the benefits of early error detection, better performance, and robust tooling support. However, it also provides a flexible environment typical of dynamically typed languages, thanks to features like type inference, type parameters, and type members. Built upon a variant of the Hindley-Milner type system, Scala's type inference system supports subtyping, generics, and type bounds, as Java does. Additionally, Scala offers many type system features absent in Java, such as *type variance*, *type aliases*, *type members*, *covariant and contravariant overriding*, *higher-kinded types* Furthermore, starting with Scala 3, the type system features *opaque type aliases*, *structural types*, *dependent function types*, improved *type lambdas*, *polymorphic function types*, *context function types*, and *match types*. In the following paragraphs, the most relevant features of the Scala 3 type system are explained.

Listing 3.10: Example of opaque type alias and covariant override in Scala 3.

```scala
class A:
  def method[T](it: Seq[T]): Iterable[T] = it

class B extends A:
  opaque type X[T] = Seq[T] // opaque type alias
  override def method[T](it: Seq[T]): X[T] = // X is subtype of Iterable
    it.reverse

val b = new B()
val res1: B#X[Int] = b.method(Seq(1, 2, 3))
println(res1) // prints List(3, 2, 1)

val res2 = b.method(res1) // Found: (Playground.res1 : Playground.B#X[Int])
    Required: Seq[Any]
```

Type variance allows specifying how the subtyping relationship between two generic types is related to the subtyping relationship between their type arguments. In Scala, the variance of a type parameter can be declared with the `+` and `-` symbols, which are used to declare a type parameter as covariant or contravariant, respectively, else the type parameter is invariant by default. A usage example of a variant annotation is present in Listing 3.6.

Type aliases are used to define new names for existing types, which are often more descriptive or shorter than the original names. Type aliases can be parameterized and can be recursive. Their syntax is the same for defining type members, using the `type` keyword, as shown in Listing 3.10. Opaque type aliases are a new feature of Scala 3 that allows defining a type alias that is not interchangeable with its underlying type, hiding it from consumers.

Covariant overrides enable the overriding of superclass methods with methods that have more specific return types, as shown in Listing 3.10.

Higher-kinded types allow the definition of types and methods that work on generic types regardless of their actual type arguments, only requiring a fixed arity of type parameters. Scala types are partitioned into kinds based on the top type of which it is a subtype, such as `Any`, `[+X] =>> Any`, `[X, +Y] =>> Any`. Higher-kinded types have a kind that counts at least one type arrow, such as `List` of kind `[+X] =>> Any`, `Option` of kind `[+X] =>> Any`, and `Map` of kind `[X, +Y] =>>`

Listing 3.11: Example of match types in Scala 3.

```
type Elem[X] = X match
  case String => Char
  case Array[t] => t
  case Iterable[t] => t
```

`Any`, but potentially even a type whose kind is `[X] =>> [Y] =>> Any`, such as a type class for a type constructor. Scala 3 adds support for *kind polymorphism*, allowing the definition of type parameters that accept types of any kind, through the special syntax `T <: AnyKind`.

Type lambdas are simply anonymous type constructors, that starting from Scala 3 have a new concise syntax thanks to the type operator `=>>`. For instance, `[X, Y] =>> Map[Y, X]` is a binary type constructor that maps arguments `X` and `Y` to the type `Map[Y, X]`.

Context functions are functions that accept only context parameters as input. Starting with Scala 3, context functions can be treated as values thanks to context function types, which can be distinguished from standard function types by the presence of the `?=>` operator in place of the `=>` operator.

Match types are conditional type aliases that allow the definition of a type as the result of a pattern match between types, and are available only in Scala 3. An example of match types can be found in Listing 3.11.

3.2.10 Explicit nulls and the Scala 3 type hierarchy

The Dotty compiler offers experimental features that alter the language in various ways. Two examples of experimental features relevant to ScaFi-XC are *explicit nulls* and *multiversal equality*.

Explicit nulls are enabled by the "`-Yexplicit-nulls`" flag, which modifies the type hierarchy of Scala making reference types non-nullable. With explicit nulls disabled, the type system looks like the one in Java, pictured in Figure 3.2. Conversely, with explicit nulls enabled, the type system resembles Kotlin's, where

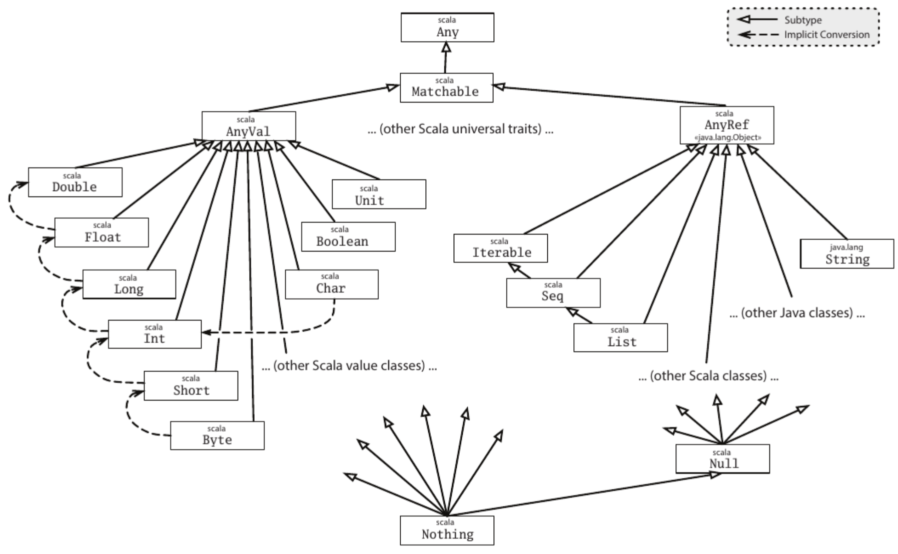


Figure 3.2: Scala type hierarchy with explicit nulls disabled.

null safety is a key part of the language, as depicted in Figure 3.3. Nullable values with that option enabled can still be defined using sum types, such as in `Type | Null`. Scala 3 provides the extension method `.nn` to convert a nullable value to a non-nullable one through casting. If the value was `null` at runtime, this forced conversion results in a *NullPointerException*.

3.2.11 Multiversal Equality

The Scala 3 compiler with the "`-language:strictEquality`" flag enabled forbids *universal equality*, that allowed comparing any two values regardless of their types using the `==` operator, which under the hood invoked the `equals` method. With that flag enabled, universal equality is replaced with *multiversal equality*, which allows comparing two values of type `X` and `Y` only if a given instance of `CanEqual[X, Y]` can be found. Implementing `CanEqual[X, Y]` instances can be automated with *type class derivation*, a feature of Scala 3 that allows the compiler to synthesize given instances following rules defined by the user, often based on compositions of algebraic data types.

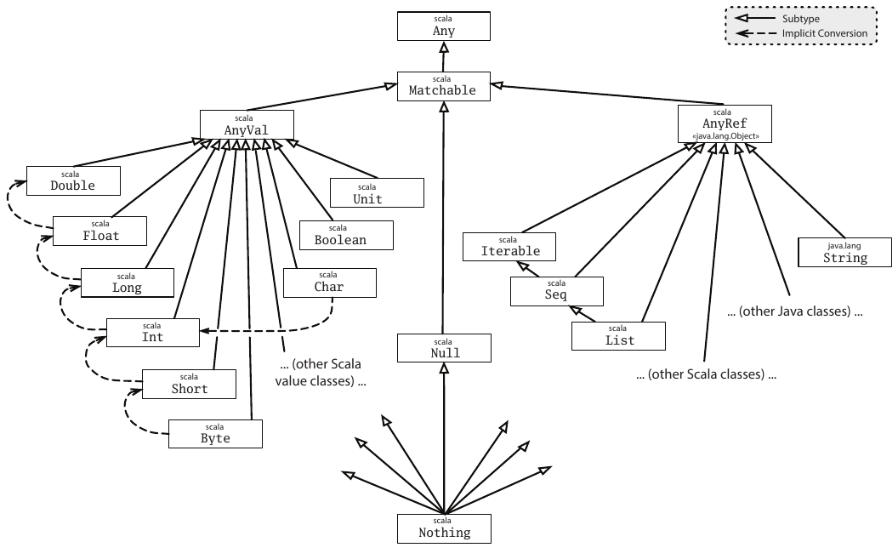


Figure 3.3: Scala type hierarchy with explicit nulls enabled.

Chapter 4

Analysis

This chapter defines the scope, requirements, and use cases of ScaFi-XC, based on the expectations of the stakeholders, consisting of ScaFi developers and researchers. A priority analysis is conducted using the MoSCoW method, ensuring that the work stays focused on delivering key features while meeting technical constraints and user preferences.

4.1 Requirements analysis

Since the very beginning, ScaFi-XC aimed to redesign ScaFi using Scala 3 to improve the quality of the code, including but not limited to readability, maintainability, and reusability. This time, the implementation is based on XC as the theoretical foundation, through which the FC constructs could be implemented.

The project started interviewing the stakeholders, including developers and researchers who developed the original ScaFi and still use it, in order to understand their needs and expectations.

The *key users* identified for the library are:

- *End users*, that are the developers who will use the library to implement their aggregate programs;
- *Library developers*, who will extend the library with new features, constructs, and syntax;

- *Researchers*, who will experiment with the calculus foundations and could benefit from reusing existing libraries.

During the stakeholder interviews, a comprehensive list of functional and technical requirements emerged, which were then carefully curated and prioritized through extensive discussions and feedback sessions.

Functional requirements are the features that the library must provide to the users. The following are the most relevant functional requirements identified:

F.1 redesign and implement a new Application Programming Interface (API) for the `core` packageSection 4.2;

F.2 redesign and implement the `tests` package with acceptance tests, easy to read and understand, and that can be used as examples;

F.3 use the XC as the foundation of the (default) implementation of constructs, while still providing a FC based API;

F.4 develop an *Alchemist incarnation*[1] [25], enabling ScaFi-XC programs to run on the well-tested and widely used Alchemist simulator;

F.5 develop a minimal, pure Scala 3 simulator to run tests and examples without the need for external dependencies;

F.6 provide a new API for the `core` package that allows developers to import arbitrary ScaFi libraries and constructs into their programs without conflicts, in a seamless way;

F.7 prefer keeping the original, abbreviated names for core constructs like `nbr` and `rep`, as they are widely adopted and recognized within the community, promoting consistency and familiarity;

F.8 conduct experiments with Scala 3 to explore and achieve new compile time features for ScaFi, to enhance code quality and elevate the overall user experience.

[1]`https://github.com/AlchemistSimulator/Alchemist`

Technical requirements are the constraints and guidelines that the library must follow to ensure the quality of code and user experience.

T.1 use Scala 3 as the host language to leverage its advanced features and enhancements;

T.2 enable quality options on the Scala 3 compiler such as explicit nulls (Section 3.2.10) and multiversal equality (Section 3.2.11);

T.3 employ Simple Build Tool (SBT) as build system;

T.4 cross-build the project for *scala-js*[2];

T.5 cross-build the project for *scala-native*[3];

T.6 lint the code with *scalafix*[4] and/or *scalafmt*[5];

T.7 avoid using third-party libraries for the `core` package dependencies.

Following the identification of requirements, a comprehensive discussion and prioritization process was undertaken, guided by the Must, Should, Could, Won't (MoSCoW) method. The results are outlined in detail in Table 4.1.

Finally, before starting with the design of the solution, given that the project is a redesign of an existing library, the requirements were compared with the existing ScaFi library to identify the differences and similarities.

4.2 ScaFi

The ScaFi repository[6] is organized in modules, as illustrated in Figure 4.1. For most of the use cases, only a small subset of the modules should be imported. The ScaFi-XC project primarily focuses on redesigning and implementing the `core` package, implicitly including `commons` module, as well as the `simulator` and `tests` packages. However, it should be noted that the simulator implementation planned

[2]`https://www.scala-js.org`
[3]`https://scala-native.org`
[4]`https://scalacenter.github.io/scalafix`
[5]`https://scalameta.org/scalafmt`
[6]`https://github.com/scafi/scafi`

Table 4.1: Requirements prioritization.

Requirement	**MoSCoW**	**Priority**
F.1	must	high
F.2	must	high
F.3	must	high
F.4	could	low
F.5	must	high
F.6	should	average
F.7	should	average
F.8	won't	low
T.1	must	high
T.2	should	high
T.3	must	high
T.4	could	low
T.5	could	low
T.6	should	low
T.7	should	average

for this project will be a minimal version written in pure Scala 3, as opposed to the existing `simulator` module, which is a complex and feature-rich suite of tools for simulating aggregate programs in space and time. Porting the remaining modules, which encompass various Graphical User Interface (GUI) implementations, demos, and integration with *Akka*[7] for actual deployment of real-world aggregate applications, is out of the scope of ScaFi-XC, and will be addressed in Chapter 8.

