ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

MSc in Engineering and Computer Science

Towards Aggregate Programming in pure Kotlin through compiler-level metaprogramming

Thesis in

LABORATORY OF SOFTWARE SYSTEMS

Supervised by:
Prof. DANILO PIANINI

Presented by:
ELISA TRONETTI

Fourth Graduation Session
Academic Year 2021-2022

There is life on the other side of fear.

Contents

List of Figures

Listings

Acknowledgements

I would like to express my sincere gratitude to my parents **Catia** and **Fabrizio**, and brother **Stefano** for their love and support. Your constant encouragement and belief in my abilities have been my greatest source of inspiration. You saw all the struggles and all the victories on my path, and you have stood by my side, never stopping to cheer for me.

I am also grateful to my loving boyfriend, **Alessandro**, for his unconditional support and presence during the highs and lows of my academic pursuits. You have been my rock during the most challenging times of my studies. You have never doubted my skills even when I was the first one not believing in myself. Having you by my side all along this journey was my biggest treasure.

Furthermore, I cannot express enough gratitude to my exceptional colleagues at the university. Together, we spent countless hours working on assignments and projects, and your genuine support and encouragement have made all the difference. A special thank goes to **Angelo**, **Luca**, **Schiaro**, **Andrea** and again, **Alessandro**.

To my dear friends **Francesca**, **Sofia**, and **Dariana**, thank you for making the journey enjoyable with your endless hours of companionship and laughter. Your friendship has been a bright light that lifted the weight of my studies.

My esteemed thesis supervisor, the professor **Danilo Pianini**, deserves a special mention for his invaluable guidance, unwavering support, and expertise throughout my research journey. Your mentorship and advices have been crucial to the successful completion of this project.

Finally, I extend gratitude to myself for never giving up, even when it felt impossible. This achievement proves my determination and resilience, and I am proud of myself for persevering through the toughest times. This is the end of an incredible journey, and hopefully the beginning of a new beautiful one, which I begin with the certainty that I can achieve incredible goals, such as this one.

Abstract

The last few decades have seen significant technological advancements in computing, the internet, and mobile technology, leading to the growth of the Internet of Things (IoT). This has resulted in a network of physical devices embedded with sensors, software, and connectivity, which can collect and share data. However, this growth has also brought new challenges, such as the need for complex software engineering to take advantage of the computational infrastructure available while considering unpredictability and communication heterogeneity.

This thesis explores the aggregate programming, which is a paradigm based on field calculus, and it allows for the easy manipulation of data across devices, making it possible to perform operation on the data of distributed systems, in a simple and efficient manner. The paradigm has been implemented in various programming languages and platforms, such as *Protelis*, *Scafi* and *FCPP*.

This thesis proposes a new implementation of the aggregate programming paradigm, called *Collektive*.
The aggregate programming paradigm requires the communication of the devices to be coordinated through the alignment, which keeps track of the computational state of each device. The work done in this thesis explores different Kotlin metaprogramming techniques in order to solve this problem, illustrating the final solution achieved through the implementation of a Kotlin compiler plugin, which is totally transparent and portable.

The project provides the user a minimal DSL, which is compatible with multiple platforms, such as JVM, JavaScript and Kotlin Native. This is possible because of the features offered by Kotlin Multiplatform, which is used for the implementation of the DSL. Moreover, this thesis addresses the validation process carried out to test the correct behavior of the system, which guarantees that Collektive can be considered at the same level of the existing aggregate programming implementations.

Chapter 1

Introduction

In the last few decades, there has been a significant technological development in various fields. Computing technology has advanced rapidly, with the introduction of faster and more powerful processors, and the widespread adoption of cloud computing.
The Internet has also become an integral part of modern life, connecting people and devices across the globe; in addition, the development of faster and more reliable connectivity networks has made it possible to transfer and share data at an even faster rate.
Mobile technology has grown exponentially, thanks to the adoption of smartphones and other mobile devices.
One of the fields that has seen a substantial growth, due to the increasing availability and affordability of devices, sensors and other components, is the Internet of Things (IoT). The term IoT refers to the growing network of physical devices, vehicles, buildings and other items that are embedded with sensors, software and connectivity, which enables these *Things* to collect and share data. These capabilities have the potential to bring significant benefits to society and economy, such as improving public services, increasing efficiency and productivity, and reducing costs.

This technological development is closely connected with the growth of distributed systems. As a consequence of the increase in computing power and the availability of fast and reliable networks, it has become possible to allow devices and systems to work together and share data, even when they are physically separated.

The general growth just discussed brought new challenges, such as the need of engineering complex software which has to take full advantage of the computational infrastructure available, taking in consideration the unpredictability of changes and the heterogeneity of communication required.

In order to face the new complexities it is necessary to rethink and renovate the process of software development.

Aggregate Programming is a paradigm which aims to address these requirements. It allows for the easy manipulation of data across devices, making it possible to perform operation on the data of distributed systems, in a simple and efficient manner.

This paradigm has been implemented in different programming languages and platforms: for example **Protelis** [1], **Scafi** [2] and **FCPP** [3]. All of them present strengths, but also weaknesses: in order to address those, a new framework can be a solution.

In the following sections of this chapter, these themes are described in details, in order to define the goal of this thesis. Specifically Section 1.1 describes the aggregate programming characteristics and its main complexities, in Section 1.2 are introduced the implementations of the aggregate paradigm that constitute the current state of the art, and Section 1.3 presents the motivations behind this work and the goals that aims to achieve.

1.1 Context

In this section it is going to be discussed in details the paradigm of Aggregate Programming, in order to provide the context of its development and an overview about how the paradigm works.

1.1.1 Aggregate Programming

Since the last few decades have witnessed technological advances, the traditional way of engineering software is, in some cases, suboptimal.
The systems developed are becoming more and more complex, with requirements that are sometimes difficult to achieve.

These challenges are enhanced when coming into terms with distributed systems, because there is the need to handle the always increasing number of connected devices, that share and compute data.
Some relevant complexities are the following [4]:

- **Scalability**: as the number of devices in a distributed system grows, the coordination becomes increasingly complex. This requires the development of scalable algorithms and protocols that can handle large numbers of devices;

- **Heterogeneity and interoperability**: in a distributed system, devices may be made by different manufacturers and use different protocols, which can make it difficult for them to communicate with each other. The interoperability between heterogeneous devices and systems is a major challenge;
- **Synchronization**: devices are usually not connected to a common clock, which can make it difficult to synchronize their activities. This can lead to inconsistencies and errors within the system;
- **Latency**: the distance between devices in a distributed system can lead to high latency, which can affect the performance of the system. Managing and reducing latency is a challenge while coordinating numerous devices;
- **Consistency**: devices may have different views of the system state, which can lead to inconsistencies. Guarantee consistency across devices can be challenging and requires the development of distributed algorithms and protocols;
- **Fault tolerance and error control**: devices may fail or become disconnected, which can disrupt the system. Ensuring fault tolerance and the ability to detect and recover from failures is essential in this kind of systems;
- **Security and privacy**: coordinating devices in a distributed system requires the sharing of sensitive information, which can create security risks.

Traditionally, the development of this kind of architectures is focused on the single device, instead of the system as a whole. This means that the developers need to concern about all the aspects just discussed, such as the coordination between devices, creating specific algorithms, but also about security, consistency, fault tolerance, which creates unnecessary complexities during the engineering of a system.
This approach is not feasible when dealing with large-scale, decentralized and adaptive infrastructures, such as the one represented in Figure 1.1, which shows an example of a real life environment, with numerous heterogeneous devices that need to communicate with each others.

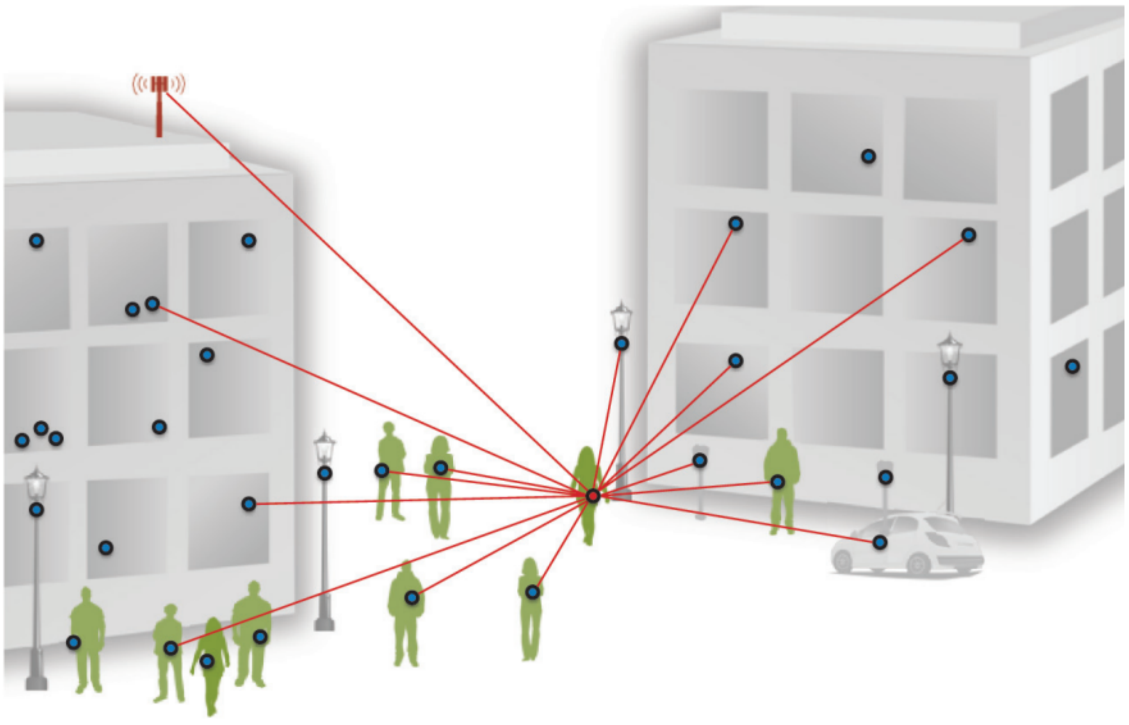


Figure 1.1: Example of a real life environment, where heterogeneous devices have the possibility to communicate with each other wirelessly creating complex distributed systems [5]

Aggregate programming is a paradigm that focuses on the manipulation of complex data structures as a whole, rather than their individual components [5]. The focus is on what a system is required to do, rather than on how is going to achieve it.
This paradigm is closely related to the **field calculus** [6], a mathematical framework for modeling and reasoning about distributed systems.
Aggregate programming in combination with field calculus, provides a powerful tool for modeling, reasoning, and programming distributed systems. It allows for the efficient manipulation of large and complex data in distributed systems, and facilitates the representation of relationships and interactions between devices. It is a useful approach to adopt for IoT and other distributed systems where scalability, coordination, and security are critical concerns.

The basic concept used in aggregate programming is the **computational field**, which is a mathematical function that assigns a value to each point in a space, and it is used to represent and manipulate the state of a distributed system.

In order to explain better what a computational field is and how it is used, it is necessary to discuss the assumed computational model [7].
Supposing a program P executed by a network of devices defined by a dynamic neighboring relationship. The computation of P can be analyzed by a local and a global point of view. Considering the **local** perspective, P is computing on a round-based scheme in single devices. A round is executed in the following steps:

- the device sleeps for a delta of time and at some point it wakes up;
- it retrieves the information received from neighbors while it was sleeping. The messages sent by the neighbors are fields, and they map the neighbor device identifier with the values computed;
- it gathers information about its context;
- it retrieves the information stored locally in the previous round about the context;
- it executes P, which might manipulate the data of the current context, the data retrieved from the previous round and the neighbors values;
- at the end of the execution of P, the device stores the current context's value in the local memory and sent a message to all the neighbors about the current computation;
- at the end of the round, the device returns to sleep, until another round begins.

On the other hand, the **global** point of view is the key that makes aggregate computing a powerful tool. In this case, the computation considers the entire network of interconnected devices as a single, unified entity.
The data is represented as a distributed space-time field evolution, which maps the computation events to values computed by devices.
A computational field is a snapshot of the state of the distributed system's network at a certain moment in time, which maps device identifiers to their values.

The main constructs provided by the field calculus are the following:

- **Rep-expression** $rep(e_0)\{(x) \rightarrow e\}$

  This is the repeat construct, and it represents the time evolution. This allows the field to change dynamically: if in the previous round the rep expression has not been evaluated, the initialization value is e_0, otherwise it is the value obtained in the last computation of rep;

- **Nbr-expression** $nbr\{e\}$
  The neighboring is used to model the device-to-neighbor interaction. In order to do this, it is necessary to construct neighboring fields: in this data structure each event is associated to a value, that is mapped to a neighbor identifier and its last value computed. Using this construct it is possible for a device to understand its surrounding by manipulating the neighboring field obtained;

- **Branching** $if(e_0)\{e_1\}else\{e_2\}$
  Branching causes a domain restriction, which means that devices compute e_1 only in the restricted domain where e_0 is true, otherwise they compute e_2. This behavior and its consequences will be discussed in details in Section 1.1.1.1;

- **Function calls** $e(e_1, ..., e_n)$
  This construct model a function call where e evaluates to a field of function values.

As previously said, the data structures manipulated are distributed over space, and they evolve over time. These two aspects are handled, in the case of field calculus, by two separate constructs: the evolution in time is manipulated by *rep-expression*, the space management, which provide neighbors interactions, is provided by the *nbr-expression*. The use of time evolution and neighbor interaction operators in the traditional way leads to a slowdown in the efficient spread of information. This is due to the separation of state sharing (nbr) and state updates (rep), which requires information received through nbr to be stored through rep in the local memory of the device before it can be passed on during the next execution of nbr [8].

It is possible to overcome this limitation by extending the field calculus with a new construct, which has been called **share** [8]. The share operator allows to observe the neighbors' field, updated the local values and sharing immediately the updated state in a single computation. It is possible to observe the differences between the behavior of the share operator and the combination of rep and nbr in Figure 1.2.

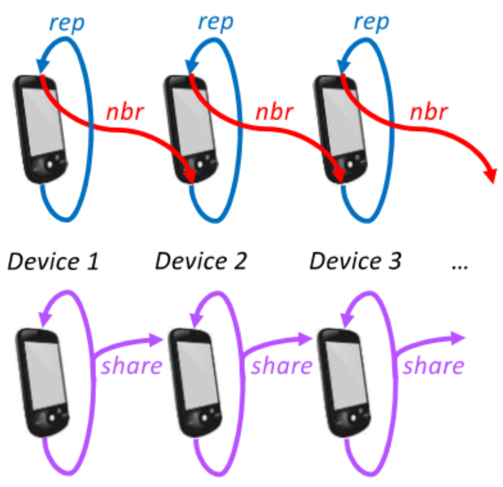


Figure 1.2: Comparison between the combination of rep-expression and nbr-expression, and the share construct [8]

1.1.1.1 Domain restriction: alignment

The expected consequence of a branch construct is to determine the portion of code that is going to be computed. In the field calculus, the branch construct also has an unusual behavior, which is called domain restriction.

```
if (e0) {e1} else {e2}
```

Taking in consideration this portion of pseudocode, it is possible to identify two different restricted subdomains:

1. D_{true}: this is how the domain is called when $e0$ is true;
2. D_{false}: the name of the domain in the case that the condition $e0$ is false.

For example, if a device is in the domain D_{true}, the following are the implications [7]:

- it does not compute $e2$, which is the typical branch behavior;
- if $e2$ involves a nbr-expression, this is not going to be evaluated by the considered device. This means that the neighboring devices in D_{false} can not obtain the value of this device for the nbr-expression in $e2$, because it has not been computed. Similarly, the nbr-expression in $e1$ evaluated by this device is not going to be shared from D_{true} to D_{false};
- if the device evaluated $e2$ instead of $e1$ in its previous round, then all the rep-expressions in $e2$ are going to be computed using the initialization value.

Given this behavior, the result is that devices operating in different subdomains are computing in isolation. As a direct consequence of that they can not communicate with each other unless they are in the same domain. This characteristic is called **clustering**, which is an important feature: it allows restricting computations in subdomains in an easy and efficient way, which is extremely useful in many use-cases.

The domain restriction is a crucial trait of aggregate programming, and it introduces to a problem that, necessarily, has to be taken in consideration: different devices that operate in a highly distributed system do not possess a shared memory that can be used to keep track of the computation being executed on each device. On the other hand, this domain restriction characteristic requires that each device must communicate only with devices in the same subdomain, which means that they are computing the same operation.

This is defined as the **alignment problem**, and it is necessary to explore all the information of the system in order to find a solution.

First, it is important to keep in mind that each device has its own identifier, which can be used to define the author of a message.
Second, when dealing with a computational field, such as when retrieving neighboring values using a nbr-expression, the device is currently computing the operation that it is actually looking for in the field.

In order to explain this concept better, here is a snippet of pseudocode:

```
if (e0) {nbr(e1)} else {nbr(e2)}
```

When a device δ_1 is in the domain D_{true}, it computes $nbr(e1)$: this means that it is retrieving from the neighbors that computed $e1$, while evaluating $e1$ itself.
Consequently, δ_1 and the other devices have to be aligned in order to exchange information: they are all in the same subdomain, in which $e0$ is evaluated as true, and they are computing $nbr(e1)$.
Given these concepts, it is possible to understand how to deal with the alignment problem: thanks to the current computational location of a device, it is possible to understand which values it is currently trying to retrieve, and, since each device has an identifier, the values can be mapped to those identifiers. Then, combining all the information together, a computational field is obtained.

One thing to notice is that the alignment is not a concern of the branch construct only. In general, the alignment feature is necessary all along an aggregate program, but it can be more clearly noticed in the branch occurrences, which is the reason it has been discussed as the first example.

Considering this pseudocode:

```
fun f1() {nbr(e1)}
fun f2() {nbr(e1)}

f1()
f2()
```

Even though the body of the functions $f1$ and $f2$ is the same, they are two different expressions. This means that the neighboring expression in $f1$ is not the same of the one in $f2$, and when one of the neighboring expression is retrieving the values computed by the neighbors, it has to consider only its current computational location. For example if the neighboring expression in $f1$ is executing, it has to get the neighbors' evaluation of the $nbr(e1)$ inside the body of $f1$. The resolution of this problem is the same explained for the branch expression.

Concluding, the alignment is provided by keeping track of the computational location, which must be done every time an aggregate programming expression is involved. In this way it is possible to solve any ambiguity that might appear during the computational field construction.

1.2 State of the art

The aim of this section is to give an introduction about the state-of-the-art technologies that have been developed in the field of aggregate programming. This overview is going to take in consideration how each implementation works and the choices that led their development, with the goal of depicting a representation of the context in which the project of this thesis is going to be set in.

1.2.1 Protelis

Protelis [1] is a functional language inspired by Proto. It implements the field calculus providing a modern specification language.

The architecture is based on Java, and the reasons are multiple:

- Java is portable across different systems and devices;
- It is easy to import libraries and APIs;
- The variety of low-cost embedded devices that are able to support Java is increasing;
- Java optimizations in terms of speed and resources consumption are making it a valid alternative to low-level languages, such as C.

Since Protelis is a functional language, but its syntax is Java-like, its adoption is not difficult. For example, the following is a snippet of code written in Protelis, which is used to calculate the distance between each device and the source, using the gradient algorithm:

```
def myPosition() = self.getDevicePosition()
def nbrRange() = nbr(myPosition()).distanceTo(myPosition())
share (d <- POSITIVE_INFINITY) {
    let shortestPathViaNeighborhood =
        foldMin(POSITIVE_INFINITY, d + nbrRange())
    if (env.has("source")) {
        0
    } else {
        shortestPathViaNeighborhood
    }
}
```

Listing 1.1: Protelis example

Protelis is very versatile, and it can be used in a wide range of application domains. It is possible to create Protelis modules and execute them in simulations using Alchemist [9], which is a simulator for pervasive computing and distributed systems.

