ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Static Analysis of Aggregate Programs through Compiler Plugins

Tesi di laurea in:
Software Process Engineering

Relatore
Danilo Pianini

Candidato
Francesco Magnani

Correlatori
Nicolas Farabegoli
Angela Cortecchia

IV Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Static analysis is crucial in ensuring software quality, detecting potential errors, and enforcing coding standards. However, such tools are often unavailable for novel programming paradigms, limiting their adoption and development. This work explores the application of static analysis in the context of Aggregate Computing, a paradigm for designing and managing distributed systems. Specifically, it focuses on *Collektive*, a Kotlin-based framework that provides an internal Domain Specific Language (DSL) for Aggregate Computing.

To improve the reliability of Collektive programs, a compiler plugin was developed to perform static analysis during compilation. The plugin extends the Kotlin compiler using the new and experimental Frontend Intermediate Representation (FIR) checkers to detect potential issues in aggregate computations, particularly regarding *domain alignment* and misuse of DSL constructs. Various detection patterns were identified and implemented, using techniques ranging from direct API inspections to tree traversal via the visitor pattern.

The development process, testing methodologies, challenges encountered, and the trade-offs of integrating static analysis within the compilation pipeline are discussed. While the plugin provides useful checks and integrates with development environment, it also highlights the limitations of the approaches explored, suggesting potential directions for further refinement and improvements in static analysis for internal DSL-based frameworks.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Analyzing characteristics of the source code without necessarily building and executing it — i.e., static analysis — is a process that has been studied and implemented in various forms during the last decades. Various tools have been developed to perform such a task (e.g., *Checkstyle*[1] for Java, *Detekt*[2] for Kotlin and also multi-language ones, like *PMD*[3] and many more), each with its own strengths and weaknesses. The need for such a tool is often evident in the software development process, where ensuring the **quality** and **robustness** of software systems is a critical concern. Using code quality analysis techniques is also a powerful mean to avoid situations of "technical debt" [EBO+15], that targets the system quality in maintenance and evolution.

As the system grows in complexity, so does the variety of errors and vulnerabilities that can be detected through tools (e.g., concurrency management issues, error handling, etc.). At the same time, adhering to coding standards helps to avoid both trivial and non-trivial errors. Many tools exist to enforce these coding standards (e.g., *Ktlint*[4] for Kotlin) and to detect violations and, moreover, these are usually among the easiest types of tools to integrate, often already included within Integrated Development Environments (IDEs) [Tho21]. The effectiveness of static analysis tools has been the subject of various studies [LPS+23], evaluat-

[1] `https://checkstyle.org/`
[2] `https://detekt.dev/`
[3] `https://pmd.github.io/`
[4] `https://pinterest.github.io/ktlint/latest/`

ing their detection capabilities, agreement, and precision. Some of these studies revealed a low degree of agreement among the tools and highlighted the need for a better understanding of their actual capabilities. Highly advanced tools have been used also to rewrite code and help with the development of very complex systems, like *Coccinelle* for *collateral evolutions* inside the Linux kernel [PLHM08]. More over, in the last years, the static analysis tools have become more popular and easier to use, becoming protagonists of many Continuous Integration (CI) pipelines [ZSO+17] that automatically performs checks on the entire source code, embracing change and evolution of the software without making it a threat, backed by a solid safety net.

1.1 Static Analysis for novel paradigms

Despite all this, these tools are not always available out-of-the-box, especially in the case of new and experimental language paradigms. Developing useful static analysis tools means having a **deep understanding of the programming language** used to write the code, as well as **knowing the paradigm well enough** in order to reason about the main pitfalls and points of failure present within it. Beyond this, static analysis tools are inherently difficult to develop, as they must be exceptionally reliable and robust. They serve as critical foundations for software development, and they are among the last places where one would want to encounter a bug. Any flaw in these tools can lead to incorrect analysis, misguiding developers and potentially introducing severe issues into a codebase.