Within the `core` package of the `core` module, the `Core` trait serves as the root of the family of traits. It defines the set of abstract type members and traits, such as `Context`, that represents the context of the aggregate program execution. A trait `Language` defines a `Constructs` trait with the core syntax of FC, such as `rep`, `nbr`, and `foldhood`, plus additional utilities such as `mid(): ID` for retrieving the device own identifier and `sense[A](name: CNAME): A` for sensing a value from the environment. Furthermore, the `RichLanguage` trait extends `Language` with additional constructs, such as `branch`, `foldhoodPlus`, and `maxHood`. The `Semantics` trait implements the FC constructs, the value tree building, and the *foldhood context semantics*. More insights on the foldhood semantics are provided

[7] `https://akka.io`

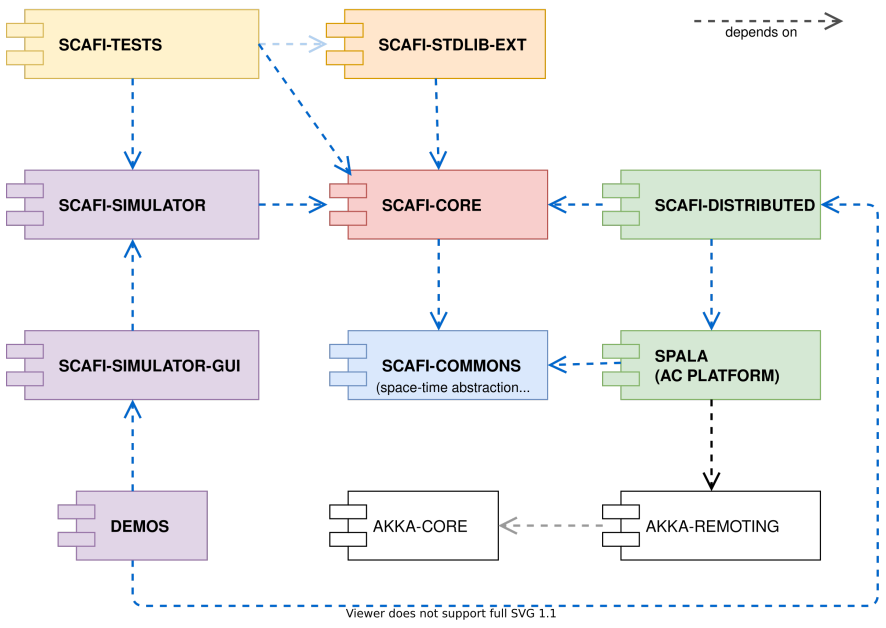


Figure 4.1: ScaFi project organization.

in Section 4.2.1. In addition to that, the `core` module contains other packages, such as `lib` offering standard libraries, and the utilities to provide a fully-fledged aggregate program execution environment called *incarnation*. The `distributed` and `spala` modules enable ScaFi to run on real-world distributed applications by leveraging integration with the *Akka framework*[8] and with other supportive libraries such as *Java Swing* and the *Play Framework*[9]. Lastly, the `simulator` and affiliated modules provide an advanced simulation suite featuring spacial and temporal simulation alongside multiple GUI options, including a tridimensional renderer.

To use a library component in ScaFi, the user must mix in the library trait with the `AggregateProgram` base class, as shown in Listing 4.1. As a consequence, all the transitive dependencies of the mixed-in libraries become visible within the scope of the program, necessitating careful construction of libraries to avoid conflicts.

Once the program is defined within the `main` method of the `AggregateProgram`

[8]`https://akka.io`
[9]`https://www.playframework.com`

Listing 4.1: Using a library component in ScaFi.

```
class MyProgram extends AggregateProgram with BlockG with BlockS { /*...*/ }
```

extension, the user can execute the program on the simulator through the `Launcher`. This specialized extension of Scala's `App` trait provides a `launch` method to start the simulation with one of the available GUIs.

ScaFi-XC aims to redesign and implement the `core` package to maintain the feasibility of integrating an equally sophisticated simulation suite while improving the overall quality of the code and the user experience.

4.2.1 Foldhood semantics in ScaFi

ScaFi employs the concept of a stateful "virtual machine" as its core to monitor the context of expression evaluation within the aggregate program. The virtual machine not only tracks nested invocations of core constructs, but also the scope of foldhood expressions. These are evaluated for each neighbor of the device, and the results of the evaluations are combined into a single value. By adopting this approach, ScaFi avoids the definition of an explicit type for fields, resulting in a clean syntax but leaving the awareness of the underlying field-like nature of the foldhood semantics to the user. For instance, the invalidity of expressions like `nbrnbrx` within a foldhood is something that the user must be aware of, as it is not enforced by the compiler or the virtual machine.

Chapter 5

Design

The design of ScaFi-XC has been divided into three main phases. In the first part, four design prototypes have been developed in order to determine the optimal user experience for three key users: the *program developer*, the *library developer*, and finally *a new foundation researcher*, representing a novel addition to the use cases of an aggregate programming library.

In the second part, the final version of the DSL has been designed, taking the best features from the prototypes and integrating them into a new core DSL for ScaFi-XC. Additionally, an execution context for aggregate programs made with ScaFi-XC, called “engine”, has been designed.

In the third and final part, the design process concerned the execution engine and the simulator.

The first and second part cover requirements *F.1*, *F.3*, *F.6*, *F.7*, and *T.1*, while the third covers requirement *F.5* from Section 4.1.

5.1 Designing a scalable internal Domain Specific Language

The process of designing a new core DSL for ScaFi-XC has been carried out through rapid prototyping of four competing designs of different DSLs, each coming with a set of advantages and disadvantages, highlighted using code snippets for every key user. Prototipation has been necessary to explore the design space and to understand the trade-offs between different design options, as well as to test in practice the interactions of different combinations of language features. For each of the four prototypes, a brief description of the design choices and the programming experience is provided, followed by the final design described in Section 5.2. Each prototype is named after the Scala 3 feature it is mainly based on, and each aims to separate the definition of the syntax from the definition of the semantics, and separate the definition of the semantics from the actual implementation. This way, more than one semantics can be defined for the same syntax, and more than one implementation can be defined for the same semantics, allowing for more flexible customization and composition. While the XC [5] is the only semantics considered for implementation within the scope of ScaFi-XC, the design should be flexible enough for potential future "calculi" to be implemented as alternative foundations. It is worth noting that there exists a subtle distinction between the semantics and syntax of the same formal calculus in practice. In the case of XC, both the syntax and the semantics consist of two methods, `branch` and `exchange`. The difference lies in the role of the two methods in the two contexts: in the syntax, they are constructs that the programmer uses to write programs, and should be treated as an API at all effects, while in the semantics, they are constructs that the programmer uses to implement all the supported syntaxes API, meaning they are not part of an API used by aggregate programs and are focused on being as complete and simple to implement as possible. For instance, the `exchange` method of the `ExchangeCalculusSyntax` trait is focused on its usage experience, attempting to imitate the syntax provided in the paper, meanwhile the `xcexchange` method of the `ExchangeCalculusSemantics` has `protected` visibility and its signature provides only the most complete and expressive version of the `exchange` primitive presented in the paper [5].

Each of the four prototypes retains the separation between syntax definitions, semantic definitions, syntax support declaration, and semantics implementations, and considers the programmer experience of all the three key users. For brevity, only the most relevant advantages and disadvantages are highlighted, and the full code of the prototypes is available in the project repository git history, under the git tag `experiments`[1], after which the code got removed to avoid confusion with

[1]`https://github.com/ldeluigi/scafi-xc/tree/experiments`

the final design.

5.1.1 Prototype 1: Extension Methods

In this design, an `AggregateFoundation` base trait defines a common syntax for all the aggregate programming foundations, such as the existence of a type called `AggregateValue[T]` that represents a collection of values coming from neighboring devices, including self. The `AggregateFoundation` trait also defines a set of *abstract extension methods* that provide basic functionalities for aggregate values, such as *lifting* for composition and mapping, *folding* for reduction, methods for retrieving the value for the current device or exclude the current device, mimicking the functionalities of the `foldhood` and `foldhoodPlus` of ScaFi, as shown in Listing 5.1. In the example, the source of the `Liftable` and `Foldable` type classes is included for completeness, but they are not part of the `AggregateFoundation` source file as they are located in the `commons` module.

By defining an abstract type member `AggregateValue[T]`, semantics like XC can override it to model any kind of specific interface, such as *NValues*, on top of which they can provide any additional behavior and syntax, following the pattern of *family polimorphism*. Implicitly, this design abandons the "field-transparent" semantics of the `foooldhood*` methods of ScaFi in favor of having explicit *field* types, similarly to FCPP (Section 2.2) and the XC DSL experiment (Section 2.4). Nevertheless, a semantics design that replicates that lost feature can still be implemented with an extension of `AggregateFoundation` that provides a `foldhood` construct that works the same as the `foldhood*` methods of ScaFi.

Aggregate Semantics such as `ExchangeCalculusSemantics` are defined as a trait that extends `AggregateFoundation`. A semantics trait can provide a concrete type for the abstract type member `AggregateValue[T]`, even if the type refers to a trait and not a concrete implementation.

For instance, the `ExchangeCalculusSemantics` provides a concrete implementation for `AggregateValue[T]` corresponding to `NValues[ID, T]`, where `ID` is an abstract type member for device identifiers. Additionally, the semantics provides an abstract given instance of `CanEqual[ID, ID]` to provide equality comparisons between device identifiers, as well as the core XC constructs: `xcexchange` and

Listing 5.1: Prototype 1 - Aggregate Foundation and helper type classes.

```scala
// Foldable.scala
trait Foldable[F[_]]:
  extension [A](a: F[A]) def fold[B](base: B)(acc: (B, A) => B): B

// end Foldable.scala

// Liftable.scala
trait Liftable[F[_]]:
  extension [A](a: F[A]) def map[B](f: A => B): F[B]

  def lift[A, B, C](a: F[A], b: F[B])(f: (A, B) => C): F[C]

  def lift[A, B, C, D](a: F[A], b: F[B], c: F[C])(f: (A, B, C) => D): F[D]

object Liftable:
  def lift[A, B, C, F[_]: Liftable](a: F[A], b: F[B])(f: (A, B) => C): F[C] =
    summon[Liftable[F]].lift(a, b)(f)

  def lift[A, B, C, D, F[_]: Liftable](a: F[A], b: F[B], c: F[C])(f: (A, B, C) =>
      D): F[D] =
    summon[Liftable[F]].lift(a, b, c)(f)

// end Liftable.scala

// AggregateFoundation.scala
trait AggregateFoundation:
  type AggregateValue[T]

  given lift: Liftable[AggregateValue]
  given fold: Foldable[AggregateValue]
  given convert[T]: Conversion[T, AggregateValue[T]]

  // Default builtins
  extension [T](av: AggregateValue[T])
    def onlySelf: T

    def withoutSelf: AggregateValue[T]

    def nfold[B](base: B)(acc: (B, T) => B): B =
      av.withoutSelf.fold(base)(acc)

end AggregateFoundation
// end AggregateFoundation.scala
```

`xcbranch`, corresponding to the `exchange` primitive and the domain branching behavior of XC. These core constructs of the semantics are protected in visibility because they are meant to be invoked only through a facade that corresponds to one or more syntax of a *calculus*, whereas abstract given instances are public as they are meant to be available to libraries that depend on a specific semantics. The syntaxes for both FC and XC are shown in Listing 5.2, and the XC semantics is shown in Listing 5.3. Finally, Listing 5.4 shows the implementation of the syntaxes in terms of the XC semantics.

One notable feature of having a facade defined using extension methods is that the compatibility layer between a semantic and a syntax is provided through the implementation of a given instance, much like a written proof that a syntax can be obtained extending a given semantics. The proof can `summon` other given instances of supported syntax in order to define a proof dependent on another proof, as shown in Listing 5.4. Another significant feature is the possibility to import dependencies and preferred syntax/facade with an `import` statement at the beginning of the file, instead of having to mix in traits like in ScaFi. Additionally, this design allows implementing a new library by simply writing new extension methods of a generic language `L` that extends `AggregateFoundation` and the required other syntaxes, as shown in Listing 5.6. This allows libraries to be singleton objects, imported where needed with a top-level `import` statement. One hidden feature of this design is the possibility to hide, by default, transitive library dependencies, something that ScaFi could not allow because it used a mixin composition where every library was a trait that mixed in other traits, thus exposing all the dependencies of the mixed-in traits.

Even though this design proves to be very flexible and extensible, it has a few drawbacks. The most important one is the impossibility of invoking an extension method without the "`.`", having at best to invoke constructs on `this`, as in Listing 5.5. The next prototypes focus on overcoming this limitation at any cost, even if it means losing some of the flexibility and clarity of the design, in order to provide empirical evidence of the trade-offs between the different design choices.

Listing 5.2: Prototype 1 - Syntax definitions.

```scala
// ExchangeCalculusSyntax.scala
trait ExchangeCalculusSyntax[L <: AggregateFoundation]:
  extension (language: L)
    def exchange[T](initial: language.AggregateValue[T])(
        f: language.AggregateValue[T] => (language.AggregateValue[T], language.
            AggregateValue[T]) |
          language.AggregateValue[T],
    ): language.AggregateValue[T]
// end ExchangeCalculusSyntax.scala

// FieldCalculusSyntax.scala
trait ClassicFieldCalculusSyntax[L <: AggregateFoundation]:
  extension (language: L)
    def nbr[V](expr: => language.AggregateValue[V]): language.AggregateValue[V]
    def rep[A](init: => A)(f: A => A): A
    def share[A](init: => A)(f: A => A): A
// end FieldCalculusSyntax.scala
```

5.1.2 Prototype 2: Context parameter in constructors

Even though the second prototype is based on the use of a context parameter that passes a semantics instance down to every construct invocation, the folding and lifting functionalities are provided through abstract given instances of type classes declaring extension methods, as per prototype 1. Libraries, as well as core syntaxes, are defined as classes and traits, respectively, that take a context parameter of type `L`, short for `Language`, that must be a subtype of `AggregateFoundation`, as shown in Listing 5.7. Libraries dependent on other libraries must either instantiate their dependencies or require a context parameter that provides them, as shown in Listing 5.8. This approach has the side effect that the type member `AggregateValue` present in the `AggregateFoundation` is seen as a different type for every dependent method of every library, thus making passing an aggregate value from a library method to another impossible. This forced each semantics, library, and syntax to have a generic type constructor parameter `AV[_]` that generalizes and uniformizes the type of the aggregate value for all the dependencies, making them compatible again. Listing 5.8 also shows how this design, with the additional cost of a static `import` for dependencies methods and given instance, allows to invoke constructs such as `branch` or `distanceTo` with having to use the `.` operator and the same applies to user programs.