1.2.2 ScaFi

ScaFi [2] is a programming language and framework for Aggregate Programming. Its name stands for ***Sca**la **Fi**elds*, since it is built on top of the Scala programming language, which provides the functional programming paradigm and type-safe features. It also uses the Akka framework for actor-based concurrency and the Java Virtual Machine (JVM) for execution.

ScaFi implements a variant of field calculus, providing a domain-specific language (DSL) and API in order to allow writing, testing and running aggregate programs.

One of the key features of ScaFi is its support for declarative programming, which allows developers to specify the desired outcome of a computation, rather than how the computation should be performed. This makes it easier for developers to write correct and efficient distributed algorithms, as they can abstract away the low-level details of communication and synchronization.

ScaFi aims to provide a high-level and type-safe programming interface for building complex systems.
The combination of the Scala programming language, the Akka framework, and the JVM provides a strong foundation for building scalable and robust distributed systems.
In particular, the choice of a JVM-based architecture has the same advantages discussed previously for Protelis in Section 1.2.1, which are mainly portability, low-cost of devices, diffusion and general JVM performances optimizations.

1.2.3 FCPP

FCPP [3] is a library in C++14, and it implements the field calculus.
It provides tools for simulation of distributed systems, and its extensible component-based architecture makes it suitable for a wide range of application scenarios.

Differently from Protelis and ScaFi, FCPP does not rely on the Java Virtual Machine (JVM). It is implemented in C++, which choice gives the following advantages:

- numerous devices support C++ architectures, making it possible deploy FCPP on systems of any sort, which includes microcontrollers;
- C++ is performance-oriented, which means that the system requirements are very low, increasing the variety of devices that can use FCCP.

These two features allow the coverage of multiple use-cases that were not possible to handle using Protelis and ScaFi: for example the deployment on microcontroller-based systems or cloud applications that have strict parallelism requirements in order to scale

correctly, which is guaranteed by the performance improvements.
Since it is based on a lower level language that Protelis and Scafi, programs written in FCCP might be difficult to understand when dealing with complex systems.

1.3 Motivation and goal

When talking about aggregate computing, there are several important features that must be considered when comparing different aggregate programming frameworks, such as ScaFi, Protelis, and FCPP.
After the analysis of the previous sections, it is possible to achieve the following conclusion:

- **Protelis**
  It has a simple and expressive programming model, but its simplicity and expressiveness come at the cost of performances, as the framework does not provide much control over the underlying data structures and algorithms. Moreover, Protelis might not be the best choice for systems that require strong interoperability with other systems and technologies, since it has a limited set of integrations;

- **ScaFi**
  It has a strong focus on programmability and support for Scala, which can make the system more complex and difficult to understand for developers who are not familiar with the language or the functional programming paradigm. Additionally, ScaFi may have a higher overhead compared to other frameworks due to its use of a more expressive programming model;

- **FCPP**
  The key aspects taken in consideration from FCPP are scalability and performance, providing efficient algorithms and data structures for large-scale data processing. These features can make the framework more complex and difficult to understand, since it utilize a low-level programming model that requires a deep understanding of algorithms and data structures.

These considerations highlight crucial weaknesses of the already existent frameworks that the project of this thesis tries to overcome. Specifically, the project is an aggregate programming Domain-Specific Language (DSL) in Kotlin and the main features to achieve are the following:

- **Transparency**: refers to the ability of the DSL to provide clear and concise information about the underlying system's behavior, such as how data is processed, stored, and communicated between nodes in a distributed system. Transparency helps to reduce the complexity of the system and makes it easier to understand and maintain, especially in the case of large and complex systems;

- **Minimality**: refers to the design of the DSL to have as few constructs and abstractions as possible while still providing all the necessary functionality. This makes the system easier to understand, maintain, and debug. Minimality also helps to lower the overhead associated with the use of the DSL, which is especially important for systems that require high performance and scalability;

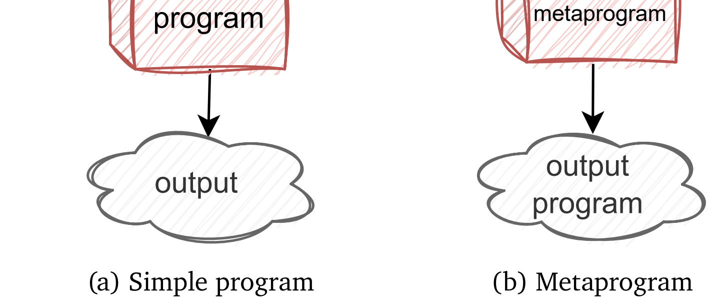
- **Portability**: refers to the ability of the DSL to run on different platforms and environments, such as different operating systems, cloud platforms, and hardware architectures. Portability helps to ensure that the systems built using the DSL can be easily deployed and run in different environments, which is especially important for systems that need to be deployed in multiple locations or need to scale to meet changing demands.

The goal of this thesis is the study and the implementation of the aggregate programming paradigm, facing the alignment problem discussed in Section 1.1.1.1, focussing on transparency, minimality and portability.
The reminder of this thesis is organized as follows. Chapter 2 debates all the possibilities of metaprogramming in Kotlin, and Chapter 3 explains in details the solution of the alignment problem. Chapter 4 describes how the DSL is implemented, shows the end usage result, and presents the validation process. Finally, Chapter 5 summarizes the work done in this thesis, adding plans for future development.

Chapter 2

Metaprogramming in Kotlin

A typical program computation can be summarized as follows: it reads data as input, computes that data and generates an output.
Metaprograms[1] are able to take as input another program, sometimes even the metaprogram itself, manipulate it and return a new version of it expressing a modified behavior. In Figure 2.1 it is shown this difference with a representation of a simple program which transforms the input data, and a metaprogram which, on the other hand, modifies an entire input program generating an output program.



(a) Simple program (b) Metaprogram

Figure 2.1: Representation of an execution of a simple program compared to a metaprogram

[1]https://devopedia.org/metaprogramming

Metaprogramming in Kotlin[2] is a powerful feature that allows for the manipulation and generation of code at compile time. The possible benefits of this technique are:

- **Code generation**: it generates repetitive or boilerplate code automatically, reducing the amount of code that needs to be written and maintained by hand. Consequently, this can increase code readability and maintainability, as well as reducing the risk of introducing bugs or errors, that can be easily done by humans during the executions of repetitive tasks;
- **Readability**: it provides a higher level of abstraction, allowing to abstract away complex logic and make code more readable and easier to understand. This can also help to simplify the implementation of elaborated algorithms and data structures;
- **Reusability**: by generating code automatically, it is possible reuse the same logic across multiple parts of applications, increasing overall code reuse and maintainability;
- **Supports Domain-Specific Languages** (DSLs): it enables to create custom domain-specific languages (DSLs) that are optimized for specific tasks or use cases, which can help to simplify complex operations and make the code more readable and intuitive;
- **Custom annotations**: it allows creating custom annotations, which can be used to provide additional information about the code and to automate tasks such as code generation. This can improve code quality and make it easier to understand intentions and goals behind the code.

The main cost that has to be paid when dealing with metaprogramming is the increased **language complexity**.

2.1 Metaprogramming techniques

A multitude of different techniques can be used in Kotlin in order to create metaprograms. In the following sections the main approaches provided by Kotlin are going to be analyzed

[2]https://www.droidcon.com/2022/04/28/meta-programming-with-kotlin-for-android/

in details, in order to give an overview of the possible alternatives when coming into terms with metaprogramming.

2.1.1 Annotations

Annotations[3] in Kotlin are a type of metadata that can be added to provide additional information about the code, as well as to automate certain tasks.

To create a custom annotation in Kotlin, first it is necessary to declare the annotation type, using the **annotation** keyword, as shown in the snippet of code in Listing 2.1.

```kotlin
annotation class MyAnnotation
```

Listing 2.1: Example of creation of a custom annotation in Kotlin

Kotlin provides the option to use additional attributes by annotating the annotation class with **meta-annotation**. The available meta-annotation alternatives are:

- **@Target**: it is used to specify the elements that can be annotated, which can be classes, functions, properties and expressions;
- **@Retention**: it defines if an annotation is stored in the compiled class files and if it is visible at runtime using reflections;
- **@Repeatable**: it allows an annotation to be used on the same element multiple times;
- **@MustBeDocumented**: it can be used when the annotation is part of a public API, and it must be included in the generated API documentation.

For example, this is a possible custom annotation that uses meta-annotation:

```kotlin
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
annotation class MyAnnotation
```

Listing 2.2: Example of custom annotation with meta-annotations in Kotlin

In this snippet of code in Listing 2.2, the **@Target** annotation specifies the elements that the annotation can be applied to, in this case either classes or functions. The **@Retention**

[3] https://kotlinlang.org/docs/annotations.html

annotation refers to the retention policy, which determines that the annotation is available at runtime.

Annotations can be used to provide metadata, such as the purpose of a function, the intended usage of a class, or the preferred behavior of a method. One possible use case is to use annotations to specify that a certain method should only be called on a background thread, or that a certain class is serializable.

Another feature provided by annotations, which is considered metaprogramming, is the possibility to use them to automate tasks, such as code generation. For example, annotations can be used to automatically generate code for serializing and deserializing objects.

It is possible to use the annotation in the code by placing the @ (at) symbol, followed by the annotation type before the element that is going to be annotated, such as a class, function, or property.

The snippet of code reported in Listing 2.3 shows how to actually use a custom annotation, specifically the one created in Listing 2.2.

```kotlin
@MyAnnotation
class MyClass

@MyAnnotation
fun myFunction() {
    val myVariable = 42
}
```

Listing 2.3: Example of usage of a custom annotation in Kotlin

In this example, the **MyAnnotation** annotation is applied to both a class and a function. The annotations can be used by other code, either at compile time or at runtime, to provide additional information about the annotated elements.

2.1.2 KSP

KSP stands for **Kotlin Symbol Processing**, and it is an API that can be used to build lightweight compiler plugins[4]. It offers an easy compiler plugin API that takes advantage

[4] https://kotlinlang.org/docs/ksp-overview.html

of Kotlin's capabilities, but it keeps the learning curve low.

KSP has been developed as an alternative technology to **kapt**[5], which was used to allow annotation processing. Currently, *kapt* is in maintenance mode, meaning that it is being kept up-to-date with the new Kotlin and Java releases, but without adding any features.

In comparison to *kapt*, annotation processors that utilize KSP can be executed two times faster.

Kotlin Symbol Processing is a feature of the Kotlin compiler that allows to manipulate the code at compile time by processing the symbols contained in the code. As a consequence of that, it is possible to write metaprogramming code that analyzes, transforms, and generates other code, without the need to access the code at runtime.

KSP reads the source code, generates new modules, and during the compilation the Kotlin compiler uses the original code and the generated source without distinctions, as it is shown in Figure 2.2, creating one executing program.

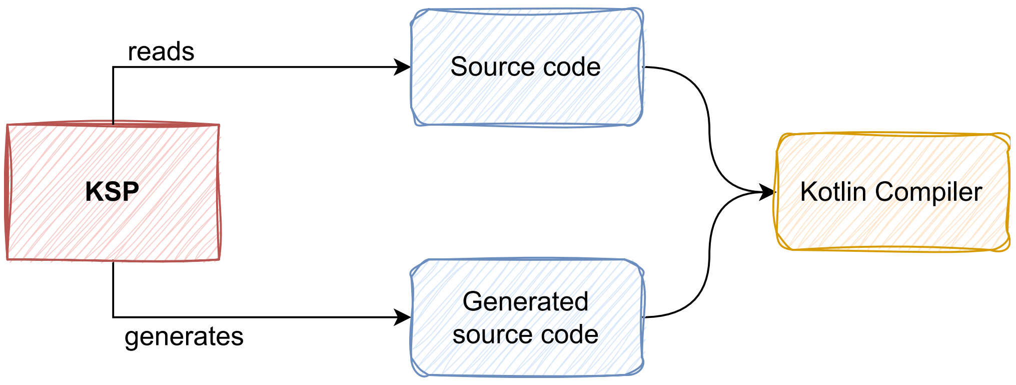


Figure 2.2: Representation of KSP behavior

This API allows processing Kotlin programs in a native way, with an understanding of Kotlin-specifc features, which includes extension functions, local functions and declaration-site variance. The API also models types explicitly, enabling type checking.

KSP models the structure of Kotlin programs, making class declarations, class members, functions and parameters accessible for processing, while elements such as if blocks and for loops are not.

It can also be seen as a preprocessor framework, making a KSP plugin a symbol processor that follow three main steps:

1. It analyzes the source program and its resources;

[5] https://kotlinlang.org/docs/kapt.html

2. It generates an output, which might be code or in another form;
3. Finally, the Kotlin compiler compiles the source program with the generated output.

Differently from a typical compiler plugin, KSP restricts processors from modifying the source code, as it is considered read-only. This choice can be justified by the intention to prevent confusion that may arise from a plugin that modifies the language semantics.

The source code, from the KSP perspective looks like this when being analyzed:

```
KSFile
  packageName: KSName
  fileName: String
    declarations: List<KSDeclaration>
      KSClassDeclaration //class, interface, object
        simpleName: KSName
        qualifiedName: KSName
        containingFile: String
        typeParameters: KSTypeParameter
        parentDeclaration: KSDeclaration
        classKind: ClassKind
        primaryConstructor: KSFunctionDeclaration
        superTypes: List<KSTypeReference>
        //contains inner classes, functions, properties
        declarations: List<KSDeclaration>
```

Listing 2.4: The source code from KSP perspective

The Listing 2.4[4] shows the data structure that is accessible using KSP. Specifically, in this code it is contained a class declaration (*KSClassDeclaration*), and it is possible to get its name, type, parameters, inner declarations, and so on.
This representation allows to have a complete overview of the code structure.

KSP expects two elements to be implemented: the **SymbolProcessorProvider** and the **SymbolProcessor**.
The *create* function is invoked when KSP creates an instance of the *SymbolProcessor*: this means that, without the *SymbolProcessorProvider*, it would not be possible to instantiate a *SymbolProcessor*. The *SymbolProcessorProvider* is separated from the *SymbolProcessor* to

allow more freedom during the development. The *SymbolProcessorProvider* interface is defined like shown in Listing 2.5[4].

```
interface SymbolProcessorProvider {
    fun create(environment: SymbolProcessorEnvironment):
        SymbolProcessor
}
```

Listing 2.5: SymbolProcessorProvider interface

In Listing 2.6[4] is shown the interface of the *SymbolProcessor*.
The only function that needs to be overridden is *process*, which is typically used to read files and pass the elements to the visitors.

```
interface SymbolProcessor {
    fun process(resolver: Resolver): List<KSAnnotated>
}
```

Listing 2.6: SymbolProcessor interface

The *Resolver* is the entry point of the functionalities provided by KSP: it can be used, for example, to get all the files of the source code, the symbols with annotation, a class, or a function declaration by name.
As many other compiler related APIs, KSP supports the **visitor pattern**, allowing to examine each element in an object-oriented way.
For example, the code in Listing 2.7[4] reports an implementation of a visitor that inspects the declarations and collects the function names in a list.

```
class FindFunctionsVisitor: KSTopDownVisitor<Unit, Unit>() {
    override fun visitFunctionDeclaration(
        function: KSFunctionDeclaration,
        data: Unit
    ) {
        functions.add(function.toString())
    }
}
```

Listing 2.7: Visitor that collects function declarations

In conclusion, one of the biggest strength of KSP is that it provides an API built on top of the compiler plugin: for this reason, it is able to hide the compiler changes, minimize maintenance efforts, and its API makes possible to implement a light-weight compiler plugin without the complexities and the knowledge required to create a real compiler plugin.
On the other hand, since the goal of KSP is to be a simple solution for common problems, it comes with some limitations [6]:

- it can not examine expression-level information;
- it can not modify the source code, but it can only generate new code;
- it is not integrated within the IDE, meaning that the IDE does not have any information about the code generated.

2.1.3 Kotlin compiler plugins

Kotlin compiler plugins[7] are a way to extend the functionalities of the Kotlin compiler by adding custom processing to the compilation. This means that the compiler plugin code runs at compile-time, and, since this is a feature of *kotlinc*, it only works on Kotlin source code.

Compiler plugins allow automating tasks and enforce coding standards. For example, they can be used to generate boilerplate code, such as accessor methods, or to enforce specific naming conventions or coding styles. Compiler plugins can also be used to perform code analysis and modification, such as adding custom checks and warnings, or refactoring the code automatically.

Kotlin compiler provides a powerful API, which makes it possible to even modify the internals of functions and classes. This enables to solve metaprogramming problems that were impossible using annotation processors, for example with KSP. Another advantage is that compiler plugins work on every Kotlin target, without having to write multiple plugins.

However, it is important to be aware of the potential downsides of using Kotlin compiler plugins. One of the main cons is that in order to write even a really simple plugin, it

[6]https://kotlinlang.org/docs/ksp-why-ksp.html
[7]https://resources.jetbrains.com/storage/products/kotlinconf2018/slides/5_Writing%20Your%20First%20Kotlin%20Compiler%20Plugin.pdf

is necessary to have compiler background knowledge. Because of the high learning curve, it is important to consider very carefully if it is necessary to develop a plugin. The investment of time and work can be significant and, beside understanding how a compiler plugin works, it is necessary to develop:

- **An IntelliJ plugin**: whenever they are used synthetics members, such as functions that are going to be created by the plugin at compile time, the IntelliJ plugin is used to avoid errors highlights, making the IDE understand what is going on;
- **A Gradle or Maven plugin**: in order to make the user able to use and configure the compiler plugin;
- **The actual compiler plugin**.

Moreover, compiler plugins can slow down the compilation process and make it more complex. Additionally, this can cause compatibility issues with different versions of the Kotlin compiler, and may require significant effort to maintain and update. It is also worth mentioning that because compiler plugins modify the behavior of the compiler, they can potentially introduce bugs and cause compatibility issues with other plugins.

Finally, another aspect that must be taken in consideration is the fact that the Kotlin compiler API is not documented. This means that working with it and understanding the functions' behaviors is not an easy task, and it requires even more time.

A Kotlin compiler plugin architecture is organized as shown in Figure 2.3[7].

Two different modules can be distinguished in the architecture: a Gradle module and a Kotlin module.

The **Gradle module** is a wrapper for the Kotlin module, and it is an entry point for Gradle. The Gradle module is composed by:

- **plugin**: this part is unrelated to Kotlin, and it is based on the Gradle API. It provides an entry point from a *build.gradle* script, and it makes possible to configure the plugin via Gradle extensions;
- **subplugin**: this is the bridge that allows the interaction between Gradle and the Kotlin APIs. It reads the Gradle extension options and defines the compiler plugin's identifier, which is a unique key used in order to avoid collision with other plugins. The subplugin defines also the location of the compiler plugin artifact, which is

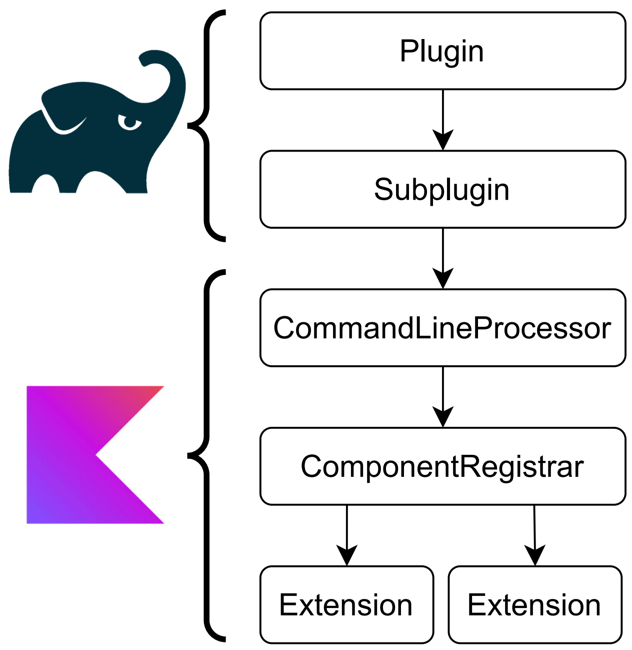


Figure 2.3: The Kotlin compiler plugin architecture

going to be used to download it at compile time. The location can be local or a Maven coordinate.

