ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

MSc in Engineering and Computer Science

Feasibility of Reactive Aggregate Programming via Kotlin Flows

Thesis in:
Laboratory of Software Systems

Supervisor
Prof. Danilo Pianini

Co-supervisor
Dott. Gianluca Aguzzi

Candidate
Filippo Vissani

IV Graduation Session
Academic Year 2022-2023

Abstract

The field of engineering self-organizing systems, encompassing realms such as robot swarms, collectives of wearables, and distributed infrastructures, has witnessed extensive exploration through diverse methodologies. These approaches range from deriving algorithms inspired by natural phenomena to leveraging design patterns, utilizing learning techniques to synthesize behavior based on emergent expectations, and exposing pivotal mechanisms and abstractions at the programming language level. Among these, a predominant focus has been on employing round-based execution models in state-of-the-art languages for self-organization. While such models offer simplicity in reasoning, they often exhibit limitations concerning flexibility and granular management of sub-activities. Drawing inspiration from the Functional Reactive Approach to Self-organisation Programming (FRASP) model implemented in Scala, this thesis aims to showcase the feasibility of reactive aggregate programming in Kotlin. Leveraging the Flow functional reactive library, we demonstrate a functional reactive implementation of aggregate programming, separating program logic from the scheduling of its sub-activities. The resulting framework maintains the expressive power of aggregate scheduling while enhancing scheduling controllability, flexibility in the sensing/actuation model, and execution efficiency.

"A great adventure without success is far superior to a climb where everything goes as planned."
— Tommy Caldwell

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Developing *artificial self-organizing systems* with *collective intelligence* poses a significant research challenge that spans multiple disciplines in science and engineering [1, 2, 3, 4]. A central problem involves guiding the *self-organizing behavior among a group of agents or devices*, a task often referred to as "guided self-organization" [5] or "controlled self-organization" [6]. This challenge revolves around defining the control program that each agent must execute [7]. Solutions to this problem can be approached through automatic approaches such as multi-agent reinforcement learning [8] or manual approaches [7] involving the definition of control rules or designs in terms of patterns involving, e.g., information flows and control loops [9].

This thesis focuses on leveraging programming languages for self-organizing systems. Here, developers craft the self-organizing control program using a *macro-programming language* [10, 11], which aims to express the system's macro-level behavior. This language may be general-purpose or domain-specific, tailored to specific applications such as robotic swarms [12] or wireless sensor networks [13]. The overarching objective is to design a programming language that is expressive, practical, and declarative. This language should enable programmers to abstract away operational details, allowing the underlying platform to handle them automatically [14, 15].

Existing languages often use a *round-based* execution model, where devices repeatedly evaluate their context program cyclically or periodically. While this

approach is simple, it lacks flexibility in scheduling and managing subtasks, especially in response to contextual changes. The primary objective of this thesis is to demonstrate the feasibility of implementing reactive aggregate programming in Kotlin and to analyze the ergonomics of the proposed language in comparison to the proactive model, taking inspiration from the FRASP model [16]. Specifically, we aim to assess the suitability of Kotlin for developing self-organizing systems using a reactive programming approach.

Thesis structure The structure of this thesis is designed to provide a comprehensive exploration of the topics, starting with an in-depth Background section (Chapter 2). Here, we delve into functional programming, reactive programming, and aggregate computing, elucidating their core concepts and implementations in Kotlin, which serves as the foundation for the subsequent analyses. Moving forward, the Analysis section (Chapter 3) evaluates the current state of the art, examining notable frameworks for aggregate computing, such as Protelis, ScaFi, FCPP, Collektive, and FRASP. Building upon this analysis, we proceed to detail the design of FRASP and Collektive. At the end of the chapter, possible solutions are identified to provide a re-implementation of FRASP within Collektive. Chapter 4, Design, delineates the architectural and detailed designs of the proposed models, setting the stage for their Implementation (Chapter 5), where we describe the practical realization of these models, divided into sections for the PRM and the RMSM. Subsequently, the Validation section (Chapter 6) examines the testing procedures and evaluates the ergonomic aspects of the proposed models. Finally, the Conclusion section (Chapter 7) synthesizes our findings, encapsulating the contributions of this thesis and suggesting avenues for future research in this domain.

Chapter 2

Background

This chapter establishes the essential theoretical foundation for the subsequent exploration undertaken in this thesis. We focus on three key programming paradigms: functional programming (Section 2.1), reactive programming (Section 2.2), and aggregate computing (Section 2.3).

We begin by examining the core concepts of functional programming and its practical implementation in Kotlin. This understanding underpins our exploration of reactive programming, where we delve into its evaluation model, the reactive operators, and their application in Kotlin. Finally, we explore the fundamental abstractions of aggregate computing, including the field calculus and the contrasting reactive and proactive computational models.

2.1 Functional Programming

The functional paradigm, in the context of computer science, involves building programs through the application and composition of functions. It adopts a *declarative* approach, where function definitions are represented as trees of expressions mapping values to other values, rather than relying on a sequence of imperative statements to update the program's running state.

2.1.1 Concepts

Functional programming is built upon a rich set of fundamental concepts that serve as the foundation of its paradigm. This section aims to provide a comprehensive understanding of these concepts, elucidating their significance and practical implications in software development. From higher-order functions to referential transparency, each concept plays a fundamental role in shaping the declarative nature of the functional paradigm.

Higher-order functions higher-order functions possess the ability to either receive functions as arguments or produce them as results. The nuanced difference lies in the mathematical concept denoted by "higher-order", which involves functions operating on other functions.

These functions facilitate partial application or currying, enabling a technique where a function is applied to its arguments one at a time. With each application, a new function is created to handle the next argument. This approach allows programmers to express ideas succinctly, such as representing the successor function by partially applying the addition operator to the natural number one.

Purity pure functions, or expressions, lack side effects. This absence of side effects endows pure functions with various advantageous properties, many of which can be leveraged for code optimization. A pure function, to be defined as such, must meet the following properties:

- If the result of a pure expression is not used, it can be removed without influencing other expressions.
- If a pure function is invoked with arguments that do not introduce side effects, the result remains constant concerning that set of arguments. Repeatedly calling the pure function with the same arguments yields identical results.

Recursion functional languages typically employ recursion for iteration. Recursive functions call themselves, allowing an operation to iterate until it meets

the base case. Generally, recursion involves managing a stack, consuming space proportional to the recursion depth. This characteristic might render recursion less favorable compared to imperative loops due to potential space inefficiency. Nonetheless, a specific type of recursion called *tail recursion* can be identified and optimized by a compiler, producing code similar to that used for iteration in imperative languages. Implementing tail recursion optimization involves transforming the program using a continuous passing style during compilation.

Evaluation strategies in functional languages, various methods are commonly provided for evaluating arguments during their passage to functions. Three primary approaches include:

- *Call-by-value*: This involves evaluating arguments before the function application.
- *Call-by-name*: Here, arguments are assessed each time their value is needed within the function body.
- *Call-by-need*: Also known as *lazy evaluation*, this approach involves evaluating arguments only when their value is first required within the function body.

Type systems functional programming languages have leaned towards employing typed lambda calculus. This approach involves rejecting all invalid programs at compilation time, even at the risk of encountering false positive errors. In contrast, untyped lambda calculus, accepts all valid programs at compilation time, running the risk of false negative errors, as it rejects invalid programs only at runtime when there is sufficient information to distinguish valid from invalid programs. The incorporation of *algebraic data types* enhances the ease of manipulating complex data structures. Additionally, the robust compile-time type checking contributes to program reliability, offering a level of assurance even in the absence of other reliability techniques. Furthermore, type inference alleviates the need for manual declaration of types by the programmer in most cases.

Referential transparency in functional programming, there are no assignment statements; once a variable is defined, its value remains constant throughout the program's execution. This characteristic eliminates the possibility of side effects since any variable can be substituted with its actual value at any given point in the program. Consequently, functional programs are characterized by referential transparency.

Data structures purely functional data structures are often represented differently from their imperative counterparts. While arrays, providing constant access and update times, are fundamental in most imperative languages, purely functional alternatives might employ maps or random access lists. Although these alternatives allow for a purely functional implementation, they come with logarithmic access and update times. One distinguishing feature of purely functional data structures is persistence, which involves maintaining unmodified previous versions of the data structure.

2.1.2 Functional Programming in Kotlin

Kotlin[1], an open-source programming language characterized by static typing, accommodates both object-oriented and functional programming paradigms. Variants of Kotlin are designed to target different platforms, including the Java Virtual Machine (JVM), JavaScript, and native code.

In Kotlin, functions are treated as *first-class entities*, implying their ability to be stored in variables and data structures. Additionally, they can be passed as arguments to and returned from other higher-order functions. The operations that can be performed on functions are equivalent to those applicable to other non-function values.

To support these functionalities, Kotlin, being a statically typed programming language, employs a family of function types to represent functions. Moreover, it incorporates specialized language constructs, such as *lambda expressions*.

An illustrative instance of a higher-order function in Kotlin is the functional programming idiom `fold` (Listing 2.1) employed for collections. This function

[1] `https://kotlinlang.org/`.

Listing 2.1: `fold` function.

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

receives an initial accumulator value and a combining function. Subsequently, it constructs its return value by iteratively combining the current accumulator value with each element in the collection. Importantly, the accumulator value is replaced with each iteration.

the `combine` parameter has the function type $(R, T) \rightarrow R$, so it accepts a function that takes two arguments of types R and T and returns a value of type R. It is invoked inside the `for` loop, and the return value is then assigned to `accumulator`.

Kotlin uses function types, such as $(Int) \rightarrow String$, for declarations that deal with functions. Each function type in Kotlin is characterized by a parenthesized list specifying the parameter types and a return type. For example, $(A, B) \rightarrow C$ represents a type indicative of functions that accept two arguments of types A and B, yielding a result of type C. The parameter list may be empty, denoted by $() \rightarrow A$. It is essential to note that the return type cannot exclude the declaration of `Unit`. Function types have the option to include an additional receiver type, indicated before the dot in the notation. For instance, the type $A.(B) \rightarrow C$ signifies functions that can be invoked on a receiver object A, accepting a parameter B, and producing a result of type C.

2.2 Reactive Programming

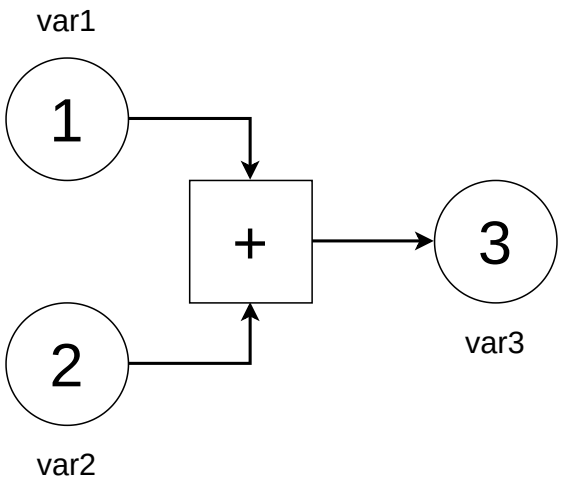
Reactive programming, as defined in [17], is a programming paradigm centered on the concept of *continuous time-varying values* and the seamless propagation of changes. It streamlines the declarative creation of event-driven applications

by enabling developers to articulate programs in terms of desired actions, leaving it to the language to autonomously handle the timing of execution. Within this paradigm, alterations in the state are automatically and efficiently disseminated throughout the network of interdependent computations by the intrinsic execution model.

Consider the example of calculating the sum of two variables (Listing 2.2). In conventional sequential imperative programming, the value of the variable `var3` will always contain 3, which is the sum of the initial values of variables `var1` and `var2` even when `var1` or `var2` is later assigned a new value (unless the programmer explicitly assigns a new value to the variable `var3`). In reactive programming, the value of the variable `var3` is always kept up-to-date. In other words, the value of `var3` is automatically recomputed over time whenever the value of `var1` or `var2` changes. This is the key notion of reactive programming. Values change over time and when they change all dependent computations are automatically reexecuted. In reactive programming terminology, the variable `var3` is said to be dependent on the variables `var1` and `var2`.

Listing (2.2) Example of a program with reactive values.

```
1 var1 = 1
2 var2 = 2
3 var3 = var1 + var2
```



(a) Graphical representation of expression dependencies in Listing 2.2.

2.2.1 Evaluation Model

The evaluation model of a reactive programming language focuses on how changes propagate across a dependency graph of values and computations. From the programmer's perspective, the automatic propagation of changes is a fundamental aspect of reactive programming. Essentially, any alteration in a value should be automatically transmitted to all computations dependent on it. When an event occurs at a source, computations reliant on that event should be notified of the changes, potentially triggering a recomputation.

At the language level, a crucial design decision involves determining who initiates the propagation of changes. This entails deciding whether the source should *push* new data to its dependents (consumers) or if the dependents should *pull* data from the event source (producer). In the pull-based model, the computation that requires a value needs to "pull" it from the source. That is, propagation is driven by the demand for new data. In the push-based model, when the source has new data, it pushes the data to its dependent computations. That is, propagation is driven by the availability of new data.

2.2.2 Reactive Operators

The primary advantage offered by libraries furnishing the reactive streams APIs lies in the provision of operators. These operators are essentially functions applicable to a data stream, adept at solving problems related to the processing of reactive streams, encompassing tasks such as filtering, mapping (Figure 2.2), and aggregating.

Furthermore, these operator functions are intentionally designed to be composable, signifying their capability to be consecutively linked to construct processing pipelines.

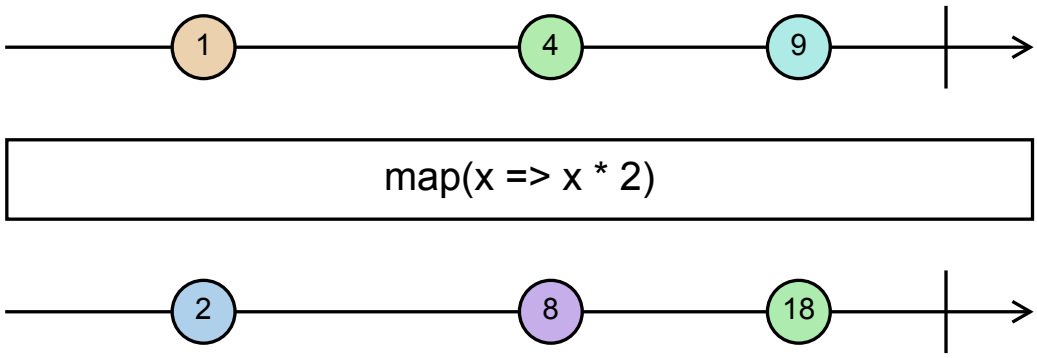


Figure 2.2: Map operator in reactive applications.

2.2.3 Reactive Programming in Kotlin

Kotlin Flow[2] is a part of the Kotlin Coroutines library, introduced to provide a reactive programming model for asynchronous *cold*[3] and *hot*[4] data streams.

A `Flow` is an asynchronous data stream that sequentially emits values and completes normally or with an exception. Intermediate operators on the flow such as `map`, `filter`, `take` and `zip` are functions that are applied to the *upstream* flow or flows and return a *downstream* flow where further operators can be applied to. Intermediate operations do not execute any code in the flow and are not suspending functions[5] themselves. They only set up a chain of operations for future execution and quickly return. This is known as a cold flow property. *Terminal operators* on the flow are either suspending functions such as `collect`, `single`, `reduce` and `toList` or `launchIn` operator that starts collection of the flow in the given scope. They are applied to the upstream flow and trigger the execution of all operations. Execution of the flow is also called "collecting the flow" and is always performed in a suspending manner without actual blocking. Terminal operators complete normally or exceptionally depending on the successful or failed execution of all the flow operations in the upstream.