Having to explicitly import the methods from dependencies inside every library

Listing 5.3: Prototype 1 - Exchange Calculus Semantics.

```scala
trait ExchangeCalculusSemantics extends AggregateFoundation:
  type ID
  given idEquality: CanEqual[ID, ID] = CanEqual.derived

  case class NValues[ID, +V](default: V, values: MapView[ID, V]):
    def apply(id: ID): V = values.getOrElse(id, default)

  override type AggregateValue[T] = NValues[ID, T]

  protected def xcbranch[T](cond: AggregateValue[Boolean])(th: => AggregateValue[T
      ])(
      el: => AggregateValue[T],
  ): AggregateValue[T]

  protected def xcexchange[T](init: AggregateValue[T])(
      f: AggregateValue[T] => (AggregateValue[T], AggregateValue[T]),
  ): AggregateValue[T]


  def self: ID // the id of the current device

  def neighbors: AggregateValue[ID] // the ids of the neighboring devices


  override given lift[ID]: Liftable[[V] =>> NValues[ID, V]] with
    extension [A](a: NValues[ID, A])
      override def map[B](f: A => B): NValues[ID, B] =
        NValues[ID, B](default = f(a.default), values = a.values.mapValues(f))

       override def lift[A, B, C](a: NValues[ID, A], b: NValues[ID, B])(f: (A, B)
          => C): NValues[ID, C] =
        ??? // omitted for brevity

      override def lift[A, B, C, D](a: NValues[ID, A], b: NValues[ID, B], c:
          NValues[ID, C])(
            f: (A, B, C) => D,
        ): NValues[ID, D] = ??? // omitted for brevity

  override def fold: Foldable[AggregateValue] = ??? // omitted for brevity

  override given convert[ID, V]: Conversion[V, NValues[ID, V]] with
    def apply(v: V): NValues[ID, V] = NValues[ID, V](default = v, values = MapView
        .empty)

  extension [T](av: NValues[ID, T])
    override def onlySelf: T = av(self)
    override def withoutSelf: NValues[ID, T] = av.copy(values = av.values.
        filterKeys(_ != self))

end ExchangeCalculusSemantics
```

Listing 5.4: Prototype 1 - Syntax implementations in terms of the exchange semantics.

```scala
given ExpressiveFieldCalculusSyntax[ExchangeCalculusSemantics] with
    extension (language: ExchangeCalculusSemantics)
      override def rep[A](init: => language.AggregateValue[A])(
          f: language.AggregateValue[A] => language.AggregateValue[A],
      ): language.AggregateValue[A] =
        language.exchange(init)(prev => f(prev.onlySelf))

      override def nbr[A](expr: => language.AggregateValue[A]): language.
          AggregateValue[A] =
        language.exchange(expr)(n => (n, expr))

      override def share[A](init: => language.AggregateValue[A])(
          f: language.AggregateValue[A] => language.AggregateValue[A],
      ): language.AggregateValue[A] =
        language.exchange(init.onlySelf)(f)

given ClassicFieldCalculusSyntax[ExchangeCalculusSemantics] with
    extension (language: ExchangeCalculusSemantics)
      override def nbr[V](expr: => NValues[language.ID, V]): NValues[language.ID,
          V] =
        summon[ExpressiveFieldCalculusSyntax[ExchangeCalculusSemantics]]
          .nbr(language)(expr)
      override def rep[A](init: => A)(f: A => A): A =
        summon[ExpressiveFieldCalculusSyntax[ExchangeCalculusSemantics]]
          .rep(language)(init)(nv => nv.map(f))
          .onlySelf
      override def share[A](init: => A)(f: A => A): A =
        summon[ExpressiveFieldCalculusSyntax[ExchangeCalculusSemantics]]
          .share(language)(init)(nv => nv.map(f))
          .onlySelf
```

Listing 5.5: Prototype 1 - Example usage by an aggregate program developer.

```scala
// the runtime defines the vm implementations
// while the engine defines context and exchange implementations
object AggregateProgramDeveloper extends ExchangeCalculusRuntime with Engine:

  // a program can either subclass an interpreter
  private class MyProgram extends ExchangeCalculusInterpreter:

    override def main(): Any =
      this.rep(1)(_ + 1)

  // the following tests simulate a single round of execution
  @main def test1(): Unit =
    val p = MyProgram()
    val c: Context = EngineContext()
    println(p(c))
```

Listing 5.6: Prototype 1 - Example usage by a library developer.

```scala
object AggregateLibraryDeveloper:
  // libraries can either be syntactic or semantic
  // semantic library make sense only for a specific semantics
  // syntactic library just rely on common syntax between semantics

  // if any library is needed, the import must be explicit:
  import it.unibo.scafi.xc.extensions.language.syntax.library.BasicGradientLibrary
      .*

  // example of syntactic library that works for many foundations:
  extension [L <: AggregateFoundation: ClassicFieldCalculusSyntax: BranchingSyntax
      ](lang: L)

    def distanceToGateways[D: Numeric: UpperBounded](
        local: Boolean,
        gateway: Boolean,
        distances: lang.AggregateValue[D],
    ): lang.AggregateValue[D] =
      lang.branch(local)(summon[UpperBounded[D]].upperBound)(lang.distanceTo[D](
          gateway, distances))

  // example of semantic library which only works for one foundation
  // in this case we need differentiated messages of xc calculus
  extension (language: ExchangeCalculusSemantics)

    def randomMessages(): language.AggregateValue[Int] =
      language.neighbors.map(_ => Random.nextInt())
end AggregateLibraryDeveloper
```

Listing 5.7: Prototype 2 - Field Calculus syntax definition.

```scala
trait ClassicFieldCalculusSyntax[AV[_], L <: AggregateFoundation[AV]](using L):
  def nbr[V](expr: AV[V]): AV[V]
  def rep[A](init: A)(f: A => A): A
  def share[A](init: A)(f: A => A): A
```

Listing 5.8: Prototype 2 - Example usage by a library developer.

```
object AggregateLibraryDeveloper:
  // libraries can either be syntactic or semantic
  // semantic library make sense only for a specific semantics
  // syntactic library just rely on common syntax between semantics

  // if any library is needed, the import must be explicit, and it must be
      instantiated

  // example of syntactic library that works for many foundations:
  class MyLibrary1[AV[_], L <: AggregateFoundation[AV]](using
      lang: L,
      dependency: BasicGradientLibrary[AV, L],
      branching: BranchingSyntax[AV, L],
  ):

    import branching._
    import dependency._
    import lang.{ _, given }

    def distanceToGateways[D: Numeric: UpperBounded](
        local: Boolean,
        gateway: Boolean,
        distances: lang.AggregateValue[D],
    ): lang.AggregateValue[D] =
      branch(local)(summon[UpperBounded[D]].upperBound)(distanceTo[D](gateway,
          distances))

  // alternative example of syntactic library that instantiate the dependencies
      itself:
  class MyLibrary1b[AV[_], L <: AggregateFoundation[AV]](using
      lang: L,
      calculus: ClassicFieldCalculusSyntax[AV, L],
      branching: BranchingSyntax[AV, L],
  ):
    private val dependency = BasicGradientLibrary[AV, L]()

    import dependency._
    import lang.{ _, given }
    import branching._

    def distanceToGateways[D: Numeric: UpperBounded](
        local: Boolean,
        gateway: Boolean,
        distances: lang.AggregateValue[D],
    ): lang.AggregateValue[D] =
      branch(local)(summon[UpperBounded[D]].upperBound)(distanceTo[D](gateway,
          distances))

  // example of semantic library that works only for a specific foundation:
  class MyLibrary2[ID](using lang: ExchangeCalculusSemantics[ID]):
    import lang._

    def randomMessages(): NValues[ID, Int] =
      neighbors.map(_ => Random.nextInt())
end AggregateLibraryDeveloper
```

of programs, added to the necessity of instantiating libraries, makes this design very cumbersome to use. The next prototype attempts to improve the usability by returning to libraries defined as singleton objects, having each construct take the context parameter that were in the library class constructor in this prototype.

5.1.3 Prototype 3: Implicit parameter in methods

This prototype design shares some similarities with the previous one, but it has a different approach to the problem of passing the semantics instance to the constructs, and to dealing with library dependencies. In this design, libraries are singleton objects, where each method takes a context parameter for the semantics and a context parameter for every syntax needed as a dependency, as shown in Listing 5.9. Alternatively, a library method can instantiate its syntax dependencies and only take a context parameter for the semantics, in a similar way to the previous prototype. Even though the issue of library dependencies is solved, syntax dependencies are still cumbersome to use and still require static imports inside the body of the methods to invoke the constructs without the `.` operator. In addition to that, `AggregateFoundation` still needs the generic aggregate value type constructor parameter to make syntax dependencies compatible with each other.

The last prototype represents a return to the original mixin-oriented design used in ScaFi and its purpose is closer to a comparison baseline rather than a design alternative, but it is still useful for the last design phase where the best features of all the prototypes will be cherry-picked and combined.

5.1.4 Prototype 4: Mixin composition

In this design, the `AggregateFoundation` trait is the same as in prototype 1 (see Listing 5.1), without the need for the generic type constructor for aggregate values, because syntaxes, semantics, and the foundation are meant to become part of the same type hierarchy, and a type member for aggregate values will be the same for all the mixed-in traits. For instance, given a semantics such as `ExchangeCalculusSemantics`, giving proof for the support for a syntax means to define a trait to be mixed in with the semantics, that implements the syntax in

Listing 5.9: Prototype 3 - Example usage by a library developer.

```scala
object AggregateLibraryDeveloper:
  // libraries can either be syntactic or semantic
  // semantic library make sense only for a specific semantics
  // syntactic library just rely on common syntax between semantics

  // if any library is needed, the import must be explicit, and it must be
      instantiated

  // example of syntactic library that works for many foundations:
  object MyLibrary:

    def distanceToGateways[AV[_], L <: AggregateFoundation[AV], D: Numeric:
        UpperBounded](using
        lang: L,
        branching: BranchingSyntax[AV, L],
        classicFieldCalculusSyntax: ClassicFieldCalculusSyntax[AV, L],
    )(
        local: Boolean,
        gateway: Boolean,
        distances: lang.AggregateValue[D],
    ): lang.AggregateValue[D] =
      import lang.convert, branching._
      branch(local)(summon[UpperBounded[D]].upperBound)(distanceTo[AV, L, D](
          gateway, distances))

  // example of semantic library that works only for a specific foundation:
  class MyLibrary2[ID](using lang: ExchangeCalculusSemantics[ID]):
    import lang._

    def randomMessages(): NValues[ID, Int] =
      neighbors.map(_ => Random.nextInt())
end AggregateLibraryDeveloper
```

Listing 5.10: Prototype 4 - Example usage by a library developer.

```
trait BasicGradientLibrary:
  self: AggregateFoundation with ClassicFieldCalculusSyntax with BranchingSyntax =
      >

  def distanceEstimate[D: Numeric: UpperBounded](
      estimates: AggregateValue[D],
      distances: AggregateValue[D],
  ): D = Liftable
    .lift(estimates, distances)(_ + _)
    .nfold(summon[UpperBounded[D]].upperBound)(
      summon[Numeric[D]].min,
    )

  def distanceTo[D: Numeric: UpperBounded](
      source: Boolean,
      distances: AggregateValue[D],
  ): D =
    rep[D](summon[UpperBounded[D]].upperBound)(n =>
      branch(source)(summon[Numeric[D]].zero)(distanceEstimate[D](n, distances)).
          onlySelf,
    )

  def hopDistance[D: Numeric: UpperBounded](source: Boolean): D =
    distanceTo(source, summon[Numeric[D]].one)
end BasicGradientLibrary
```

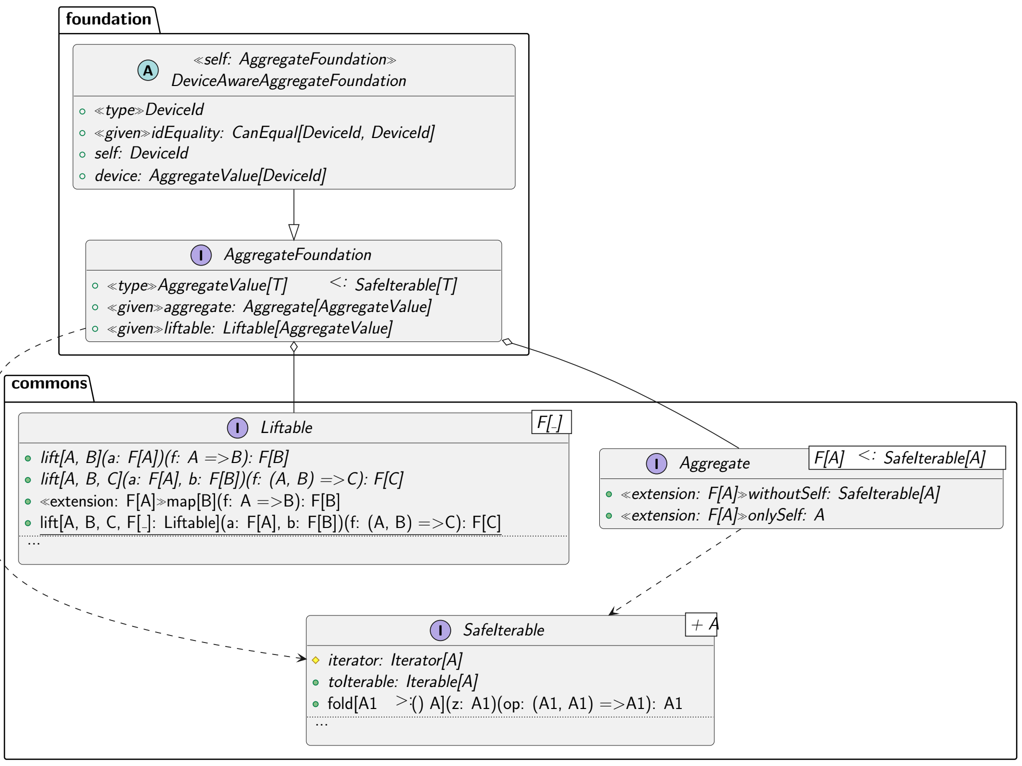
terms of the semantics. Libraries are defined, like in ScaFi, with mixin traits that declare their dependencies using self-type annotations, as shown in Listing 5.10.