The **Kotlin module** is dived in three main parts:

- **CommandLineProcessor**: the options created in the Gradle subplugin are loaded and used by the CommandLineProcessor. Whenever a Kotlin program is launched, *kotlinc* is invoked and the arguments that are set in the subplugin are passed to the Kotlin compiler;
- **ComponentRegistrar**: it registers the extension components, which are going to be used when the project is compiling;
- **Extensions**: they are used to actually generate code. There are a lot of different extensions, which are used based on the use case.

Since the components of a plugin architecture have been explained from a general point of view, it is now necessary to explain each element more in depth.
Specifically, the following sections are organized as follows: Section 2.1.3.1 introduces how to create a Gradle plugin necessary to use a compiler plugin, Section 2.1.3.2 goes into details of the **IR** (**I**ntermediate **R**epresentation) that is the foundation that makes a compiler plugin possible, in Section 2.1.3.3 it is explained how to inspect and navigate

Kotlin IR, in Section 2.1.3.4 are covered advanced features, such as how to build new elements and how to transform the Kotlin IR, and Section 2.1.3.5 brings all together, showing the highlights of a working example of a simple Kotlin compiler plugin.

2.1.3.1 Gradle Plugin

A Gradle plugin[8] is typically used to handle a set of tasks that extend the project's capabilities.
In the context of Kotlin compiler plugins, the Gradle plugin is responsible for:

- Giving the **artifact coordinates** of the compiler plugin: this is used to download the artifact from Maven Central or another location, which might be a local submodule;
- Defining the **identification string** of the compiler plugin: used to avoid conflicts with other compiler plugins options, since the identification string of the plugin is used to separate the command line options;
- Translating the Gradle configuration to **command line options**.

In order to create the artifact coordinates and the ID it is possible to use the *buildconfig* plugin[9].

```
buildConfig {
    val project = project(":compiler-plugin")
    packageName(project.group.toString())
        buildConfigField("String", "KOTLIN_PLUGIN_ID", "\"${
  project.group}.${project.name}\"")
    ...
}
```

Listing 2.8: Example of a *buildconfig* that creates the compiler plugin artifact

The code in Listing 2.8 shows how to create references to a subproject called *compiler-plugin* and how to create the identification string, which is the concatenation of the project group and name. Similarly, it is possible to reference the name, the version and the project group of the compiler plugin, creating a unique reference to its artifact.

[8] https://docs.gradle.org/current/userguide/custom_plugins.html
[9] https://github.com/gmazzo/gradle-buildconfig-plugin

After creating the configuration just presented, it is possible to create the actual Gradle plugin, such as in Listing 2.9. Gradle exposes an interface that is specifically created for Kotlin compiler plugins, which is called **KotlinCompilerPluginSupportPlugin**.

```kotlin
class GradlePlugin : KotlinCompilerPluginSupportPlugin {
    ...
}
```

Listing 2.9: Gradle plugin class example

From this class they can be defined the compiler artifact, the identification string and any needed extension. For example, a Gradle Extension can be used to accept a command line parameter that enable and disable the plugin.

Basically, the creation of a Gradle plugin when dealing with Kotlin compiler plugins is always the same, and it is based on the notions just discussed in this section.

2.1.3.2 Kotlin IR

The Kotlin compiler is organized in two parts: the *frontend* is used to analyze the code, and the *backend* is the one that generates the executables.
Kotlin used to have three different backends: Kotlin/JVM, Kotlin/JS and Kotlin/Native. These backends were used to generate JVM byte code, JavaScript and LLVM IR, which is the representation for Kotlin Native.

When the Kotlin/Native backend was first developed, it was based on a new infrastructure, which used an internal representation for Kotlin code[10]. After its stabilization, Kotlin developers started to migrate the other two backends to the same representation. This allows to share the backend logic, and to have most of the feature and optimization done only once for all the targets. Moreover, the common backend infrastructure grants the possibility to create **multiplatform** compiler plugins.

Currently, the structure of the compiler can be represented as in Figure 2.4[11]: the frontend takes as input source file written in Kotlin, and then, each backend creates a syntax tree with all the information, using the IR representation. Only after this step, the code is transformed in the specific bytecode of each backend, which is different for each

[10] https://blog.jetbrains.com/kotlin/2021/02/the-jvm-backend-is-in-beta-let-s-make-it-stable-together/
[11] https://twitter.com/kotlin/status/1453741469148270593/photo/1

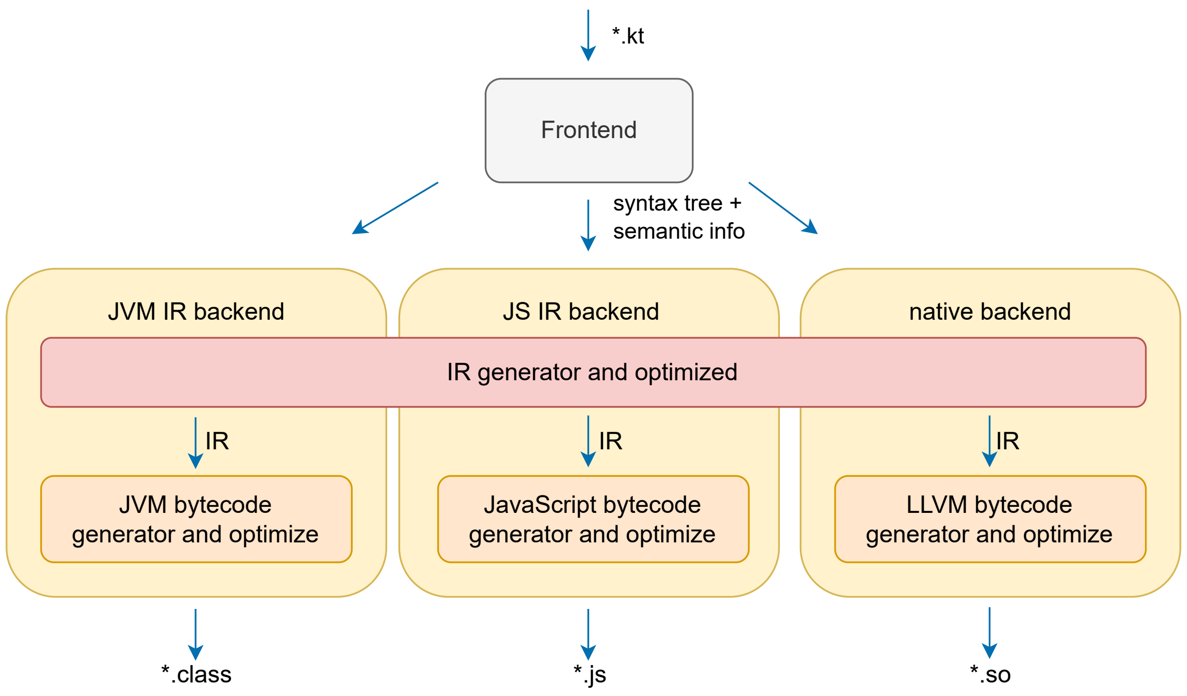

*.kt
Frontend
syntax tree +
semantic info
JVM IR backend
JS IR backend
native backend
IR generator and optimized
IR
IR
IR
JVM bytecode generator and optimize
JavaScript bytecode generator and optimize
LLVM bytecode generator and optimize
*.class
*.js
*.so

Figure 2.4: The new Kotlin compiler backend that uses IR

target.

The IR representation is an **abstract syntax tree** (**AST**). An AST[12] is a data structure that represents the abstract syntactic structure of a program. It is a tree representation of the source code, where each node in the tree corresponds to a construct in the source code. The nodes in the tree contain information about the construct, such as its type, location, and properties.

For example, considering the simple snippet of pseudocode in Listing 2.10.

```
if (a < b) {
    x = a + b
} else {
    x = a - b
}
```

Listing 2.10: Pseudocode of a simple assignment and expression

The AST that is generated from the code structure of Listing 2.10 is shown in Figure 2.5. Each node in the tree describes a construct in the source code. The root node represents the assignment of the result of the *branch* expression to the variable *x*. The *branch* defines

[12] https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38

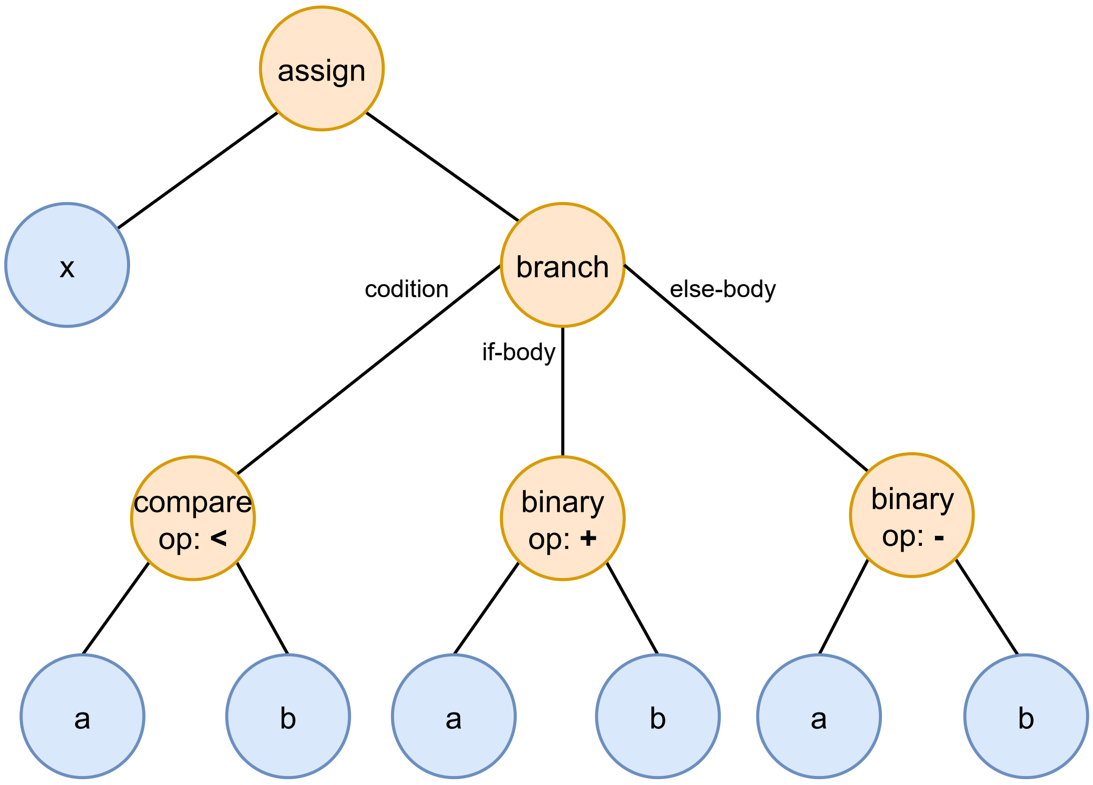


Figure 2.5: The Abstract Syntax Tree of the source code in Listing 2.10

the conditional construct, and it has three children: the *condition*, the *if-body* and the *else-body*. Each of these blocks is represented by a separate subtree, with <, + and - nodes representing the comparison, addition and subtraction operations, respectively. The nodes *a* and *b* are the variables, that are used as operands of these operations.
The condition node is crucial, because it is used to decide whether to evaluate the *if-body* or the *else-body*.

The AST is generated by a parser, which takes the source code as input and constructs the tree representation based on the syntax rules of the language. Once the AST is generated, it can be used for various purposes, such as type checking, code analysis, optimization, and code generation.
An AST provides a more abstract and structured representation of the source code compared to the raw source code. This makes it easier to analyze and manipulate the code, as well as to automatically generate code or perform other tasks. For example, compilers use the AST to perform optimizations, such as constant folding, dead code elimination, and inlining, before generating machine code.

Hence, **Kotlin IR** (Intermediate Representation) backend is the part of the Kotlin compiler that generates a representation of the Kotlin code. This is used as the input for the next stage of the compilation, which can be either the code generator or another intermediate representation.

The IR backend is designed to provide a high-level, platform-agnostic representation of the Kotlin code, making it easier to target different platforms and architecture. The IR code is optimized for use by the code generator, reducing the number of intermediate representations required, and allowing for better optimization.

Kotlin IR backend is an important part of the Kotlin compiler as it enables the compiler to target different platforms, such as the JVM, JavaScript, and Native code, while still providing a high-level representation of the code. This makes it easier to maintain and evolve the compiler and provides a more flexible way to target new platforms in the future.

2.1.3.3 Basics

In order to be able to build a Kotlin compiler plugin it is necessary to understand how the Kotlin IR syntax tree looks like.

First, one important information is that every node in the tree implements the **IrElement** interface. IrElement has an extension function, called **dump()**, that allows to get as output the IR tree from any point.
This feature is extremely useful when trying to generate code, because, in the case of errors, the exception thrown does not indicate what caused the problem. The message associated to a thrown exception just says the following:

```
org.jetbrains.kotlin.backend.common.BackendException: Backend
    Internal error: Exception during IR lowering
```

Listing 2.11: Kotlin IR lowering Exception

Typically, this is symptom of the generation of not well-formed elements, that are caught during compilation. This can be verified by writing the code that wants to be generated and comparing its *dump()* output against the one created by the compiler plugin, finding the necessary changes that needs to be done.

For example, it is going to be analyzed the generated IR tree of this snippet of code:

```kotlin
fun main() {
    println("test")
}
```

Listing 2.12: Kotlin basic code to demostrate the Kotlin IR representation

This is the complete Kotlin IR syntax tree that is generated from the source code of Listing 2.12:

```
FILE fqName:<root> fileName:Main.kt
FUN name:main visibility:public modality:FINAL <> () returnType:
    kotlin.Unit
 BLOCK_BODY
  CALL 'public final fun println (message: kotlin.String): kotlin.
    Unit [external] declared in kotlin.io' type=kotlin.Unit origin=
    null
   message: CONST String type=kotlin.String value="test"
```

Listing 2.13: Kotlin IR tree of Listing 2.12

The indentation of the IR allows to understand the relationship between each node, increasing its readability. In order to provide more clarity, they are going to be analyzed the most significant nodes of the IR.
Starting from the following node:

```
FUN name:main visibility:public modality:FINAL <> () returnType
    :kotlin.Unit
```

Listing 2.14: Kotlin IR tree of the main function in Listing 2.12

This represents the definition of the *main()* function and the IR tree representation provides a lot of information about it:

- **FUN** indicates that the current node is a declaration of a function;
- **name** is used to express how the function is called;
- **visibility** shows that the main function visibility is public;
- **modality** declares that main is final;
- **<>** refers to the fact that the function does not have a generic type;
- **()** means that main does not take parameters;

- **returnType** indicates that the return type of the main function is *kotlin.Unit*.

The child of the function IR element is the initialization of its body, which is declared like this:

```
BLOCK_BODY
```

Listing 2.15: Kotlin IR tree of the body block of the main function in Listing 2.12

When analyzing the IR tree, two different keywords are going to be found for bodies. The first one is **BLOCK_BODY**, which is used in this specific case in Listing 2.15: this means that the body is expected to have multiple statements. The second one is **EXPRESSION_BODY**, which is indicated when the body holds a single expression, such as a variable assignment.

Finally, the Listing 2.16 shows the *println()* function call and its parameter:

```
CALL 'public final fun println (message: kotlin.String): kotlin
    .Unit [external] declared in kotlin.io' type=kotlin.Unit
    origin=null
  message: CONST String type=kotlin.String value="test"
```

Listing 2.16: Kotlin IR tree of the body block content of the main function in Listing 2.12

Similarly to the *FUN* keyword, **CALL** is used to indicate that a function is being called. Specifically, it declares the call of the *println* function, alongside with information about where to find its declaration, which is in this case in *kotlin.io*, since the print is a Kotlin library function. Moreover, the function has one parameter, which is called message. The second line in Listing 2.16 gives all the details about it: it is a constant, defined by the **CONST** keyword, it is a *String* and its value is *test*.

Another basic feature that needs to be explained is how to actually visit the IR syntax tree from the code point of view. One key concept that is necessary to cover before diving into it is the **visitor design pattern**[13]. This is a behavioral design pattern which purpose is to add new operations to existing classes without modifying the classes themselves. The operation defined can be performed on elements of an object structure, which is defined in a separate class, called the **Visitor**, that implements the desired behavior. The elements of the object structure are updated to accept the Visitor instance, allowing it to traverse the structure and perform the operation on each element. The structure of the design pattern just described is reported in the UML in Figure 2.6[13].

[13]https://www.baeldung.com/java-visitor-pattern

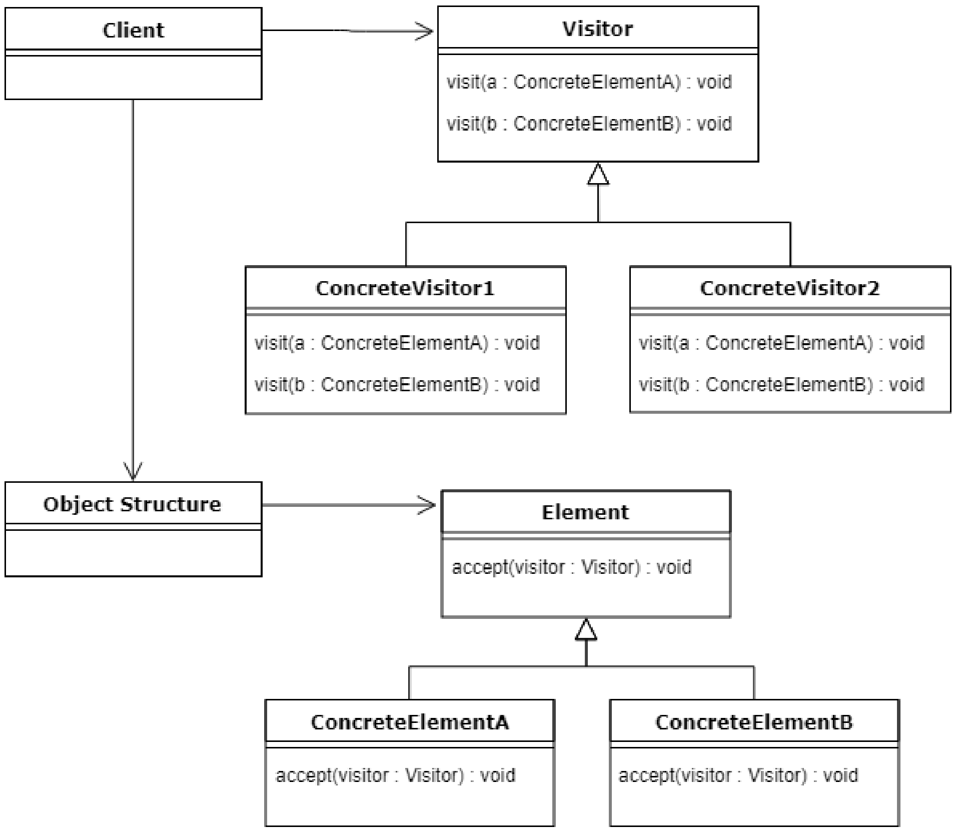

Client
Visitor
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
ConcreteVisitor1
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
ConcreteVisitor2
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
Object Structure
Element
accept(visitor : Visitor) : void
ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void

Figure 2.6: The Visitor design pattern UML

In the context of the Kotlin compiler, the visitor pattern is applied as in Figure 2.7. The *IrElement* interface represents the object structure that needs to be visited. As anticipated previously in this section, this interface is implemented by all the node of the IR syntax tree, which means that every element can be visited through this pattern.
For example, the two implementations of *IrElement*, shown in Figure 2.7, are *IrClass* and *IrFunction*, which represents class declarations and function declarations respectively. A lot of other different elements that are not specified in the UML exists; this omission is done in order to increase its readability. Examples of the omitted elements are: *IrConst* for constant, *IrCall* for function calls, *IrBranch* that includes if-else blocks, etc.
The *IrElement* has two functions that enable the visitor pattern that have two completely different behaviors:

- **accept**: it allows the visit of only the current element. For example, if trying to visit a class, only the class element is going to be analyzed and returned;

- **acceptChildren**: this function implementation consists on calling the *accept* function on each child of the current element, including itself.