By default, flows are *sequential* and all flow operations are executed sequentially

[2] `https://kotlinlang.org/docs/flow.html`.
[3] A flow that emits values only when there is an active collector.
[4] A flow that produces values regardless of whether there are active collectors.
[5] A function that can be paused and resumed at a later time.

in the same coroutine[6], with an exception for a few operations specifically designed to introduce concurrency into flow execution such as `buffer` and `flatMapMerge`.

The `Flow` interface does not carry information on whether a flow is a cold stream that can be collected repeatedly and triggers execution of the same code every time it is collected, or if it is a hot stream that emits different values from the same running source on each collection. Usually flows represent cold streams, but there is a `SharedFlow` subtype that represents hot streams. In addition to that, any flow can be turned into a hot one by the `stateIn` and `shareIn` operators, or by converting the flow into a hot channel via the `produceIn` operator.

Listing (2.3) Kotlin Flow example.

```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // this represents an operation that takes time
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
```

Listing (2.4) Kotlin Flow example result.

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

The example Listing 2.3 demonstrates the asynchronous nature of Kotlin Flow and how it allows concurrent execution without blocking the main thread. The `simple` function creates a flow using the `flow` builder. Inside the flow, it emits values from 1 to 3 with a delay of one hundred milliseconds between each emission.

[6]A concurrency design pattern that allows to write asynchronous, non-blocking code in a sequential style.

The delay simulates an operation that takes time, such as network calls or disk I/O. In the `main` function, a coroutine is launched using `launch` to run concurrently with the main thread. This coroutine prints messages indicating that it is not blocked and introduces delays between each message. As the flow is collected in the main coroutine, the emitted values are printed and interleaved with the messages from the concurrent coroutine (Listing 2.4). This demonstrates that the main thread is not blocked during the execution of the flow, thanks to the asynchronous nature of flows.

2.3 Aggregate Computing

Aggregate computing is a method for designing intricate coordinations in distributed systems, particularly for *Collective Adaptive Systems (CAS)* [18]. The approach primarily centers on the notion that understanding system interactions is more straightforward when viewed in the context of information flowing through the system as a whole, as opposed to focusing on individual devices and their interactions with peers and the environment [19].

Aggregate computing is especially suitable for scenarios where the problem at hand involves a network of devices possessing the following characteristics:

- **Openness**, indicating that the surrounding environment where devices operate can undergo unforeseen changes and faults.
- **Large scale**, involving a potentially extensive number of devices or agents that necessitate effective abstractions for coordination.
- **Intrinsic adaptiveness**, signifying the capability to respond to significant events to ensure the overall resilience of the system.

Addressing these considerations requires an approach grounded in *self-organization*, where a cohesive and resilient coordination behavior arises from localized coordination abstractions. Another objective of aggregate computing is to provide developers with a means to articulate the behavior of distributed systems possessing the aforementioned features in a composable and declarative manner. This enables

the creation of diverse layers that progressively align with specific application domains. This layered approach enhances scalability by effectively addressing the complexities inherent in the domain.

Aggregate computing builds upon the principles of *Field Calculus (FC)* (Section 2.3.2) but adds abstraction layers to address scalability and resilience challenges (Figure 2.4). These layers hide the complexity of distributed coordination and support efficient system engineering. The methodology ensures simplicity and transparency in module composition, tailoring coordination mechanisms to different subsystems based on varying requirements. Additionally, it abstracts away intricate implementation details, enabling programmers to focus on high-level system design rather than low-level intricacies. The introduction of "*resilient coordination operators*" is fundamental in concealing complexity and ensuring system robustness. By providing standardized ways to handle failures and adapt to changing conditions, these operators contribute to the overall efficiency and reliability of distributed coordination systems.

2.3.1 Abstractions

Aggregate computing models a distributed system as a set $\mathcal{D}$ of devices, ranged over by δ. On top of that, a reflexive [7] *neighboring relation* indicates the devices with which one can communicate (which is application-dependent and can be used to describe logical or physical proximity). In addition, each device has a set of *sensors* that enable the perception of the environment.

The primary abstraction introduced by aggregate computing is the *computational field* (or simply *field*), which is a function $\phi : \mathcal{D} \mapsto \mathcal{L}$ mapping each device $\delta \in \mathcal{D}$ to a local value $l \in \mathcal{L}$ [20]. A *field evolution* is a dynamically changing field, and a *field computation* takes field evolutions as inputs and produces field evolutions as outputs. These outputs are defined in such a way that they change tracking input changes.

The key idea of aggregate computing is that any field computation (*global interpretation*) can be mapped to a *single-device behavior* that is iteratively executed by all the devices in the network (*local interpretation*). Each iteration executed

[7] Each device is a neighbor of itself.

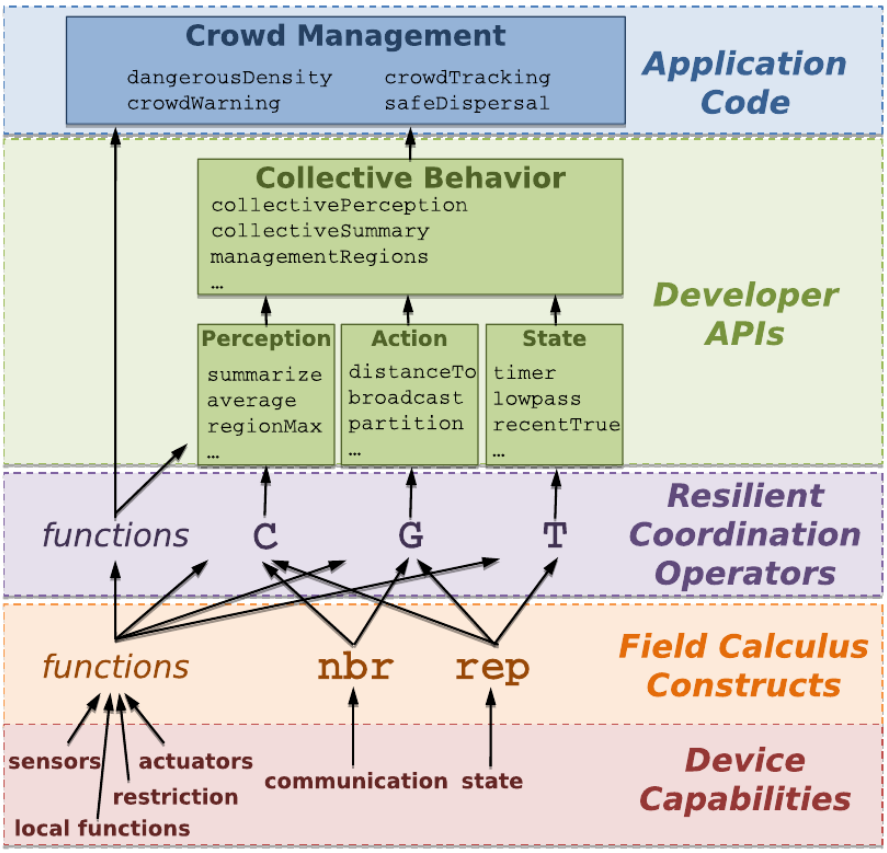


Figure 2.4: Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined to produce a user-friendly API for developing situated IoT systems.

by a device is called a *computation round* and can be subdivided into three steps:

- **sense**: the device gathers information coming from its neighbors and local sensors, which are collected to create its *local context* (or *local state*) for the current round;
- **eval**: the computation defined by the behavior is evaluated against the local context, producing an *export*;
- **broadcast**: the export is broadcasted to all the device's neighbors, which in turn collect and use this information in their future rounds.

2.3.2 Field Calculus

Aggregate computing builds from a foundation of the field calculus, a functional programming model for the specification and composition of collective behaviors with formally equivalent local and aggregate semantics.

The concept of field calculus was introduced in [21] as a fundamental core calculus designed to encapsulate the essential elements found in languages utilizing computational fields. These include functions operating over fields, functional composition involving fields, the progression of fields over time, the creation of fields of values based on neighboring elements, and the limitation of a field computation to a specific sub-region within the network. While its syntax, typing, and semantics are deeply discussed in [19] and are omitted here for simplicity, a brief description of its elements is presented below and in Figure 2.5:

- a *field calculus program* `P` consists of a sequence of *function declarations* $\bar{\texttt{F}}$ followed by the *main expression* `e`;
- an expression `e` can be:
  - a *variable* `x`, e.g., a function parameter;
  - a *local value* l, such as a boolean, number, string, pair, tuple, etc;
  - a *neighboring field value* ϕ, e.g., a map of neighbors to the distances to those neighbors;

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x | v | f($\overline{\texttt{e}}$) | if(e){e}{e} | nbr{e} | rep(e){(x)=>e}	expression
f	::=	d | b	function name
v	::=	ℓ | ϕ	value
ℓ	::=	c($\overline{\ell}$)	local value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value

Figure 2.5: Abstract syntax of the field calculus.

- a *function call* f($\bar{\texttt{e}}$) to a *user-declared function* or a *built-in function*, such as a mathematical or logical operator, a data structure operation, or a function returning the value of a sensor;
- a *branching expression* if($\texttt{e}_1$){$\texttt{e}_2$}{$\texttt{e}_3$} which splits computation into isolated sub-regions, where devices belonging to one subregion cannot communicate with those belonging to the other, resulting in $\texttt{e}_2$ where and when $\texttt{e}_1$ evaluates to true, and in $\texttt{e}_3$ otherwise;
- a nbr(e) construct, which creates a neighboring field value that maps each neighbor to the latest result of evaluating e;
- a rep($\texttt{e}_1$){(x) $\Rightarrow$ $\texttt{e}_2$} construct, which models state evolution over time. This construct retrieves the value v computed for the whole rep expression in the last evaluation round (the value produced by evaluating the expression e1 is used at the first evaluation round) and updates it with the value produced by evaluating the expression obtained from e2 by replacing the occurrences of x by v.

To work properly, the semantics of nbr and rep require a way to access, respectively, the last registered state of each neighbor and the last registered output of the device itself. In addition, this process should be made in such a way that different instances of rep and nbr cannot inadvertently "swap" their respective value. This process is called *alignment*, and it has the consequence that two branches of an if expression execute in isolation, meaning that two devices that execute

different branches cannot communicate with each other inside their branches. In practice, this process is done by carefully constructing the export of an expression as an *evaluation tree* that represents the aggregate computation. The complete semantics of export construction and alignment can be found in [20].

2.3.3 Field Calculus Extensions

The Share Operator

In recent research on the universality of the field calculus, a limitation in the efficiency of information propagation has been identified [22]. This limitation arises from the combination of time evolution and neighbor interaction operators in the original field calculus, resulting in a delay that restricts the speed at which information can be effectively propagated.

The delay stems from the separation between state sharing (`nbr`) and state updates (`rep`). Specifically, when information is received through a neighbor operation, it must be retained and remembered through a state update before it can be shared onward during the subsequent execution of the neighbor operation. This process is illustrated in Figure 2.6.

This delay in information propagation has implications for the efficiency and effectiveness of systems or models built upon the field calculus framework. Researchers may need to explore alternative approaches or optimizations to overcome this limitation and enhance the speed of information dissemination within such systems.

In [22] is proposed a solution to the limitation mentioned by introducing the `share` construct as an extension to the field calculus. This extension is designed to overcome the delay in information propagation by integrating time evolution and neighbor interaction into a single atomic coordination primitive.

The `share` construct leverages the asynchronous protocol of the field calculus, enabling it to perform several crucial operations simultaneously:

1. observation of neighbors' values;

2. reduction to a single local value;

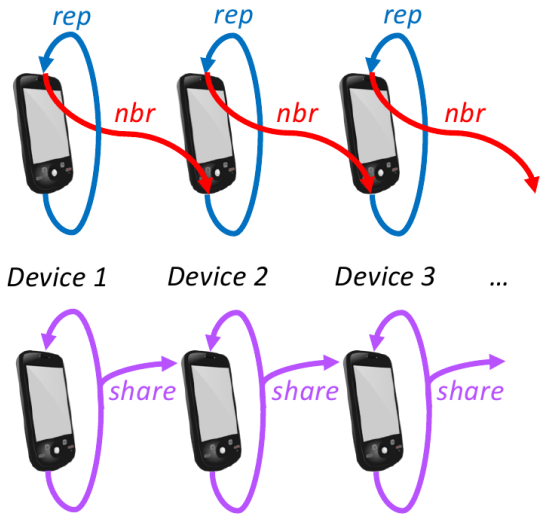


Figure 2.6: Handling state sharing (`nbr`) and memory (`rep`) separately injects a delay while information "loops around" to where it can be shared (top) while combining state sharing and memory into the new share operator eliminates that delay (bottom).

3. update of a local variable and sharing of the updated value.

By incorporating these functionalities into a single atomic operation, the `share` construct enables the immediate sharing of information received from neighbors as soon as it is integrated into the system's state. This eliminates the need to wait for the next computation round, effectively addressing the delay in information propagation identified in the original field calculus framework.

The XC Language

Programming distributed systems presents significant challenges, primarily stemming from issues such as concurrency, asynchronous execution, message loss, and device failures. These complexities are particularly pronounced in homogeneous distributed systems, wherein devices are similar and interact with neighboring devices while executing identical programs.

XC is a programming language introduced in [23], tailored for the development of homogeneous distributed systems. Within XC, developers craft a singular program that each device executes, facilitating collective emergent behavior. The lan-

guage's framework abstracts away complexities such as concurrency, asynchronous execution, message loss, and device failures. A minimalist approach is adopted, incorporating a single declarative primitive responsible for communication, state management, and connection oversight (`exchange`). The alignment mechanism within XC enables developers to abstract over asynchronous execution while preserving composability.

XC features a single communication primitive:

$$\texttt{exchange(}e_i\texttt{, (}\underline{n}\texttt{) => return }e_r\texttt{ send }e_s\texttt{)}$$

which de-sugars to:

$$\texttt{exchange(}e_i\texttt{, (}\underline{n}\texttt{) => (}e_r\texttt{,}e_s\texttt{))}$$

and is evaluated as follows:

1. the device computes the local value l_i of e_i (the initial value);
2. it substitutes variable $\underline{n}$ with the *nvalue* (neighboring value) $\underline{w}$ of messages received from the neighbors for this exchange, using l_i as default. The exchange returns the (neighboring or local) value v_r from the evaluation of e_r;
3. e_s evaluates to a nvalue $\underline{w}_s$ consisting of local values to be sent to neighbor devices δ', that will use their corresponding $\underline{w}_s\ (\delta')$ as soon as they wake up and perform their next execution round.

Often, expressions e_r and e_s coincide, hence we provide:

$$\texttt{exchange(}e_i\texttt{, (}\underline{n}\texttt{) => retsend }e\texttt{)}$$

as a shorthand for:

$$\texttt{exchange(}e_i\texttt{, (}\underline{n}\texttt{) => (}e\texttt{,}e\texttt{))}$$

Another common pattern is to access neighbors' values, which we support via:

$$\texttt{nbr(}e_i\texttt{, } e_s\texttt{) = exchange(}e_i\texttt{, (}\underline{n}\texttt{)} \Rightarrow \texttt{return } \underline{n} \texttt{ send } e_s\texttt{)}$$

In `nbr(`e_i`,` e_s`)`, the value of expression e_s is sent to neighbors, and the values received from them (gathered in $\underline{n}$ together with the default from e_i) are returned as a nvalue, thus providing a view on neighbors' values of e_s. It is crucial for the expressivity of XC that `exchange` (hence `nbr`) can send a different value to each neighbor, to allow custom interaction.