A major challenge in static analysis is balancing accuracy while minimizing **false positives** and **false negatives** [Tho21]. A false positive occurs when the tool incorrectly flags a valid piece of code as problematic. This can lead to unnecessary developer frustration, wasted time, and, in extreme cases, a loss of trust in the tool itself. On the other hand, a false negative happens when the tool fails to detect an actual issue, giving developers a false sense of security and allowing critical bugs to slip through undetected: this kind of error is also harder to detect when evaluating and testing the tool. For novel paradigms, where best practices and common pitfalls are still being explored, finding this balance is even more complex, increasing the risk for these kinds of errors when there are no well-established

rules and patterns to guide analysis. Finally, from a technical standpoint, developing such tools also requires building a substantial integration layer with other systems to ensure their usability. For instance, they need to seamlessly integrate with IDEs, code editors, and development pipelines, further increasing the complexity of their implementation.

Even considering all of this, the challenge of developing these tools extends beyond technical complexity. For a static analysis tool to be developed, the target paradigm often needs to generate sufficient interest and gain adoption among developers, otherwise it is not worth the effort of creating it. However, the growth and adoption of a new paradigm are often inhibited by the very absence of these tools, which have become essential in most development contexts. This creates a self-perpetuating cycle: without adequate tooling, a paradigm struggles to gain traction, yet without widespread adoption, there is little incentive to develop the necessary tools. As a result, developers and industry stakeholders tend to fall back on well-established tools with mature and certified ecosystems, reinforcing the dominance of existing paradigms and limiting the emergence of new possibilities. This, in turn, leads to significant *technical debt*, as developers are forced to work within paradigms that may not be the most suitable for the problem at hand.

Knowing how to break this cycle could represent a great incentive to the development of new paradigms and tools.

1.2 The role of Domain Specific Languages

When introducing a new programming paradigm, DSLs are often among the preferred means of implementation. A DSL is a specialized language designed for a specific domain, rather than a general-purpose programming language. Since many new paradigms do not require extensive general-purpose functionality, DSLs often provide a natural and efficient way to express the paradigm's core concepts. One of the key advantages of DSLs is their ability to naturally represent the domain within the code, as they are specifically designed for it. This often results in a syntax that is more intuitive, sometimes even resembling natural language, and encourages a more declarative way of writing code. Additionally, because DSLs

are inherently more restricted in scope than general-purpose languages, developing static analysis tools could focus on a smaller set of rules and patterns, making it easier to ensure their accuracy and reliability. Essentially, their limited expressiveness reduces the complexity of the analysis, making it more feasible to create robust and effective tooling [MHS05].

Recently, many new DSLs have been implemented as *internal* to a host language (e.g., the Kotlin-based DSL for the Gradle build tool[5]). Internal DSLs are *embedded* within a general-purpose language, using the host language's syntax and semantics to define the domain-specific constructs. This approach offers several advantages, including the ability to leverage the host language's ecosystem and tooling, without having to develop a compiler from scratch and manage all the necessary checks. It also makes easier for the tool to be distributed and used by a broader audience, as it can be published as a library for the host language. Finally, internal DSL users can also exploit already existing libraries available for the host language to make their life easier. This approach, however, also comes with its disadvantages, the main one being the limitations in their expressiveness, as the host language's capabilities can **constrain syntax and semantics**. For this reason, developing internal DSLs could suffer from scalability issues when the lack of full control over the language becomes an obstacle to the paradigm reflection in the code. Moreover, implementing custom static checks over the code could be difficult when the host language does not provide the necessary mechanisms to do so (i.e., meta-programming capabilities).

In conclusion, even though internal DSLs can be a solution to the first half of the problem — i.e., the adoption and implementation of the paradigm with less technical challenges — they still suffer from the lack of proper tooling for specific static analysis.