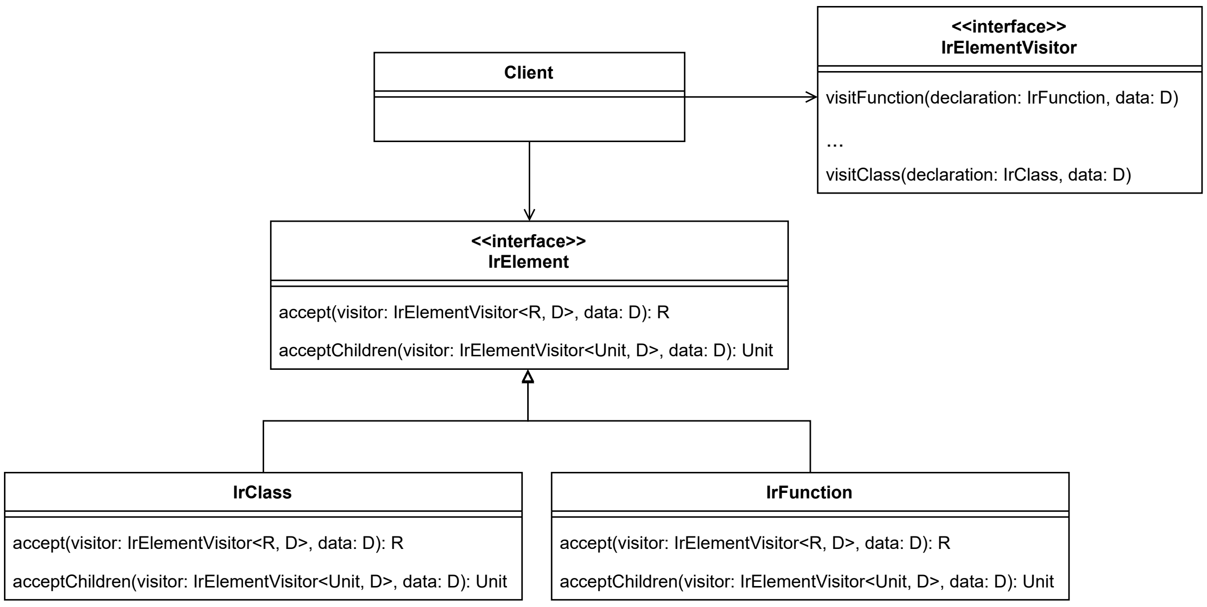

Client
<<interface>>
IrElementVisitor
visitFunction(declaration: IrFunction, data: D)
...
visitClass(declaration: IrClass, data: D)
<<interface>>
IrElement
accept(visitor: IrElementVisitor<R, D>, data: D): R
acceptChildren(visitor: IrElementVisitor<Unit, D>, data: D): Unit
IrClass
accept(visitor: IrElementVisitor<R, D>, data: D): R
acceptChildren(visitor: IrElementVisitor<Unit, D>, data: D): Unit
IrFunction
accept(visitor: IrElementVisitor<R, D>, data: D): R
acceptChildren(visitor: IrElementVisitor<Unit, D>, data: D): Unit

Figure 2.7: The Visitor design pattern UML applied in the Kotlin compiler plugin context in order to visit the IR syntax tree

Both functions possess two parameters, where the first one is the *IrElementVisitor*, the second one is *data*, which is used to pass information about the context to the IR visitor. The return type in the *accept* function can be used to get as output the elements visited, and it is useful, for example, when the navigation aims to find a specific element. Otherwise, the return can just be set as *Unit*.

```kotlin
class CustomVisitor(
  private val elements: MutableList<IrElement>
) : IrElementVisitor<Unit, Nothing?> {
  override fun visitElement(element: IrElement,data: Nothing?){
    elements.add(element)
    element.acceptChildren(this, data)
  }
}

fun collect(element: IrElement) = buildList<IrElement> {
  element.accept(CustomVisitor(this), null)
}
```

Listing 2.17: Example of a custom visitor and a function that supports the collection of elements

In the case of *acceptChildren*, its return type is *Unit*, which means that when visiting recursively the children of an element, no data can be returned. Whenever it is necessary to collect information from a recursive visitor, it is usually created a builder object that can be updated and then returned. In Listing 2.17[14] the role of the *collect* function is to create a mutable list that is updated on the visit of each child of the root element.

As previously said, the **IrElementVisitor** is the interface that has to be implemented in order to create a custom visitor. It requires two type parameters: the first one indicates the return type of the visit functions, the second one the type of the data that can be used to pass contextual information. The interface provides a lot of different visit methods, that can be used based on the wanted behavior. In Figure 2.7 are reported *visitFunction* and *visitClass*, but other examples are *visitBody*, *visitVariable*, *visitDeclaration*, etc.

The **IrElement** and the **IrElementVisitor**, with the understanding of the IR syntax tree, allows creating basic Kotlin compiler plugins, without modifying the existing source code.

2.1.3.4 Advanced features

There are two advanced aspects that need to be introduced, which are the building of new IR elements and the transformation of the IR syntax tree.

In order to understand how to build IR elements, it is important to keep in mind that the Kotlin compiler entry point provides the **IrPluginContext**, which contains the information about the context of the plugin. This is a critical element because it can be used to obtain an instance of the **IrFactory**, which is a factory that can create IR elements, such as *IrClass* for creating new classes or *IrSimpleFunction* used for creating functions.

```kotlin
pluginContext.irFactory.buildFun {
    name = Name.identifier("main")
    visibility = DescriptorVisibilities.PUBLIC
    modality = Modality.FINAL
    returnType = typeUnit
}
```

Listing 2.18: Main function IR building by using the IrFactory

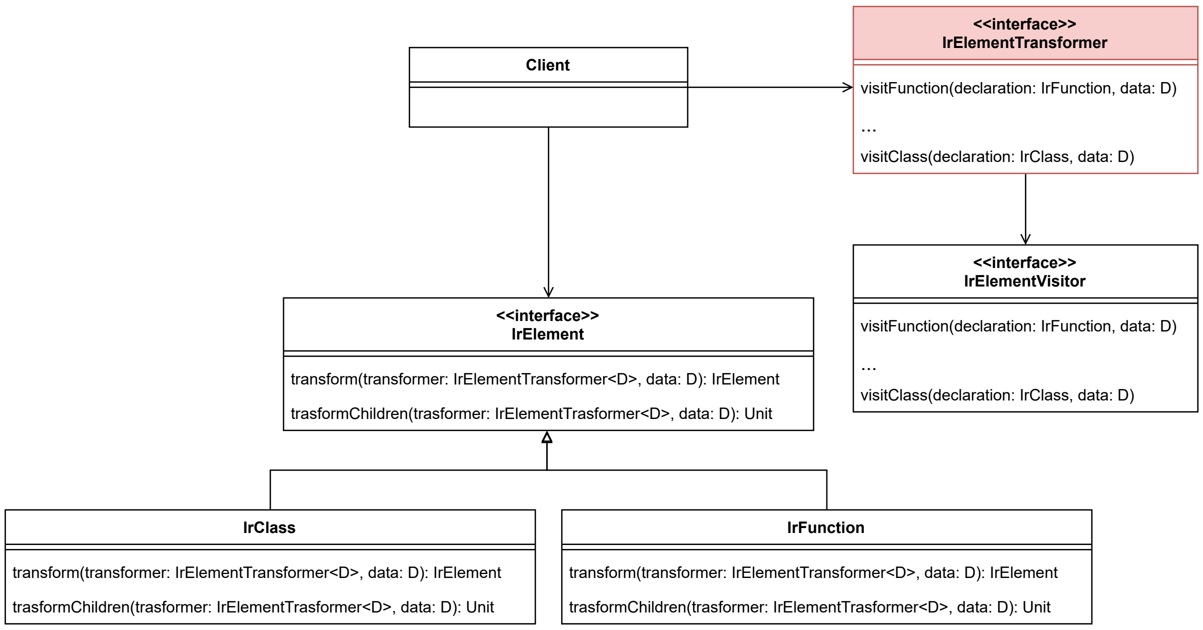
[14] https://blog.bnorm.dev/writing-your-second-compiler-plugin-part-3

For example, the code in Listing 2.18[15] shows how to use the factory to create a function called *main* and that does not have a return type.

The factory can only be used to create new element declarations, so it is necessary another way in order to build statements and expressions. This is provided by the **IrBuilder** and by the **IrBuilderWithScope**, which is really convenient when, for example, a new statement is created in a specific function scope.

The transformation of the IR can be done similarly to the navigation seen in Section 2.1.3.3, but it is performed by another interface that extends *IrElementVisitor*, which is called **IrElementTransformer**. The UML that shows the organization of the interfaces in this context can be seen in Figure 2.8.

Considering the *IrElement*, there are two transforming functions, that matches the



<<interface>>
IrElementTransformer
visitFunction(declaration: IrFunction, data: D)
...
visitClass(declaration: IrClass, data: D)
Client
<<interface>>
IrElementVisitor
visitFunction(declaration: IrFunction, data: D)
...
visitClass(declaration: IrClass, data: D)
<<interface>>
IrElement
transform(transformer: IrElementTransformer<D>, data: D): IrElement
trasformChildren(trasformer: IrElementTrasformer<D>, data: D): Unit
IrClass
transform(transformer: IrElementTransformer<D>, data: D): IrElement
trasformChildren(trasformer: IrElementTrasformer<D>, data: D): Unit
IrFunction
transform(transformer: IrElementTransformer<D>, data: D): IrElement
trasformChildren(trasformer: IrElementTrasformer<D>, data: D): Unit

Figure 2.8: The Visitor design pattern UML applied in the Kotlin compiler plugin context in order to transform the IR syntax tree

navigation functions:

- **transform**: it is like the *accept* function considered for the visiting of the IR tree, and it is used to transform only a specific element;
- **transformChildren**: similarly to *acceptChildren*, it transforms recursively all the children of the element, including itself.

[15] https://blog.bnorm.dev/writing-your-second-compiler-plugin-part-4

The *IrElementTransformer* provides the same methods of the *IrElementVisitor*, such as *visitClass* and *visitDeclaration*, but the return type is always set as the element itself. Basically, when wanting to transform a function it is sufficient to create an *IrElementTransformer* and to override the *visitFunction* method, implementing whatever transformation needed. For example the element can be modified or created by using the *IrFactory* or the *IrBuilder*, or existing elements can be deleted.

As a whole, all these functionalities discussed constitute powerful tools that allow all kind of compile-time modifications, that can be encapsulated in a Kotlin compiler plugin.

2.1.3.5 Example

The complexities involved in Kotlin compiler plugins, discussed in the previous sections, need a further exploration. For this reason, it has been developed a simple plugin[16], which aims to dive deeply into the details of basic and advanced features implementation.

For example, considering the following code:

```kotlin
fun test() {
    println("Hello!")
}

fun main() {
    test()
}
```

Listing 2.19: Kotlin code without the modification of the compiler plugin created as an example

When executing the code in Listing 2.19, this is the result shown in the console:

```
main declaration
test declaration
Hello!
```

Listing 2.20: Output of the execution of Listing 2.19 with the application of the compiler plugin created as an example

[16]https://github.com/ElisaTronetti/compiler-plugin-kmp

The goal of this compiler plugin is to modify at compile time every function declaration, inserting a print of the function name. This means that after the compilation, the code in Listing 2.19 is going to be transformed as in the code in Listing 2.21. The arrows added are used to identify the lines of code created by the compiler plugin.

```kotlin
fun test() {
    -> println("test declaration")
    println("Hello!")
}

fun main() {
    -> println("main declaration")
    test()
}
```

Listing 2.21: Kotlin code with the modification of the compiler plugin created as an example

One thing to notice is that, typically, the functions' names are not passed from compilation time to run time, this information would be usually lost. By the intervention of the plugin, that information is not lost, but it is inserted in the print function and shown to the user.

Moreover, the module of the project that contains the code that is going to use the compiler plugin is a **Kotlin Multiplatform** project. The reasons behind this choice are going to be explained afterwards in Section 4.1, but this example is also a **proof of concept** of the feasibility of creating Kotlin compiler plugins for multiplatform projects.

This project example is composed by three submodules:

- *gradle-ir-plugin*: it creates the subplugin, identifying the location of the compiler plugin, which is in the same Gradle project but in a different submodule;
- *compiler-ir-plugin*: the actual compiler plugin, that is going to be discussed in details;
- *kmp-sample*: it is the Kotlin multiplatform project, which uses the Gradle plugin to include the compiler plugin;

Since the Gradle module is just based on mandatory but standard settings, it is only going to be analyzed the compiler module.

The first component of the compiler plugin is the **CommandLineProcessor**, which sets one command line option: when including the plugin to a project, it can be specified a boolean parameter that enables or disable the plugin, making it easier to use.
The creation of this parameter consists on building an option using the *CliOption* class, which is used to inform the compiler plugin what parameters to expect and what is their role. The option created for this example is shown in the code in Listing 2.22.

```
CliOption(
    optionName = ARG_ENABLED,
    valueDescription = "bool <true | false>",
    description = "If the compiler plugin should be applied",
    required = false
)
```

Listing 2.22: Creation of compiler option that enables or disables the plugin

The entry point of the compiler plugin is the **ComponentRegistrar**. It checks whether the plugin is enabled or not, and, in case it is, it registers a generation extension, which is going to actually perform code changes. In the *ComponentRegistrar* implementation, the function that perform this feature is the following:

```
override fun registerProjectComponents(
    project: MockProject,
    configuration: CompilerConfiguration
) {
    if (configuration.get(ARG_ENABLED)) {
        IrGenerationExtension.registerExtension(
            project,
            IrGenerationExtensionImpl()
        )
    }
}
```

Listing 2.23: Example of registration of extension if the plugin is enabled performed by the *ComponentRegistrar*

In Listing 2.23 it is shown how to register the extension that is going to generate code. The **IrGenerationExtension** implemented method is *generate*, and it is shown in Listing 2.24.

```kotlin
override fun generate(
    moduleFragment: IrModuleFragment,
    pluginContext: IrPluginContext
) {
    val funPrintln = pluginContext.referenceFunctions(
        FqName("kotlin.io.println")
    ).first()

    moduleFragment.transform(
        TransformerImpl(pluginContext, funPrintln),
        null
    )
}
```

Listing 2.24: Example of *IrGenerationExtension* implementation of methos generate

The *generate* method just shown has two different purposes:

- It aims to find the reference of the Kotlin print function, which is going to be used in the code generation. This is done by specifying the name of the function and its package, then it is going to be searched in the whole plugin context;
- It creates an instance of the transformer implementation, giving it as argument the plugin context and the print function found, which are all the information that is going to be used to modify the code.

The most important element of all, that actually perform the code transformation, is the **TransformerImpl**, which implements the **IrElementTransformer** interface.
Since the modifications that needs to be achieved are applied to all the function declaration bodies, the *visitFunction* needs to be overridden as reported in the code in Listing 2.25, which works in the following way:

- for each function declaration found in the context, verify if it has a body. For example, abstract declarations do not have a body;

- if a body is found, it modifies the declaration it passing it to the *irDebug* function.

```kotlin
override fun visitFunction(
    declaration: IrFunction
): IrStatement {
    val body = declaration.body
    if (body != null) {
        declaration.body = irDebug(declaration, body)
    }
    return super.visitFunction(declaration)
}
```

Listing 2.25: Example of *visitFunction* implementation of the transformer, used to visit all the function declarations

The implementation *irDebug* is shown in Listing 2.26, and it is used to add an *irEnter* element and then all the statements in the body of the function are inserted again in the function body, in order to maintain the previous behavior. The body is recreated and returned by using the *DeclarationIrBuilder*.

```kotlin
private fun irDebug(
    function: IrFunction,
    body: IrBody
): IrBlockBody {
    return DeclarationIrBuilder(
            pluginContext,
            function.symbol
        ).irBlockBody {
            +irEnter(function)
            for (statement in body.statements) +statement
        }
}
```

Listing 2.26: Example of implementation of a function that adds a new element into the function body

In the code in Listing 2.27 it is shown the extension function *irEnter*, which is used to build a new *IrCall*. The *IrCall* aims to create a new function call, specifically of the print function retrieved previously. It also adds a new argument, which is a string of the name of the function it is going to be created into.

```kotlin
private fun IrBuilderWithScope.irEnter(
    function: IrFunction
): IrFunctionAccessExpression {
    return irCall(logFunction).also { call ->
        call.putValueArgument(
            0,
            irString("${function.name} declaration")
        )
    }
}
```

Listing 2.27: Example of creation of a new function call and adding to it arguments

In conclusion, this transformation, applied to all the function declarations, creates the behavior that this project example wanted to achieve.
It proves that the source code can be modified completely and that the compiler plugin can be also applied smoothly to multiplatform projects.

Chapter 3

Transparent alignment in Kotlin

The alignment, discussed in Section 1.1.1.1, is a crucial feature that must be provided in order to create a working implementation of the **Aggregate programming** paradigm.

The first step necessary to find a solution for a problem is to define a simple use case, which is going to be the base of the resolution attempts. A good starting point to illustrate this is represented by the code in Listing 3.1, which was also discussed previously in Section 1.1.1.1. In order to be coherent with the final solution, the code is now written in Kotlin and the nbr-expression is now called *neighboring*.

```kotlin
fun f1() {neighboring(e1)}
fun f2() {neighboring(e1)}

f1()
f2()
```

Listing 3.1: Starting point code to resolve the alignment problem

The goal that needs to be achieved is to find an alignment solution that is able to keep track of the current computational state of a device, making it possible to align only the correct expressions with each other. In this example, it is necessary to find a unique way to identify the execution of the neighboring expression in the body of the *f1* function from the one in the *f2* function.

The computing device identification is not considered right now, since it is not a core problem in these attempts. Once the base case is solved, then further complexities of the alignment problem are going to be analyzed, such as the branch construct.

There are key aspects that are considered for this study and that are used to compare the different approaches:

- **Availability for Kotlin multiplatform**: it is important to keep in mind that DSL is developed by using Kotlin multiplatform, which choice is discussed in Section 4.1. The approaches chosen must be available for a Kotlin multiplatform architecture;
- **Portability**: the alignment solution has to work for all the targeted platforms by Kotlin Multiplatform, which are Kotlin Native, JavaScript and JVM. Multiple devices computing on the same platform have to be able to align correctly;
- **Interoperability**: the solution should allow devices executing on different platform to align;
- **Efficiency**: key aspect to consider, since it can not be produced an elaborated solution that needs devices with high computational requirements;
- **Transparency**: finally, this feature has to be transparent to the final user, without making the developer write ulterior code to make the alignment work.

Different approaches have been tried to find a solution, and, for each attempt, the advantages and the disadvantages are discussed. The following sections analyze all the possibilities taken in consideration: Section 3.1 goes into details of the attempts with stacktraces and hashes, Section 3.2 exploit the problem with annotations and KSP, understanding their limits. Finally, Section 3.3 describes in details the solution adopted, by developing a Kotlin compiler plugin. In each section there is a table that recaps the characteristics for each technology tried for the alignment based on the key aspects highlighted previously.

3.1 Stacktraces and hashes

Since the alignment problem is quite complex, the firsts attempts regards simple strategies without the concern of efficiency, trying to use already existing Kotlin functionalities to generate a unique identifier.