2.3.4 Reactive and Proactive Models

Aggregate computing emerged as a prominent approach for programming self-organization, with the benefits of formality, abstraction, compositionality, and pragmatism. Formality stems from building the approach over field calculus with well-defined language semantics.

Though conceptually simple, in the round-based model, discussed in [20], each round of a device is alternated with some sleeping time during which it collects information from neighboring devices. This way of managing computation can be thought of as a *proactive model* since it is the device that decides when computation should occur based on its internal scheduler.

The round-based model could be more efficient because it fully re-evaluates the context and the whole program without tracking change. Though it might be acceptable for predictable patterns of environmental change, this becomes largely suboptimal for highly variable dynamics. Indeed, the round-based approach seems to be a legacy of imperative languages or solutions featuring limited compositionality. Given this motivation, taking inspiration from the functional reactive paradigm, in [16] a *reactive self-organization programming language*, called FRASP, is proposed. This model enables the decoupling of program logic from its scheduling; the details will be discussed more deeply in Chapter 3.

Chapter 3

Analysis

This chapter delves into an in-depth examination of various state-of-the-art frameworks and methodologies in the field (Section 3.1). It presents a comprehensive overview of Protelis, ScaFi, FCPP, Collektive, and FRASP, analyzing their respective contributions. Through a comparative lens, this chapter aims to elucidate the evolution of these technologies and their impact on the domain.

Following the exploration of existing frameworks, the chapter transitions into the analysis of FRASP (Section 3.2) and Collektive (Section 3.3), shedding light on their architecture and detailed design.

Furthermore, the chapter delves into the re-implementation of FRASP into Collektive (Section 3.4), outlining possible issues and solutions designed to overcome them. In particular, it evaluates the feasibility of reactive aggregate programming in Kotlin, offering valuable perspectives on the re-implementation process and its implications.

3.1 State of the Art

3.1.1 Protelis

Protelis [24] is based on field calculus and is closely related to Proto [25]. It inherits spatial computing features from the field calculus, which provides universality, consistency, and self-stabilization properties. However, Protelis improves over Proto by offering a richer API through Java integration, support for code mobility

```
P ::= Ī F̄ s̄;                                   ;; Program
I ::= import m | import m.*                    ;; Java import
F ::= def f(x̄) {s̄;}                            ;; Function definition
s ::= e | let x = e | x = e                    ;; Statement
w ::= x | l | [w̄] | f | (x̄)->{s̄;}              ;; Variable/Value
e ::= w                                        ;; Expression
    | b(ē) | f(ē) | e.apply(ē)                 ;; Fun/Op Calls
    | e.m(ē) | #a(ē)                           ;; Method Calls
    | rep(x<-w){s̄;}                            ;; Persistent state
    | if(e){s̄;} else {s̄′;}                     ;; Exclusive branch
    | mux(e){s̄;} else {s̄′;}                    ;; Inclusive branch
    | nbr{s̄;}                                  ;; Neighborhood values
```

Figure 3.1: Protelis abstract syntax.

through first-order functions, and a syntax inspired by C-family languages.

The syntax of Protelis (Figure 3.1) is presented in abstract form. It uses meta-variables to represent names of user-defined functions (`f`), variables and function arguments (`x`), literal values (l), built-in functions and operators (b), Java method names (`m`), and aliases of static Java methods (`#a`). The syntax employs conventions like comma-separated lists and semi-colon separators for sequences of elements.

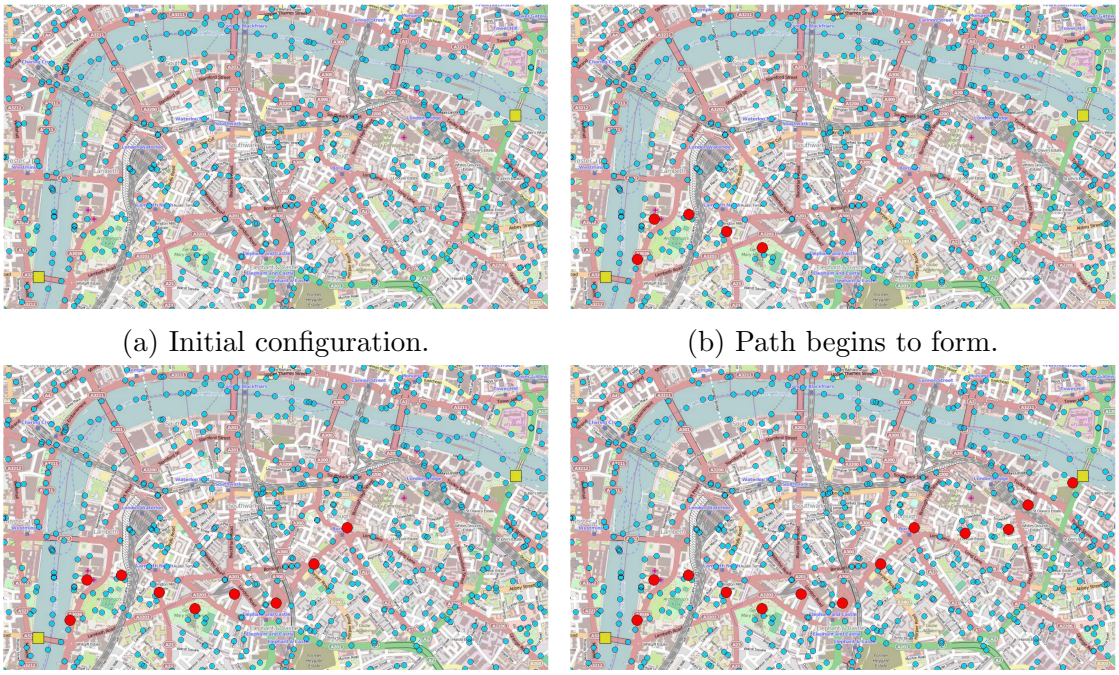
Protelis adopts a familiar C- or Java-like syntax, making it more accessible and reducing barriers to adoption. Despite its syntactic similarity to imperative languages, Protelis is purely functional. Programs consist of a sequence of function definitions, followed by a main block of statements. Functions are defined with curly brackets and can contain sequences of statements or expressions. Each statement is an expression to be evaluated (`e`), possibly in the context of the creation of a new variable (`let x = e`) or a re-assignment (`x = e`).

Example: Rendezvous at a Mass Event

In large public events, it can be challenging to meet with companions due to crowded areas, inaccessible rendezvous points, or difficulty in accessing cloud-based services.

Utilizing peer-to-peer geometric calculations across a network of devices to

compute a *rendezvous*[1] route is the proposed solution to the problem. The solution is demonstrated in a simulated city center environment (Figure 3.2), using London as an example, with devices distributed randomly across the city streets. Each device has a communication range, and the goal is for two individuals (represented by their devices) to meet at a specific location.



(a) Initial configuration. (b) Path begins to form.

(c) Path continues to extend. (d) Path computation complete.

Figure 3.2: Example of computing a rendezvous route for two people in a crowded urban environment.

The implementation (Listing 3.1) involves injecting the environment of the devices with properties representing their owners (e.g., Alice and Bob). The algorithm measures the distance to one of the participants, creates a potential field, and builds an optimal path from the other participant, descending the distance potential field to reach the first participant at zero distance. The algorithm utilizes two main functions: `distanceTo` and `descend`. `distanceTo` measures the distance to one of the participants. Given a device and a potential field, `descend` builds a path of devices connecting the device with the source of the potential field. The

[1] A meeting at an agreed time and place.

Listing 3.1: Rendezvous implementation in Protelis.

```
// Follow the gradient of a potential field down from a source
def descend(source,potential) {
    rep(path <- source) {
        let nextStep = minHood(nbr([potential, self.getId()]));
        if (nextStep.size() > 1) {
            let candidates = nbr([nextStep.get(1), path]);
            source || anyHood([self.getId(), true] == candidates)
        } else {
            source
        }
    }
}

def rendezvous(person1, person2) {
    descend (person1 == owner, distanceTo(person2 == owner))
}

// Example of using rendezvous
rendezvous("Alice", "Bob");
```

algorithm elegantly compresses the entire process into a few lines of code, utilizing the `nbr` operator to exchange required information without explicitly declaring any communication protocol.

As Figure 3.2 shows, the simulation rapidly identifies a chain of devices (represented by red dots) that marks a sequence of waypoints for both device owners to walk and meet in the middle. The algorithm dynamically adjusts the path if one of the device owners moves in a different direction, ensuring it continues to recommend the best path for rendezvous.

3.1.2 ScaFi

ScaFi [26] is a Scala[2]-based library and framework designed for aggregate programming. It facilitates the development of distributed algorithms where computations are performed by individual devices in a network, and the results are aggregated across the network. The core concepts and constructs of ScaFi's API are outlined as follows:

Expression Evaluation An expression written using the ScaFi API is evaluated by each device once per computation round.

[2] `https://www.scala-lang.org/`.

Fields Fields are represented as atomic values without any particular wrapper. They indicate the value of the field at the device performing the computation.

Neighboring Field The concept of "neighboring field" from field calculus is not explicitly represented (not reified). Spatial computation (`nbr` and `nbrvar` constructs) is only available inside a special scope provided by the `foldhood` construct.

Export The export for each iteration is constructed by the ScaFi engine. It applies side effects to an internal data structure as the constructs are invoked, thereby constructing the evaluation tree.

Constructs The semantics of the constructs defined in ScaFi are described below:

- `rep(init)(f)`: captures state evolution, starting from an `init` value that is updated each round through `f`;
- `nbr(e)` captures communication, of the value computed from its `e` expression, with neighbors; it is used only inside the argument `expr` of

  `foldhood(init)(acc)(expr)`, which supports neighborhood data aggregation, through a standard "fold" of functional programming with initial value `init`, accumulator function `acc`, and the set of values to fold over obtained by evaluating expr against all neighbors;
- `branch(cond)(th)(el)` captures domain partitioning (space-time branching): essentially, the devices for which `cond` evaluates to `true` will run subcomputation `th`, while the others will run `el`;
- `mid` is a built-in sensor providing the identifier of devices;
- `sense(sensorName)` abstracts access to local sensors;
- `nbrvar(sensorName)` abstracts access to "neighboring sensors" that behave similarly to `nbr` but are provided by the platform: i.e., such sensors provide a value for each neighbor.

Gradient Implementation in ScaFi

A *(self-healing) gradient* (Figure 3.3) is a distributed behavior that self-stabilizes, in each device of the distributed system, to a value denoting its minimum distance from the closest source node (for instance, computed by summing the neighbor-to-neighbor distances along the shortest path to the source), adapting to changes in the source set and distances. By following the neighbors of maximum decrease (resp. increase) of the gradient value, i.e., by descending (resp. ascending) the gradient, it is possible to implement efficient hop-by-hop information flows, that can be useful for data propagation and collection.

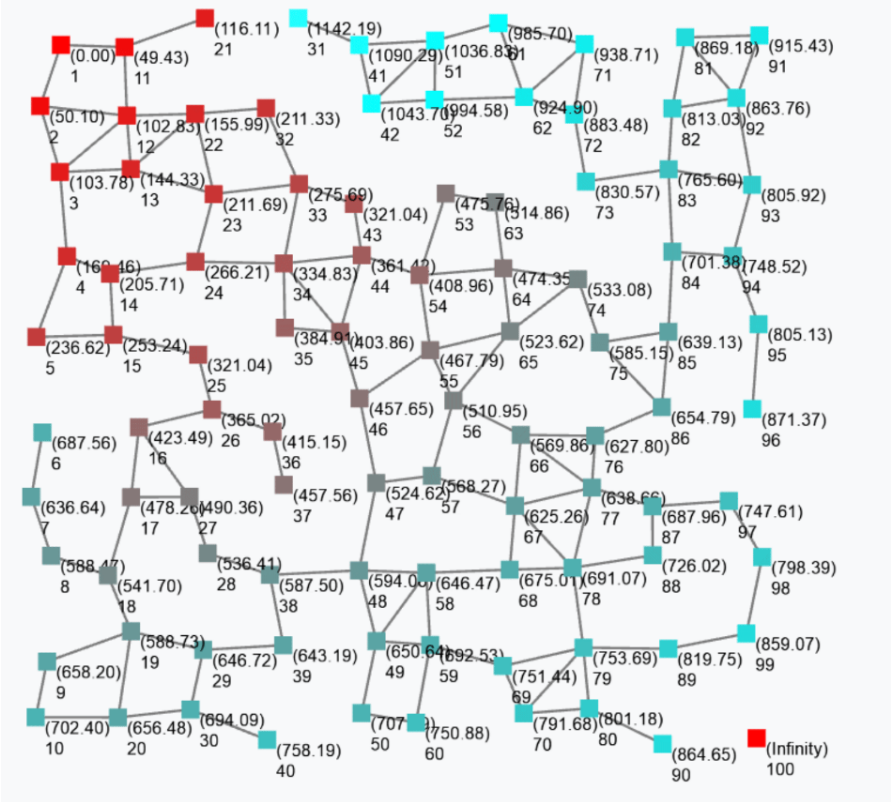


Figure 3.3: A graphical representation of the gradient implementation in ScaFi after stabilization. Each device of the network is labeled with its distance from the source (in parenthesis) and its ID. The source device is the one with ID 1. Note that devices that are not connected to the source are considered to be at an infinite distance from it.

The implementation of a gradient using ScaFi is presented in Listing 3.2, the fol-

lowing is a brief description of the program: The gradient value at each node is dynamically evolved using `rep`. This is necessary to allow a node to share its previous gradient value with neighbors. The default value is `Double.PositiveInfinity` since by default a node is at an infinite distance from a source (since it may not be reachable in general). The `mux(c)(th)(el)` evaluates its expression `th` and `el` and then uses the Boolean condition `c` to select either the former (when `c` is true) or the latter (when `c` is false). If a node is a source (i.e., if sensing the Boolean sensor `source` returns `true`), then its gradient value is `0` (by definition). If a node is not a source, then will take as its gradient value the output of the expression `minHoodPlus(nbr{distance} + nbrRange)`. `minHoodPlus(e)` is a variant of `foldhood` which does not consider the device itself when folding over the neighborhood. Namely, it selects the minimum value among those obtained by evaluating `e` against the neighbors. The argument of `minHoodPlus` is `nbr{distance} + nbrRange()`, which amounts to calculating, for each neighbor, the sum of the neighbor's most recent gradient value and the corresponding distance to that neighbor (obtained by neighboring sensor `nbrRange`, which is `nbrvar[Double](''nbrRange'')`).

Listing 3.2: Implementation of gradient in ScaFi.

```
object MyAggregateProgram extends AggregateProgram {

  override def main() = gradient(isSource)

  def gradient(isSource: Boolean): Double =
    rep(Double.PositiveInfinity)(distance =>
      mux(isSource){
        0.0
      }{
        minHoodPlus(nbr{distance} + nbrRange)
      }
  )

  def isSource = sense[Boolean]("source")
  def nbrRange = nbrvar[Double]("nbrRange")
}
```

3.1.3 FCPP

FCPP [27] is a C++14 library implementing field calculus and providing tools for distributed system simulation.

Its extensible component-based architecture allows customization for diverse application scenarios, such as Internet-of-Things (IoT) deployments, simulations, and self-organizing cloud applications, which require fine-grained parallelism to scale and for which performance improvements translate into a cost reduction. Users can add components tailored to specific functionalities, enhancing flexibility and applicability. The library incorporates compile-time optimizations and supports parallel execution, enabling efficient simulation of both systems and self-organizing cloud applications. Currently, FCPP focuses on distributed system simulations but already significantly reduces simulation costs, accelerating the development of new distributed algorithms. These features offer a path for a convenient extension to address previously ineffective scenarios.