[5] `https://docs.gradle.org/current/userguide/kotlin_dsl.html`

1.3 Enabling static analysis through Compiler Plugins

If looked from another perspective, automated static analysis tools and compilers share significant similarities in their operations. Both perform thorough examinations of source code without executing it, aiming to identify errors, enforce coding standards, and optimize performance. Most of what compilers do on the static analysis side is to facilitate code optimization and error detection during the compilation process. Essentially, a compiler can be viewed as a form of static analysis tool, as it analyzes code to generate executable programs and associated debugging information [Tho21]. Specialized static analysis tools, on the other hand, extend beyond the capabilities of standard compilers by offering additional functionalities and broader diagnostic capabilities. They enable detection for specific and uncommon bugs that compilers might overlook but are also, however, distributed as separated software programs, that need to be integrated in some way with the development process.

In the end, compilers offer a possibly more limited range of already included static checks (as the compiler is a necessary tool for building the software) while specialized tools can offer a broader range of checks but need to be integrated externally. What if, however, the compiler were extended beyond its core functionality, incorporating specialized capabilities that go beyond the scope of a general-purpose environment? These extensions could be designed to address the unique requirements of a specific project or domain. This is precisely where **compiler plugins** come into play.

Compiler plugins are dynamic modules that interact with the compiler during its various phases, enabling the introduction of new functionalities or the modification of existing behaviors. They serve as intermediaries that can inspect, modify, or enhance the compilation process, providing developers with the flexibility to implement domain-specific checks, optimizations, or transformations, all without altering the compiler's core architecture. For instance, in the context of the GNU Compiler Collection (GCC), plugins allow for the addition of new features without necessitating modifications to the compiler itself (mostly, again, for optimization purposes).

1.3.1 Types of Compiler Plugins

Compiler plugins can be broadly categorized based on the phase of compilation they target:

- **Frontend Plugins**: These plugins operate during the initial stages of compilation, focusing on tasks such as syntax analysis, semantic analysis, and Intermediate Representation (IR) generation (i.e., an internal data structure used by the compiler). They are useful also for implementing custom syntax extensions, enforcing coding standards, or performing static code analyzes. For example, in the *Rust* programming language, compiler plugins can introduce new syntax extensions and lint checks.

- **Backend Plugins**: Functioning in the latter stages of compilation, backend plugins are concerned with code optimization, machine code generation, and platform-specific adjustments. They can be utilized to implement custom optimizations, support additional hardware architectures and more.

Compiler Plugins in Kotlin

Kotlin, a statically typed programming language developed by JetBrains, offers robust support for compiler plugins, allowing developers to highly customize the compilation process to their specific needs. Kotlin's compiler architecture facilitates the creation of plugins that can modify or extend its behavior during compilation. For example, a popular plugin is the *all-open* one: some frameworks (e.g., *Spring*[6]) require all classes to be `open` (i.e., classes that can have subclasses, the contrary of `final` classes, which are default in Kotlin). Instead of having to manually annotate each class with the `open` keyword, this plugin does it automatically, facilitating this operation.

When guiding developers towards the creation of compiler plugins, JetBrains compares them to **Annotation Processors**[7]. Annotation processors are a powerful feature in many modern programming languages, including Java and Kotlin,

[6] `https://spring.io/projects/spring-framework`

[7] `https://resources.jetbrains.com/storage/products/kotlinconf2018/slides/5_Writing%20Your%20First%20Kotlin%20Compiler%20Plugin.pdf`

that allow developers to generate code, validate code, and perform various compile-time checks based on **annotations** present in the source code. In Java, annotation processors are part of the Java Compiler API and can be used to generate additional source files, validate the correctness of the code, and even modify the Abstract Syntax Tree (AST) of the code being compiled. They are commonly used in frameworks and libraries to reduce boilerplate code and enforce coding standards. Kotlin also supports annotation processors through the Kotlin Annotation Processing Tool (KAPT) — which is, in fact, a compiler plugin itself — that allows Kotlin code to interoperate with Java annotation processors and, more recently, the Kotlin Symbol Processing (KSP), another compiler plugin introduced as "an API that you can use to develop lightweight compiler plugins". The former enables developers to leverage existing Java annotation processors in their Kotlin projects and the latter, on the other hand, provides a more efficient and Kotlin-specific way to generate code at compile time, offering a new approach that is much more integrated with Kotlin symbols.