The main advantage of this design is the possibility to invoke constructs without the `.` operator, as shown in Listing 5.10, while the main disadvantage is the need to inherit from all the transitive dependencies together in the program class, and having to honor a global construct naming consistency both in all the libraries and in the semantics implementation.

5.2 Final design of the core DSL

Taking inspiration from the best features of all the prototypes, the final design was developed and showcased with a presentation in front of the research group, which provided positive feedback on the resulting user experience. The final design consists of an `AggregateFoundation` similar to prototypes 1 and 4, with core syntaxes and libraries defined as traits for a mixin composition. The twist is that libraries are instead defined as singleton objects, able to be imported with a top-

Figure 5.1: Final design: UML diagram of the `AggregateFoundation`.

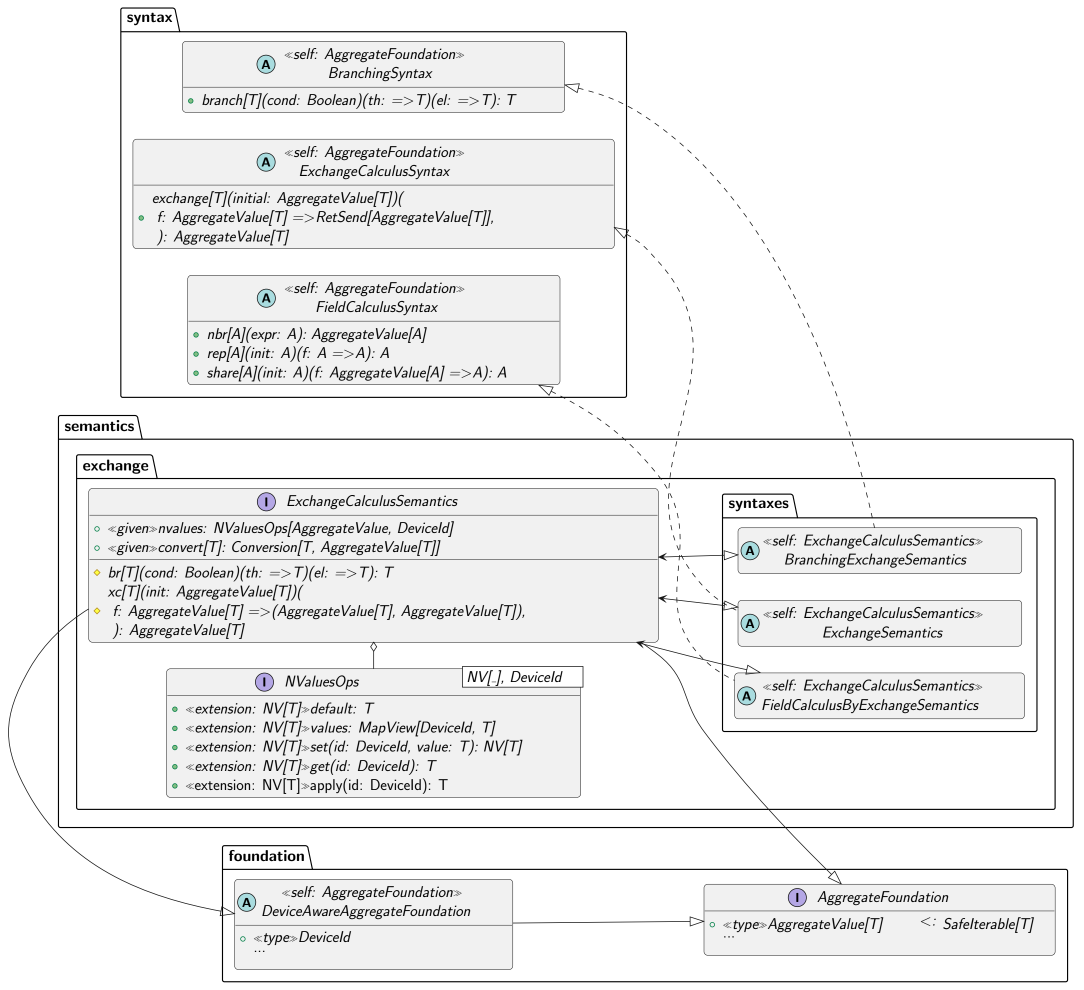

foundation
A
«self: AggregateFoundation»
DeviceAwareAggregateFoundation
«type»DeviceId
«given»idEquality: CanEqual[DeviceId, DeviceId]
self: DeviceId
device: AggregateValue[DeviceId]
I
AggregateFoundation
«type»AggregateValue[T] <: SafeIterable[T]
«given»aggregate: Aggregate[AggregateValue]
«given»liftable: Liftable[AggregateValue]
commons
I
Liftable
F[_]
lift[A, B](a: F[A])(f: A =>B): F[B]
lift[A, B, C](a: F[A], b: F[B])(f: (A, B) =>C): F[C]
«extension: F[A]»map[B](f: A =>B): F[B]
lift[A, B, C, F[_]: Liftable](a: F[A], b: F[B])(f: (A, B) =>C): F[C]
...
I
Aggregate
F[A] <: SafeIterable[A]
«extension: F[A]»withoutSelf: SafeIterable[A]
«extension: F[A]»onlySelf: A
I
SafeIterable
+A
iterator: Iterator[A]
toIterable: Iterable[A]
fold[A1 >: A](z: A1)(op: (A1, A1) =>A1): A1
...

level `import` statement, without having visibility on transitive dependencies and without having to mix them in together with the semantics in the program class. The disadvantage of this design would have been the different invocation syntax for library constructs versus core syntax constructs such as `nbr` and `rep`. This disadvantage has been overcome by defining a facade library for every core syntax, hiding the "`language.`" prefix necessary for invoking core syntax constructs, with the small cost of having to write a facade library for every future syntax developed by researchers. An Unified Modeling Language (UML) diagram of the final model for the foundation is shown in Figure 5.1.

Figure 5.2 shows the UML diagram of the `ExchangeCalculusSemantics` mixin composition, which is the only semantics implemented for the scope of this project.

Thanks to this design, the gradient construct, also known as `distanceTo`, can be defined to work with any aggregate semantics that supports the field calculus syntax, and thanks to Scala context bounds it can also be defined to work with any

Figure 5.2: Final design: UML diagram of the `ExchangeCalculusSemantics` mixin composition.


syntax
«self: AggregateFoundation» BranchingSyntax
branch[T](cond: Boolean)(th: => T)(el: => T): T
«self: AggregateFoundation» ExchangeCalculusSyntax
exchange[T](initial: AggregateValue[T])(f: AggregateValue[T] => RetSend[AggregateValue[T]],): AggregateValue[T]
«self: AggregateFoundation» FieldCalculusSyntax
nbr[A](expr: A): AggregateValue[A]
rep[A](init: A)(f: A => A): A
share[A](init: A)(f: AggregateValue[A] => A): A
semantics
exchange
ExchangeCalculusSemantics
«given» nvalues: NValuesOps[AggregateValue, DeviceId]
«given» convert[T]: Conversion[T, AggregateValue[T]]
br[T](cond: Boolean)(th: => T)(el: => T): T
xc[T](init: AggregateValue[T])(f: AggregateValue[T] => (AggregateValue[T], AggregateValue[T]),): AggregateValue[T]
NValuesOps
NV[_], DeviceId
«extension: NV[T]» default: T
«extension: NV[T]» values: MapView[DeviceId, T]
«extension: NV[T]» set(id: DeviceId, value: T): NV[T]
«extension: NV[T]» get(id: DeviceId): T
«extension: NV[T]» apply(id: DeviceId): T
syntaxes
«self: ExchangeCalculusSemantics» BranchingExchangeSemantics
«self: ExchangeCalculusSemantics» ExchangeSemantics
«self: ExchangeCalculusSemantics» FieldCalculusByExchangeSemantics
foundation
«self: AggregateFoundation» DeviceAwareAggregateFoundation
«type» DeviceId
...
AggregateFoundation
«type» AggregateValue[T] <: SafeIterable[T]
...

numeric type that supports an upper bound, such as `Double`. The resulting code for the `distanceTo` construct is shown in Listing 5.11, which resembles the syntax of the prototype DSL of Listing 2.4. The provided snippet demonstrates how this design effectively achieves the goals pursued by the other design prototypes:

- invocation of library and core constructs without the `.` operator;
- declaration of library dependencies with top-level imports, that hide transitive dependencies;
- generic definition of the constructs for high reusability;
- clean declaration of the core syntax dependencies, using a "`language`" context parameter and intersection types;
- possibility to resolve naming conflicts with import aliases;
- reuse of all the libraries and programs dependent on a set of syntaxes by re-implementing them through the next generation of aggregate programming calculi;

5.2.1 Design of the XC operational semantics

By abstracting common features of aggregate programming languages such as FC and XC into the `AggregateFoundation` trait, the `ExchangeCalculusSemantics` can focus on the peculiarities of the XC semantics, such as NValues and the `exchange` primitive. `AggregateFoundation` provides an abstract definition of an `AggregateValue` type, with the only feature to be iterable. The reason is that whatever the next aggregate calculus semantics will be, it is expected to provide some kind of notion of field, which should allow iterating over values from neighbors, including self. Additionally, `AggregateFoundation` provides the API to exclude the value for self from an `AggregateValue`, as well as to retrieve the value for self only, using extensions method inspired by the XC DSL experiment of Section 2.4. Referencing self has an important role in aggregate programs, and it was put here to provide all libraries and programs with that feature. Finally, `AggregateFoundation` provides the means to combine and map aggregate values

Listing 5.11: Final design - `distanceTo` implementation in the `GradientLibrary`.

```scala
import it.unibo.scafi.xc.abstractions.Liftable.*
import it.unibo.scafi.xc.abstractions.boundaries.UpperBounded
import it.unibo.scafi.xc.language.foundation.AggregateFoundation
import it.unibo.scafi.xc.language.syntax.FieldCalculusSyntax

import FieldCalculusLibrary.share
import CommonLibrary.mux
import Numeric.Implicits.*

object GradientLibrary:
  def distanceEstimate[N: Numeric: UpperBounded]
  (using language: AggregateFoundation)(
      neighboursEstimates: language.AggregateValue[N], // path dependent type
      distances: language.AggregateValue[N],
  ): N = lift(neighboursEstimates, distances)(_ + _).withoutSelf.min

  def distanceTo[N: Numeric: UpperBounded](using
      language: AggregateFoundation & FieldCalculusSyntax,
  )(source: Boolean, distances: language.AggregateValue[N]): N =
    share[N](summon[UpperBounded[N]].upperBound)(av =>
      mux(source)(
        summon[Numeric[N]].zero
      )(
        distanceEstimate(av, distances)
      )
    )
```

with the `lift` operator and the `map` extension method. This is a necessary difference from the formal calculi, where aggregate values such as NValues or fields allow to be treated as their underlying generic type, and are transparent when combined or mapped as if they were local values. For example, in XC, an expression e written to work with local values of type `T` can be used with an aggregate value of type `NValues[T]` without any modification. In Scala, instead, this is not possible, and operations on local values need to be *lifted* to work on aggregate values too. Unary operations have to be lifted too and work as the mapping function `f: A => B` for the `map` extension method available on an `AggregateValue[A]`. As mentioned in the previous section, abstracting common features into a common foundation for all the semantics allows the reuse of all the libraries and programs depending on those features, while also retaining the possibility to re-implement differently in a new aggregate calculus semantics.