One way of keeping track of the functions called during a program execution is by checking the **stacktrace**. This leads to one of the possible solutions, which is throwing

exceptions whenever an aggregate programming construct is computed, and then using the generated exception stacktrace as the unique identifier.
The stacktraces can be generated in all the platforms that the DSL aims to target. On each platform, the stacktrace is identical for every program execution, meaning that it is possible to align different devices that are executing the same program on the same target.
Moreover, this solution offers transparency to the final user, because the alignment can work without having the developer to write additional code.
On the other hand, this option presents two main problems that can not be avoided:

1. The first problem regards the efficiency of this solution. Since an aggregate program runs continuously on each computing device, an enormous amount of exceptions would be thrown, causing delays that can create issues in a distributed system;

2. The second problem refers to the interoperability of device computing on different targeted platforms. The stacktraces generated by the Native target are completely different from the one generated by JavaScript and JVM, and they represent information in a non-identical way. This means that devices running on different platforms can not align correctly, since the identifier generated for the sequence of the functions called does not match.

Following a similar line of reasoning, the next attempt involved the **hash-codes**. Kotlin provides a method that, given an object, returns its hash-code. It is guaranteed that the hash-code generated is always the same whenever two objects are equal to each other. In order to take advantage of this feature, it was created a new class called *Event*, which has the role to encapsulate the DSL expression that is currently being computed by a device, and then uses the object hash-code as identifier.
In general, this is not a suitable solution: the hash generated is the same during a single execution, but this is not the case when running multiple devices that have to communicate, meaning that the alignment can not be achieved, since the identifiers can be different even though they should not be.
This issue is also present when trying to align devices running on different platforms, but there is an additional problem: the hash-code generated from a target is not the same in another target, creating a discrepancy impossible to overcome.

	Available for Kotlin Multiplatform	Allows same target interoperability	Allows different targets interoperability	Acceptable efficiency	Transparent to the final user
Stacktrace	yes	yes	no	no	yes
Hash	yes	no	no	yes	yes

Table 3.1: Comparison between stacktrace and hash for solving the alignment problem

Considering these attempts and their characteristics, it is possible to conclude that they are not suitable as resolution for the alignment problem. In order to give a clear view of their pros and cons, the characteristics of the alternative just discussed are recapped in Table 3.1.

3.2 Annotations and KSP

Since simple Kotlin functionalities are not feasible for solving the alignment problem, another possibility is to try different metaprogramming alternatives, such as the ones discussed in Chapter 2.

The first metaprogramming technique taken in consideration is **annotations**. As explained in Section 2.1.1, annotations can be used to attach additional metadata to elements in the code, which can be used to create specific behaviors. For example, it is possible to use them to define that a function should be causing the alignment and specify through the annotations how to handle it.
Before trying to develop such solution, a few considerations must be highlighted:

- A key aspect of this project is the interoperability, achieved by using Kotlin multiplatform for the chosen targets (Native, JavaScript and JVM). While this thesis is being redacted, Kotlin\JS does not support annotations, creating a compatibility issue that is difficult to overcome;

- In order to take advantage of the metadata provided by annotations, the developer should annotate manually every element in the source code whenever the alignment is required. This is not acceptable, since it would violate the requirement of transparency for the final user.

This results in the impossibility to use annotations for the alignment, since a possible solution developed using said methodology would not meet the basic requirements of the project.

The next alternative that involves metaprogramming is **KSP** (Kotlin Symbol Processing), which is described in details in Section 2.1.2. Starting from KSP 1.0.1, it is possible to use KSP on multiplatform projects[1], which includes also the targets necessary to reach successfully the outcome desired.

KSP allows the creation of lightweight compilers, providing an API that hides all the complexities that a complete compiler plugin would involve. While designing a possible solution, it is necessary to keep in mind the biggest limitation that this technology brings within: the impossibility to modify the source code.

Since KSP can access the code at compile time, it would be possible to extract information that can be used to build a custom stacktrace that keeps track of the sequence of functions called. Then, this custom stack can be used as identifier when needed. For this reason, it has been created a class *Stack*, which interface is reported in Listing 3.2.

```kotlin
interface Stack {
    fun currentPath(): String
    fun align(token: String): Unit
}
```

Listing 3.2: Stack interface for KSP

By using the *Stack* it is possible to add to a data structure everything that is necessary for the alignment, which might be for example function calls. Considering the base case cited previously in Listing 3.1, when executing *f1*, the stack list should contain *[f1, neighboring]*, and when computing *f2* would be *[f2, neighboring]*, making it possible to uniquely identify the two different expressions.

Trying to obtain the result just described, the problem can be divided in two smaller steps:

1. The aggregate programming constructs are part of the DSL functions exposed to the final user, meaning that it is not possible for the user to change their implementation in any way while using it. Moreover, the name of these functions is known in advanced. This leads to create a simple solution for handling them, which consists on changing their implementations and adding a line of code that it is responsible to

[1] https://kotlinlang.org/docs/ksp-multiplatform.html

insert in the *Stack* data structure the function name. For example, in the *neighboring* implementation there would be a function call responsible for the alignment, that add its name to the stack.

2. The second problem, which is more complex, can not be solved in the same way, since it would be the final user to manually solve the alignment problem. The solution would be to modify the functions *f1* and *f2* using KSP and adding the same line of code discussed for the previous case. On the other hand, the direct modification of the source code is not allowed by KSP, which means that the only solution would be to recreate the same code of the user, with the added function calls. This solution would be feasible for simple use cases, but it is not efficient when dealing with complex systems.

This leads to the conclusion that it is crucial for the solution of the alignment problem to have the possibility to modify the source code. KSP can be used as a temporary solution for some use cases, also taking advantage of the simple and powerful API that it provides, but the downsides are important factors to consider.

	Available for Kotlin Multiplatform	Allows same target interoperability	Allows different targets interoperability	Acceptable efficiency	Transparent to the final user
Stacktrace	yes	yes	no	no	yes
Hash	yes	no	no	yes	yes
Annotation	no	yes (manually)	yes (manually)	yes	no
KSP	yes	yes	yes	no	yes

Table 3.2: Comparison between stacktrace, hash, annotation and KSP for solving the alignment problem

Concluding, the characteristics of annotation and KSP are added in Table 3.2 alongside with the stacktrace and hash ones, summarizing the considerations made on all the alternatives considered up until this point.

3.3 KCP: solution with total transparency and portability

Kotlin compiler plugins are another metaprogramming alternative. As discussed previously in Section 2.1.3, the development of a compiler plugin requires a lot of time and study, but it also provides a total freedom when trying to modify the code at compile time.

The following considerations prove that a plugin can be used to solve the alignment problem:

- When developing a project using Kotlin multiplatform, the compiler translates under the hood the Kotlin code into the code for the targets. Before doing that, it is transformed into an *Intermediate Representation*, called **IR**, which as been discussed in Section 2.1.3.2. This representation allows the creation of multiplatform compiler plugins, without the need to distinguish the plugin between the different platforms;
- Since the source code is completely available at compile time, it is possible to analyze it in order to make the plugin understand when the alignment is needed. Then, new code can be generated to guarantee the correct alignment, which can rely to a stack similar to the one discussed for KSP;
- Since the generation is based on the intermediate representation, this allows the interoperability between the different project targets, and also on devices running on the same platform;
- The plugin modifications are applied at compile time, which does not impact in a relevant way the execution time;
- Finally, the code generation is also totally transparent to the user, meaning that he does not have to worry about the alignment.

The Kotlin compiler plugin solution is the chosen one for creating this crucial feature, because, comparing it with the other alternatives, it is the one that respects all the requirements. All the characteristics of the analyzed possibilities are summarized in Table 3.3.

	Available for Kotlin Multiplatform	Allows same target interoperability	Allows different targets interoperability	Acceptable efficiency	Transparent to the final user
Stacktrace	yes	yes	no	no	yes
Hash	yes	no	no	yes	yes
Annotation	no	yes (manually)	yes (manually)	yes	no
KSP	yes	yes	yes	no	yes
KCP	yes	yes	yes	yes	yes

Table 3.3: Comparison between stacktrace, hash, annotation, KSP and Kotlin compiler plugin for solving the alignment problem

The chosen approach is similar to the one discussed for KSP in Section 3.2: the best way to understand if two different devices are aligned is by using a custom stack, which can keep track of the sequence of functions called during the computation. In order to do so, the stack is used to push the function name into the data structure when a function is called, and then it is popped when the control flow exits that function. A similar behavior is provided in the case of the domain restriction caused by the branch construct, pushing in the stack the condition evaluated.

This important data structure has been called **Stack** and its interface is shown in Listing 3.3.

```
interface Stack {
    fun currentPath(): Path
    fun alignRaw(token: X?): Unit
    fun dealign(): Unit
}
```

Listing 3.3: Stack interface for Kotlin compiler plugin solution

The stack has an internal mutable list that is updated when calling the method *alignRaw* and *dealign*. Specifically, *alignRaw* is used to push a generic identifier into the stack, while, on the other hand, *dealign* is used to pop it.
When a device needs to know its computational state, it can retrieve its path using the function *currentPath*. A **Path** is a data class that is used as a wrapper of the current stack state, in order to return the immutable list, following Kotlin best practice. Its implementation is in Listing 3.4.

```
data class Path(val path: List<Any?>)
```

Listing 3.4: Path dataclass for Kotlin compiler plugin solution

The data structures that can be used to store the information about the computation has been defined, which lays the foundations of the alignment solution.
It is now necessary to provide a method that actually updates the stack correctly, without having to do it manually. This is also going to be the function call generated by the compiler plugin to perform the alignment.
The method *alignedOn* in Listing 3.5 needs to be explained further, since its behavior is central in the solution proposed.

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

Listing 3.5: Function that perfoms the alignment in the Kotlin compiler plugin solution

The function signature accepts two parameters: the first one is the *pivot*, which is the identifier that is going to be pushed in the stack, the second one is *body*, which is the element that is going to be computed.
For instance, considering the base example reported at the beginning of this chapter in Listing 3.1, and supposing the computation of *f1*, the *pivot* would be the name of the function, that is the string *f1*, and the body would be *f1()*, which performs the computation of that function.
The implementation of *alignedOn* is entitled to update the stack pushing the pivot, then it executes the function taken as parameter and returns as output the value computed. Once the computation of the function is completed, it also resets the stack at its previous state.

The basic information to explain the solution has been presented, and it is now necessary to divide the problem in two smaller steps. In Section 3.3.1 it is going to be discussed how to perform the alignment when dealign with function calls, giving also an example of the generation of the code created by the compiler plugin. Then, in Section 3.3.2 it is going to be explained how to guarantee the alignment when there are branches that cause domain restriction.

3.3.1 Function alignment with KCP

Before describing the mechanism used to generate code using the compiler plugin, it is necessary to define what the Kotlin compiler plugin should generate in the context of functions.

Listing 3.6 shows how the code presented as base case example in Listing 3.1 is modified during compilation by the compiler plugin.

To each function call now corresponds a new call to the *alignedOn* function reported previously in Listing 3.5, creating the mechanism of alignment.

```
1  fun f1() {
2      alignedOn("neighboring"){
3          neighboring(e1)
4      }
5  }
6  fun f2() {
7      alignedOn("neighboring"){
8          neighboring(e1)
9      }
10 }
11
12 alignedOn("f1"){
13     f1()
14 }
15 alignedOn("f2"){
16     f2()
17 }
```

Listing 3.6: Generation goal to handle the alignment of Listing 3.1

While executing the code in Listing 3.6, this is what happen to the stack:

- First, the *alignedOn* on line 12 is executed and the state of the stack is now *[f1]*;
- Then, *f1* is computed, which causes another *alignedOn* call. The stack is updated, and its current value is now *[f1, neighboring]*;
- While executing the neighboring function, the device looks into the messages received by its neighbors in order to find fields that match the current stack value *[f1, neighboring]*. Those neighbors are the one aligned with the considered computing device;
- Then the *alignedOn* call at line 2 has completed its execution, so there is another update on the stack, which dealign from the current state. The stack is *[f1]*;
- Also the *alignedOn* on line 12 is finished, causing the stack to be empty again;

- The exact behavior happens when computing *f2*, with the difference that the stack when computing the neighboring function is *[f2, neighboring]*. In this way, only the neighbors values found that executed *f2* and then *neighboring* are considered.

Now that the end result that wants to be achieved for this base case is clear, the compiler plugin can be explained in details.
Since most of a basic plugin implementation has been presented in the example in Section 2.1.3.5, only the crucial part of the transformation of the code are going to be discussed in this section.

The **IrGenerationExtension**, before registering the transformer necessary to perform the code modification, needs to retrieve two elements, which absence would preclude the execution of the plugin. The user would be informed with a console message in the case that, during the compilation, the required elements were not found.
The referred elements are:

1. **The alignedOn function declaration**: in order to be able to call an already existent function, it is necessary to specify to the compiler to look for that function. In this case it is necessary the reference to *alignedOn*, discussed in Listing 3.5;

2. **The aggregate context**: the functions exposed by the DSL are encapsulated in the **AggregateContext** class, which also contains the reference to the stack instance. Since it is possible to modify the stack only through its instance, the aggregate context class retrieved here can be used to get the instance of the object when calling the *alignedOn* function. More details about the *AggregateContext* class are going to be discussed in Chapter 4.

In order to optimize the solution, not every element of the source code is going to be modified by the compiler plugin, but only the necessary ones. If the final user needs a particular alignment not covered by the developed plugin, he can use the *alignedOn* function manually.
The entry point of the DSL is the function **aggregate**, which takes as parameter a function type with receiver. The receiver is the *AggregateContext*, meaning that all the call members of the receiver can be used inside the *aggregate* function call.
After retrieving the necessary elements, the *IrGenerationExtension* registers the **Aggregate-CallTransformer**, which visits through the visitor pattern all the function calls looking for

all the *aggregate* call. Because of that, the compiler plugin knows that inside that function calls there might be the necessity to modify the code to provide the alignment feature.

For each aggregate function call the transformation is then delegated to the **AlignmentTransformer**, which handles the alignment required by the function calls and the branch constructs.
Starting from the function calls, this is what happen:

- The *visitCall* provided by the transformer's interface is overridden, providing access to all the children of the aggregate function call that are also calls;
- Then the behavior can be divided in three different cases:
  1. If the function call has been already aligned, any modification is applied to the code;
  2. Otherwise, in order to provide another optimization, it is checked if the considered function call or any of its children has a reference to the *AggregateContext*. Only the functions provided by the DSL, which are **neighboring**, **repeating** and **sharing** have a reference to that context. This means that everything not related to the DSL does not require to be modified for the alignment problem. This research is performed by checking the current function call's receivers and by the **AggregateRefChildrenVisitor** whenever it is necessary to visit the children.
     If any of the elements analyzed have a reference to the *AggregateContext*, then the code is not transformed;
  3. If the *AggregateContext* reference is found, then the transformation proceeds.

  Of the three possible outcomes, only the third one makes the computation continue, in all the other cases the not modified function call is returned. For this reason, it is necessary to assume that an *AggregateContext* reference has been found in order to continue to explain how the transformation works;
- In order to be able to create a new function call in the source code, it has to be declared inside an **irStatement**, which is used to define also the scope where the function is created;
- Inside the *irStatement* is then created the *alignedOn* function call;

- Finally, the transformed function call is returned.

The creation of the *alignedOn* function call is crucial for this transformation, and it is going to be described in more details.
Once created the *irStatement* it is possible to create a new **irCall** like shown in Listing 3.7.

```kotlin
irCall(alignedOnFunction).apply {
    // Set generics type
    putTypeArgument(expression.type)
    // Set aggregate context
    putArgument(
        alignedOnFunction.dispatchReceiverParameter!!,
        aggregateContextReference
    )
    // Set the argument that is going to be pushed in the stack
    putValueArgument(
        irString(
            expression.symbol.owner.kotlinFqName.asString()
        )
    )
    // Create the lambda that is going to call expression
    val lambda = buildLambdaArgument(
        pluginContext,
        aggregateLambdaBody,
        expression
    )
    putValueArgument(1, lambda)
}
```

Listing 3.7: Generation of the *alignedOn* function call in Listing 3.5

The *irCall* builds a new *alignedOnFunction*, which is the reference to the *alignedOn* function responsible for the alignment. It is set the type argument and the dispatch receiver, which are both necessary to create a correct element. Then, the value argument is the first parameter of the *alignedOn* function, and it is created using the name of the originating expression. Finally, the second value argument is the body of the call of the original function, which is recreated as a lambda.

The lambda as argument is created in another function, which is implemented as follows:

```
1  pluginContext.irFactory.buildFun {
2      name = Name.special("<anonymous>")
3      this.returnType = expression.type
4      this.origin = IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA
5      this.visibility = DescriptorVisibilities.LOCAL
6  }.apply {
7      this.patchDeclarationParents(this@buildLambda.parent)
8      if (expression.symbol.owner.returnType.isUnit()) {
9          this.body = context.
10                         irBuiltIns.
11                         createIrBuilder(symbol).
12                         irBlockBody {+expression}
13     } else {
14         this.body = context.
15                         irBuiltIns.
16                         createIrBuilder(symbol).
17                         irBlockBody {+irReturn(expression)}
18     }
19 }
```

Listing 3.8: Generation of the lambda body of the *alignedOn* function call in Listing 3.5

From line 2 to 5 of Listing 3.8, standard parameters are set, such as the return type, the lambda parent and data that informs the compiler that it is dealing with an anonymous function. Then, if the original expression return type is unit, then the body of the lambda is just the expression itself. Otherwise, a return block is added, which contains the original function call.

To recap the process it is possible to start from the simple code in Listing 3.9. It is reported a snippet of an aggregate program before its execution, and the behavior expected from the Kotlin compiler plugin can be summarized in three aspects:

1. It identifies the starting point of the aggregate program;
2. It does not perform the alignment on the *println* function call, since it does not involve aggregate constructs;
3. It aligns the *neighboring* function call.

```
aggregate {
    println("do not align")
    fun calculate() {
        neighboring("test")
    }
}
```

Listing 3.9: Base example of aggregate program

The Figure 3.1 shows how the various elements developed for the compiler plugin work together. The *AggregateCallTransformer* finds the aggregate call, the *AlignmentTransformer*

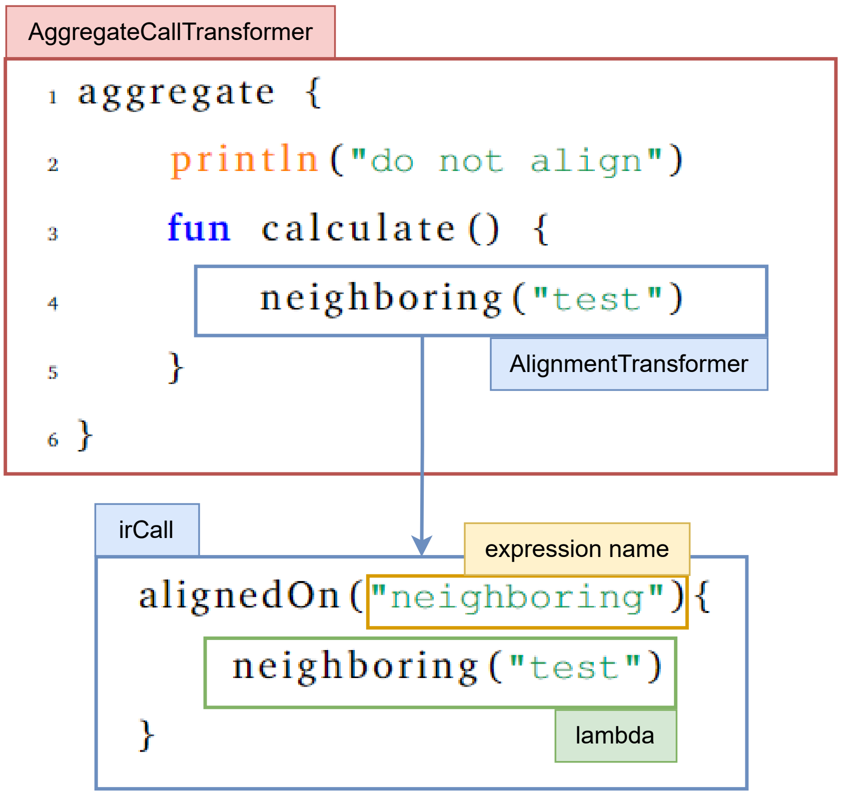


Figure 3.1: Specification of each compiler plugin element that modify the source code in Listing 3.9

is able to understand which function needs to be aligned, which in this example is *neighboring*. Then the new function call is created, with the two parameters: the name of the original function and its call.