1.3.2 Advantages of Compiler Plugins

Compiler plugins and Annotation Processors, however, have some very distinct functionalities. While Annotation Processors are limited to generating source code and performing checks based on annotations, compiler plugins can exploit a very powerful API that can create and modify byte-code, elements inside the IR and more, allowing the developers to solve a whole new class of meta-programming problems. In addition, the use of annotations in the code could make it more "cluttered" from the perspective of a DSL, since it would add elements that strictly belong to the host language and that clash with a more natural language-like view. The compiler plugin, instead, would not need these annotations, and could remain clear and untouched. Of course, Annotation Processors are typically easier to write and maintain than compiler plugins, but this extra cost can be worth in several cases, for example in the scenario that will be presented in this thesis.

1.3.3 Main challenges and requirements

At the time of writing, the development of frontend compiler plugins in Kotlin is still a relatively less explored area compared to the backend ones. Due to the limited documentation and examples available, the development of frontend compiler plugins can be a challenging task, that quite often requires inspecting the Kotlin compiler source code directly to understand how to interact with it. Frontend compiler plugins can be implemented using *Extensions* to the Kotlin compiler, a topic which will be explored in more detail in the following chapters.

This thesis presents the development process of a frontend compiler plugin designed to build upon an existing *backend* plugin, built for enhancing the capabilities of an internal DSL. The primary purpose of this frontend plugin is to perform static checks on the source code, ensuring compliance with specific rules related to the functionality of the pre-existing target framework. To better understand the target rules developed within this frontend plugin and its context, it is necessary to first introduce the backend plugin and the project it is part of.

Structure of the Thesis This thesis follows the development process of a frontend compiler plugin from its initial steps, addressing the key challenges encountered as well as the solutions proposed during the research and implementation phases. The next chapter, chapter 2, provides an overview of the ongoing project for which this frontend plugin is being developed, alongside the technical background necessary to understand how a plugin can interact with the Kotlin compiler. Chapter 3 delves into the core development process of the plugin, discussing the design decisions, alternative approaches considered, and the final implementation of the proposed **checkers**. Subsequently, chapter 4 evaluates the plugin's behavior, with a particular focus on the testing methodology adopted, including the integration of a custom testing framework named *Subjekt*, developed specifically for this purpose. Lastly, chapter 5 summarizes the primary contributions of this thesis and outlines potential directions for future work, building on the results and insights gained throughout this research.

Chapter 2

Background: the Collektive case

As mentioned earlier, the development of the frontend compiler plugin presented in this thesis will be built on top of an existing backend plugin. This one is part of a larger project named *Collektive*, a Kotlin multiplatform framework that provides an *internal* DSL for the **Aggregate Computing** [BPV15] paradigm. This chapter will provide an overview of the concepts behind Collektive as well as the features behind its backend plugin and Kotlin compiler plugins development in general.

2.1 Aggregate Computing: a novel paradigm

The Aggregate Computing (AC) paradigm is a novel *macro-programming* approach that enables **collective behaviors** within a heterogeneous set of devices, inside an *adaptive* Internet of Thing (IoT) system [BPV15]. The paradigm shifts the focus from individual devices to regions of devices, abstracting away the details of their number, position, and behavior. This abstraction enables developers to reason about distributed systems in terms of *collective* operations over computational fields, rather than device-to-device interactions.