To allow the definition, in the future, of an aggregate calculus without explicit device identifiers exposed in the API, all the features related to explicit device identifiers in the calculus have been grouped and modeled as an optional mixin

Listing 5.12: Supported syntaxes for invoking the `exchange` primitive.

```
def exchange[T](initial: AggregateValue[T])(
    f: AggregateValue[T] => RetSend[AggregateValue[T]]
): AggregateValue[T]

import RetSend.{ *, given } // necessary to enable some of the styles below

// To send and return the same value:
exchange(0)(value => f(value))
exchange(0)(value => retsend(f(value)))

// To send and return potentially different values:
exchange(0)(value => (f(value), f2(value)))
exchange(0)(value => ret (f(value)) send f2(value)) // infix style
exchange(0)(value => ret(f(value)).send(f2(value)))
exchange(0)(value => RetSend(f(value), f2(value)))
```

called `DeviceAwareAggregateFoundation`. The mixin defines an abstract type member `DeviceId` and the means to compare them with `==`, that is a given instance of `CanEqual[DeviceId, DeviceId]`. In addition to that, it provides two abstract methods, one called `self` that returns the device identifier of the current device, and another called `device` that returns an aggregate value of device identifiers, including self, which is always known thanks to the network and it doesn't need to be computed with, for example, a `nbr(self)` invocation.

Therefore, the `ExchangeCalculusSemantics` is left only with the definitions of its specific features, which are NValues additional operation on aggregate values, automatic conversion from local values and NValues, the core constructs `exchange`, here called `xc`, and `branch`, called `br`, to avoid conflicts with their counterparts defined in the implemented syntaxes. The signature of the `xc` method has been simplified to simplify its implementation, written only in its complete form with both the `return` and `send` values explicitly passed as a couple, whereas the `exchange` method of the syntax allows different call signatures to imitate the syntax of the paper [5], as shown in Listing 5.12.

The concrete implementation of the XC operational semantics is discussed partly in the next section, and partly in Section 6.1.

5.2.2 The Engine

An `Engine` has been designed to be able to execute aggregate programs that use an instance of the XC semantics trait as a context parameter. The `Engine` offers a method called `cycle` which implements all the steps to be executed in a single round of an aggregate program. The steps can be summarized into the following:

1. instantiation of the semantics, whose implementation is called `Context`, using information coming from the `Network`, such as inbound messages and the device identifier;

2. execution of the aggregate program, which is a function that takes the `Context` as a context parameter and returns a `Result`, which is the result of the evaluation of the aggregate program;

3. collection of the `Export`, which is a bundle containing all the outbound messages;

4. sending of the `Export` to the `Network`, which will deliver the messages to the intended recipients.

A `Context` is defined as an interface that takes inbound messages as input, called `Import`, gets altered by the aggregate program round of execution, and produces outbound messages as output, called `Export`. More information about the implementation of a `Context` for the XC semantics can be found in Section 6.1. An `Import` is defined as an alias for a map from device identifiers to generic values, which for the only implemented context correspond to *value trees*. A `ValueTree` is an Abstract Data Type (ADT) containing the values exchanged between devices coupled with their path in the AST of the aggregate program, as described in the XC paper [5]. Here a `Path` is defined as an alias for a `List` of generic tokens so that every implementation of a semantic can control which type of token defines a location inside the AST of the aggregate program. An `Export` instead is defined as an alias for a `MapWithDefault`, because it can send a dedicated message to a known neighbor and a default message to every other new neighbor of the device. The `Network` is responsible for properly dispatching the messages to the intended recipients and is pictured with UML in Figure 5.3.

Figure 5.3: The engine: UML diagram of the `Network` interface.

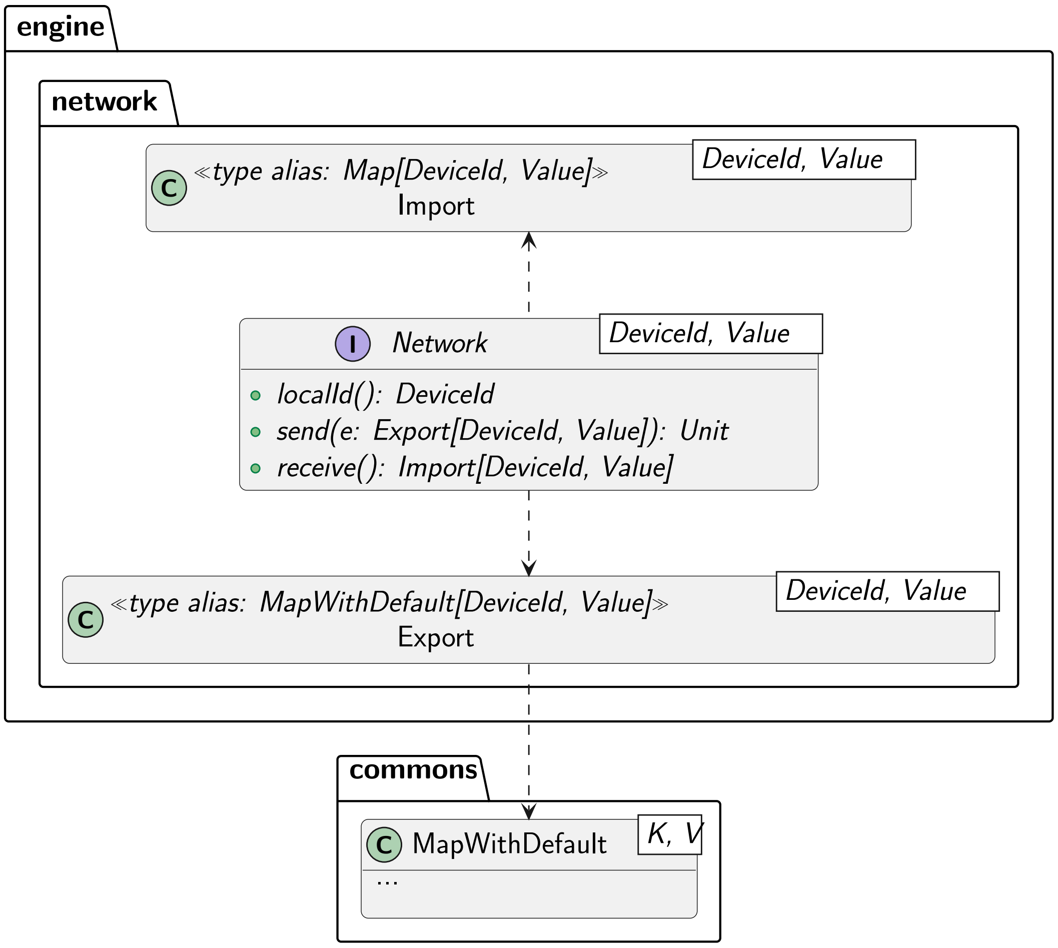
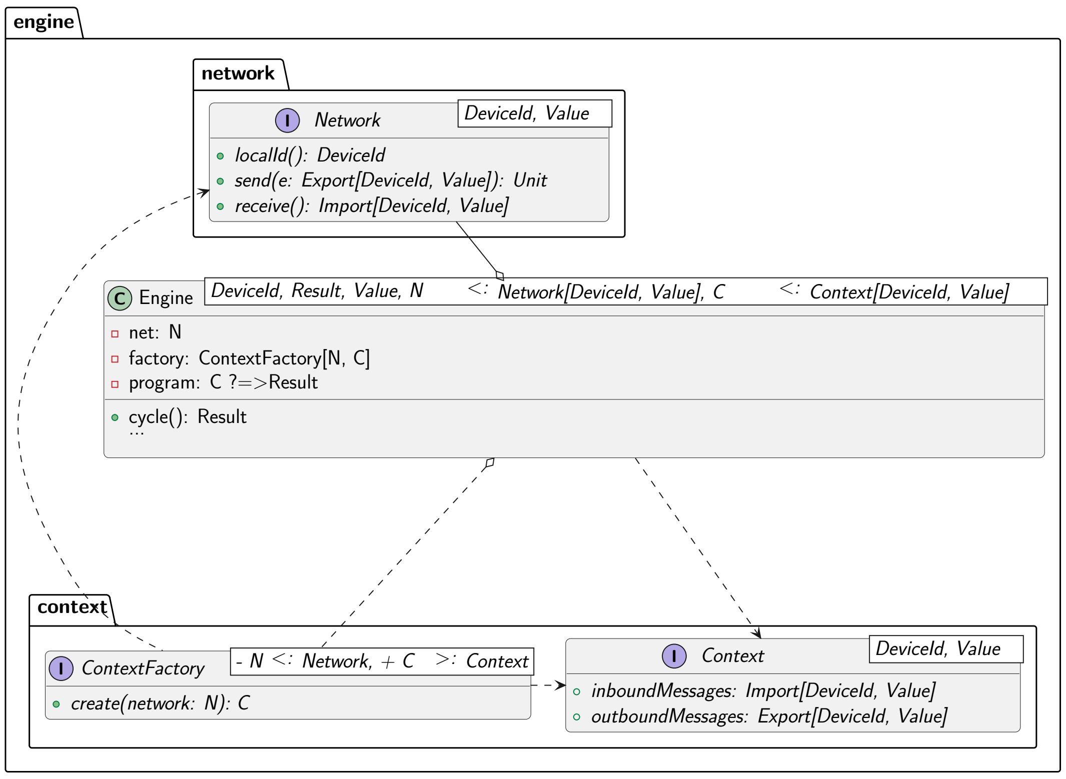

engine
network
C «type alias: Map[DeviceId, Value]» Import
DeviceId, Value
I Network
DeviceId, Value
localId(): DeviceId
send(e: Export[DeviceId, Value]): Unit
receive(): Import[DeviceId, Value]
C «type alias: MapWithDefault[DeviceId, Value]» Export
DeviceId, Value
commons
C MapWithDefault
K, V
...

Figure 5.4: The engine: UML diagram of the `Context` interface and the `Engine` class.


engine
network
Network
DeviceId, Value
localId(): DeviceId
send(e: Export[DeviceId, Value]): Unit
receive(): Import[DeviceId, Value]
Engine
DeviceId, Result, Value, N <: Network[DeviceId, Value], C <: Context[DeviceId, Value]
net: N
factory: ContextFactory[N, C]
program: C ?=>Result
cycle(): Result
...
context
ContextFactory
- N <: Network, + C >: Context
create(network: N): C
Context
DeviceId, Value
inboundMessages: Import[DeviceId, Value]
outboundMessages: Export[DeviceId, Value]

In summary, the `Engine`, uses a `ContextFactory` to repeatedly instantiate a new `Context` for every cycle of the aggregate program execution, executes the program against the context, and finally sends the result to the network. An UML diagram of the `Context` and the `Engine` can be found in Figure 5.4.

5.3 Network-based sensors

In general, sensors and actuators are not part of the DSL of ScaFi-XC. In ScaFi, sensors are modeled as a key-value dictionary, from `String` to `Any`, that casts the value to the expected type. To improve the type safety and to have less error-prone sensor access, in this design sensor and actuator design and implementation are left to the user, that can provide them in three main ways:

1. by implementing an external library that interacts with the sensor and ac-

tuator hardware, which gets invoked by the aggregate program;

2. by extending the `Context` implementation with additional, type-safe sensor and actuator methods, invoked in the aggregate program in the form `ctx.sensorName()` or `summon[ContextImpl].sensorName()`;

3. by both extending the `Context` and implementing an external library as its facade, to provide an API with static method signatures that can be invoked in the aggregate program, in line with the style of the DSL.

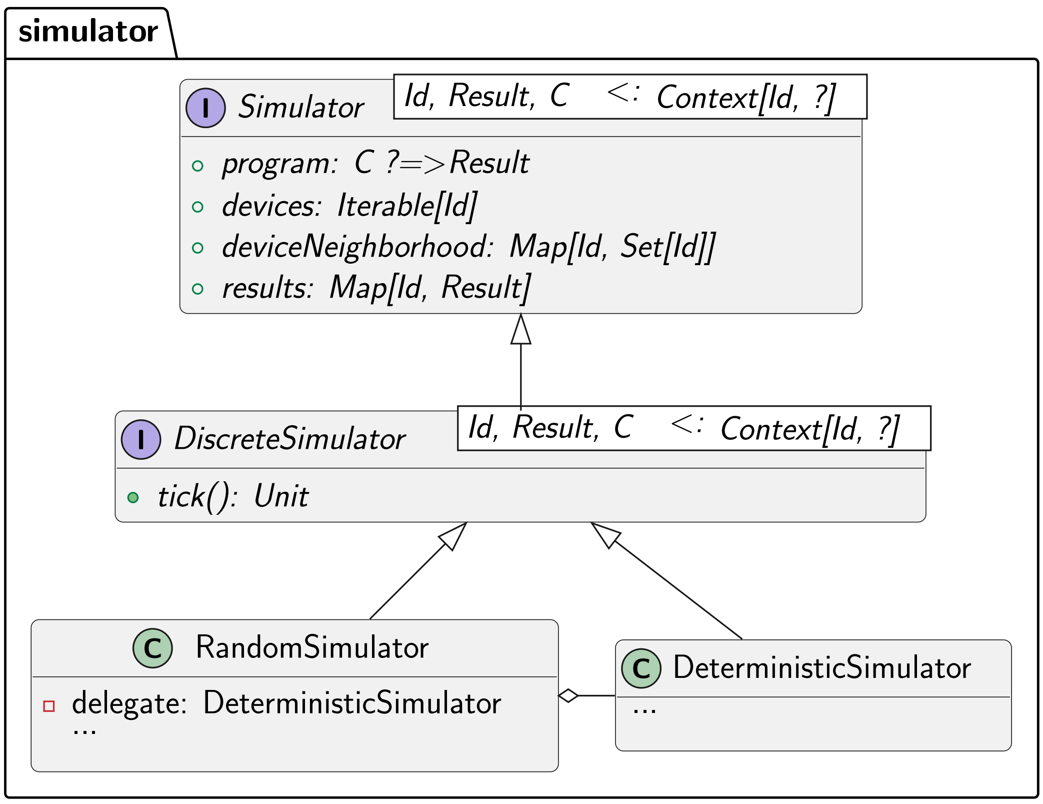
This design choice has been made to keep the DSL as simple as possible and to allow the user to choose the best way to interact with the sensor and actuator hardware, as well as to allow the user to choose the best way to model the sensor and actuator data, which can be very different from one application to another. Nevertheless, network-based sensors, which are sensors whose measured values differ for every visible neighbor, represent a special case. These sensors must either be implemented following the second or the third way of the list above, because a measurement must return an `AggregateValue[T]` where `T` is the type of the measurement and the aggregate value contains a measurement for every visible neighbor, including self.