Existing implementations often have high-performance requirements, unsuitable for resource-constrained microcontrollers. FCPP's lightweight nature makes it well-suited for these systems.

Self-organizing cloud applications necessitate fine-grained parallelism for scalability, and performance improvements directly translate to cost reduction. FCPP's support for parallelism caters to this need.

Aggregate Program Example with FCPP

The example function provided in Listing 3.3 utilizes the Adaptive Bellman-Ford algorithm to estimate distances from devices where the `source` parameter is `true`. This function explicitly takes a `node` object as input, enabling access to its functionalities, including the `nbr_dist()` method. This method returns a `field<double>` representing the estimated distances to neighboring nodes.

The `call_point` parameter serves two purposes:

- Updating the `node.stack_trace` (shared functionality across all aggregate functions, as noted in the first line).

Listing 3.3: Implementation of the Adaptive Bellman Ford algorithm in FCPP.

```
template <class node_t>
double abf(node_t& node, trace_t call_point, bool source) {
    trace_call trace_caller(node.stack_trace, call_point);
    return nbr(node, 0, INF, [&] (field<double> d) {
        double v = source ? 0.0 : INF;
        return min_hood(node, 1, d + node.nbr_dist(), v);
    });
}
```

- Facilitating the aggregation of function calls (e.g., `nbr` and `min_hood`) by providing an incrementing index.

3.1.4 Collektive

Collektive[3] provides the user with a DSL, implemented in Kotlin, that allows to create aggregate programs transparently. It was designed with the following principles in mind: transparency, minimality and portability.

Transparency refers to the clear and concise information it provides about how the underlying system behaves, such as data processing, storage, and communication between nodes. Transparency helps to reduce complexity, making it easier to understand and maintain large and complex systems.

Collektive is designed with the fewest possible constructs and abstractions while still offering the required functionalities. This reduces the complexity of the system, making it easier to maintain and debug, and lowers the overhead associated with using the DSL, which is particularly important for systems that require high performance and scalability.

Portability refers to its ability to run on various platforms and environments, including different operating systems, cloud platforms, and hardware architectures. This enables systems built with the DSL to be easily deployed and run in different environments, which is crucial for systems requiring deployment in multiple locations or scalability to meet changing demands.

Constructs implemented in Collektive are defined in Listing 3.4, while the semantics are described below:

[3] `https://github.com/Collektive/collektive`.

Listing 3.4: Base constructs implemented in Collektive.

```
interface Aggregate<ID : Any> {
    fun <Initial> exchange(
        initial: Initial,
        body: (Field<ID, Initial>) -> Field<ID, Initial>,
    ): Field<ID, Initial>
    fun <Initial, Return> exchanging(
        initial: Initial,
        body: YieldingScope<Field<ID, Initial>, Field<ID, Return>>,
    ): Field<ID, Return>
    fun <Initial> repeat(initial: Initial, transform: (Initial) -> Initial):
        Initial
    fun <Initial, Return> repeating(initial: Initial, transform: YieldingScope<
        Initial, Return>): Return
}
```

- `exchange`: It manages the computation of values between neighbors in a specific context. It computes a `body` function starting from the `initial` value and the messages received from other neighbors, then sends the results from the evaluation to specific neighbors or everyone, it is contingent upon the origin of the calculated value, whether it was received from a neighbor or if it constituted the initial value. The result of this function is a field with as messages a map with as key the ID of the devices across the network and the result of the computation passed as relative local values.

- `exchanging`: Same behavior of `exchange` but this function can yield a `Field` of `Return` value.

- `repeat`: Iteratively updates the value computing the `transform` expression at each device using the last computed value or the `initial`.

- `repeating`: Iteratively updates the value computing the `transform` expression from a `YieldingContext` at each device using the last computed value or the `initial`.

Example of Gradient in Collektive

In the Listing 3.5, the implementation of the gradient in Collektive is presented. It uses the `share` construct with `POSITIVE_INFINITY` as the initial value. The `when`

Listing 3.5: Gradient implementation in Collektive.

```
fun Aggregate<Int>.gradient(source: Boolean): Double =
    share(POSITIVE_INFINITY) {
        val dist = distances()
        when {
            source -> 0.0
            else -> (it + dist).min(POSITIVE_INFINITY)
        }
    }
```

construct is used to select the result of the expression based on the type of the node:

- if the node is the source the result is 0.0;
- if the node is not the source it must consider the neighbor where the value of the gradient is smallest and sum the distance from that neighbor.

3.1.5 FRASP

As said in Section 2.3.4, aggregate computing makes use of a round-based execution model, that can be defined as proactive. This approach is simple to reason about but limited in terms of flexibility in scheduling and management of sub-activities (and response to contextual changes). In [16] is proposed a reactive self-organization programming approach, called FRASP, that enables the decoupling of the program logic from the scheduling of its sub-activities. This model maintains the same expressiveness and benefits of aggregate programming while enabling significant improvements in terms of scheduling controllability, flexibility in the sensing/actuation model, and execution efficiency.

Reactive Model

FRASP is based on the functional reactive programming (FRP) paradigm and considers *continuous time*, $Time = \{t \in \mathbb{R} \,|\, t \geq 0\}$. Time-varying values are called *cells* and may be conceptually modeled by generic functions of type $Cell\ a$: $Time \rightarrow a$. Then, *streams* are discrete-time values and may be modeled by generic functions of type $Stream\ a$: $[Time] \rightarrow [a]$, namely, mapping a sequence

of (increasing) sample times to a sequence of corresponding values. While cells model state, streams model state changes.

Abstractions and Primitives

One of the main differences between the proactive and reactive models is that the latter allows the self-organizing collective computation to be expressed as a graph of reactive sub-computations. Each sub-computation is called *flow* and represents it programmatically through type `Flow[T]`, where `T` is the type of the output of the wrapped computation. A `Flow` is essentially a function that takes a `Context` and returns a cell of `Export`s, possibly depending on the exports of other `Flow`s, recursively.

The details of the syntax and semantics of FRASP are discussed in detail in Section III of [16] while in this section they are presented in a simplified manner:

- `constant(e)` returns a constant flow that always evaluates to the argument that has been passed;
- `sensor(name)` returns the flow of values produced by the sensor with the given `name`;
- `mid()` returns the constant flow of the device ID;
- `mux(c){t}{e}` is an expression that returns a flow with the same output of flow `t` when the Boolean flow `c` is true and the output of flow `e` when `c` is false;
- `nbr(f)` handles communication with neighbors in both directions at once, it takes a flow `f` as a parameter;
- `branch(c){t}{e}` evaluates and returns the value of expression `t` when `c` evaluates to true. This enables a form of distributed branching, where devices that happen to execute `t` will not interact with those that executed `e` (and vice versa);
- `loop(init,ft)` evolves a piece of state (initially, `init`) by applying function `ft` mapping the previous state's flow to the next state's flow.

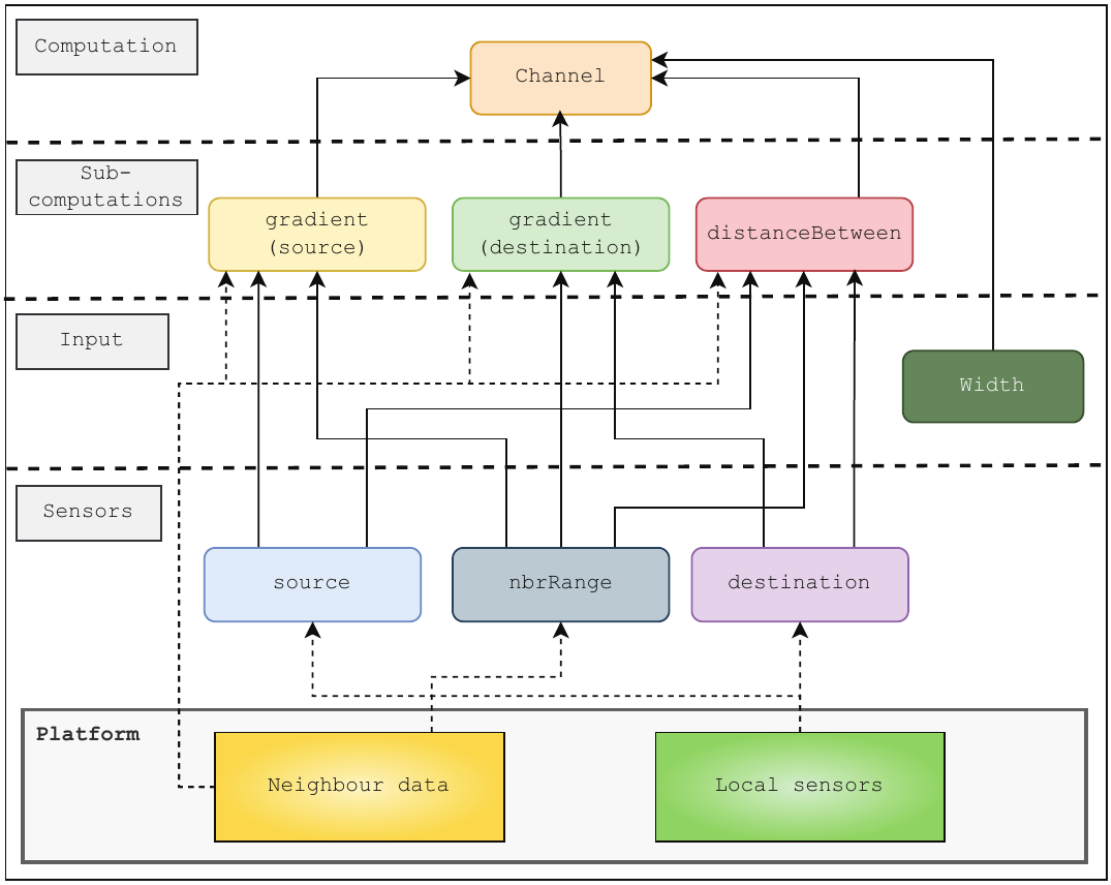
Gradient Implementation in FRASP

Listing 3.6 provides the implementation of the gradient in FRASP. The function takes the boolean `src` flow as input, denoting whether the executing node is the source of the gradient or not. The external `loop` is used to progressively evolve the current gradient value `distance` starting from an infinite value (as, initially, devices do not know whether a source is reachable). Internally to the loop, `mux` is used to select one of two values: if the node is a source, then its gradient value is `0` (base case); otherwise, the gradient should be the minimum value among the neighbors' gradient values augmented by the distance (`nbrRange`) from that very neighbor. Construct `liftTwice` is used to combine (using the sum: `_+_`) the two flows `nbrRange` (distances to neighbors) and `nbr(distance)` (neighbors' gradient values).

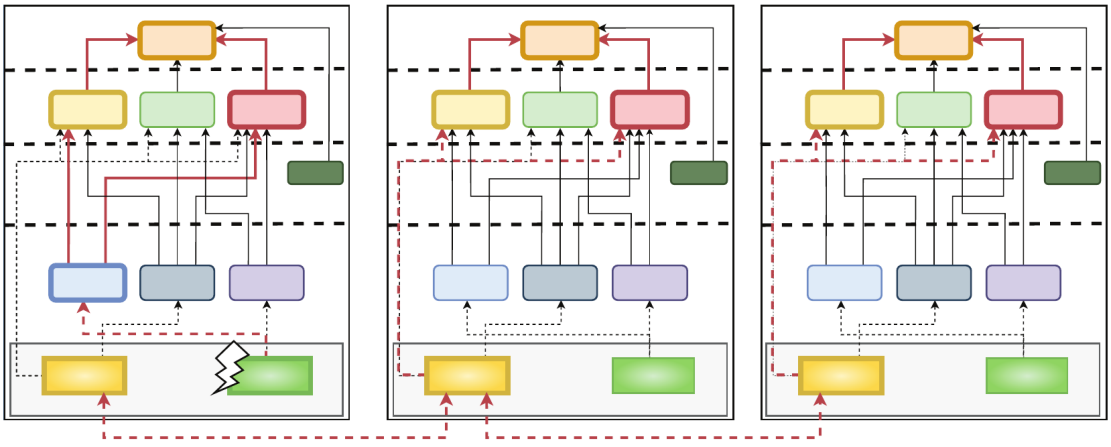
Listing 3.6: Gradient implementation in FRASP.

```
1  def gradient(src: Flow[Boolean]): Flow[Double] =
2    loop(Double.PositiveInfinity) { distance =>
3      mux(src) {
4        constant(0.0)
5      } {
6        liftTwice(nbrRange, nbr(distance))(_ + _).withoutSelf.min
7      }
8    }
```

The reactive dataflow graph in Figure 3.4a corresponds to Listing 3.6. Figure 3.4a provides the local view of the computation for a single node (where the layers denote different semantic kinds of dependencies), whereas Figure 3.4b shows the distributed dependency graph. The arrows denote dependencies. The dashed arrows denote dependencies based on platform-level scheduling and node interaction; for instance, a red block depends on changes corresponding to neighbors' red blocks and is communicated via message passing.



(a) Node view.



(b) Distributed view (with neighbor dependencies).

Figure 3.4: Dependencies between sub-computations in gradient program (Listing 3.6).

3.2 Design of FRASP

3.2.1 Architecture

The architecture of FRASP is shown in Figure 3.5. The design is organized into three packages: `core`, which includes basic type definitions (`Core`) as well as the components for the DSL (`Language` for primitives and `RichLanguage` for other built-ins) and its "virtual machine" (`Semantics`), overall captured by an `Incarnation`; `frp`, which provides an interface to the FRP engine (`FrpEngine`), possibly also decoupling from the specific FRP library adopted, as well as extensions (`FrpExtensions`) useful for the definition of FRASP constructs; and simulation, which provides basic simulation support.

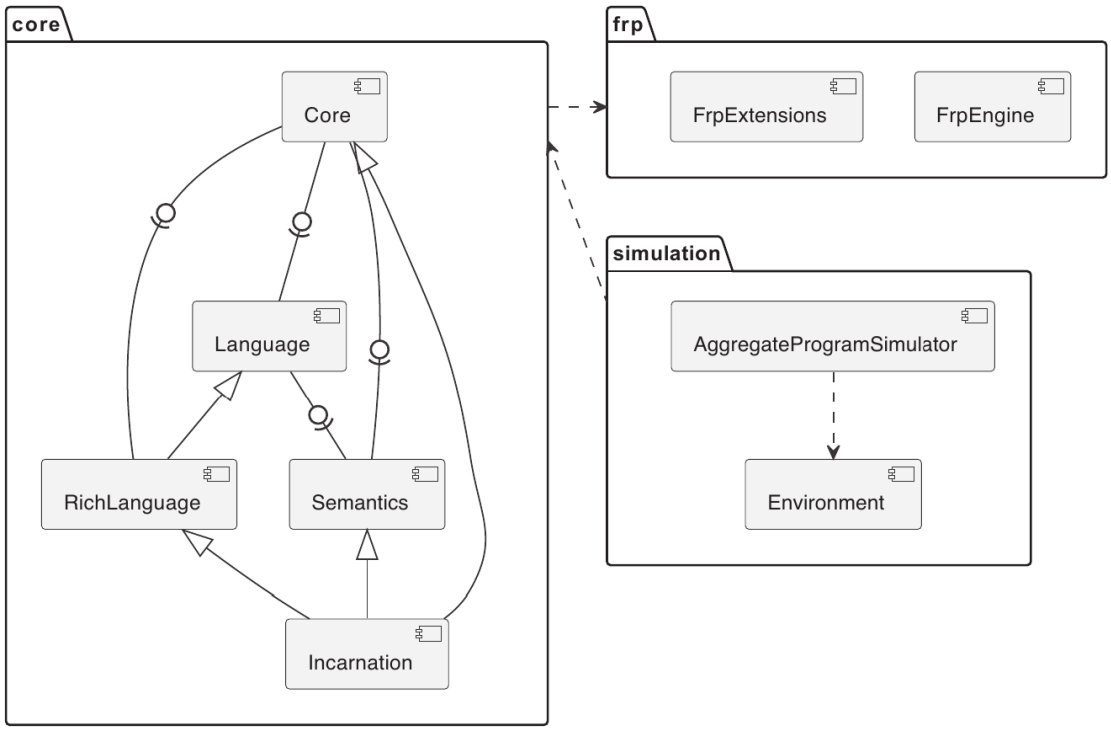


Figure 3.5: FRASP architecture.