The foundation of Aggregate Programming is built on **field calculus** [BPV15], a set of constructs that enable manipulations of data structures named *Fields*, which map points in space-time. These constructs enable the implementation of robust coordination mechanisms that are *self-stabilizing*, meaning they can adapt to

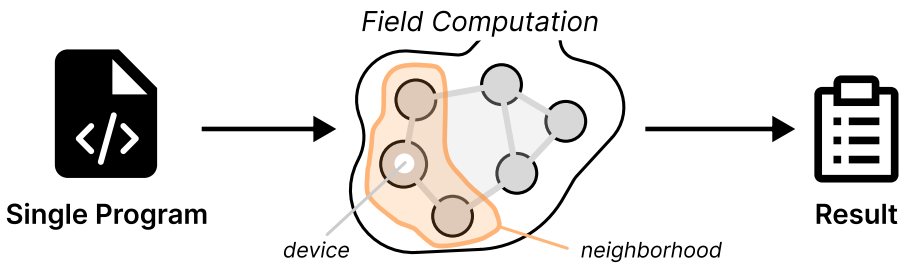

Figure 2.1: Execution model of an Aggregate Computing system.

changes in the environment and input values, gaining scalability to large networks and preserving resilience properties [BPV15] [VAB+18] [VBD+19]. Applications of aggregate programming are particularly impactful in large-scale scenarios, such as crowd management during public events, where distributed devices coordinate to provide services like crowd density estimation, dispersal advice, and navigation support.

Aggregate computing formalism has been proposed in various ways, introducing syntaxes and semantics to support distributed, collective behaviors in dynamic systems. Tools like Protelis [PBV17], ScaFi [CV16] and FCPP [Aud20] extend field calculus principles, providing programming frameworks and language constructs to bridge the gap between theoretical models and practical implementations.

2.1.1 Applications and critical aspects

In Aggregate Computing, the main model of the system consists of a *network of intercommunicating devices*: each device can be *close* to one another therefore introducing a concept of **neighborhood** of devices. Equipped with *sensors* and *actuators*, they can interact with an environment and communicate with other devices through a *message-passing* system.

A key aspect of AC regards the execution model, which is based on a local program identical for all devices. The system is governed by a continuously executed loop that makes the devices (1) *receive messages*, (2) *produce a result through a round of execution of the aggregate program* and finally (3) *send values to neigh-*

bors. This structure allows for the system to show collective behaviors emerging from the network of devices, and contemplates even complex interactions of the devices with the environment through their sensors and actuators.

Finally, as already said, the *field calculus* model can be implemented through an ad-hoc API and syntax to perform operations on a *computational field* (i.e., a mapping from device locations to values, in our example) manipulating it over time, defining interactions between devices and finally creating the emergent behavior.

2.1.2 The Domain Alignment problem

In AC, distributed devices execute the same program and interact with each other to compute a collective result. To do so, data is exchanged during its execution and a *field* of values is created, mapping the devices' locations to values in time. In aggregate computations, devices can "observe" these values in *neighbors* as they are being computed, therefore mapping the other devices to the result of an *intermediate computation*. Since this mapping evolves over time, devices must be able to observe only a *specific portion* of it, corresponding to neighbor devices that have executed the same intermediate program and are, therefore, called *domain-coherent*. This necessary restriction is called **domain alignment**, and represents a main challenge in Aggregate Computing, faced in several studies [DVPB15] [ADVC16].

In other words, domain alignment is a necessary step to ensure that when a device computes a value that depends on neighbor devices (e.g., through particular Aggregate constructs), those devices have computed the *same expression in the same evaluation round*. This guarantees that shared computations remain consistent across devices: it is necessary in order to **maintain consistency between field values** and **prevent information leakage**. Without domain alignment, devices may execute the same function in different rounds, making it impossible to refer to the same part of the aggregate program, which is the same for all devices, and therefore referring to different "domains" during the execution.

The alignment problem has been approached with several strategies [ADVC16] for its run-time management, each one with different degrees of tolerance on the domain of the devices. Even though these differences will not be explored in details in this thesis, it is important to note that **domain alignment cannot be guar-**

anteed in all situations. In some Aggregate programs, domain alignment could fail due to the impossibility for the devices to align on some specific computation, resulting in inconsistencies. As proven in [ADVC16], the problem of *statically* determining whether an aggregate program guarantees domain alignment is in general undecidable, and many implementations disallow certain expressions that could lead to domain misalignment only via run-time checks.