The standard library included in the `core` module provides a network-based sensor called `DistanceSensor`, due to its importance in common aggregate programs. In ScaFi, the corresponding construct is called `nbrRange`. If an aggregate context implements the `DistanceSensor[N: Numeric]` trait, `senseDistance: N` is available to be invoked in the aggregate program, and it returns a value of type `AggregateValue[N]` containing the distance to every visible neighbor, including self. The availability of the distance sensor in the context enables the invocation of library constructs based on that, such as the `sensorDistanceTo` of the `GradientLibrary` which uses the sensors as metric for the distance to the neighbors. The distance sensor is generic on the type of the measurement, as long as it is a numeric type, to allow for different configurations, such as measuring with floats, integers, or custom types that provide a `Numeric` given instance.

5.4 The simulator

The simulator module serves both as a tool for developers to test their programs in a controlled environment and as a fundamental component for the acceptance tests. For the scope of this project, the simulator has been designed to be as simple as possible, providing a minimal set of functionalities that are enough to test the DSL and the libraries developed for it. As a result, the simulator is deterministic, with discrete time, and models neighborhoods as a map from device identifiers to a set of device identifiers. Nevertheless, the simulator implements basic real-world network phenomena such as message loss and delay, as well as customizable message retention time and device reboot/failures. Inside tests, a *deterministic* simulator allows control of every aspect of the aforementioned features through the use of policies, implementing the strategy pattern. In addition to that, in manual tests, a *random* simulator could be used. The random simulator allows the generation of randomized device networks and randomized policies to simulate an environment closer to the internet, following a set of given parameters for the probability distributions used in the implementation. This is particularly useful to test the self-healing, self-organizing properties of aggregate programs. Tests can be reproduced deterministically even in the random simulator thanks to the `seed` parameter that controls the generation of pseudo-random numbers. The resulting UML diagram of the simulator module design is shown in Figure 5.5.

Figure 5.5: UML diagram of the `simulator` module.



Chapter 6

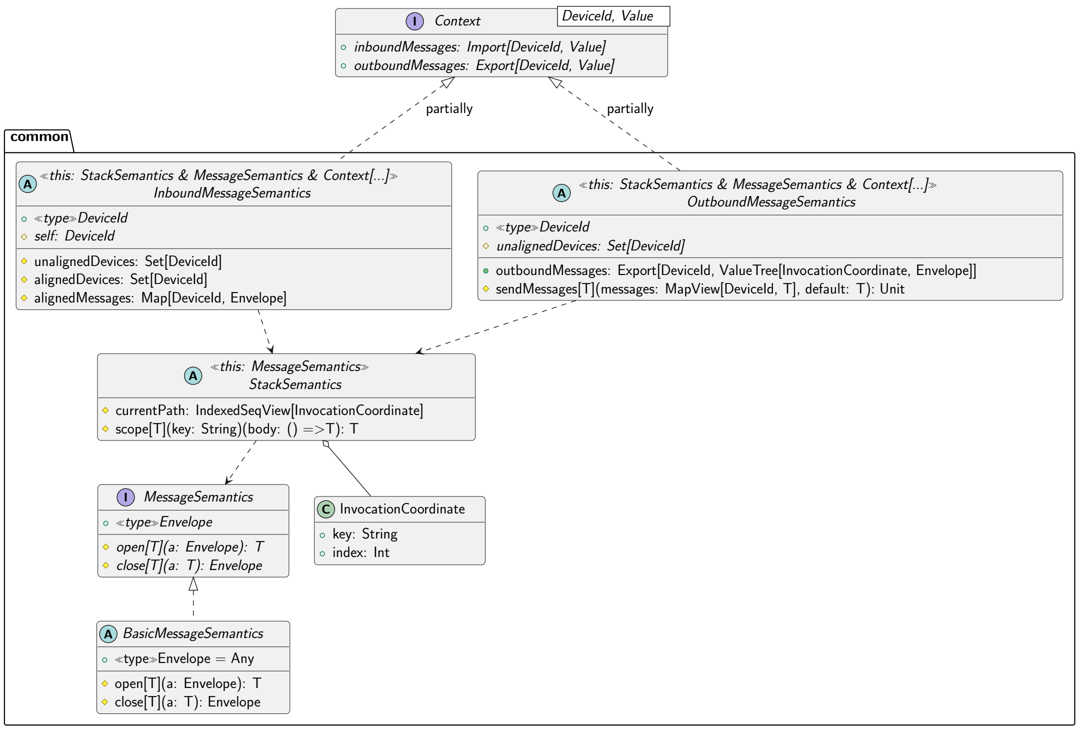
Implementation

This chapter documents the implementation details of the models and libraries designed, as well as additional tools and extensions developed as research experiments and proof of concepts. The implementation of the `core` and `simulator` modules has no external, third-party dependencies apart from the Scala standard library, thus satisfying the requirement *T.7* from Section 4.1. In particular, the chapter covers the implementation of an experimental `FoldhoodLibrary`, that demonstrates the expressiveness of the ScaFi-XC design by implementing an API for `foldhood` and `foldhoodPlus` similar to the original ScaFi, and the prototype of a different implementation of the `AggregateFoundation` trait, which adds compile time assertions on the user code to prevent common mistakes and improve the quality of aggregate programs, at the expense of more complicated signatures of library methods, following requirement *F.8* from Section 4.1. The chapter also covers the integration of ScaFi-XC with the Alchemist simulator (requirement *F.4* from Section 4.1), which enables graphical and more realistic simulations, as well as additional proof of the functionality of ScaFi-XC.

6.1 Implementation of the XC operational semantics

The implementation of the operational semantics as described in paper [5] follows the design of Section 5.2.1 by defining a concrete class that inherits from

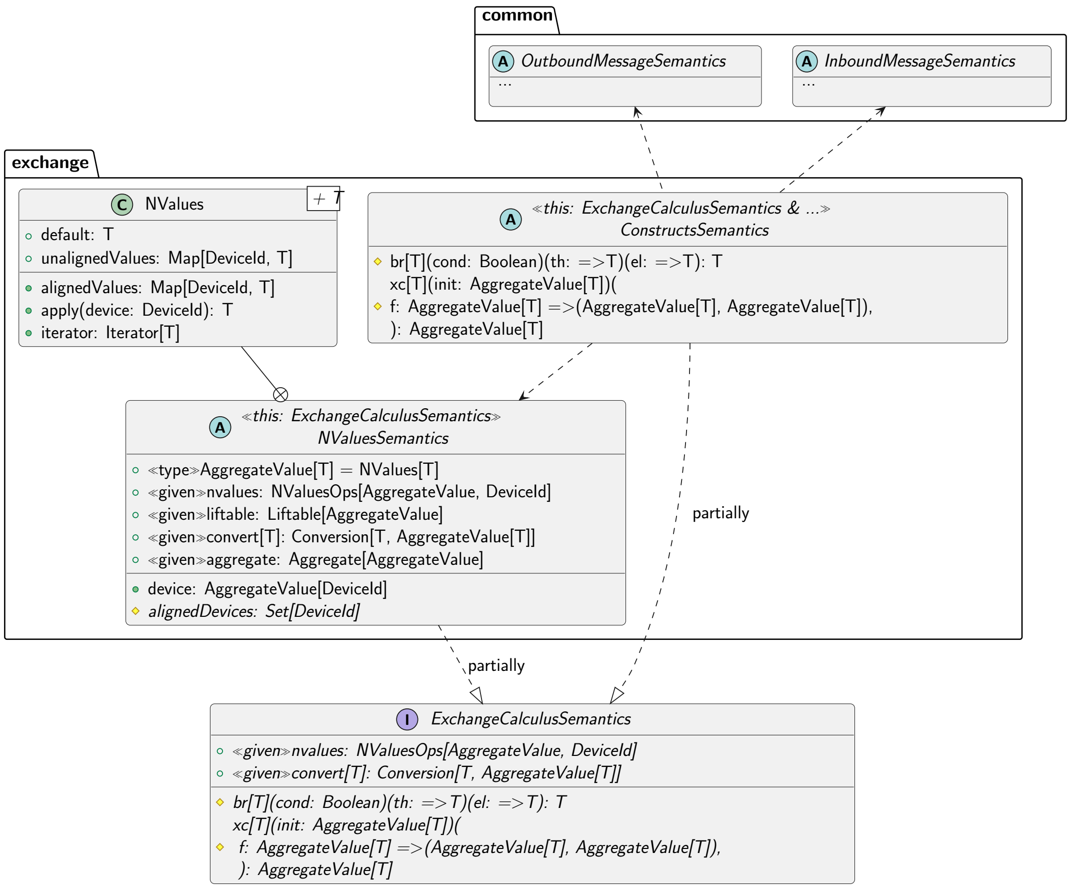
Figure 6.1: Exchange Calculus context mixins: UML diagram of the mixin layers in package `common`, stripped of transitive dependencies.


Context
DeviceId, Value
inboundMessages: Import[DeviceId, Value]
outboundMessages: Export[DeviceId, Value]
partially
partially
common
«this: StackSemantics & MessageSemantics & Context[...]»
InboundMessageSemantics
«type»DeviceId
self: DeviceId
unalignedDevices: Set[DeviceId]
alignedDevices: Set[DeviceId]
alignedMessages: Map[DeviceId, Envelope]
«this: StackSemantics & MessageSemantics & Context[...]»
OutboundMessageSemantics
«type»DeviceId
unalignedDevices: Set[DeviceId]
outboundMessages: Export[DeviceId, ValueTree[InvocationCoordinate, Envelope]]
sendMessages[T](messages: MapView[DeviceId, T], default: T): Unit
«this: MessageSemantics»
StackSemantics
currentPath: IndexedSeqView[InvocationCoordinate]
scope[T](key: String)(body: () =>T): T
MessageSemantics
«type»Envelope
open[T](a: Envelope): T
close[T](a: T): Envelope
InvocationCoordinate
key: String
index: Int
BasicMessageSemantics
«type»Envelope = Any
open[T](a: Envelope): T
close[T](a: T): Envelope

`ExchangeCalculusSemantics`. Given that the same class serves as context for the execution of aggregate programs' rounds, following the engine design of Section 5.2.2, it implements the `Context` interface too.

The implementation is named `BasicExchangeCalculusContext`, because it is meant to provide a simple yet readable and reliable implementation, without pursuing premature optimizations or additional features. A more advanced implementation could be developed in the future, maybe specifically tailored to some destination platform or network implementation. In order to maximize the reusability of its code, the logic and behavior that compose the operational semantics has been broken down into several mixin layers, with their dependencies declared through self-type annotations and abstract members. These mixin layers have been organized into two packages based on their reusability: `context.common` with the most general and reusable mixins, and `context.exchange` with the mixins that are specific to the exchange calculus, as shown in Figures 6.1 and 6.2.

Figure 6.2: Exchange Calculus context mixins: UML diagram of the mixin layers in package `exchange`, stripped of transitive dependencies.


common
A OutboundMessageSemantics
...
A InboundMessageSemantics
...
exchange
C NValues
+ T
default: T
unalignedValues: Map[DeviceId, T]
alignedValues: Map[DeviceId, T]
apply(device: DeviceId): T
iterator: Iterator[T]
A «this: ExchangeCalculusSemantics & ...» ConstructsSemantics
br[T](cond: Boolean)(th: =>T)(el: =>T): T
xc[T](init: AggregateValue[T])(
f: AggregateValue[T] =>(AggregateValue[T], AggregateValue[T]),
): AggregateValue[T]
A «this: ExchangeCalculusSemantics» NValuesSemantics
«type»AggregateValue[T] = NValues[T]
«given»nvalues: NValuesOps[AggregateValue, DeviceId]
«given»liftable: Liftable[AggregateValue]
«given»convert[T]: Conversion[T, AggregateValue[T]]
«given»aggregate: Aggregate[AggregateValue]
device: AggregateValue[DeviceId]
alignedDevices: Set[DeviceId]
partially
partially
I ExchangeCalculusSemantics
«given»nvalues: NValuesOps[AggregateValue, DeviceId]
«given»convert[T]: Conversion[T, AggregateValue[T]]
br[T](cond: Boolean)(th: =>T)(el: =>T): T
xc[T](init: AggregateValue[T])(
f: AggregateValue[T] =>(AggregateValue[T], AggregateValue[T]),
): AggregateValue[T]

Listing 6.1: Example of aggregate program that produces the value tree in Figure 6.3.