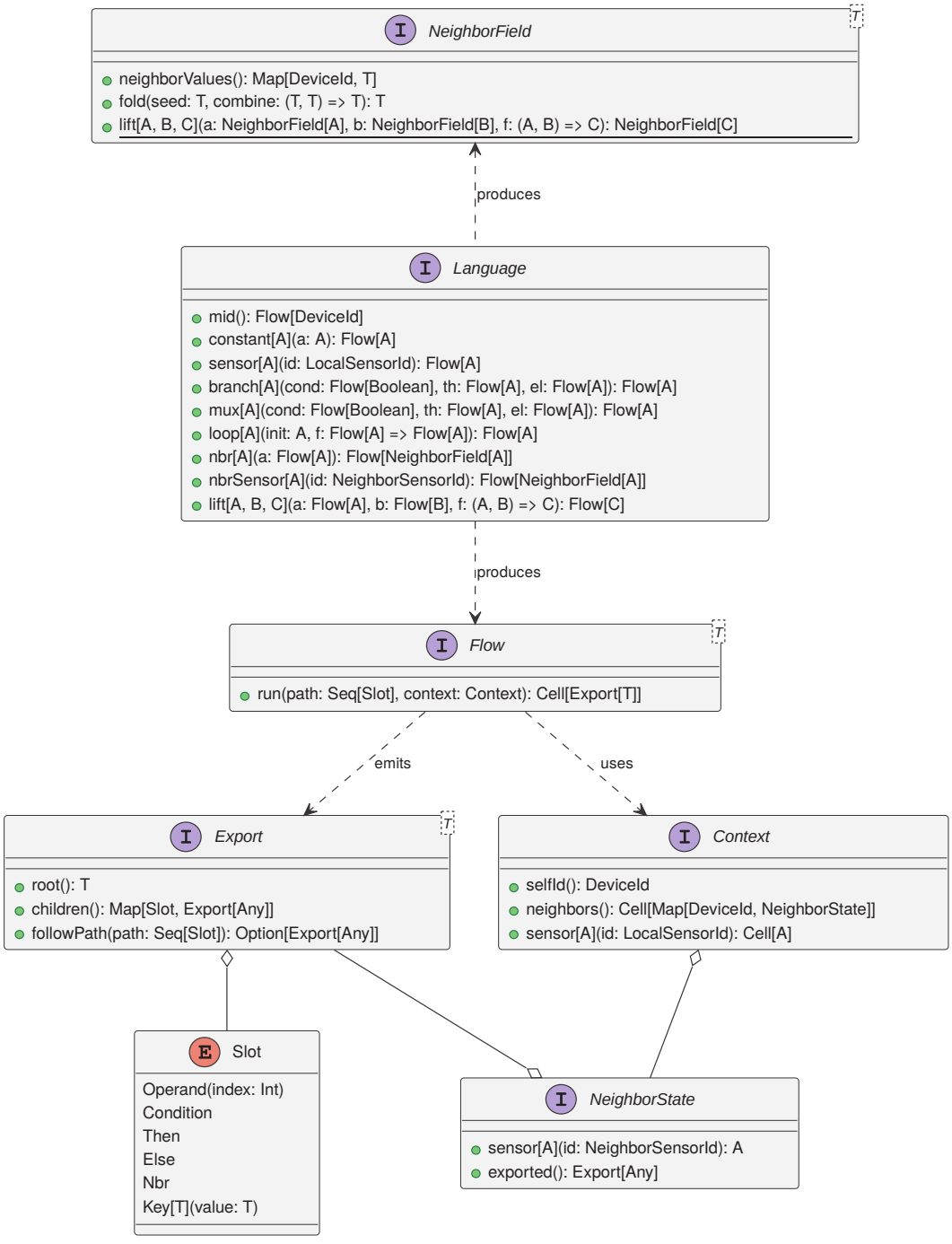


Figure 3.6: Detailed design of FRASP.

3.2.2 Detailed Design

FRASP has been implemented in Scala, using Sodium[4] as FRP library. Scala is well known for its suitability as a host for embedded DSLs and aggregate computing embeddings as well. The design of the FRASP DSL is detailed in Figure 3.6.

Following the system/execution model described in Section 3.1.5, the input and output of a (sub-)program are modeled through an interface `Context`, providing access to local sensor data and neighbor data; and an interface `Export`, capturing outputs and data that must be shared with neighbors. In particular, an `Export` is modeled as a tree where each node is a `Slot` (corresponding to a particular language construct) with an associated value, and can be located through a path of slots—e.g., `S1/S2/S3` identifies a node in the export tree, where `S1` depends on `S2` which depends in turn on `S3` (so, a change in the output `S3` will cause the expression corresponding to `S2` to re-evaluate, and possibly `S1` in turn). `Flow` is the type of a reactive (sub-)computation, which takes a `Context` (providing its inputs), a `Seq[Slot]` as path (indicating its position in the export tree), and returns `Cell` (i.e. a time-varying value) of `Export`. Each `Language` construct returns a `Flow`: therefore, the constructs do not immediately run upon evaluation, but rather an executable, reactive object denoting a computation graph whose nodes will execute as a response to change (Figure 3.4a). Access to neighbor-related data is mediated by a `NeighborField` abstraction, which is the same provided by constructs supporting interaction with neighbors, i.e., `nbr` and `nbrSensor`.

3.3 Design of Collektive

3.3.1 Architecture

Collektive has been developed as a Gradle project composed of three different submodules. The cited submodules are, namely:

- `plugin`, that is divided into two submodules:
  - `gradle-plugin`: the necessary plugin used by a gradle project to include the compiler plugin.

[4] `https://github.com/SodiumFRP/sodium`.

  - `compiler-plugin`: the compiler plugin is used to modify the data structure which is responsible for keeping track of the stack at runtime. For each aggregate function and branch construct, the stack data structure is updated to allow alignment whenever necessary.

- `dsl`: the actual DSL implementation in Kotlin Multiplatform, where the logic is implemented and that exposes the operators of the aggregate computing.

- `alchemist-incarnation-collektive`: allows to integrate Collektive simulations in the Alchemist [28] simulator.

3.3.2 Detailed Design

The detailed design of Collektive is presented in Figure 3.7. The `Collektive` class represents a device with a specific `localId` and a `Network` to manage incoming and outgoing messages, it takes a function to apply within the `AggregateContext`. `Collektive` implements two different execution strategies:

- `cycle`: it applies once the aggregate function to the parameters of the device, then returns the result of the computation.

- `cycleWhile`: it applies the aggregate function to the parameters of the device while the condition is satisfied, then returns the result of the computation.

`cycle` and `cycleWhile` implicitly use the `aggregate` function, which is the entry point of the aggregate program. It computes an iteration of a device (`localId`), taking as parameters the previous `state`, the messages received from the neighbors and the `compute` with `AggregateContext` object receiver that provides the implementation of the aggregate constructs. Another version of the `aggregate` function computes an iteration of a device, over a `network` of devices, optionally from a previous state (`previousState`), running the `compute` aggregate program. The `aggregate` function returns an `AggregateResult`, which is the result of the aggregate computation. It represents the `localId` of the device, the `result` of the computation, the messages to send (`toSend`) to other devices and the new state (`newState`) of the device.

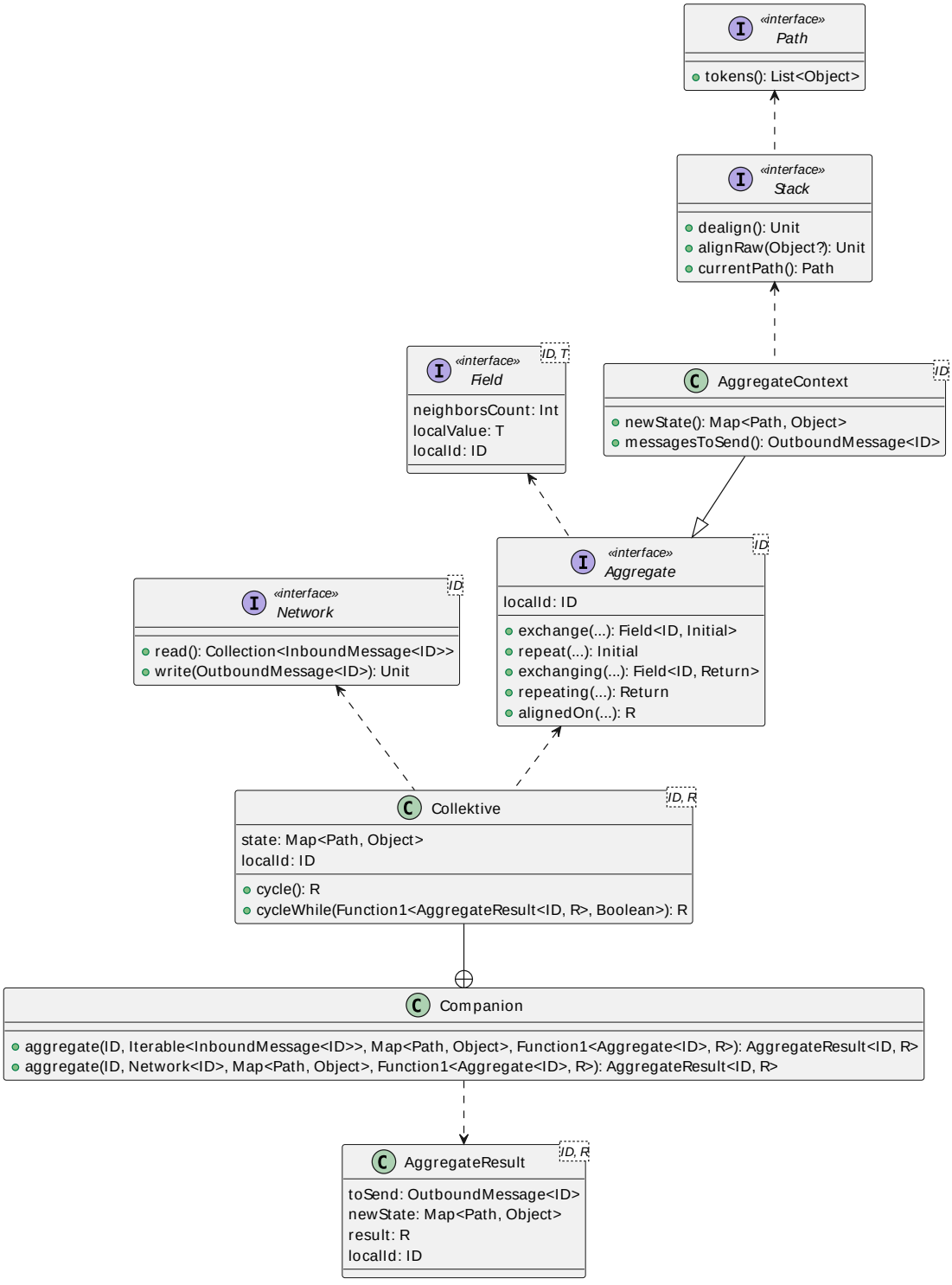


Figure 3.7: Detailed design of Collektive DSL.

The interface `Aggregate` models the minimal set of aggregate operations and holds the `localId` of the device executing the aggregate program.

The `alignOn` function is used for the alignment, it pushes in the stack the `pivot`, executes the `body` and pops the last element of the `Stack` after it is called, finally returns the `body`'s return element.

The `AggregateContext` class represents the context for managing aggregate computation. It encapsulates the `localId` of the device, the messages received from the neighbors, and the previous state (`previousState`) of the device. The actual implementation of the aggregate constructs is defined in this class.

3.3.3 Alignment Processing Strategy

The alignment processing pursues the following strategy: in the first instance, all the function definitions are visited and the ones involving aggregate computation will be subject to alignment processing. Then, for each candidate function, the plugin visits all the call sites in the body of the function and checks if the call has an aggregate reference or if it is involved in an aggregate computation. If so, the plugin will align the expression call. During the visiting of the function definition, branch conditions are also visited aligning only the branches that involve aggregate computation. If a branch body does not involve aggregate computation, the plugin will not align it. Aligning the branches in this way, by default all the branches follow the branch semantics of aggregate computing. The alignment strategy is formalized below:

1. Each function definition exhibiting the following characteristics is the target of the alignment processing:

    - The function has an `extensionReceiver` of type `Aggregate` or a subtype of it.
    - The function has a `dispatchReceiver` of type `Aggregate` or a subtype of it.
    - One or more of the function's parameters are of type `Aggregate` or a subtype of it.

2. For each candidate function, it aligns the call expressions having an aggregate reference or in-depth they involve an aggregate computation.

3.4 Re-implementation of FRASP in Collektive

Given the considerations regarding the proactive computational model made in Section 2.3.4 and the solution proposed in [16] with the related results of the evaluations performed, it is decided to introduce the FRASP model in Collektive. Analysis of the architectures of FRASP and Collektive, defined in Section 3.2 and Section 3.3, respectively, reveals substantial differences in technology and design choices. This considerably complicates the process of implementing the reactive model into Collektive. The possible issues identified during the analysis are described in Section 3.4.1.

3.4.1 Implementation Issues

Differences between Scala and Kotlin

The Scala implementation of FRASP is extremely concise, even though it models several aspects of aggregate programming. In addition, the DSL provided is particularly ergonomic, so the user can create aggregate programs easily and effectively. These characteristics of FRASP are due in part to the flexibility of Scala, which is given by the constructs that the language implements. The following are some Scala features that are used in FRASP but are not available by Kotlin:

- **Given instances and using clauses**: functional programming tends to express most dependencies as simple function parameterization. This is clean and powerful, but it sometimes leads to functions that take many parameters where the same value is passed over and over again in long call chains to many functions. Context parameters can help here since they enable the compiler to synthesize repetitive arguments instead of the programmer having to write them explicitly. Given instances define "canonical" values of certain types that serve for synthesizing arguments to context parameters.

- **Traits and self-types**: self-types are a way to declare that a trait must be mixed into another trait, even though it does not directly extend it. That makes the members of the dependency available without imports. A self-type is a way to narrow the type of `this` or another identifier that aliases `this`.

Differences in Paths and Exports Management

Typically, in aggregate computing implementations that respect the proactive model, paths are modeled as lists of tokens, while exports are represented by making use of maps that have the path as the key and the result of evaluating the sub-expression related to the path as the value. In FRASP, these entities are represented in a completely different way; this is due to the need to properly model the dependencies of reactive sub-expressions. In particular, an export is modeled as a tree (using a specially defined data structure) where each node is a token with an associated value and can be located through a path of tokens.