In the following chapters of this thesis, the domain alignment problem will be examined during the static analysis of Aggregate Computing programs, written in a specific AC framework. Before proceeding, however, it is necessary to introduce the tool in question, that will be the target of the frontend plugin in this thesis: *Collektive*.

2.2 Collektive: an Aggregate Computing framework

Collektive is a modern Aggregate Computing framework developed in Kotlin that allows developers to easily write Aggregate Computing programs (also called "aggregate programs") through a flexible *internal* DSL. The framework is designed to be multiplatform, targeting JVM, JavaScript and native platforms. Compared to some other existing AC frameworks, Collektive offers a modern and idiomatic approach to writing aggregate programs, with a static type system (as it is internal to Kotlin) and few, expressive constructs like `neighboring` and `exchange`, which can be used to implement a broad variety of interactions between devices (and also Aggregate Computing patterns).

The project is organized in modules, with the main one being the **dsl** and **compiler-plugin**, and also provides an integration for Alchemist [PMV13], a simulator for pervasive, aggregate, and nature-inspired computing.

2.2.1 Collektive DSL: main concepts

In order to understand how to work on Collektive programs, we first need to understand its main usage. Note for the reader: during this explanation and in the rest of the document, a sufficient knowledge of the Kotlin programming

Listing 2.1: Example of a Collektive program using the DSL.

```
aggregate(localId) {
  share(initialValue) { field ->
    field.max(field.localValue)
  }
}
```

language is necessary to grasp a large part of concepts exposed, and therefore will be assumed in many parts of the elaboration.

Collektive DSL is centered around the `aggregate` function, which is the entry point for the aggregate program. The function uses a local ID to identify the device on which is executed on and then accepts a function as parameter that performs the aggregate computation using the `Aggregate` interface members. This interface provides several functions that represent the main constructs of Collektive's implementation of the AC paradigm. The most important ones are:

- `exchange`: is the base construct that can be used to implement the behavior of the other constructs. This operator models an *anisotropic communication* with neighbors — i.e., different information is sent to different neighbors;
- `share`: models the space-time evolution of the field with an *isotropic communication* — i.e., information is sent uniformly in all directions;
- `neighboring`: observes expressions on neighbors, returning the related field;
- `evolve`: updates an initial value iteratively computing an expression at each device.

Some of these constructs also have a variant that allows the program to return a value of a different type than the one *sent* to the neighbors; the name of the variant is the same of the original function but with the suffix `-ing` (e.g., `exchanging`, `evolving` etc.). In listing 2.1 is shown an example of an aggregate program using the Collektive DSL: it starts identifying each device using a `localId` and then shares the maximum value of the field with neighbors starting from an `initialValue`.

Just by using these four constructs, already complex collective behaviors can be achieved. Besides that, the Collektive DSL hides several operations through the already mentioned compiler plugin.

2.2.2 Collektive Compiler Plugin

As previously introduced, Collektive provides an already integrated **backend compiler plugin**. This plugin is responsible for managing the already introduced *domain alignment* inside aggregate programs — i.e., the alignment of *devices* that execute aggregate programs. Essentially, Collektive *backend* compiler plugins works by analyzing call sites and function definitions in the code, intercepting the ones that involve aggregate computation and that should, therefore, be "aligned". Essentially, this is done by looking at `Aggregate` interface's usage in functions, especially when used as receiver, and visiting their declarations, wrapping aggregate constructs usages with special `align` and `dealign` functions that perform domain alignment under the hood. As it will be shown in section 3.5.1 of the next chapter, this is not always automatic or safe in certain situations (e.g., loops), and the frontend extension will take care of these extra checks.

But how do Kotlin compiler plugins work in general? Before diving into the core development of the frontend plugin, it is necessary to understand their main structure and how they interact with the Kotlin compiler.