3.3.2 Branch construct alignment

As anticipated in Section 1.1.1.1, also the branch construct creates a domain restriction that needs to be handled by the alignment feature. Again, it is necessary to define the required result that has to be achieved by the compiler plugin.

```kotlin
val number = (0..10).random()
if (number > 5) {
    neighboring("higher")
} else {
    neighboring("lower")
}
```

Listing 3.10: Base example of Koltin code that requires the branch alignment

Given the example in Listing 3.10 its transformation would be the one reported in Listing 3.11.

```kotlin
val number = (0..10).random()
if (number > 5) {
    alignedOn("[number > 5, true]"){
        alignedOn(neighboring){
            neighboring("higher")
        }
    }
} else {
    alignedOn("[constant, false]"){
        alignedOn(neighboring){
            neighboring("lower")
        }
    }
}
```

Listing 3.11: Generation goal to handle the alignment of branches required in Listing 3.1

The *alignedOn* function calls at line 4 and 10 are created by the mechanism explained previously is Section 3.3.1, and at line 3 and 9 there are the alignment functions for the branches.
Supposing that the random generated number is 7, the stack during the computation of the code in Listing 3.11 is the following:

1. Since the value of number is 7, the condition of the *if* at line 2 is true and its body is executed;

2. The alignment function at line 3 allows putting in the stack *[[number > 5, true]]*, which is a pair of the condition evaluated and its actual value, which is true;

3. Then, to the stack it is added the neighboring call, leading to *[[number > 5, true], neighboring]*. This means that the neighboring function align only whenever the condition of the *if* is true, meaning that it is higher than 5;

4. After the computation of neighboring, it is popped from the stack, leaving again only *[[number > 5, true]]*;

5. Finally, the body of the *if* has been successfully executed and the stack can be cleared.

Before diving into the creation of the *alignedOn* parameters, it is necessary to understand how the IR syntax tree handled the branches when there are multiple conditions, which is slightly different from the single condition just presented. Kotlin IR works with binary trees, which means that whenever it has to handle branches with multiple conditions, it creates nested branches.
For example, considering Listing 3.12, it is represented as shown in Figure 3.2.

```kotlin
if (a && b) {
    function()
}
```

Listing 3.12: Example of code where a branch has multiple conditions

The Figure 3.2 shows that a branch is handle like follows:

- The first branch of the if evaluates the condition a;

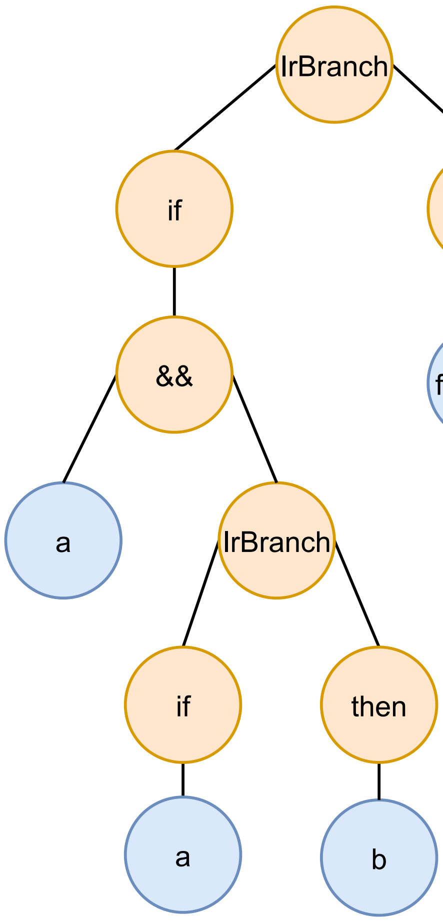


Figure 3.2: The IR of a branch with multiple condition shown in Listing 3.12

- If a is true, then it is evaluated the condition b. This creates the chance to align b based on the evaluation of a, because after the evaluation of a the stack is *[[a, true]]*, which means that b align only with the neighbors that evaluated a as true;
- After obtaining both the result of a and b, it is calculated the value of *a && b*;
- If the full condition is evaluated as true, the stack is *[a && b, true]*;
- Then it is computed *function()*, which is aligned based on the stack just created.

It is the **AlignmentTransformer** that, as well as the functions' alignment, handles the branches. It overrides two different functions:

- **visitBranch**: it represents the *if* and the *else if* condition;
- **visitElseBranch**: it is used for the *else* branch;

The way these two types of elements are handled is the same, so just the normal *if* branch is discussed.

The result of a branch is divided in the IR in blocks or expressions:

- **IrBlock**: in this case the *if* result consists of a block, like this:

```
if(condition){
    function()
    function()
}
```

- **IrExpression**: the result of the *if* it is not encapsulated in brackets, it is composed by one expression:

```
if(condition) function()
```

The *IrBlock* and the *IrExpression* requires a slightly different implementation: if a branch was constituted by an *IrBlock*, when creating the *alignedOn* function call it is necessary to keep in mind that the second parameter is going to be a block and not a single expression.
The building of the *alignedOn* function call is the same of the code shown in Listing 3.8, the only difference is the creation of the parameters.

The first parameter is the name of the element that is going to be put in the stack. As seen before, just the name of the condition is not sufficient, since the alignment is based on the name and the condition value. For this reason, the first parameter is a pair constituted by the condition name and its value, like this: *[condition, true]*.
When the condition is a variable, a function call or an anonymous function, the stack will contain the exact name of the element. In the case of a constant, for example just the value *true*, then it is used the placeholder *constant*.

The second parameter is constituted by a lambda that contains the body of the *if*. Since it is considered in this discussion that the body is an *IrBlock*, then it is necessary to create a new block that is going to create the lambda.
There are three main things that need to be done when creating the lambda body, and they are shown in Listing 3.13:

1. **Setting the return type**: since the body of the *if* is a block, it is constituted by multiple expression. The last expression return type defines the return type of the

block. In order to be able to get the result of the *if* construct, it is necessary to set it correctly;

2. **Creating the IrBlock**: the block created contains all the expressions of the previous block but the last one. In this way, the behavior of the block is not going to change;

3. **Setting the return type**: finally, the last element to add to the block is the return, which is defined by the last expression. It must be created inside the *irReturn* element, which is used to create a special block that is expected to match the return type.

```kotlin
this.returnType = getReturnType(lastExpression)
this.body = context.
            irBuiltIns.
            createIrBuilder(symbol).
            irBlockBody {
                for (bodyStatement in expression.statements) {
                    +bodyStatement
                }
                +irReturn(lastExpression)
            }
```

Listing 3.13: Creation of the lambda body when modifying a *IrBranch*

This constituted the whole behavior of the compiler plugin when dealing with a *IrBranch*, but the transformation is almost identical when a *IrElseBranch* is considered. One final detail of the transformation of the branches is in common with the functions handling: the alignment is required if and only if there are aggregate programming construct calls, which means that the modification of an *IrBranch* when its body does not involve any aggregate function is useless.
For this reason, the same controls performed for the function calls are applied here, which means that, when visiting any branch, only if the *AggregateContext* reference is found in any of it body expression or expressions' children, then the alignment transformation is actuated. This enhances the performances of the Kotlin compiler plugin created.

Chapter 4

Collektive: aggregate programming in pure Kotlin

The name given to the project is **Collektive**[1], which emphasize the aggregate programming aim to dispose of numerous devices that work together to achieve a certain goal. Moreover, the name contains the term **'kt'**, which refers to the development in Kotlin.

The goal of Collektive is to provide to the user a minimal DSL that makes it possible to create aggregate programs transparently. It is necessary to keep in mind that the solution needs to respect these requirements:

- **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;

- **Minimality**: the goal is to design it 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 differ-

[1] https://github.com/ElisaTronetti/collektive

ent environments, which is crucial for systems requiring deployment in multiple locations or scalability to meet changing demands.

The following sections are organized in order to present in details the project developed. Specifically, Section 4.1 discuss the main technological choice involved in the development of the DSL, in Section 4.2 is shown the project structure, in Section 4.3 is analyzed in details the DSL created. Then, Section 4.4 presents the final result and how the DSL can be used, and Section 4.5 is used to highlight the validation methods applied to test the correct behavior of the project developed.

4.1 Technology

This section presents the main technological choice taken regarding the development of the DSL, in order to meet the requirements cited previously.
It is important to achieve a certain level of portability, specifically for devices running on JVM, JS and Kotlin Native platforms, which makes also possible to gain interoperability between different targets. Moreover, an ideal solution would not require writing the DSL code in three different programming languages to match the required platforms.
For the reasons just presented, the choice made for this project development is **Kotlin Multiplatform**.

4.1.1 Kotlin Multiplatform

Kotlin Multiplatform technology is specifically designed to streamline the development process for cross-platform projects.
It achieves this by minimizing the amount of time developers spend writing and maintaining identical code for different platforms. This approach saves valuable resources and time, as the code can be written once and used across multiple platforms.

Kotlin Multiplatform allows to write code in the Kotlin programming language and use it across multiple platforms, including Android, iOS, web, and desktop. As can be seen in Figure 4.1[2], Kotlin Multiplatform code behaves in the following way[2]:

- **Common Kotlin**: the code written in common Kotlin can be used across multiple platforms without requiring modifications. This code can include business logic,

[2]https://kotlinlang.org/docs/multiplatform.html

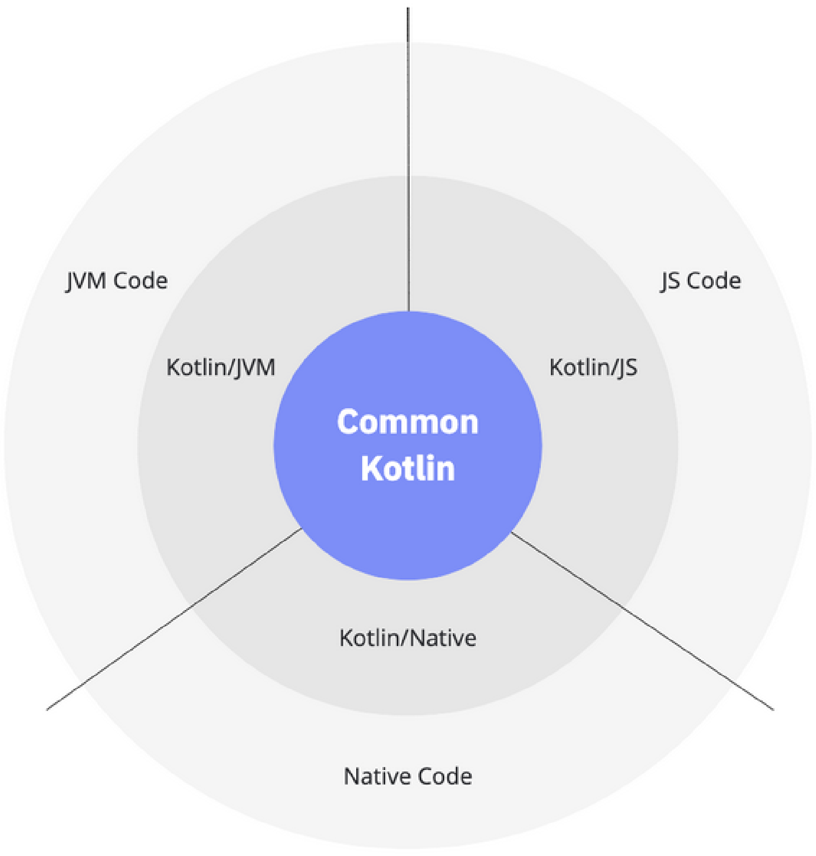


Figure 4.1: How Kotlin Multiplatform works

data models, and other non-platform-specific functionalities. Common code can rely on a set of libraries that are available for Kotlin Multiplatform and that cover everyday tasks, such as serializations or coroutines;

- **Platform-specific versions of Kotlin**: this refers to **Kotlin/JVM**, **Kotlin/JS** and **Kotlin/Native**, which include extensions to the Kotlin language, allowing developers to use platform-specific APIs, features and tools;
- **Platform native code**: finally, through these platforms it is possible to access the platform native code (JVM, JS and Native), with the possibility to use all the native features.

The way that Kotlin Multiplatform avoid the necessity to write and maintain the same code over and over again for all the targeted platforms, is by providing the possibility to share the same code across multiple platforms.
It is possible to share the code in different ways[3]:

[3]https://kotlinlang.org/docs/multiplatform-share-on-platforms.html

- **Share code on all platforms**: this is typically done when some business logic is common to all platforms, in order to write it only once in the common code and then share it on all the targets, as shown in Figure 4.2[3];

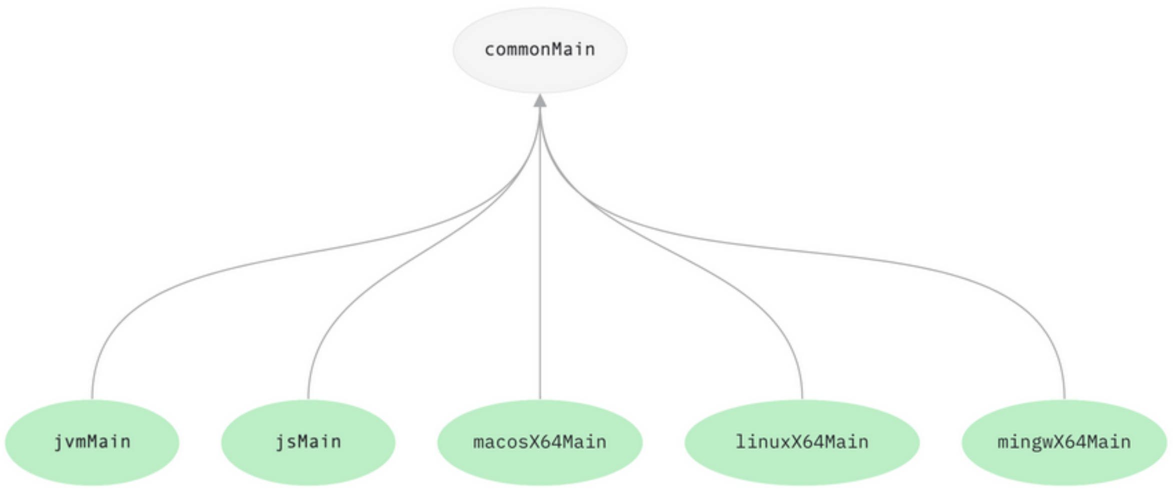


Figure 4.2: Kotlin Multiplatform sharing code on all platforms

- **Share code among some platforms**: this organization is usually applied when similar platforms share a big portion of code. As it can be seen in Figure 4.3[2], it is possible to define hierarchies that allow to organize the shared code, such as the *desktopMain* folder that shares some code with its dependencies, which does not include the whole code present in *commonMain*.

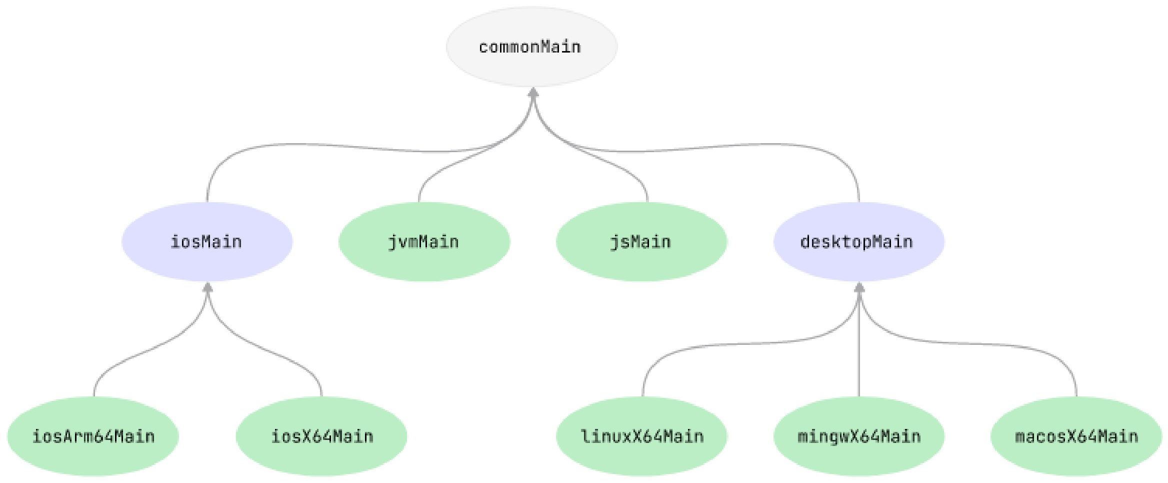


Figure 4.3: Kotlin Multiplatform reusing code among some platforms of the project

In some cases it might be necessary to access platform-specific APIs from the common code. This can be done by using the specific Kotlin mechanism of expected and actual declarations[4].

In Figure 4.4[4] it is shown an example of this mechanism. For instance, in the common

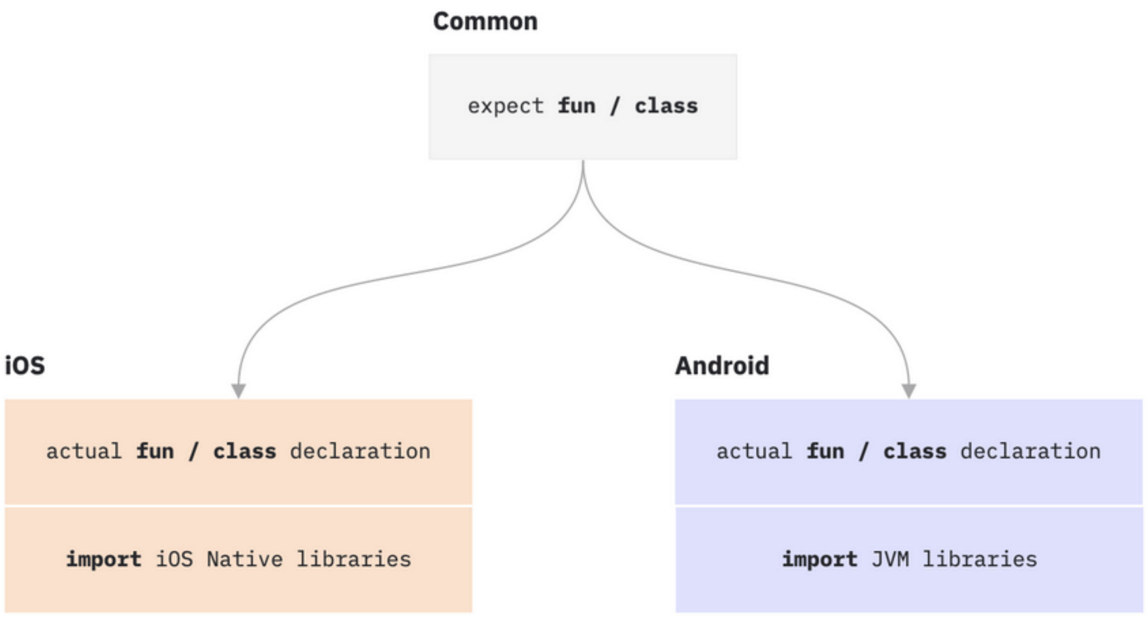

Common
expect fun / class
iOS
actual fun / class declaration
import iOS Native libraries
Android
actual fun / class declaration
import JVM libraries

Figure 4.4: Kotlin Multiplatform expect and actual dependency mechanism

code it is created a new function or class, and it is declared using the keyword **expect**. The keyword informs the compiler that it should look for the implementation of this element in the platform-specific folders. This can be done by declaring the same function or class by using the **actual** keyword, which allows taking advantage of platform-specific APIs. In the cited example, the actual implementation required are for the iOS and Android platforms.