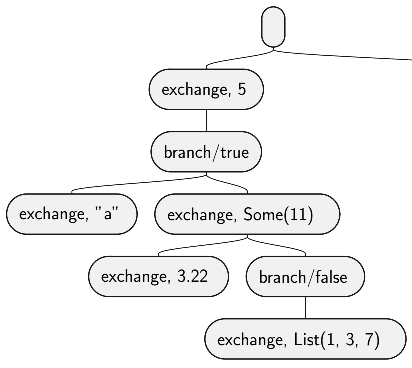
```scala
def myProgram(using ExchangeCalculusSemantics): Int =
    val _ = exchange(5)(nv1 =>
      val _ = branch(true) {
        val _ = nbr("a").fold("")(_ + _)
        share(Some(2))(nv3 =>
          val _ = exchange(3.22)(nv4 => nv4).fold(0.0)(_ + _)
          Some(branch(false) { -1 } {
            exchange(List(1))(nv => nv.map(_ ++ List(3, 7))).map(_.sum).onlySelf
          }),
        )
      } {
        None
      }
      nv1,
    ).onlySelf
    branch(false) { 1 } { exchange(2)(nv => nv).onlySelf }
```

6.1.1 The stack-based semantics implementation

Without using Scala 3 macros everywhere an aggregate expression is written, the AST of an aggregate program is not directly available to the semantics implementation. Consequently, the semantics implementation has to track the invocation of its primitives into an explicit stack-like ADT, while building the `Export` value tree using the paths traced by the stack. If a primitive invocation is nested inside another, the invocation trace is pushed into the stack, and the nested expression is evaluated, potentially growing the stack but leaving it unchanged at the end of the evaluation, and then the trace is popped from the stack. This logic is implemented in the `scope` method of `StackSemantics` and is employed by the `xc` implementation of `exchange.ConstructsSemantics`. For instance, the program in Listing 6.1, at the first execution round with the `BasicExchangeCalculusContext`, would produce the value tree in Figure 6.3.

Alignment is implemented by comparing the current stack with the neighbor's value trees: if a path prefix is in common, the device is aligned with the given neighbor. In order to distinguish `exchange` invocations that are not part of the same conditional branch, it is necessary that the `branch` construct is invoked in place of Scala's `if`, and that the `branch` construct is implemented to push a branch identifier into the stack using the scope method, thus misaligning devices who took different branches. A known limitation and pitfall of this solution, besides

Figure 6.3: Example of value tree produced by the program in Listing 6.1.



making programmers understand the need to use `branch` instead of `if`, is boolean short-circuiting, which behaves similarly to the `if` construct, in the sense that it can happen to skip the evaluation of some exchange calls without tracing the conditional branch in the stack. When that happens, the resulting behavior is unpredictable and will probably lead to runtime errors.

`InboundMessagesSemantics` is responsible for providing the set of currently aligned devices as well as retrieving the values corresponding to the current traced path from their value trees. `OutboundMessagesSemantics` is responsible for building the export value tree using the current path and the values passed to the `sendMessages` method. Once the program round has completed, and the `Engine` asks for the `Export` value tree, the sent messages are reorganized into a different value tree for every known neighbor, plus the current device, because memory is modeled as a self-message, and a default value tree for every new neighbor that appears during sleep time. This is the reason why the `Export` is an alias for a `MapWithDefault` while the `Import` is an alias for a `Map`, with both ADT immutable.

Inside a value tree, values of different types are stored together under a common type, and they need to be converted back to their original types when extracted from the tree. For this reason, `MessageSemantics` offers two methods, `open` and `close`, responsible for the conversion from and to the common type, defined with an abstract type member called `Envelope`. In a real-world distributed system, an `Envelope` could be a sequence of bytes, containing the serialized stream from shared objects. For the scope of ScaFi-XC, all the simulations are executed using the `MessageSemantics.Basic` implementation, which simply casts the values to and from `Any`, which works because the simulations are run in a single JVM, and the `Any` type is the root of the Scala type hierarchy.

6.2 The build system

Following the requirements listed in Section 4.1, the chosen build system for the project is SBT, in particular version `1.9.8`, following requirement *T.3* from Section 4.1. The build tool has been customized with the following plugins:

- `sbt-scalafix` to lint the code with *scalafix*, further explained in Section 7.4, following requirement *T.6* from Section 4.1;
- `sbt-scalafmt` to lint the code with *scalafmt*, further explained in Section 7.4, following requirement *T.6* from Section 4.1;
- `sbt-scalajs` and `bt-scalajs-crossproject` to cross-build the project for *JavaScript* with *scala-js*, following requirement *T.4* from Section 4.1;
- `sbt-scala-native` and `sbt-scala-native-crossproject` to cross-build the project for *native* with *scala-native*, following requirement *T.5* from Section 4.1.

In addition to that, the Dotty compiler has been customized with flags that enhance the quality of the code, such as the aforementioned *explicit nulls* and *multiversal equality*, but also enforcement for indentation over curly braces style, warnings as errors, safe initialization checks, warnings on value discards, and more (requirement *T.2* from Section 4.1).

6.3 The "FoldhoodLibrary"

As a proof of concept of the expressiveness of the ScaFi-XC design, an experimental library has been developed, called `FoldhoodLibrary`, which provides an API for the `foldhood` and `foldhoodPlus` constructs as defined in the original ScaFi library, Listing 6.2: Usage example of the `FoldhoodLibrary`.

```scala
def foldhoodingPlusProgram(using ExchangeCalculusContextWithHopDistance): Int =
  foldhoodPlus(0)(_ + _) {
    nbr(self) + nbr("3").toInt + nbrRange
  }
```

representing in a way an internal DSL written in terms of another. The library works for any aggregate context that supports the `FieldCalculusSyntax`, and implements `foldhood`, `foldhoodPlus`, `nbr`, and `nbrRange` for contexts that also support `DistanceSensor`. The resulting API can be seen used in Listing 6.2, where `nbr` is not the same as the one defined in `FielCalculusLibrary` but has a different signature, that takes a lazy expression of type `=> T` and returns a `T`. When evaluating a foldhood, the expression is evaluated as is and the values passed to `nbr` are recorded and returned in order, then shared with neighbors. Then, for each aligned neighbor, the same expression is re-evaluated, this time substituting the `nbr` return values with the ones coming from neighbors, in the right order to match the expression. If the context implements the `DistanceSensor` trait, `nbrRange` can be invoked to return the distance from the current node to the neighbor evaluated in the foldhood. The only difference between `foldhood` and `foldhoodPlus` is that the former does not include the expression value of the current node in the folding result, while the latter does. The example of Listing 6.2 demonstrates that `nbr` can be used with arguments of any type, as long as they type check in the foldhood expression.

6.4 Context-based constraints on shared values

This proof of concept has been implemented on a different feature branch of the repository, as it represents a very impactful change on the entire framework. The idea is to use context bounds on every method that is supposed to share values with neighbors or self, such as `exchange`, `nbr`, `share`, `distanceTo`, and so on. Through these context bounds, the types able to be shared can be restricted by a set of rules, that, if not satisfied, won't provide the context parameter needed to invoke the method. The type class used as bound is called `Shareable`, and its counterpart

Listing 6.3: Definition of the `Shareable` and `NotShareable` type classes.

```scala
package it.unibo.scafi.xc.language.foundation

import scala.util.NotGiven
import scala.annotation.implicitNotFound

object DistributedSystemUtilities:
  @implicitNotFound(
    "Cannot share value of type ${T}. ${T} must be a primitive value type or a
        serializable type, and it must not be marked as NotShareable",
  )
  open class Shareable[T](using NotGiven[NotShareable[T]])

  final class NotShareable[T]

  given [T <: AnyVal | Serializable](using NotGiven[NotShareable[T]]): Shareable[T
      ] = Shareable[T]()
end DistributedSystemUtilities
```

negating it is named `NotShareable`, as shown in Listing 6.3. By default, in order to allow libraries that abstract over the semantics implementation to share at the very least primitive types and classes marked as `Serializable`, a global given instance of `Shareable` is provided for subtypes of `AnyVal` or `Serializable`. Given that the `Shareable` type class is marked as `open` and has a public constructor, semantics and their implementations can extend the set of types that satisfy the constraints, by providing additional given instances of the type class, and optionally adding ad-hoc behavior to the type class with extensions. As a side-effect, the `Shareable` could potentially be instantiated manually to force any type to be shareable, probably resulting in a runtime error during the actual serialization phase that would have occurred anyway without this whole feature enabled. Nevertheless, in a typical scenario, the constraint works as expected, as shown in Listing 6.4, where the `Shareable` constraint is violated by the attempt to share a value of type `AggregateValue[Int]`, while it is marked as `NotShareable` by the `AggregateFoundation`. In the same snipped, the new signature of the `nbr` method with the context bound shows how signatures get affected by this change, in case it was merged and applied to the main branch.

Listing 6.4: Usage example of the `Shareable` type class, that demonstrates a violation of the constraint.

```scala
// New nbr definition:
def nbr[A: Shareable](using language: AggregateFoundation & FieldCalculusSyntax)(
  expr: A,
): language.AggregateValue[A] = language.nbr(expr)

// Bad usage example:
val _ = nbr(nbr(1))

// Compilation error:
//[error] -- [E172] Type Error: Test.scala:7:19
//[error]   7 |    val _ = nbr(nbr(1))
//[error]     |                   ^
//[error]     |Cannot share value of type c.NValues[Int]. c.NValues[Int] must be a
    primitive value type or a serializable type, and it must not be marked as
    NotShareable.
```

6.5 Integration with the Alchemist simulator

As stated in the GitHub repo[1], Alchemist is a simulator for pervasive, aggregate, and nature-inspired computing. Originally, Alchemist was conceived as a chemical-oriented multi-compartment stochastic simulation engine, generic enough that could be adapted to simulate a wide range of systems, even if unrelated to the chemistry domain [25]. The integration with Alchemist enables a whole new simulation experience, thanks to the graphical interface and the ability to simulate most of the relevant properties of real-world CAS. Integrating with the simulator consists, in practice, of implementing an *incarnation*, which is a set of classes that bridge the Alchemist models of molecule, reactions, and concentration with the desired models of the simulation. Inside an Alchemist simulation of a network of devices, each device is represented as a `Node`, and the incarnation implementation adds to each node an implementation of a `Network`, an engine instance, and a reference to the aggregate program to simulate. Each node has visibility over its neighbors and takes care to send outbound messages to the right recipients when invoked, after the execution round. During each of the rounds, the engine will invoke the aggregate program using *Java reflections*, because the program's fully qualified name is passed as a string through the configuration file written in *yaml*.

[1] `https://github.com/AlchemistSimulator/Alchemist`

Listing 6.5: Example of Alchemist configuration file.

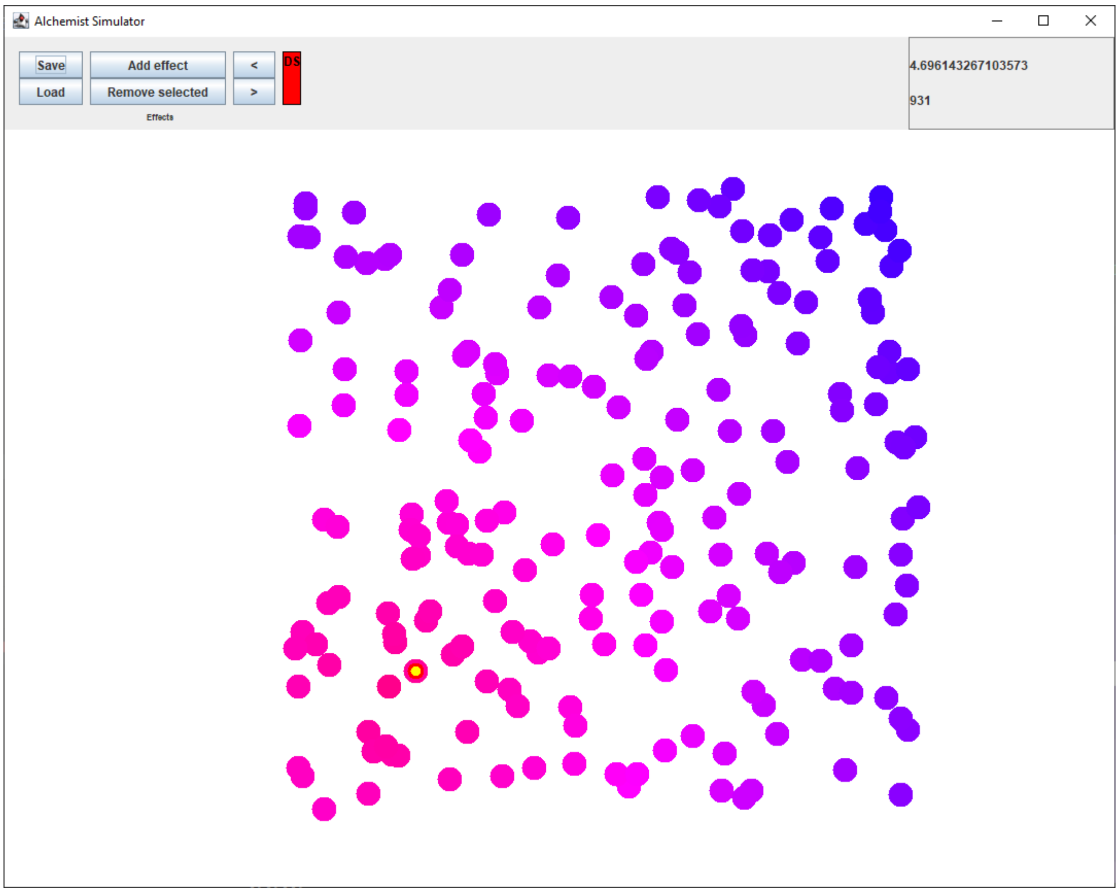
```yaml
incarnation: exchange

network-model:
  type: ConnectWithinDistance
  parameters: [5]

_pool: &program
  - time-distribution: 1
    type: Event
    actions:
      - type: it.unibo.scafi.xc.alchemist.actions.RunScafiProgram
        parameters: [it.unibo.scafi.xc.alchemist.main.TestProgram.
            myProgram]

deployments:
  - type: Rectangle
    parameters: [200, 0, 0, 20, 20]
    programs:
      - *program

monitors:
  type: SwingGUI
  parameters:
    graphics: alchemist-incarnation/src/main/resources/effect.json
```

Listing 6.6: Example of aggregate program that can be run with the Alchemist simulator.

```scala
def myProgram(using AlchemistContext[?]): Double =
  sensorDistanceTo(self == 0)
```

For instance, if the configuration of Listing 6.5 is run with the program of Listing 6.6, the expected output will be similar to the snapshot of Figure 6.4.