2.3 Kotlin Compiler Plugins: general structure

Since Kotlin is a multiplatform language, the same source code can be compiled into low-level code specific to different targets, such as the JVM, JavaScript, and native platforms. In order to work with different targets, the Kotlin compiler architecture is divided into two sub-parts: the *frontend* and the *backend*[1]. The frontend is independent of the target, and for this reason the result of its part of the pipeline — i.e., the frontend IR — can be reused when targeting different

[1]The following explanation is greatly inspired from the work of Marcin Moskala in his book "Advanced Kotlin (Kotlin for Developers)", which provides a more comprehensive overview of the Kotlin compiler plugins architecture.

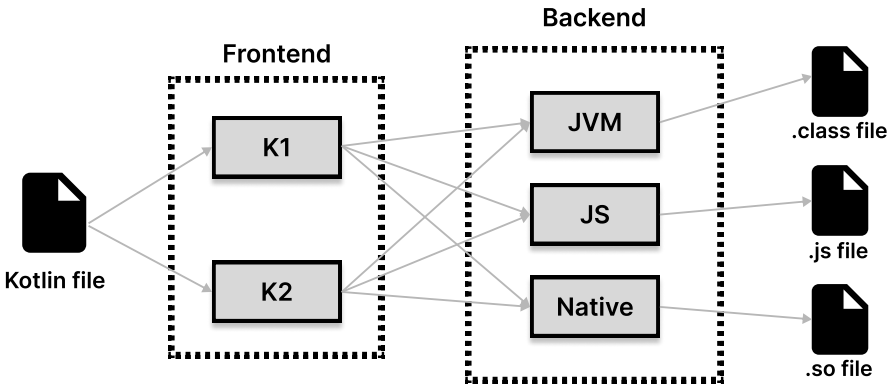

Figure 2.2: Kotlin compiler architecture.

platforms. Starting from the version 2 of the Kotlin Compiler, the frontend has been upgraded to a new structure called K2, which is supposed to be much more efficient than the previous K1 version. The backend, on the other hand, is mostly specific to the target platform, and uses the output of the frontend to generate the final code. In reality, the backends for JVM, JS and Native share some parts that will be analyzed later. The general structure is summarized in fig. 2.2.

2.3.1 Kotlin K2 and frontend plugins

The frontend's output is not only used by the backend for the final one, but it is also responsible for communicating with IDEs and build tools, providing APIs to present errors, warnings, code completions and so on. To make this architecture modular, the Kotlin compiler provides a set of IRs, that the various steps of the workflow can use to process the preceding steps' output. Both the frontend and the backend creates this data structure, although they are very different. The backend's one is created starting from the output IR of the frontend, while the frontend's one is created from the Kotlin source code. The general workflow is summarized in fig. 2.3.

Before the so-called K2 frontend, the compiler's frontend worked by building the Programming Structure Interface (PSI), a syntactic model of the parsed source code, and the `BindingContext`, that holds semantic information such as types and symbol bindings (represented in fig. 2.3 as a whole). The new K2 frontend, on the other hand, builds the FIR, a more powerful and complete representation of Kotlin