Diversity of Implemented Constructs

FRASP implements a reactive version of the constructs defined by the field calculus, this allows a sub-expression to be automatically re-evaluated as one of the sub-expressions on which it depends changes. On the other hand, in addition to the field calculus constructs, Collektive implements a proactive version of `exchange` and `share`, so a reactive version of these two constructs must be provided.

Divergences between Reactive and Proactive Models

In the proactive model, at each round, the aggregate expression is reevaluated entirely, taking into account the following parameters passed in as input: previous state, neighbor messages, and sensors states. This behavior differs completely from the reactive model, where it is necessary to think in terms of dependencies between information flows instead of computational rounds. In other words, it is necessary to revisit the design of Collektive by modifying some aspects of it. The state, sensors, and messages of neighbors must be modeled as reactive entities, of which the values change over time; furthermore, in addition to modeling the

aggregate constructs so that they are reactive, it is necessary to adequately define the dependencies between them and the context (state, sensors and neighbors messages) in which they are executed.

3.4.2 Feasibility of Reactive Aggregate Programming in Kotlin

Even before addressing the problems presented in Section 3.4.1, it is necessary to understand whether it is possible to create a Kotlin version of FRASP and, if so, analyze the similarities and differences with the model implemented in Scala, considering, in particular, the ergonomics of the DSL. In this regard, an implementation of FRASP in Kotlin that makes use of the Flow library is proposed in the analysis phase to demonstrate the actual feasibility. Reactive values are modeled by making use of `StateFlow<T>`, whose behavior is similar to that of a `Cell`, described in Section 3.1.5.

Figure 3.8 shows the design of the Kotlin version of FRASP. The implementation[5] includes only the subset of features from the Scala version of FRASP needed to demonstrate the feasibility of reactive aggregate programming. An aggregate expression is represented by a dedicated data structure (`AggregateExpression`) that produces a `StateFlow<ExportTree<T>>`, its behavior is similar to `Flow[T]` described in Section 3.1.5. The context of a device (`Context`) includes neighboring states, self ID and the state of the local sensors. Exports are modeled through a tree-like recursive data structure (`ExportTree`) holding data of type `T`. It exposes methods for accessing the `children` and the `root` element of a node and navigating through the tree using a `path`. `Semantics` implements field calculus functions (plus others useful in the reactive context) for constructing an `AggregateExpression`. These functions include `branch`, `constant`, `loop`, `mux`, `neighbor`, `selfID`, and `sense`. The enum `Slot` defines different types of tokens used within the `ExportTree` structure.

Given the proposed design in Figure 3.8 and its implementation, to demonstrate the ergonomics of the related DSL, the implementation of the gradient is provided in Listing 3.7. The results obtained demonstrate the actual feasibility of

[5] `https://github.com/FilippoVissani/kotlin-distributed-frp`

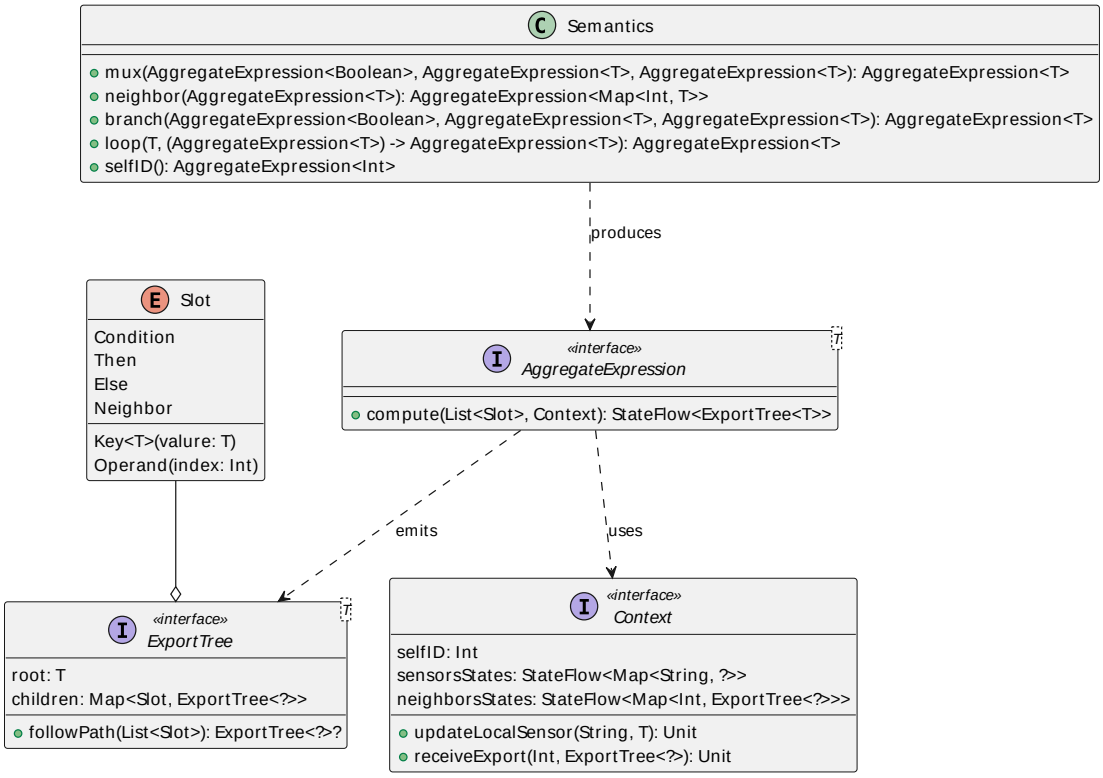


Figure 3.8: Detailed design of Kotlin Distributed FRP.

the solution, not only that, the DSL appears ergonomic and easy to use, just like the one proposed in Scala (Listing 3.6).

3.4.3 Solutions Identified

The main goal is to introduce the reactive paradigm into Collektive, replacing it with the proactive one. At the same time, we want to make sure that the resulting DSL is ergonomic and thus it is easy for the user to understand and to use it. It should be noted that, given the diversity of the two architectures, it is not possible to implement the FRASP model directly in Collektive, so an intermediate solution must be identified.

Given the results obtained from the analysis reported in Section 3.4.2 and taking into consideration the possible implementation issues identified in Section 3.4.1, we propose two possible solutions for implementing the reactive paradigm into

Listing 3.7: Gradient implementation in Kotlin Distributed FRP.

```
fun gradient(): AggregateExpression<Double> {
    return loop(Double.POSITIVE_INFINITY) { distance ->
        mux(
            sense(Sensors.IS_SOURCE.sensorID),
            constant(0.0),
            neighbor(distance)
                .withoutSelf()
                .min()
                .map { it + 1 }
        )
    }
}
```

Collektive:

- **RMSM**: it consists of building the reactive model on top of the proactive one. This can be achieved by making the messages and sensors reactive; every time the value of one of these is changed the expression is re-evaluated entirely, executing a round. This solution has the advantage that the Collektive DSL remains identical to the current one. On the other hand, it is not possible to exploit the partial updates of the sub-expressions proposed in FRASP, this translates into lower efficiency.

- **PRM**: it consists of re-engineering all aggregate constructs and context definitions in Collektive to be reactive. In this case, important changes must be made to the current design of Collektive, making the implementation more complicated than the first solution. Furthermore, it is not possible to guarantee the ergonomics of the resulting DSL, as the definition of the aggregate constructs needs to be revised. The advantage of this solution is that, as in FRASP, it is possible to model the dependencies of the sub-expressions so that the latter can be updated only when necessary.

Chapter 4

Design

This chapter delves into the design of a reactive extension for the Collektive framework. Collektive enables the execution of aggregate computations across distributed devices. This chapter introduces two novel models that incorporate reactive principles into Collektive's design.

The chapter begins by providing an overview of the current Collektive architecture. It then outlines the proposed reactive architecture, highlighting the introduced components and their functionalities (Section 4.1).

Following the architectural overview, the chapter dives into the detailed design of two reactive models proposed (Section 4.2).

4.1 Architecture

As mentioned in Section 3.3.1, Collektive consists of three main modules:

alchemist-incarnation-collektive It is responsible for enabling the integration of Collektive simulations into Alchemist.

compiler-plugin It takes care of visiting the abstract syntax tree of the aggregate expression and modifying the function call stack to correctly align the devices that execute the aggregate program.

dsl This module defines the following components:

- `aggregate`: deals with defining the context related to a device, the semantics of aggregate constructs, and the data structures necessary for path definition and device alignment;
- `field`: contains the definition of computational field and the related functionalities for manipulating the latter;
- `state`: defines the association between the paths and the results of their evaluations;
- `path`: defines the data structures necessary to represent the abstract syntax tree relating to the aggregate expression;
- `networking`: defines the data structures necessary for distributed device communication.

Given the solutions proposed in Section 3.4.3, in both cases, it is necessary to review some of the entities present in Collektive so that it is possible to detect and react to their changes. Regardless of the detailed solution chosen, given that the Collektive design allows it, it is possible to introduce the necessary functionalities as an extension of the current ones. The proposed architecture is shown in Figure 4.1, the components in gray are part of the current Collektive architecture, and those in orange introduce the entities that enable reactive aggregate programming. The component `reactive` extends `aggregate` to introduce a reactive version of the entities described above and `network` to allow reactive distributed communication between devices. The component `flow.extensions` is used to simplify some operations for combining and mapping flows. According to this design choice, the other modules in the project (`compiler-plugin` and `alchemist-incarnation-collektive`) are not altered; consequently, the reactive model introduced continues to make use of the compiler plugin for the definition of the paths.

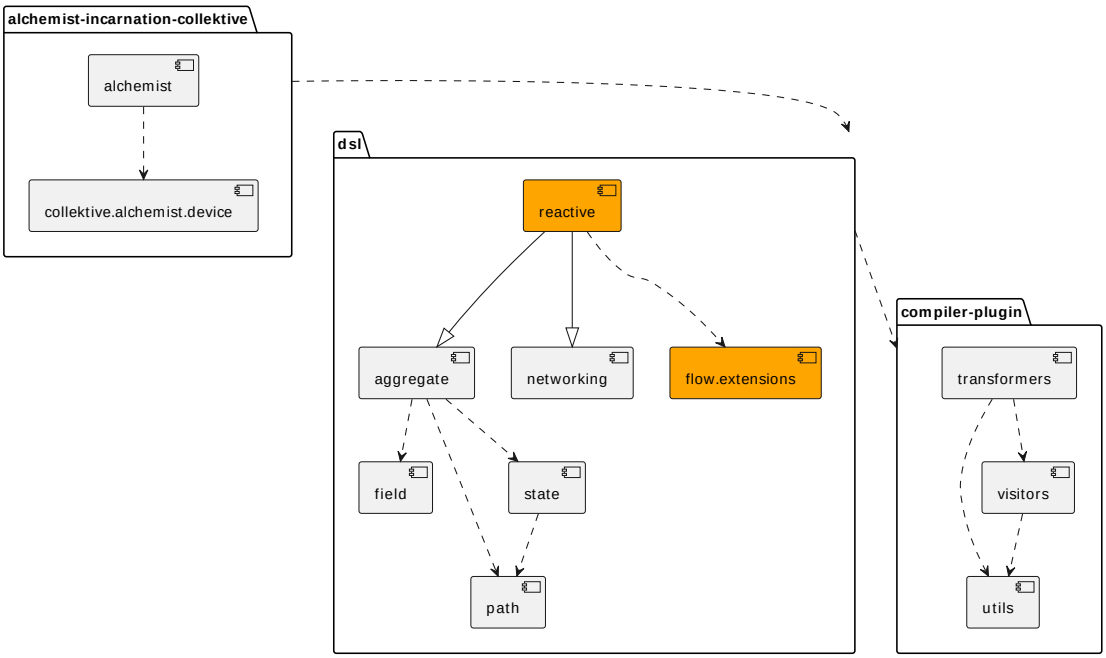


Figure 4.1: Architecture of the reactive model proposed. The gray-colored components are part of the original Collektive architecture, while the orange-colored components are used to introduce the reactive paradigm.

4.2 Detailed Design

4.2.1 Purely Reactive Model

The detailed design of the PRM is shown in Figure 4.2. Within the `RCollektive` class, the methods for executing aggregate programs have been augmented to accommodate reactive functionalities. The method `execute` now takes a

`MutableStateFlow<List<InboundMessage<ID>>>` as parameter, enabling the system to react to incoming messages from neighboring devices. Similarly, the `execute` method now also accepts a `ReactiveNetwork<ID>` parameter, facilitating reactive communication among devices within the network. These additions signify a departure from the static nature of traditional aggregate computations, allowing for dynamic adjustments based on real-time changes in the environment.

In the PRM, the concept of expression result (`RAggregateResult`) is revisited to imbue reactivity. This ensures that the aggregate expression's result is not static but rather dynamic, adapting to alterations in the underlying data or environmental conditions. Moreover, the `Aggregate` interface undergoes modifications to accommodate reactive versions of aggregate constructs. Parameters within these constructs are bound to `StateFlow`, enabling reevaluation whenever their inputs experience changes.

The `RAggregateContext` class extends the `Aggregate` interface, providing concrete implementations of reactive aggregate constructs. Additionally, it hosts reactive data structures essential for managing outbound messages and states. Leveraging `StateFlow` extensions (`StateFlowExtensions`), this class simplifies the mapping and combining of hot flows, enhancing the efficiency and scalability of the system.

Behavioral Characteristics

The PRM exhibits distinct behavioral characteristics that distinguish it from traditional static computation approaches:

- **Reactive computations**: computation occurs reactively in response to environmental changes, minimizing resource wastage and ensuring optimal responsiveness.

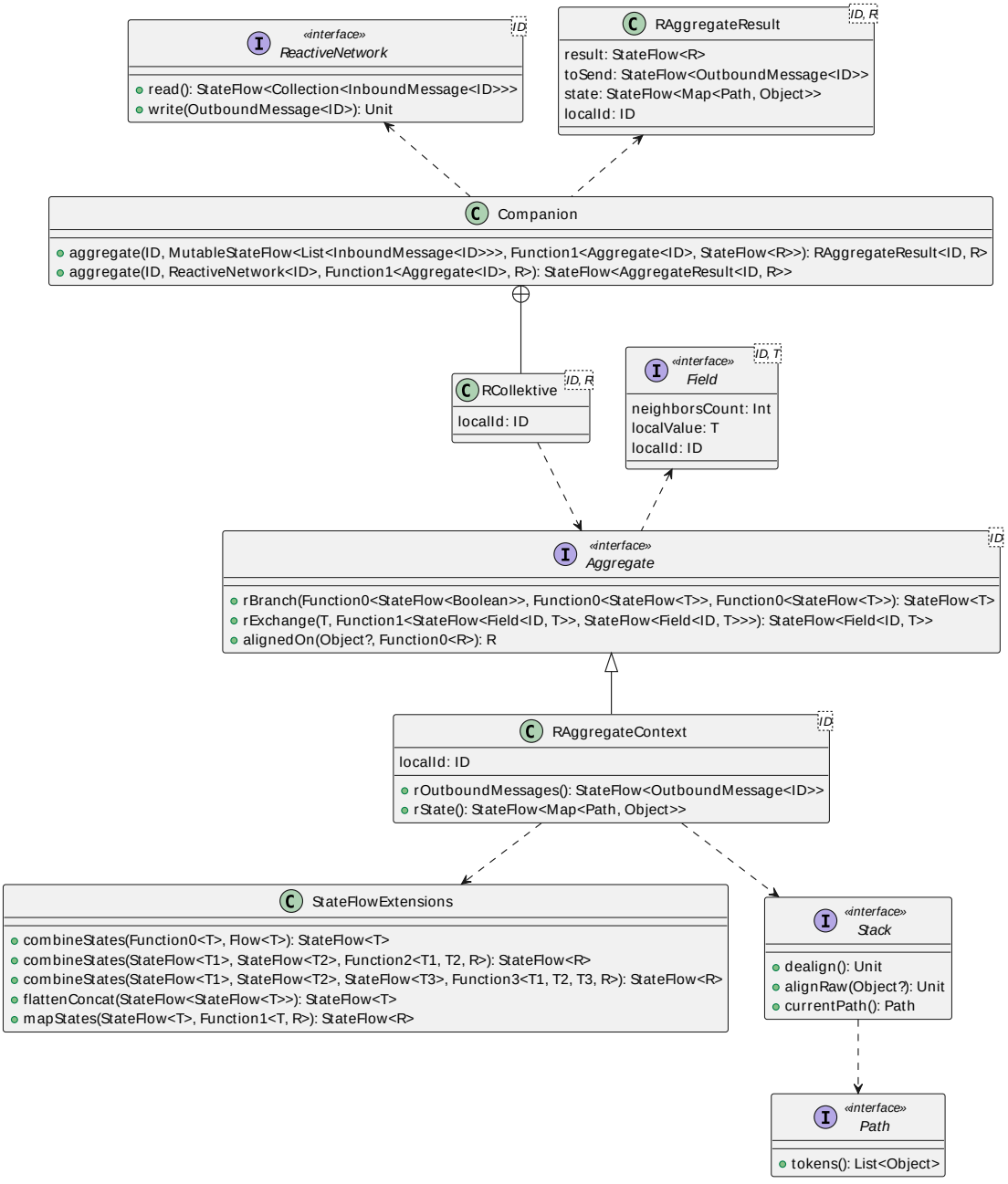


Figure 4.2: Detailed design of the PRM proposed.