Kotlin Multiplatform also includes tools to help developers manage their codebase, such as Gradle plugins that allow for building and testing code across multiple platforms.

One advantage of Kotlin Multiplatform is, for example, the ability to share code between Android and iOS. This can be especially valuable for companies that want to develop apps for both platforms, as it can help reduce development time and costs. With Kotlin Multiplatform, developers can write shared code for common features, such as user authentication or data storage, and then write platform-specific code for the UI and other platform-specific features.

Concluding, Kotlin Multiplatform is the perfect technology to achieve this thesis project, because it allows creating a unique code base that is possible to use on three different

[4] https://kotlinlang.org/docs/multiplatform-connect-to-apis.html

platforms: JVM, JavaScript and Native. Moreover, the implementation of platform-specific behaviors is not required, meaning that only the common code has been developed.

4.2 Project structure

Collektive has been developed as a Gradle project composed by three different submodules, as shown in Figure 4.5. The cited submodules are:

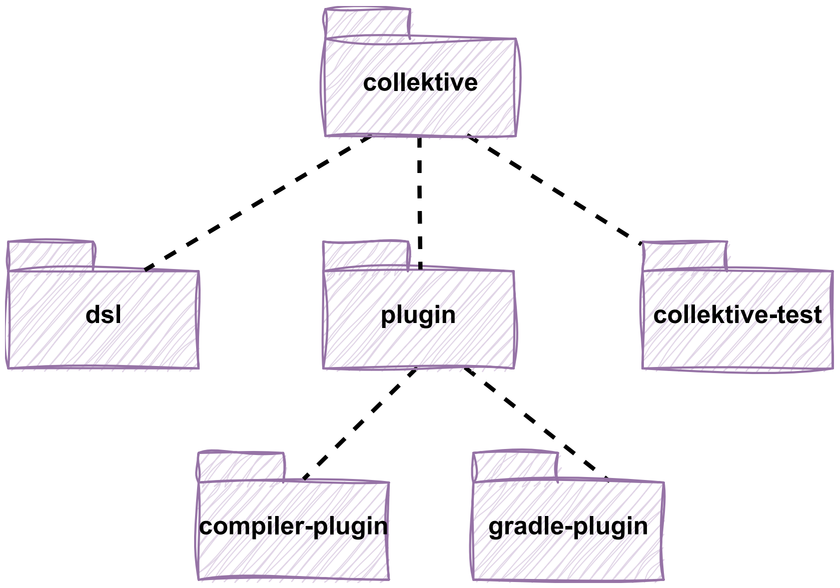


Figure 4.5: Package diagram of Collektive project

1. **plugin**: it is divided in two submodules itself:
   (a) **gradle-plugin**: necessary plugin used by a gradle project in order to include the compiler plugin. Its structure has been introduced previously in Section 2.1.3.1;
   (b) **compiler-plugin**: the compiler plugin is used to modify a data structure, which to keep tracks of the stack at runtime. For each aggregate function and branch construct, the stack data structure is updated in order to allow the alignment whenever necessary. The specific functionalities of the compiler plugin have been explained in depth in Section 3.3;
2. **dsl**: the actual DSL implementation in Kotlin Multiplatform, where the logic is implemented and that exposes the operators of the aggregate computing. The description of this submodule is going to be presented in Section 4.3;

3. **collektive-test**: this submodule is used to test an aggregate program by running it on a simulated environment provided by the Alchemist Simulator [9]. More details about this are in Section 4.5.

4.3 DSL

A **DSL** (Domain-Specific Language)[5] is a programming language or language construct that is designed to be highly specific to a particular domain, or problem space. Unlike general-purpose programming languages, which are intended to be applicable across a wide range of domains and problem types, DSLs are created to meet the specific needs of a particular application or system. For this reason, general-purpose languages, such as Java, are generally more complex than DSLs.
DSLs can be implemented in a number of ways, including: standalone programming languages, libraries that extend existing programming languages, or annotations or macros that modify the syntax or behavior of an existing language. DSLs can be used to provide a higher level of abstraction over complex systems or processes, allowing developers to express ideas and concepts in a more concise and intuitive way.
To ensure that DSLs are fit for their intended purpose, they are typically developed in close collaboration with domain experts. In fact, many DSLs are not designed to be used by programmers, but rather by non-technical individuals who are knowledgeable in the relevant domain. This collaborative approach helps to ensure that the DSL is intuitive and expressive for its intended users, allowing them to more easily and effectively express complex ideas and processes.

The DSL developed for this project is organized as shown in the package diagram in Figure 4.6.
As previously described, Kotlin Multiplatform allows to create common code that is then compiled on three different targets, which are JVM, JavaScript and Native. Since the behavior of the DSL does not require platform-specific features, it is present in the project only the **commonMain** package, which contains the whole implementation.
It is also present the **commonTest** folder, which is used to define the tests to verify the correct behavior of the system and that is possible to run over the different platforms, to

[5] https://www.jetbrains.com/mps/concepts/domain-specific-languages/

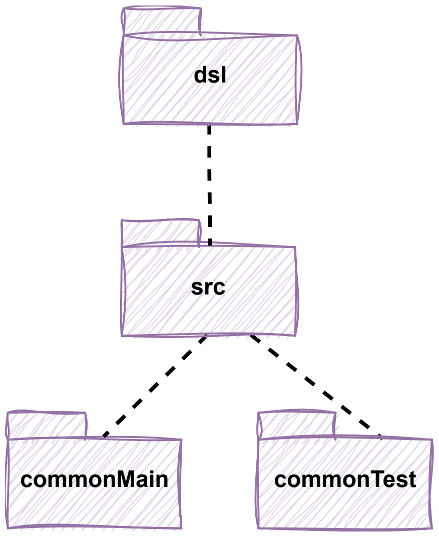


Figure 4.6: Package diagram of Collektive DSL

ensure that the implementation works as predicted for all the targets. In Section 4.5 are going to be presented more details about the tests.

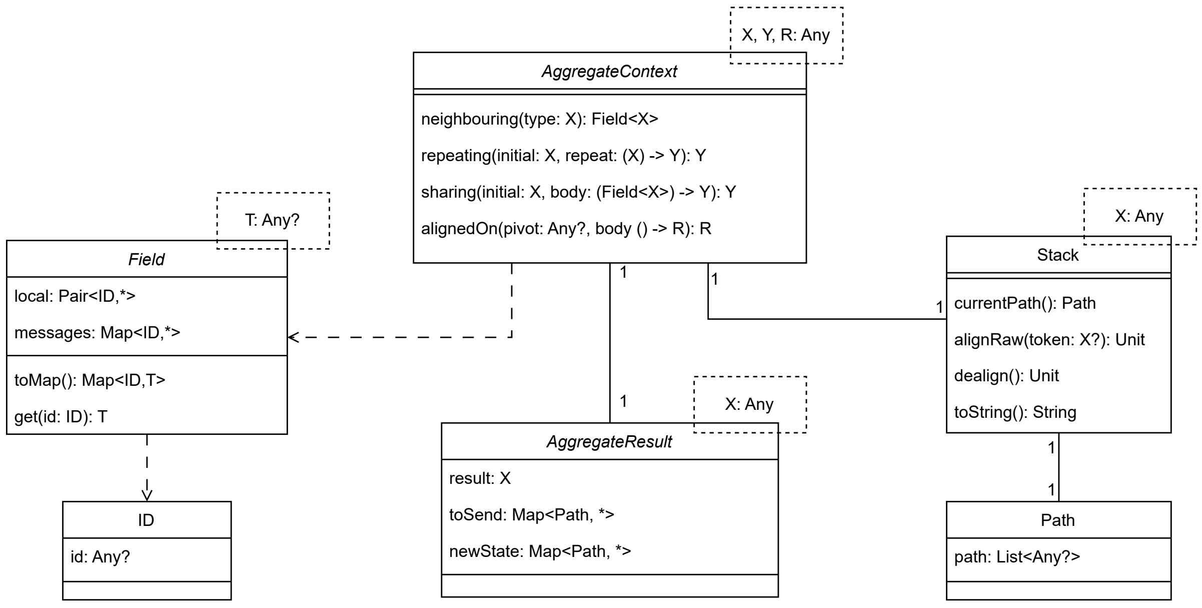


Figure 4.7: Class diagram of Collektive DSL

The implementation of the project involves the elements presented in the class diagram in Figure 4.7, which are:

- **AggregateContext**: it is the class that contains the implementation of the aggregate programming constructs and the function that performs the alignment. Moreover, this class is used to handle all the data necessary for the computation: it stores the local state of the device computed in the previous round, the messages received from the neighbors, the device identifier, and an instance of the stack used to keep track of the computational state. The *AggregateContext* contains also the data class *AggregateResult*, making it possible to instantiate it when the context receiver is *AggregateContext*;

- **AggregateResult**: the computation of an aggregate program returns as output an *AggregateResult*, which contains the results obtained from the current iteration, and it is composed by: the actual result returned by the aggregate program, the messages to send to the neighbors to notify them of the current results, and the new device local state;

- **Field**: this class represents the **computational field**, and it is used by the aggregate constructs to handle this data structure. It contains the local value of the device that is computing and all the messages received by the neighbors. Specifically, the local value is a pair that contains the ID and a map of paths and computed values. Similarly, the messages are composed by the identifier of the neighbor and the same map containing the path and the registered result obtained in the computation corresponding to that path;

- **ID**: a simple identifier which goal is to be able to uniquely recognize different computational device in the aggregate system. Any type of identifier can be used;

- **Stack**: this is the data structure discussed previously in Section 3.3, and it is used by the compiler plugin to save the current computational state in order to provide the alignment feature. Through an instance of this class it is possible to retrieve the current path in order to create a computational field or to get the neighboring devices values computed on that specific path;

- **Path**: the path class has already been introduced in Section 3.3, and its role is to return an immutable list whenever it is necessary to retrieve the current stack state.

A key aspect of this project is the implementation of the aggregate programming constructs contained in the *AggregateContext* class.
The implementation of the **neighboring** function is reported in Listing 4.1. It accepts in input any parameter, which is the value computed by the current device and that is going to be shared with the neighborhood.

```kotlin
fun <X> neighboring(type: X): Field<X> {
    toBeSent[stack.currentPath()] = type
    val messages = messagesAt(stack.currentPath())
    return FieldImpl(Pair(localId, type), messages)
}
```

Listing 4.1: Neighboring implementation

The behavior of this construct consists on:

- Saving the value computed and the current stack in order to send the message to the neighbors;
- Retrieving the messages of the neighbors that correspond to the current path;
- Finally, returning a field with the local value and the neighbors values, ready to be manipulated.

```kotlin
fun <X,Y : Any> repeating(initial: X, repeat: (X) -> Y): Y {
    val res =
        if (previousState.containsKey(stack.currentPath())) {
            repeat(previousState[stack.currentPath()] as X)
        } else {
            repeat(initial)
        }
    state[stack.currentPath()] = res
    return res
}
```

Listing 4.2: Repeating implementation

The **repeating** function is used to represent the time evolution, and it allows the field to change dynamically.
This expression needs two parameters:

1. **initial**: the initial value used whenever the function is evaluated for the first time;

2. **repeat**: the function that is going to be computed and that takes as input the initial value or the output of the previous round evaluation.

The body of *repeating* can be broken down in the following steps:

- It checks whether a previous state exists for the current path in the *perviousState* map;
- If a previous state exists, it is invoked the *repeat* function with the value associated with the current path in the previous round;
- If a previous state does not exist, the function invokes *repeat* with the *initial* value;
- Furthermore, the result of the *repeat* evaluation is stored in the *state* map associated to the current state. This creates the evolution of the local device state, that is going to be used in the next aggregate program iteration;
- Finally, the computed value returned by *repeat* is returned.

The last aggregate programming constructs is **sharing**, and its implementation is shown in Listing 4.3. As described before in Section 1.1, the *sharing* allows to observe the neighbors' field, updated the local values and share immediately the updated state in a single operation.
This function accepts two parameters:

1. **initial**: similarly to the *repeating* constructs, the initial value is used whenever a *sharing* function is evaluated for the first time;
2. **body**: it is used to transform immediately the field, resulting a value that is going to be shared with the neighborhood.

The behavior of the *sharing* function can be described as follows:

- The function retrieves the messages of the neighbors for the current path from the stack instance;
- It checks whether a previous state exists for the current path, similarly to the *repeating* function;

- If a previous state exists, the function retrieves it from the *previousState* map. Otherwise, it is used the *initial* value;
- Taken the retrieved value, whether it is the *initial* value or the one computed in the previous round, and the messages from the neighbors, a new field instance is created;
- The expression invokes the *body* function with the new created field as argument;
- The returned value of the *body* function is stored in the *toBeSent* map assigned to the current path. This map contains all the messages to send to the neighbors before the next round begin;
- Finally, the output value of *body* is returned by the *sharing* function.

```kotlin
fun <X, Y: Any?> sharing(
     initial: X,
    body: (Field<X>) -> Y
): Y {
    val messages = messagesAt(stack.currentPath())
    val previous =
        if (previousState.containsKey(stack.currentPath())){
            (previousState[stack.currentPath()])
        } else {
                initial
        }
    val subject = FieldImpl<X>(
        Pair(localId, previous),
        messages
    )
    return body(subject).also {
        toBeSent[stack.currentPath()] = it
    }
}
```

Listing 4.3: Sharing implementation

The main entry point of an aggregate program is shown in the code in Listing 4.4.

```kotlin
fun <X> aggregate(
    localId: ID = IntId(),
    messages:Map<ID,Map<Path,*>>=emptyMap<ID,Map<Path,Any>>(),
    state: Map<Path,*> = emptyMap<Path, Any>(),
    init: AggregateContext.() -> X
) = singleCycle(localId, messages, state, compute = init)
```

Listing 4.4: Aggregate entry point

The *aggregate* function accepts four parameters:

1. **localId**: it is the identification number of a device, which can be used when it is necessary to make the same device compute another round of the aggregate program. This parameter is not mandatory, and, in case it is not defined, a new random identifier is generated;

2. **messages**: it might be sometime necessary to give the aggregate program some contextual information about the previous round outcome. This parameter is used to provide the messages received from the neighbors, and it is initialized ad an empty map as default value;

3. **state**: similarly to the *messages* parameter, this is used to propagate the previous state of the device in the current round. It is not mandatory, and it is initialized as an empty map whenever it is not found;

4. **init**: it is a lambda expression that takes an instance of *AggregateContext* as a receiver object.
   A lambda expression is a block of code that can be executed later, and it can be passed around as a value. In Kotlin, lambda expressions can have a receiver object, which is an object on which the lambda is invoked.
   In Kotlin, a context receiver is a way of providing a context or a scope to a block of code, such as a lambda expression. A context receiver allows the block of code to access the properties and functions of the receiver object using the *this* keyword, without having to explicitly specify the receiver object in the code.
   This means that when the lambda expression *init* is invoked, it has access to the

properties and functions of the *AggregateContext* object. By using the aggregate function, it is then possible to define the lambda that is going to be computed, and, by doing so, the aggregate constructs defined in the *AggregateContext* can be used freely.

This *aggregate* function in Listing 4.4 allows the computation of a single cycle, which means that only one round is going to be performed. This requires the user of the DSL to define how the different devices should communicate.
In Section 4.5 it is going to be presented a different *aggregate* function that allows to run multiple rounds of an aggregate program, used to test the correct behavior during the validation process.

The *singleCycle* function returns an **AggregateResult**, which is the output of an aggregate program in Collektive. The implementation of the *singleCycle* function is reported in Listing 4.5. This class contains the value computed from the aggregate lambda, a map of all the messages to send to the neighbors, and the new state of the device.

```
with(AggregateContext(localId, messages, state)) {
    AggregateContext.AggregateResult(
        compute(),
        messagesToSend(),
        newState()
    )
}
```

Listing 4.5: Single cycle *AggregateResult* output

4.4 Usage example

One important aspect to consider during the development of a DSL is its usability. It is important to provide the user with a minimal set of operators, each of them with a clear purpose.
This is what has been achieved with Collektive, since the constructs are coherent with the aggregate programming original operators.

In order to use Collektive, it is necessary to perform some steps to set up the environment:

1. **Include the compiler plugin**: since the alignment is crucial for the DSL to work, it is important to include into the Gradle project the custom Gradle plugin created to expose the Kotlin compiler plugin.
   Whenever a new submodule of Collektive is created, the Gradle plugin can be included as shown in the code in Listing 4.6;

```
plugins {
    id("io.github.elisatronetti.kotlinAlignmentPlugin")
  version "0.1.0"
}
```

Listing 4.6: Inclusion of the custom Gradle plugin to a local Gradle submodule

2. **Include the DSL**: it is also necessary to include the DSL to the project, in order to be able to access all the aggregate programming constructs developed. The code in Listing 4.7 shows how to include it into a new Collektive submodule.

```
dependencies {
    implementation(project(path=":dsl"))
}
```

Listing 4.7: Inclusion of the DSL to a local Gradle submodule

Once the setup is finished, the DSL is available into the new submodule. By calling the function **aggregate**, the DSL functions **neighboring**, **repeating** and **sharing** can be used. Moreover, the compiler plugin included provides the correct alignment of the elements.

```
val double: (Int) -> Int = { it * 2 }
val testValue: Int = 2
aggregate {
    neighboring(double(testValue))
}
```

Listing 4.8: Example of an aggregate program developed with Collektive

The snippet of code in Listing 4.8 shows an example of usage. The program shares with the neighbors the double of a local number. In this example, the value of the number is hardcoded, but in a real world example, this value might depend on the device environment, which would make different devices have a different value evaluation.

4.5 Validation

The term validation refers to the process of ensuring that the software meets the intended requirements and specifications and fits for its intended use. Validation is a critical aspect of the software development process as it helps to ensure that the software is free of defects, meets the expectations of its users, and performs its intended function correctly.

The validation process typically involves testing the software in various ways, and the goal of each type of testing is to ensure that the software meets the intended goal and that it performs as expected in various environments and scenarios.

Collektive validation consists on two main tests:

1. **Functional testing**: involves verifying that the software performs its intended functions correctly and without errors;

2. **Compatibility testing**: involves testing the software's ability to operate correctly in different environments and with different hardware and software configurations.

These validation processes are achieved in two different ways, which are going to be discussed in this section.