Figure 6.4: Snapshot of the Alchemist simulation of the program in Listing 6.6, with the configuration in Listing 6.5.



Chapter 7

Evaluation

This chapter presents the evaluation techniques employed in the project, as well as the tools used to ensure the quality of the codebase and the correctness of the implementation. Mainly, the evaluation consists of passing the test suite, which includes both unit and acceptance tests, while the quality of the code of libraries and programs is partially ensured by the continuous integration pipeline and the code style enforcement tools.

7.1 Unit tests

The unit test framework used for the project is ScalaTest, a popular testing framework for Scala. All the unit tests are defined as traits or classes that depend on a common trait called `UnitTest`, which provides a common testing DSL called `AnyFlatSpec`, enhanced with the `ShouldMatchers` trait and other utilities, that make the test assertions look like natural spoken language. Unit tests cover the `commons`, `core`, and `simulator` modules. `UnitTest` is also the base for the `AcceptanceTest` class of the `tests` module, to have a consistent testing style across the whole project.

Given that the expected behavior of the aggregate programming libraries API strictly relates to the chosen semantics that implements the necessary syntaxes, the unit tests for the libraries are tied to the semantics they are tested with. For the scope of the project, XC is the only semantics implemented, so the libraries are tested against it. Unit tests for libraries always include a sample program using the subject under test, and during the test cases, the program is executed in a test environment and inspected, both for the expected results and for the expected value tree produced by the test context.

Where possible, unit tests are written as traits, to be mixed in with the actual test classes, to avoid code duplication and to have a common set of tests applied for different implementations of the same API, which must adhere to the same behavior. Examples of these abstract tests can be found for collections and abstractions of the `commons` module.

7.2 Acceptance tests

Acceptance tests are an important validation tool for the project, as they are the only way to ensure that the libraries are working as expected in simulated scenarios. The tests are aimed to be as readable as possible, using the unit test DSL, but also hiding all the complexity of the simulation setup and execution. The idea is to use acceptance tests both as a validation tool and as a documentation tool, on one side proving the correctness of the implementations and on the other demonstrating the usage and quality of aggregate programs written with the standard libraries.

In ScaFi-XC, acceptance tests extend the `AcceptanceTest` trait, which is a subtype of `UnitTest`, and they are located in the `tests` module. As a consequence, acceptance tests inherit the same assertion DSL used in unit tests, that is `AnyFlatSpec` with `ShouldMatchers`, and the same utilities for testing, providing consistency across the whole test suite.

One of the most important acceptance tests currently present is named `GradientWithObstacleTest`, which is a simulation of a bi-dimensional grid-like network of devices that compute a gradient from a source, with an obstacle in the middle of the grid that appears halfway through the simulation, as shown in Listing 7.1. The gradient is expected to be recalculated after the obstacle appears, and the test checks that the devices adapt to the new environment, confirming the self-organizing properties of the aggregate system and the functionality of the library.

Listing 7.1: `GradientWithObstacleTest` acceptance test.

```
class GradientWithObstacleTest extends AcceptanceTest with GridNetwork:
  override type TestProgramResult = Double
  val epsilon: Double = 0.0001
  val obstacleGradient: Double = Double.PositiveInfinity
  override def rows: Int = 10
  override def columns: Int = 10
  override def ticks: Int = 1600
  def isSource(id: PositionInGrid): Boolean = id.row == 0 && id.col == 0
  def isObstacle(id: PositionInGrid): Boolean = id.row > 0 && id.col == 4

  def expectedGradient(id: PositionInGrid): Double =
    if isObstacle(id) then obstacleGradient
    else if id.col < 4 || id.row == 0 then Math.max(id.row, id.col).toDouble
    else Math.max(id.row, id.col - 4).toDouble + 4

  // Network:
  // s * * * * * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *
  // * * * * | * * * * *

  override def device(row: Int, col: Int): SleepingDevice[PositionInGrid] =
    SleepingDevice.WithFixedSleepTime(PositionInGrid(row, col), ((row + 1) * col %
        3) + 1)

  override def program(using TestProgramContext): Double =
    val round = rep(0)(_ + 1)
    branch(isObstacle(self) && round >= 200)(obstacleGradient)(distanceTo(isSource
        (self), 1.0))

  "The gradient" should "never be calculated for the obstacles" in:
    results
      .filter(kv => isObstacle(kv._1))
      .foreach: (id, value) =>
        value shouldBe obstacleGradient

  it should "be calculated correctly with obstacles" in:
    results.foreach: (id, value) =>
      value shouldBe expectedGradient(id) +- epsilon
end GradientWithObstacleTest
```

7.3 Continuous Integration

Both unit and acceptance tests are run automatically by the continuous integration pipeline, built with *GitHub Actions*[1] and hosted by GitHub[2]. The pipeline is invoked on every push to the repository, and it runs the tests on the latest version of the codebase, as well as on every push on open pull requests, given that ScaFi-XC is meant to be open source, under the *Apache License 2.0*.

7.4 Code Style

Having a consistent and "clean" coding style across the entire project contributes to the maintainability and readability of the codebase. For this reason, automatic code formatting and linting tools are used to enforce a consistent style across the project. The tools put in place are *scalafmt* and *scalafix*, each with their own configuration file and rule sets, specific for Scala 3. The code style is enforced by two dedicated phases of the continuous integration pipeline, described in Section 7.3.

[1] `https://github.com/features/actions`

[2] `https://github.com/ldeluigi/scafi-xc/actions`

Chapter 8

Conclusion and Future Work

The objective of this work has been to develop a DSL foundation, which would serve as the basis for a new framework named ScaFi-XC. The framework proposed is intended to redesign and improve the ScaFi toolkit, is implemented using the Scala 3 programming language, and is supported by the Exchange Calculus computational model and formal language. The development of ScaFi-XC involved prototyping, interviews, and the application of advanced programming patterns with Scala 3. Continuous integration and acceptance testing were utilized to ensure the quality and reliability of the framework. The requirements for ScaFi-XC, collected through interviews with stakeholders, have all been satisfied, including optional ones, so the final result can be considered a success. However, the current implementation covers only a fraction of the functionality offered by the original ScaFi framework, leaving room for further extensions.

The concrete implementations presented here aim for simplicity, readability, correctness, and reusability, deferring concerns about performance and efficiency for future iterations. Furthermore, the developed simulator offers a very restricted set of features in comparison to the original ScaFi simulator, needing improvements to support more complex scenarios and one or more graphical interfaces for the user to interact with the simulator. The following paragraphs provide a list of possible future developments.

Performance optimization The current implementation of the context and the simulator is not optimized for performance. For example, the context leverages the

`Map` data structure to represent `ValueTrees`, which is not the most efficient data structure for this purpose, as it stores multiple copies of all the common prefixes of the keys.

Complete re-implementation of the core module The `core` module is the foundation of the ScaFi framework, as it contains the basic building blocks for the development of aggregate programs. In ScaFi-XC, it has been implemented only partially, since the only advanced construct implemented is the gradient. A comprehensive extension to include the full standard library is essential for the framework's completeness.

Enhancements of the simulator The current simulator does not support many of the features of the original ScaFi simulator and is limited to discrete time. Extensions of the implementation or a complete redesign can provide a more comprehensive feature set, including support for graphical user interfaces.

Support for real-world distributed systems The original ScaFi allowed the deployment of aggregate programs on real-world distributed systems with the `spala` and `distributed` modules, currently absent in ScaFi-XC. Implementing such support is a crucial step to consider ScaFi-XC a valid replacement for the original ScaFi.

Experimental developments with aggregate programming The reusability and the modularity of the new core module allow it to be extended with new, experimental libraries and semantics, fostering research projects exploring novel aspects of aggregate programming.

Adding more acceptance tests Most of the reliability of the framework comes from the functionality and readability of its tests. In particular, acceptance tests are designed to be proofreadable by experts in the field, and they are the most important tests for the framework. Nevertheless, only a few acceptance tests are currently present. Strengthening the test suite would enhance the reliability of the framework, ensuring its robustness in a wider range of scenarios.

Improvement of the Alchemist incarnation The integration with the Alchemist simulator is still a prototype. A more complete implementation would enable more sophisticated simulation scenarios, such as those involving situated agents, with sensors and actuators actively managed by the aggregate program.

Survey evaluation of the framework A survey evaluation of the framework could be conducted to assess the usability and effectiveness of the framework in the development of aggregate programs. Additionally, it could provide conclusive results regarding the impact of the Context-based constraints on shared values discussed in Section 6.4. Whether or not the proposed changes to the core represent a valid improvement is still an open question, and a survey evaluation of the proposal could provide a definitive answer.

In conclusion, while ScaFi-XC marks a significant step towards a new ScaFi framework, there is still much work to be done. With continued development and iteration, it has the potential to significantly contribute to the field of aggregate programming, both as a development framework and as a research tool.

Bibliography

[1] G. D. Abowd, "Beyond weiser: From ubiquitous to collective computing," *Computer*, vol. 49, no. 1, pp. 17–23, 2016.

[2] M. Satyanarayanan, "Pervasive computing: vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.

[3] R. Casadei, "Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling," *CoRR*, vol. abs/2201.03473, 2022.

[4] R. Casadei, M. Viroli, G. Aguzzi, and D. Pianini, "Scafi: A scala dsl and toolkit for aggregate programming," *SoftwareX*, vol. 20, p. 101248, 2022.

[5] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, and M. Viroli, "The exchange calculus (xc): A functional programming language design for distributed collective systems," *Journal of Systems and Software*, vol. 210, p. 111976, 2024.

[6] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *Computer*, vol. 48, pp. 22–30, Sep. 2015.

[7] D. Pianini, M. Viroli, and J. Beal, "Protelis: practical aggregate programming," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, (New York, NY, USA), pp. 1846–1853, Association for Computing Machinery, 2015.

[8] G. Audrito, "Fcpp: an efficient and extensible field calculus framework," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 153–159, Aug 2020.

[9] M. Viroli, F. Damiani, and J. Beal, "A calculus of computational fields," in *Advances in Service-Oriented and Cloud Computing: Workshops of ES-OCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers 2*, vol. 393, pp. 114–128, 09 2013.

[10] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM Trans. Comput. Logic*, vol. 20, jan 2019.

[11] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, p. 100486, 2019.

[12] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, and M. Viroli, "Functional programming for distributed systems with xc," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)* (K. Ali and J. Vitek, eds.), vol. 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 20:1–20:28, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[13] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[14] S. Doeraene, "Cross-platform language design in scala.js (keynote)," in *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, (New York, NY, USA), p. 1, Association for Computing Machinery, 2018.

[15] R. Casadei, "scafi/artifact-2021-ecoop-xc: v1.2," May 2022.

[16] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Trans. Model. Comput. Simul.*, vol. 28, mar 2018.

[17] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Trans. Model. Comput. Simul.*, vol. 28, mar 2018.

[18] G. Audrito, F. Damiani, M. Viroli, and E. Bini, “Distributed real-time shortest-paths computations with the field calculus,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 23–34, 2018.

[19] G. ”Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli, “”the share operator for field-based coordination”,” in *”Coordination Models and Languages”* (H. ”Riis Nielson and E. Tuosto, eds.), (”Cham”), pp. ”54–71”, ”Springer International Publishing”, ”2019”.

[20] N. Amin, A. Moors, and M. Odersky, “Dependent object types,” 2012.

[21] D. Ghosh, J. Sheehy, K. K. Thorup, and S. Vinoski, “Programming language impact on the development of distributed systems,” *Journal of Internet Services and Applications*, vol. 3, no. 1, pp. 23–30, 2012.

[22] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, “Traits: Composable units of behavior,” in *Proceedings of European Conference on Object-Oriented Programming (ECOOP’03)*, vol. 2743 of *LNCS*, (Berlin Heidelberg), pp. 248–274, Springer Verlag, July 2003.

[23] M. Odersky and M. Zenger, “Scalable component abstractions,” *SIGPLAN Not.*, vol. 40, p. 41–57, oct 2005.

[24] M. Odersky and M. Zenger, “Scalable component abstractions,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, (New York, NY, USA), pp. 41–57, Association for Computing Machinery, 2005.

[25] D. Pianini, S. Montagna, and M. Viroli, “Chemical-oriented simulation of computational systems with ALCHEMIST,” *Journal of Simulation*, vol. 7, pp. 202–215, Aug. 2013.

[26] R. Casadei, *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields.* PhD thesis, Alma Mater Studiorum - Università di Bologna, 2016.

[27] R. Casadei, D. Pianini, and M. Viroli, "Simulating large-scale aggregate mass with alchemist and scala," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1495–1504, Sep. 2016.