- **Efficient message exchange**: devices only broadcast messages when necessary, reducing unnecessary communication.
- **Selective reevaluation**: sub-expressions are reevaluated only when their

dependencies change, optimizing computation efficiency and reducing redundant processing.

By embodying these characteristics, the PRM enhances the scalability, and robustness of the Collektive framework, empowering it to effectively handle dynamic and unpredictable environments.

4.2.2 Model with Reactive Messages and Sensors

The detailed design of the RMSM is shown in Figure 4.3. Similar to the PRM, the `Collektive` class accommodates two new versions of the `Aggregate` method to facilitate reactive functionalities. These methods accept either a

`StateFlow<Iterable<InboundMessage<ID>>>` or a `ReactiveNetwork<ID>` parameter, enabling dynamic adjustment of aggregate computations based on real-time changes in message reception.

In this model, the `Aggregate` method encompasses the logic necessary for reactive evaluation of the aggregate expression. Unlike the PRM, where the `RAggregateContext` class handles reactive evaluation, here the evaluation logic is embedded directly within the `Aggregate` method. Consequently, upon receiving a message from a neighbor, the entire aggregate expression undergoes reevaluation, resulting in a complete round of computation. While maintaining compatibility with the original Collektive DSL, this approach sacrifices some performance due to the exhaustive reevaluation process.

Behavioral Characteristics

The RMSM exhibits behavioral characteristics that align with its design principles:

- **Compatibility**: Retains compatibility with the original Collektive DSL, ensuring seamless integration with existing codebase and workflows.
- **Simplified Implementation**: Implements reactive functionalities within the `Aggregate` method, streamlining the implementation process and reducing complexity.

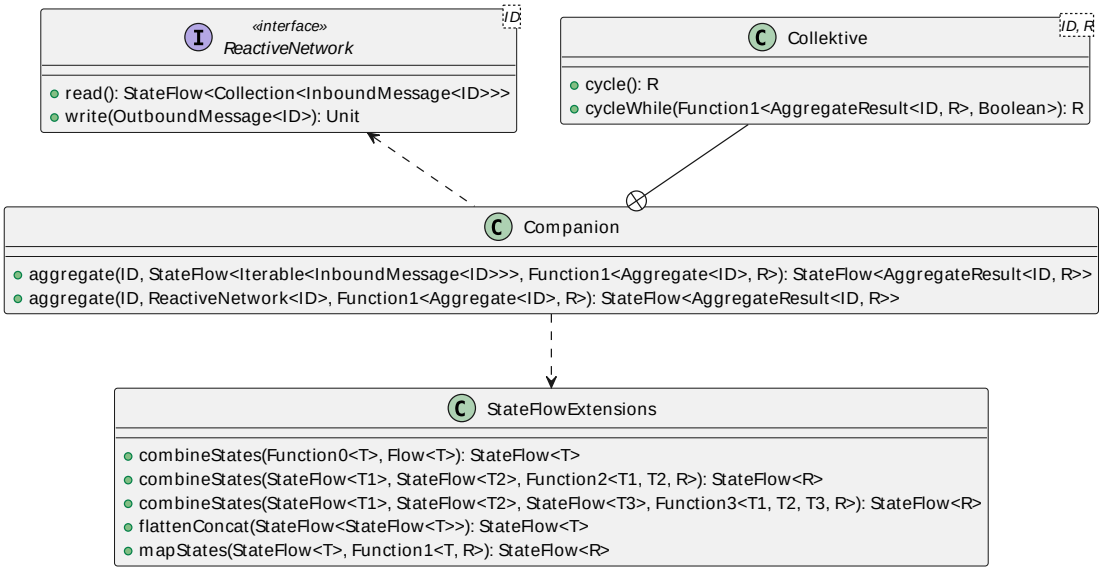


Figure 4.3: Detailed design of the RMSM proposed.

- **Performance Trade-off**: Sacrifices some performance for compatibility, as the entire aggregate expression undergoes reevaluation upon receiving a message, potentially leading to redundant processing.

Despite the performance trade-off, this model provides a pragmatic approach to introducing reactivity into the Collektive framework, catering to scenarios where compatibility and ease of integration are paramount.

In summary, both models offer distinct approaches to incorporating reactive principles into the Collektive framework, each tailored to different use cases and design priorities.

Chapter 5

Implementation

This chapter delves into the implementation details of the PRM (Section 5.1) and RMSM (Section 5.2) introduced for Collektive. Each section focuses on a specific model, exploring its key components and implementation choices.

5.1 Purely Reactive Model

In this section, the implementation of some key components of the PRM is proposed. The `aggregate` function (Listing 5.1) represents the entry point of the aggregate program; this function takes as input the device ID, a flow relating to inbound messages and an aggregate program whose result is bound to the `StateFlow<R>` type. In the body of the function, a `RAggregateContext` is created with the parameters passed as input and the aggregate program is executed in the context relating to the newly created object. The result of the function is a `RAggregateResult`, thanks to which it is possible to access the result of the aggregate expression, the outbound messages and the state of the device. The data structures within `RAggregateResult` are defined as flows, so it is possible to subscribe to and react to their changing. In this way, it is easy to establish the dependency between the inbound messages of one device and the outbound messages of another, so that the first reacts to the change of the second state.

Listing 5.2 provides the implementation of the `RAggregateContext` class, which is responsible for defining the context in which the aggregate expression is executed

Listing 5.1: Implementation of the `aggregate` function in the PRM.

```
fun <ID : Any, R> aggregate(
    localId: ID,
    rInboundMessages: StateFlow<Iterable<InboundMessage<ID>>>,
    compute: Aggregate<ID>.() -> StateFlow<R>,
    ): RAggregateResult<ID, R> = RAggregateContext(localId, rInboundMessages).run
        {
        RAggregateResult(localId, compute(), rOutboundMessages(), rState())
    }

data class RAggregateResult<ID : Any, R>(
    val localId: ID,
    val result: StateFlow<R>,
    val toSend: StateFlow<OutboundMessage<ID>>,
    val state: StateFlow<State>,
)
```

and on which the result of the latter depends. Here the actual implementation of the aggregate constructs is defined, which takes advantage of some utility functions:

- The `rMessagesAt` function takes care of returning inbound messages relating to a `path`.
- The `rStateAt` function returns the result of the evaluation of the given `path` using the `default` value if the result does not exist yet.
- The function `alignedOn` is used to define paths, it pushes on the stack the given `pivot`, executes the `body` function and pops the first token on the stack.

The functions passed as input to the aggregate constructs and the related result of the latter are bound to the `StateFlow` type so that it is possible to react to their changes.

The result of the `rBranch` construct (Listing 5.3) depends on:

- the result of the evaluation of the `condition`;
- the result of the evaluation of the `th` branch in the case that the condition is `true`;
- the result of the evaluation of the `el` branch in the case that the condition is `false`.

Listing 5.2: Implementation of the `RAggregateContext` class in the PRM.

```
class RAggregateContext<ID : Any>(
    override val localId: ID,
    private val rInboundMessages: MutableStateFlow<List<InboundMessage<ID>>> =
        MutableStateFlow(emptyList()),
) : Aggregate<ID> {

    private val stack = Stack()
    private val rState: MutableStateFlow<State> = MutableStateFlow(emptyMap())
    private val rOutboundMessages: MutableStateFlow<OutboundMessage<ID>> =
        MutableStateFlow(OutboundMessage(localId, emptyMap()))

    fun rState(): StateFlow<State> = rState.asStateFlow()

    fun rOutboundMessages() = rOutboundMessages.asStateFlow()

    @OptIn(DelicateCoroutinesApi::class)
    override fun <T> rExchange(
        initial: T,
        body: (StateFlow<Field<ID, T>>) -> StateFlow<Field<ID, T>>,
    ): StateFlow<Field<ID, T>> {...}

    override fun <T> rBranch(
        condition: () -> StateFlow<Boolean>,
        th: () -> StateFlow<T>,
        el: () -> StateFlow<T>,
    ): StateFlow<T> {...}

    private fun deleteOppositeBranch(condition: Boolean) {...}

    private fun <T> newField(localValue: T, others: Map<ID, T>): Field<ID, T> =
        Field(localId, localValue, others)

    @Suppress("UNCHECKED_CAST")
    private fun <T> rMessagesAt(path: Path): StateFlow<Map<ID, T>> = mapStates(
        rInboundMessages) { messages ->
        messages
            .filter { it.messages.containsKey(path) }
            .associate { it.senderId to it.messages[path] as T }
    }

    private fun <T> rStateAt(path: Path, default: T): StateFlow<T> = mapStates(
        rState) { state ->
        state.getTyped(path, default)
    }

    override fun <R> alignedOn(pivot: Any?, body: () -> R): R {
        stack.alignRaw(pivot)
        return body().also { stack.dealign() }
    }
}
```

Listing 5.3: Implementation of the `rBranch` construct in the PRM.

```
override fun <T> rBranch(
    condition: () -> StateFlow<Boolean>,
    th: () -> StateFlow<T>,
    el: () -> StateFlow<T>,
): StateFlow<T> {
    val currentPath = stack.currentPath()
    return condition().mapStates { newCondition ->
        currentPath.tokens().forEach { stack.alignRaw(it) }
        val selectedBranch = if (newCondition) th else el
        deleteOppositeBranch(newCondition)
        alignedOn(newCondition) {
            selectedBranch()
        }.also {
            currentPath.tokens().forEach { _ -> stack.dealign() }
        }
    }.flattenConcat()
}
```

Regardless of which branch is chosen, the result of the other branch is deleted using the `deleteOppositeBranch` function, this is because otherwise, when the condition changes, the devices would also remain aligned on the branch relating to the previous condition. The implementation of the construct also highlights that the `alignedOn` function is called explicitly when in reality it should be called by the compiler plugin. This is because the compiler plugin only aligns calls to aggregate functions, while within `rBranch` the `mapStates` function is called, which is not an aggregate function. This behavior of the compiler plugin can be considered a problem that needs to be resolved. With correct plugin behavior the `rBranch` implementation would not need calls to the `alignedOn`, `stack.dealign` and `stack.alignRaw` functions.

The `rExchange` construct (Listing 5.4) takes as input a `body` function whose result depends on the previous state and the messages received; as the result of this function changes, the outbound messages and the state of the device are updated.

5.2 Model with Reactive Messages and Sensors

The central component of the implementation with reactive messages and sensors is the `aggregate` function (Listing 5.5). This function takes as input the same parameters as the one defined in the PRM, the difference is that the re-

Listing 5.4: Implementation of the `rExchange` construct in the PRM.

```
@OptIn(DelicateCoroutinesApi::class)
override fun <T> rExchange(
    initial: T,
    body: (StateFlow<Field<ID, T>>) -> StateFlow<Field<ID, T>>,
): StateFlow<Field<ID, T>> {
    val messages = rMessagesAt<T>(stack.currentPath())
    val previous = rStateAt(stack.currentPath(), initial)
    val subject = messages.mapStates { m -> newField(previous.value, m) }
    val alignmentPath = stack.currentPath()
    return body(subject).also { flow ->
        flow.onEach { field ->
            val message = SingleOutboundMessage(field.localValue, field.
                excludeSelf())
            rOutboundMessages.update { it.copy(messages = it.messages + (
                alignmentPath to message)) }
            rState.update { it + (alignmentPath to field.localValue) }
        }.launchIn(GlobalScope)
    }
}
```

sult of the aggregate program is not bound to the `StateFlow` type. The result of `aggregate` is defined as `StateFlow<AggregateResult<ID, R>>`, this wrapping allows us to avoid having to create specific data structures for managing the aggregate result, which instead happens in the PRM. In the `aggregate` function, an `AggregateContext` flow is created that depends on the inbound message and state flows. The result of the aggregate expression depends on the flow of the `AggregateContext`; as the result of the expression changes, the flow relating to the state is updated.

Listing 5.5: Implementation of the `aggregate` function in the RMSM.

```
fun <ID : Any, R> aggregate(
    localId: ID,
    inbound: StateFlow<Iterable<InboundMessage<ID>>>,
    compute: Aggregate<ID>.() -> R,
): StateFlow<AggregateResult<ID, R>> {
    val states = MutableStateFlow<State>(emptyMap())
    val contextFlow = inbound.mapStates {
        AggregateContext(localId, it, states.value)
    }
    return contextFlow.mapStates { aggregateContext ->
        aggregateContext.run {
            AggregateResult(localId, compute(), messagesToSend(), newState()).also
                {
                states.update { this.newState() }
            }
        }
    }
}
```

Chapter 6

Validation

This chapter delves into the evaluation of the reactive extensions introduced into the Collektive framework. The chapter is divided into two main sections:

- Section 6.1 details the unit testing strategy employed to ensure the correctness of the implemented code. It highlights the chosen testing framework, and the testing style adopted.
- Section 6.2 compares the usability of the DSL for implementing aggregate programs in the two proposed reactive models. To facilitate the comparison, an example program implementing the "gradient with obstacles" scenario is presented in both DSLs. This allows for a concrete side-by-side assessment of the strengths and weaknesses of each model from a usability perspective.

6.1 Testing

This section delves into the testing strategies employed, focusing on unit testing methodologies.

The project adopts a rigorous approach to testing, leveraging the Kotest[1] framework for automated testing in Kotlin. Kotest provides a robust testing environment conducive to comprehensive test suites. Among its testing styles, the project opted for `StringSpec` due to its straightforward structure, which facilitates

[1] `https://kotest.io/`.

a behavior-driven approach to test composition. The most relevant tests within the project are those that verify the behavior of the aggregate constructs.

Unit tests are designed to verify the behavior of the aggregate constructs, ensuring they function as expected across various scenarios. Tests are crafted to cover different aspects of the reactive functionality, ensuring the accurate alignment of devices, the correctness of values exchanged and the correctness of aggregate expressions' results.