The first method tests both the functionalities and the compatibility of the system. This is obtained by taking advantage of the Kotlin Multiplatform features because, when it comes to testing, it provides several tools and frameworks that can be used to test code written in Kotlin on different platforms.
One popular testing framework for Kotlin Multiplatform is **KotlinTest**[6]; said framework is a multiplatform testing library that provides a common API for writing tests that can be run on different platforms. It supports a range of testing styles, including behavior-driven development (BDD) and property-based testing.
Kotlin Multiplatform also provides a set of common annotations that can be used to mark

[6]https://kotlinlang.org/docs/multiplatform-run-tests.html

tests and test suites, such as the **@Test** annotation. These annotations can be used in conjunction with testing frameworks like *KotlinTest* to write and run tests on multiple platforms.
Since the DSL is developed using Kotlin Multiplatform, the package **commonTest** contains the test implemented using *KotlinTest*.
The organization of the folder *commonTest* is shown in Figure 4.8. It contains only the

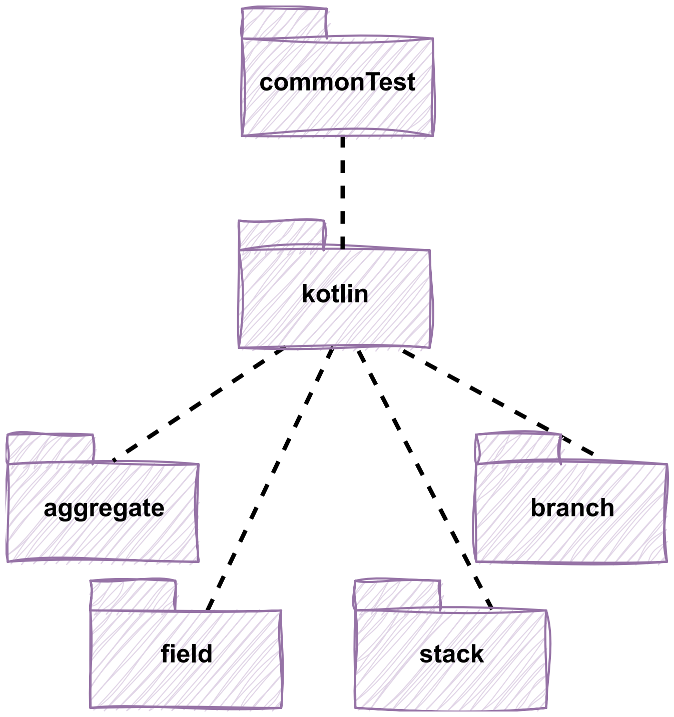


Figure 4.8: Package diagram of Collektive DSL tests

kotlin subfolder, since the implementation of platform-specific tests is not required.
The test are organized in the following folder system:

- **field**: contains unit testing of the *Field* class and the field manipulation functions, such us the retrieval of the lower comparable value contained in the field;
- **stack**: similarly to the *Field*, also the *Stack* class requires unit testing, to verify that it works as predicted;
- **branch**: it contains all the possible testing of the branching alignment. It verifies the correct alignment of simple if, else-if and else blocks and expressions. Furthermore, it tests the alignment of the *when* Kotlin expressions;
- **aggregate**: finally, in this folder are contained all the tests of the aggregate programming constructs, which are *neighboring*, *repeating* and *sharing*.

Since testing only one round of an aggregate program would not be enough to ensure the correct system behavior, it has been implemented an infrastructure that allows to test multiple rounds.
It is necessary to create a class that allows to save the messages and deliver them to other devices. This class is called **Network**, and its interface is shown in the code in Listing 4.9.

```kotlin
interface Network {
    fun send(localId: ID, message: Map<Path, *>)
    fun receive(): Map<ID, Map<Path, *>>
}
```

Listing 4.9: Network interface to simulate multiple round of an aggregate program in tests

Before creating an aggregate program, it is necessary to instantiate a *Network*, which contains an internal map of messages, and it is used to handle the communication between devices.

It is also necessary to create a new entry point for the DSL, which makes it possible to accept parameters to handle multiple rounds.

```kotlin
fun <X> aggregate(
    condition: () -> Boolean,
    network: Network = NetworkImpl(),
    init: AggregateContext.() -> X
) = runUntil(condition, network, compute = init)
```

Listing 4.10: Aggregate programming entry point used to compute multiple rounds

The new *aggregate* function is reported in Listing 4.10, and it accepts the following parameters:

1. **condition**: it is a lambda expression that takes no arguments and returns a *Boolean* value, which is used to establish whether the computation should run another round or not;

2. **network**: an instance of the *Network* class introduced in the code in Listing 4.9, which is used to handle sending and receiving messages;

3. **init**: the actual aggregate program, contained in a lambda expression which has as receiver object an *AggregateContext*.

This *aggregate* function calls **runUntil**, and in the snippet of code in Listing 4.11 it is reported the main logic its implementation.

```kotlin
while (condition()) {
    computed = singleCycle(
        localId,
        network.receive(),
        state,
        compute
    )
    state = computed.newState
    network.send(localId, computed.toSend)
}
```

Listing 4.11: Main logic of the *runUntil* function

The purpose of *runUntil* is to repeatedly execute the *compute* lambda expression until the *condition* lambda returns *true*.
In order to do that, every time it is necessary a new iteration, it is computed a **singleCycle**, which is the same function used in the normal aggregate program execution, which was presented in Section 4.3. A cycle requires the local identifier of the device, it retrieves from the network the message received from the neighbors, the current local state of the device and the lambda that define the computation.
Afterwords, it updates the local state, and it sends the via network the messages regarding the computed values of this round.

By using the mechanisms just presented, it is possible to create tests that check the correct interaction between devices, also verifying that the alignment works properly. Moreover, *KotlinTest* allows to run the implemented tests over the different targets of the project, testing the compatibility of the software over different platforms.

The second validation process regards mainly the functional testing, and it aims to provide a proof of concept of the potential of Collektive.
The goal is to implement a gradient algorithm using the constructs provided by the DSL,

and then simulate the computation of a distributed system using **Alchemist Simulator** [9]. In order to achieve this objective, it has been created a new submodule in the Collektive project called **collektive-test**. Alchemist dependencies, the DSL, and the Gradle and compiler plugin are fundamental for this purpose.

The following classes are required in order to create the correct infrastructure used to make the simulation possible:

- **CollektiveDevice**: this is the abstraction of a node present in the simulation environment. This class makes possible to send and receive messages, to store neighbors information and to uniquely identify nodes;

- **CollektiveIncarnation**: an incarnation is the interpreter that allows Alchemist to understand a language and to execute it correctly. This incarnation is specific for Collektive, and it is used to find the aggregate entry point through reflections, and it defines how each iteration of the aggregate program should be performed.

The algorithm is implemented as an *AggregateContext* extension function, and it is shown in the code in Listing 4.12.

```kotlin
fun AggregateContext.gradient(
    source: Boolean,
    sensor: DistanceSensor
) = sharing(Double.POSITIVE_INFINITY) { distances ->
    val paths: Field<Double> = sensor.distances() + distances
    val minByPath = paths.min(includingSelf = false)?.value
    when {
        source -> 0.0
        minByPath == null -> Double.POSITIVE_INFINITY
        else -> minByPath
    }
}
```

Listing 4.12: Gradient algorithm implemented using Collektive

The algorithm calculates the gradient from a node that is defined as *source*. By using the *sharing* construct, it is possible to retrieve all the neighbors' data. This is used to obtain

the distances of the neighbors from the source, and then, the distance of the current device is added to the neighbors one. After that, it is searched the minimum distance, which is going to be the current distance from the device to the source node. Finally, the algorithm returns zero if the node is the source, infinity if the minimum distance is not found, or the minimum value.
Since *sharing* has been used, the computed value is then propagated to the neighborhood.

In order to make the incarnation find the aggregate program, it is necessary to define an entry point, which is reported in the code in Listing 4.13.

```kotlin
class Aggregate(private val node: CollektiveDevice<*>) {
    private val nodeId = node.node.id
    private var state = emptyMap<Path, Any?>()
    fun entrypoint() = aggregate(
        IntId(nodeId),
        node.receive(),
        state
    ){
        gradient(nodeId == 0, node)
    }.also { state = it.newState }
}
```

Listing 4.13: Gradient algorithm entry point

The **Aggregate** class takes as parameter a *CollektiveDevice*, which is used to establish the device that is currently computing.
The function **entrypoint()** is the function that the incarnation is going to look for and execute.
The entry point creates an *aggregate* block, which takes as parameter all the contextual information about the device: the device identifier, the messages received from the neighbors, and the state obtained in the previous round. The computation consists of the execution of the gradient algorithm just presented, and it also defines that the node with the identification number *0* is going to be the source.
At the end of the single execution of the aggregate program, the current state of the device is updated.

Finally, it has been created the simulation environment, which creates **200 nodes** in a

2D space. The result of the creation of the nodes is shown in Figure 4.9.

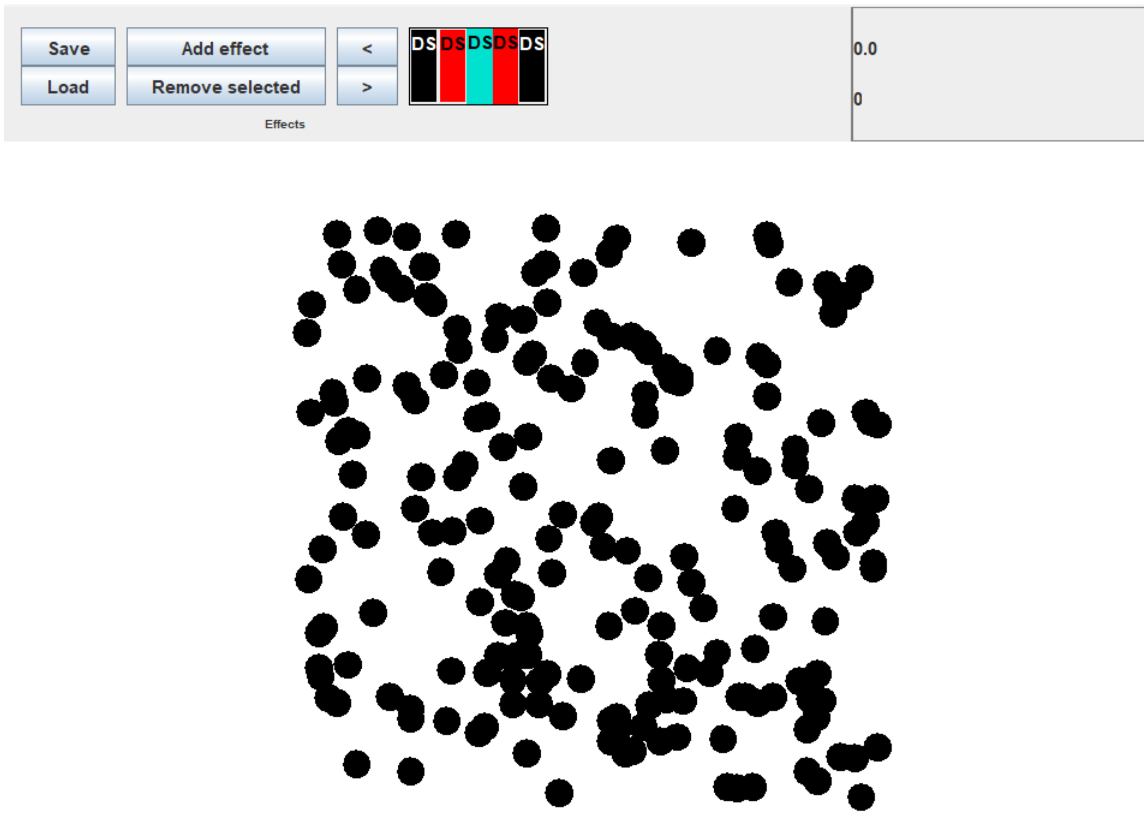


Figure 4.9: Alchemist simulation environment before running Collektive gradient algorithm

Once Alchemist has generated the nodes that are going to be involved, it is only necessary to run the simulation. The UI of Alchemist is going to color differently the nodes, basing on the distance from the source: the more the node is colored in red, the more is near to the source, the more the color tends to a colder color, the further is from the source node.

The source node can be distinguished from the other nodes from the different shape and color: it is shaped as a square, and it has a light blue border.

The user interface of the Alchemist Simulator shows also other information:

1. **Rounds performed**: it refers to the number of time the aggregate program has been executed before reaching a stable gradient. The example showed in Figure 4.10 performed **30331** rounds;

2. **Execution time**: this is a valuable indicator of the performance of the algorithm written using Collektive. The example in exam found a stable solution of the gradient in **150.73 milliseconds**.

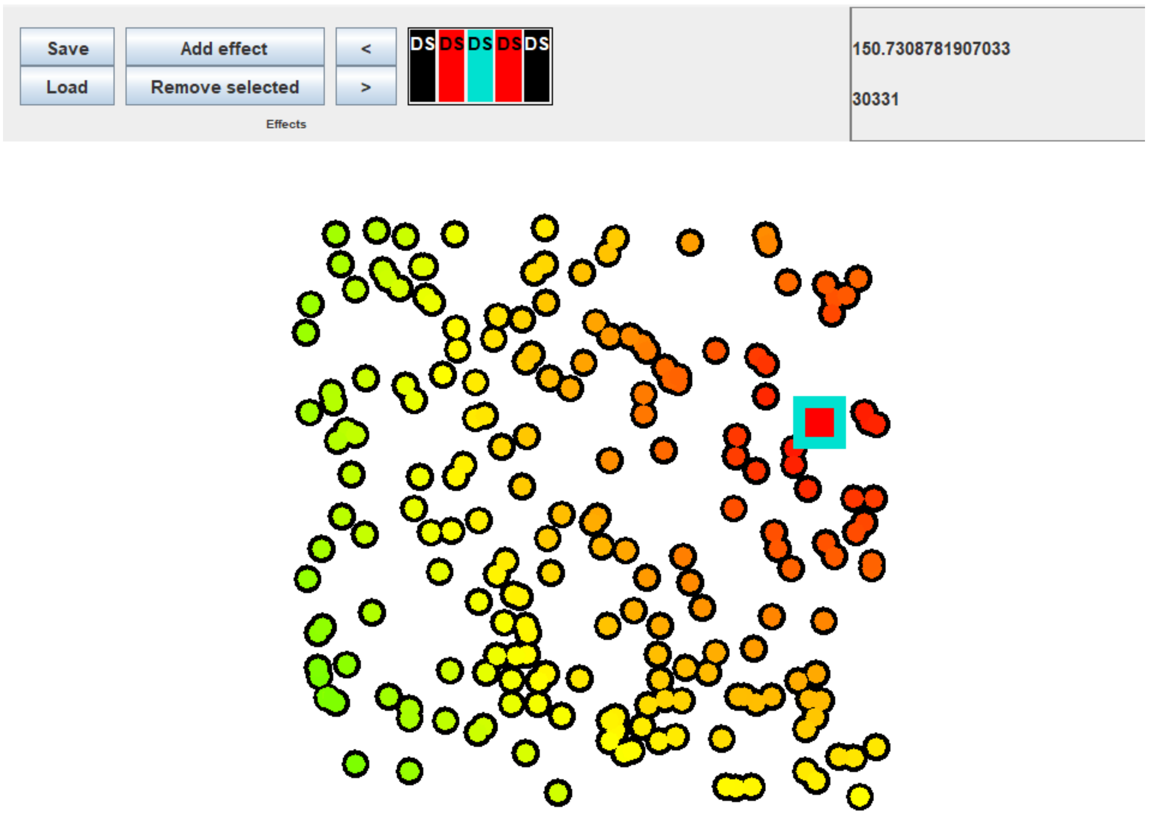


Figure 4.10: Alchemist simulation environment execution of Collektive gradient algorithm

This validation process using Alchemist Simulator shows the potential of Collektive: it is easy to implement algorithms that define the behavior of a system, and the execution is fast and reliable.

Chapter 5

Conclusion and future work

During the work done in this thesis, it was necessary to explore in details of a lot of different aspects.

The foundation is the **aggregate programming** paradigm, which introduced its constructs and the alignment problem.
Regarding the alignment, multiple technological alternatives were discussed, and some of them required to dive into the **metaprogramming**. Specifically, they were analyzed Kotlin annotations and stacktraces, KSP (which stands for Kotlin Symbol Processing) and Kotlin compiler plugins.
As seen in Chapter 3, the final solution involved the creation of a Kotlin compiler plugin, that creates a custom stack that is used to align correctly the different devices. The plugin required a deep comprehension of the compiler and the IR used to represent the syntax tree, which was used to manipulate correctly the different elements.
Since the compiler plugin is developed based on an intermediate representation maintained by Kotlin itself, it is expected to not require drastic changes over time.

Collektive project includes the Gradle plugin, the compiler plugin and the DSL, and it lays the foundation of a new aggregate programming tool.

The DSL created achieved the requirements established before its development, which are:

- **Transparency**: the behavior of the DSL is transparent to the final user, it performs exactly what an aggregate paradigm expert would expect. Moreover, the alignment is resolved without needing the developer to explicitly handle it, by using the Kotlin compiler plugin;

- **Minimality**: the DSL exposes only four functions, which are the necessary feature for the aggregate programming:
    1. **neighboring**: to model device-to-neighbors interactions;
    2. **repeating**: to handle the time evolution;
    3. **sharing**: to observe the neighbors' field, updated the local values and sharing immediately the updated state;
    4. **alignedOn**: to take advantage of the alignment in peculiar situations that the compiler plugin typically does not align;
- **Portability**: since its development is in Kotlin Multiplatform, it is possible to guarantee the compatibility with JVM, JavaScript and Kotlin Native platforms.

The validation process verifies the correct system behavior on the different targets, taking in consideration single computation and executions that involved multiple rounds to prove the correct interaction between devices.
The usage of **Alchemist Simulator** [9] proves that it is possible to create algorithms, such as the gradient, in a simple way, and that the communication between the devices works as predicted.

Different improvements can be done in the future:

1. It is possible to integrate Collektive into **Alchemist** [9], creating an incarnation that allows to create new algorithms just using the simulator;
2. Another possible development is the implementation of a library with self-stabilizing functions, which would capture families of strategies used typically to achieve flexible and resilient decentralized behaviors, in order to hide the complexities by using the field calculus constructs [10];
3. It would be also necessary in the future to publish the project to a package repository like Maven Central, JCenter, or similar, allowing others to easily download and use Collektive as a dependency in their own projects.

Bibliography

[1] Pianini Danilo, Viroli Mirko, and Beal Jacob. "Protelis". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, Apr. 2015. DOI: 10.1145/2695664.2695913. URL: https://doi.org/10.1145/2695664.2695913.

[2] Casadei Roberto et al. "ScaFi: A Scala DSL and Toolkit for Aggregate Programming". In: *SoftwareX* 20 (Dec. 2022), p. 101248. DOI: 10.1016/j.softx.2022.101248. URL: https://doi.org/10.1016/j.softx.2022.101248.

[3] Giorgio Audrito. "FCPP: an efficient and extensible Field Calculus framework". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, Aug. 2020. DOI: 10.1109/acsos49614.2020.00037. URL: https://doi.org/10.1109/acsos49614.2020.00037.

[4] Ion-Alexandru Secara. "Challenges and Considerations in Developing and Architecting Large-Scale Distributed Systems". In: *International Journal of Internet and Distributed Systems* 04.01 (2020), pp. 1–13. DOI: 10.4236/ijids.2020.41001. URL: https://doi.org/10.4236/ijids.2020.41001.

[5] Beal Jacob, Pianini Danilo, and Viroli Mirko. "Aggregate Programming for the Internet of Things". In: *Computer* 48.9 (Sept. 2015), pp. 22–30. DOI: 10.1109/mc.2015.261. URL: https://doi.org/10.1109/mc.2015.261.

[6] Giorgio Audrito et al. "Space-Time Universality of Field Calculus". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 1–20. DOI: 10.1007/978-3-319-92408-3_1. URL: https://doi.org/10.1007/978-3-319-92408-3_1.

[7] Audrito Giorgio et al. “A Higher-Order Calculus of Computational Fields”. In: *ACM Transactions on Computational Logic* 20.1 (Jan. 2019), pp. 1–55. DOI: 10.1145/3285956. URL: https://doi.org/10.1145/3285956.

[8] Giorgio Audrito et al. “Field-based Coordination with the Share Operator”. In: (Sept. 2019), pp. 1–41. DOI: 10.48550/ARXIV.1910.02874. URL: https://arxiv.org/abs/1910.02874.

[9] D Pianini, S Montagna, and M Viroli. “Chemical-oriented simulation of computational systems with ALCHEMIST”. In: *Journal of Simulation* 7.3 (Aug. 2013), pp. 202–215. DOI: 10.1057/jos.2012.27. URL: https://doi.org/10.1057%2Fjos.2012.27.

[10] Mirko Viroli et al. “Engineering Resilient Collective Adaptive Systems by Self-Stabilisation”. In: *ACM Trans. Model. Comput. Simul.* 28.2 (Mar. 2018). ISSN: 1049-3301. DOI: 10.1145/3177774. URL: https://doi.org/10.1145/3177774.