An example test case for the `rExchange` construct is presented in Listing 6.1 to illustrate the testing approach. The test case encompasses the following steps:

1. Definition of the test name and sequential execution within a coroutine.
2. Definition of the aggregate result based on the execution of the aggregate program in a specific aggregate context.
3. Launching a concurrent job to execute the simulation.
4. Introduction of a delay and subsequent cancellation of the job.
5. Assertion of the expected results against the computed values.

The provided example test serves as a template for testing other reactive constructs, ensuring thorough validation of their behavior.

Listing 6.1: Part of the test suite related to the `rExchange` construct.

```
"rExchange should return the initial value" {
    runBlocking {
        val aggregateResult0 = RCollektive.aggregate(id0) {
            rExchange(initV1, increaseOrDouble)
        }
        val job = launch(Dispatchers.Default) {
            runSimulation(mapOf(aggregateResult0 to MutableStateFlow(emptyList()))
                )
        }
        delay(100)
        job.cancelAndJoin()
        aggregateResult0.toSend.value.senderId shouldBe id0
        aggregateResult0.toSend.value.messages.values shouldContain
            SingleOutboundMessage(expected2, emptyMap())
    }
}
```

6.2 Analysis of the Ergonomics of the Proposed Models

This section evaluates the usability and effectiveness of the proposed reactive models within the Collektive framework. The evaluation focuses on readability, maintainability, flexibility, and the learning curve associated with each model.

The aggregate program chosen to carry out this evaluation is the gradient with obstacles, which maintains the properties of the classic gradient, but introduces obstacles into the environment. Figure 6.1 shows a graphical representation of what we want to achieve. There are three types of nodes in the environment: sources (green), obstacles (red) and defaults (blue). The objective is to calculate the distance of each node from the nearest source without considering the neighbors who are defined as obstacles. The environment used in this case is a grid with five columns and five rows, where each device is a neighbor of the nearest device in each horizontal and vertical direction. In addition, the device with ID 0 is a source node, while devices with ID 2, 7, and 12 are obstacles.

Listing 6.2 and Listing 6.3 present the implementation of the gradient with obstacles in the PRM and in the RMSM, respectively. In both cases the node type is defined as `StateFlow<NodeType>`, allowing to change sources and obstacles at runtime. What changes is how this flow is managed: in the purely reactive case it is used directly within the aggregate constructs, while in the other a specific simulator must be created, which reevaluates the expression as the type of node varies. As regards the use of aggregate constructs within the program, the RMSM is equivalent to the proactive model, while in the PRM, the use of functions for manipulating flows introduces greater complexity.

The differences between the two implementations are analyzed in detail below:

1. Initially, the `if` (RMSM) and `rBranch` (PRM) constructs are used to isolate obstacles from the rest of the nodes. Nodes identified as obstacles will return the value -1.0 and will not execute the gradient function. In RMSM the condition is verified directly, while in PRM it is necessary to use the `mapStates` function to convert the node type into a boolean value. In the PRM, since the return value is bound to the `StateFlow` type, to return the

Listing 6.2: Gradient with obstacles implementation in PRM.

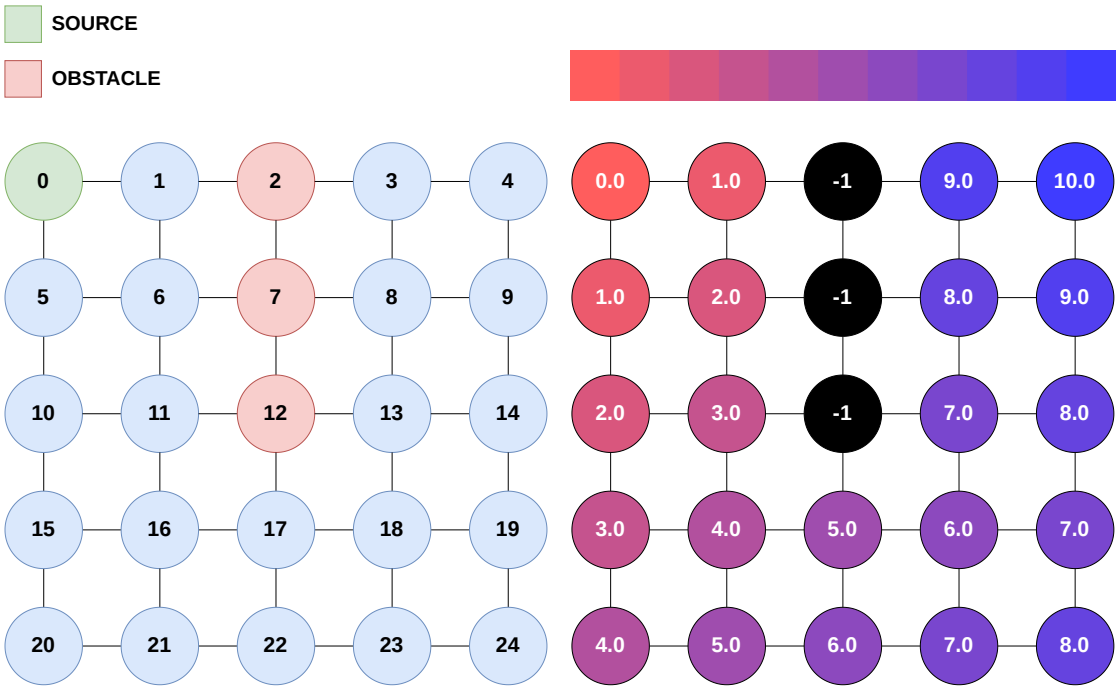
```
fun Aggregate<Int>.gradient(sourceFlow: StateFlow<Boolean>): StateFlow<Double> =
    rShare(Double.POSITIVE_INFINITY) { fieldFlow ->
        rMux(
            { sourceFlow },
            { MutableStateFlow(0.0) },
            { fieldFlow.mapStates { it.plus(1.0).min(Double.POSITIVE_INFINITY) }
                },
        )
    }

fun Aggregate<Int>.gradientWithObstacles(nodeTypeFlow: StateFlow<NodeType>):
    StateFlow<Double> =
    rBranch(
        { nodeTypeFlow.mapStates { it == NodeType.OBSTACLE } },
        { MutableStateFlow(-1.0) },
        { gradient(nodeTypeFlow.mapStates { it == NodeType.SOURCE }) },
    )
```

Listing 6.3: Gradient with obstacles implementation in RMSM.

```
fun Aggregate<Int>.gradient(source: Boolean): Double =
    share(Double.POSITIVE_INFINITY) { field ->
        when {
            source -> 0.0
            else -> (field + 1.0).min(Double.POSITIVE_INFINITY)
        }
    }

fun Aggregate<Int>.gradientWithObstacles(nodeType: NodeType): Double =
    if (nodeType == NodeType.OBSTACLE) {
        -1.0
    } else {
        gradient(nodeType == NodeType.SOURCE)
    }
```



(a) (b)

Figure 6.1: Figure 6.1a presents the environment where the gradient with obstacles was executed. The node highlighted in green represents the source, while those in red represent the obstacles. Figure 6.1b presents the output field of the gradient with obstacles after stabilization.

value -1.0 (if the condition is true) it is necessary to wrap the latter in a `MutableStateFlow`. This constraint is not present in the RMSM, since the return value does not have to be of type `StateFlow`.

2. If the node is not defined as an obstacle, the gradient is executed. The `share` (RMSM) and `rShare` (PRM) constructs are used to capture the space-time computation of the gradient. In this case, the way these two constructs are used is very similar.

3. Internally to `share` and `rShare` the `when` (RMSM) and `rMux` (PRM) constructs are used, respectively. Both constructs serve to distinguish the source from the rest of the nodes; given a node, if this is identified as the source then the value 0.0 is returned (base case), otherwise, to calculate the return value, the neighbor in which the gradient value is smaller is considered and

Listing 6.4: Gradient implementation in FRASP.

```
  def gradient(src: Flow[Boolean]): Flow[Double] =
    loop(Double.PositiveInfinity) { distance =>
      mux(src) {
        constant(0.0)
      } {
        liftTwice(nbrRange, nbr(distance))(_ + _).withoutSelf.min
      }
    }
```

Listing 6.5: Gradient implementation in Kotlin Distributed FRP.

```
fun gradient(): AggregateExpression<Double> {
    return loop(Double.POSITIVE_INFINITY) { distance ->
        mux(
            sense(Sensors.IS_SOURCE.sensorID),
            constant(0.0),
            neighbor(distance)
                .withoutSelf()
                .min()
                .map { it + 1 }
        )
    }
}
```

the distance (1.0) is added. From the two implementations, it appears that the syntax of the `when` construct is less intricate and more understandable than that used in the `rMux` construct. Furthermore, within `rMux` it is necessary to wrap the result in a `MutableStateFlow` when the condition is true, or use the functions to map flows when the condition is false; these operations are not necessary for the `when` construct.

Listing 6.4 and Listing 6.5 again show the gradient implementations (without obstacles) for FRASP and Kotlin Distributed FRP, respectively. Despite the differences regarding the aggregate constructs, languages (Scala and Kotlin) and the design of the frameworks used, there is some similarity in the four gradient implementations provided.

Based on the results obtained, the following considerations arise: in the PRM, the use of `rShare`, `rMux`, and `rBranch` might be less familiar to developers unfamiliar with this specific DSL. Understanding the syntax and purpose of these functions requires additional learning. The RMSM utilizes familiar syntax like `share` and conditional statements, potentially making it easier to read and under-

stand for developers with general programming experience. Composing complex logic using nested functions like `rMux` and `rBranch` can lead to nested code structures, potentially impacting maintainability as the codebase grows. In the RMSM conditional statements and function calls promote a more linear and explicit flow of logic, potentially improving maintainability. The DSL of the PRM provides dedicated functions for building reactive constructs, potentially offering more flexibility for complex reactive patterns. While offering less specialized syntax, the RMSM can still achieve various reactive behaviors. However, complex reactive patterns might require more verbose code compared to the purely reactive approach. The PRM requires learning the specific syntax and semantics of the DSL functions, while the RMSM leverages familiar programming constructs, potentially reducing the learning curve for developers with general programming experience.

Chapter 7

Conclusion

In this thesis, we have explored the feasibility and practicality of implementing reactive aggregate programming in Kotlin for developing artificial self-organizing systems. Our investigation has been guided by the overarching goal of crafting a programming language that enables developers to express macro-level behavior while abstracting away operational details, thus facilitating the self-organizing behavior among a group of agents or devices.

We began by delving into the foundational concepts of functional programming, reactive programming, and aggregate computing, elucidating their relevance and implementations in Kotlin. This served as the foundation on which we built our analyses and projects.

Through a critical assessment of existing frameworks such as Protelis, ScaFi, FCPP, Collektive, and FRASP, we identified key insights and gaps in the current state of the art. Subsequently, we detailed the design of FRASP and Collektive.

Our investigation into the re-implementation of FRASP into Collektive unveiled challenges, feasibility considerations, and proposed solutions, underscoring the intricacies involved in harmonizing disparate programming paradigms within a unified framework.

In the design phase, we delineated the architectural and detailed designs of the proposed models, laying the groundwork for their practical implementation. This implementation, divided into sections for the PRM and the RMSM, demonstrated the tangible realization of our theoretical constructs.

In evaluating the proposed models, we subjected them to testing procedures and analyzed their ergonomic aspects, providing valuable insights into their strengths and weaknesses.

In conclusion, our exploration has not only demonstrated the feasibility of reactive aggregate programming in Kotlin but has also contributed to advancing the discourse surrounding programming languages for self-organizing systems. By synthesizing our findings and encapsulating the contributions of this thesis, we pave the way for future research endeavors aimed at further refining and extending the capabilities of programming languages in facilitating the emergence of collective intelligence.

7.1 Future Work

In future work, several areas could be explored to further enhance the capabilities and usability of Collektive:

Support for Real-World Distributed Platforms Investigate ways to extend the framework to support deployment and execution on real-world distributed platforms. This could involve optimizations for distributed communication, fault tolerance mechanisms, and integration with existing distributed computing frameworks.

DSL Improvements Address the noise introduced in the API of the PRM due to the necessity of reactive operators to work with flows instead of local values. Research and develop a more streamlined and user-friendly API that abstracts away the complexities of dealing with flows, reducing boilerplate code and improving program transparency.

Timing Configuration Granularity Enhance the framework's flexibility in configuring the timing of computations beyond reacting solely to standard events. Explore the possibility of supporting additional strategies for scheduling and rate limiting, such as custom scheduling policies and per-construct configuration options. This could provide developers with finer control over the execution behavior

of their self-organizing systems, catering to diverse application requirements and environments.

Bibliography

[1] H. Van Dyke Parunak and Sven A. Brueckner. Software engineering for self-organizing systems. *The Knowledge Engineering Review*, 30(4):419–434, September 2015.

[2] Carlos Gershenson. *Design and control of self-organizing systems*. CopIt Arxives, 2007.

[3] V. Singh, G. Singh, and S. Pande. Emergence, self-organization and collective intelligence – modeling the dynamics of complex collectives in social and organizational settings. In *2013 UKSim 15th International Conference on Computer Modelling and Simulation*. IEEE, April 2013.

[4] Rocco De Nicola, Stefan Jähnichen, and Martin Wirsing. Rigorous engineering of collective adaptive systems: special section. *International Journal on Software Tools for Technology Transfer*, 22(4):389–397, May 2020.

[5] Mikhail Prokopenko. *Guided self-organization*. 2009.

[6] Hartmut Schmeck, Christian Müller-Schloer, Emre undefinedakar, Moez Mnif, and Urban Richter. Adaptivity and self-organization in organic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 5(3):1–32, September 2010.

[7] Georg Martius and J. Michael Herrmann. Variants of guided self-organization for robot control. *Theory in Biosciences*, 131(3):129–137, November 2011.

[8] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*, pages 321–384. Springer International Publishing, 2021.

[9] Tom De Wolf and Tom Holvoet. Designing self-organising emergent systems based on information flows and feedback-loops. In *First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*. IEEE, July 2007.

[10] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. *ACM Computing Surveys*, 55(13s):1–37, July 2023.

[11] Iwens G. S. Júnior, Thalia S. de Santana, Renato de F. Bulcão-Neto, and Barry F. Porter. The state of the art of macroprogramming in iot: An update. *Journal of Internet Services and Applications*, 13(1):54–65, November 2022.

[12] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, January 2013.

[13] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 43(3):1–51, April 2011.

[14] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani Srivastava. Ddflow: visualized declarative programming for heterogeneous iot networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19. ACM, April 2019.

[15] Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. *Future Internet*, 12(11):203, November 2020.

[16] Roberto Casadei, Francesco Dente, Gianluca Aguzzi, Danilo Pianini, and Mirko Viroli. Self-organisation programming: A functional reactive macro approach. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, September 2023.

[17] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):1–34, August 2013.

[18] Alois Ferscha. Collective adaptive systems. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers - UbiComp '15*, UbiComp '15. ACM Press, 2015.

[19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, December 2019.

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

[21] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. *A Calculus of Computational Fields*, pages 1140–128. Springer Berlin Heidelberg, 2013.

[22] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. *Space-Time Universality of Field Calculus*, pages 1–20. Springer International Publishing, 2018.

[23] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. Functional programming for distributed systems with xc. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[24] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC 2015. ACM, April 2015.

[25] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, March 2006.

[26] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, December 2022.

[27] Giorgio Audrito. Fcpp: an efficient and extensible field calculus framework. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, August 2020.

[28] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, August 2013.