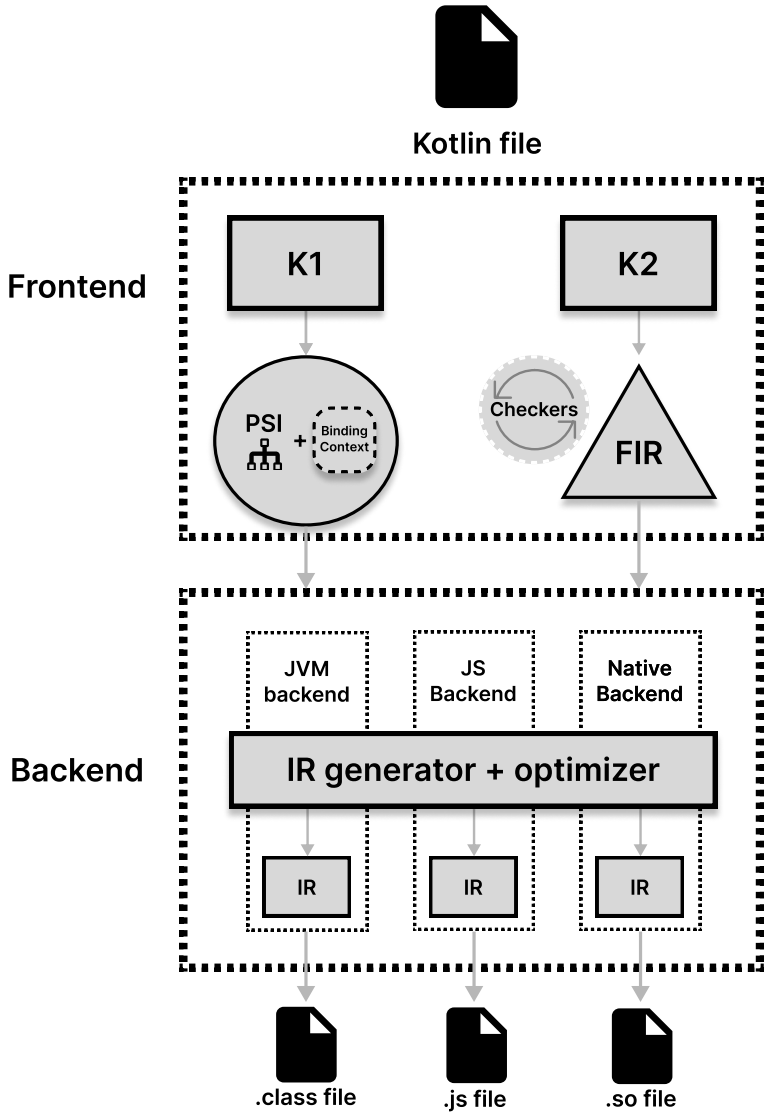

Figure 2.3: Kotlin compiler general workflow.

parsed code, capable of offloading some of the work that was previously done by the backend and enabling powerful optimizations and caching mechanisms. The raw PSI is still being produced, but it is now transformed into the *raw FIR*, which again transforms in different stages, filling the tree with semantic information. Finally, the resolved tree is passed to the backend, which takes care of the platform-specific (backend) IR, used to generate the final code.

As we can see in fig. 2.3, there is still another process that hasn't been mentioned yet: the *checkers*. During their stage of execution, the checkers can inspect the FIR and reports different diagnostics. If some of them is considered "critical", meaning that the compilation should not proceed (e.g., a type error), the compilation is stopped, and the backend is not executed: otherwise, the final IR is passed to it. Checkers play a crucial role in this thesis, and will be the main focus of the frontend plugin development, explored in chapter 3.

2.4 DSL and Compiler Plugins

In relation to what stated in section 1.2, the feature that is missing in most *internal* DSLs is the ability to perform static analysis on the code that is written, and this is particularly difficult since the developers struggle to intervene in the compilation process of the host language. Compiler plugins seem perfect for this task since, in the case of Kotlin compiler plugins at least, they can greatly influence the compilation process on the frontend side and take care of the static analysis of the code. Collektive is in fact internal to Kotlin and its main pitfalls — and the ones of Aggregate Computing in general — which are currently not captured can be spotted by a frontend plugin made for the job.

In other words, the frontend plugin can take care of the "second half of the problem" introduced in section 1.2, that is the *static analysis* of the code that is written in the internal DSL, made specifically for approaching new paradigms with less technical challenges.

2.4.1 The importance of Build Tools

Considering all of this, it is important to mention the role of build tools in the development process of a compiler plugin. Without a seamless integration of the compiler plugin during the compilation process (and, as it will be shown, the testing process as well), the development of a compiler plugin would be cumbersome and error-prone due to integration steps that are not automated. A well-configured build tool appeared as a fundamental requirement in the context of this thesis.

The tool used for the development of the frontend plugin is *Gradle*[2], a build automation tool that is used for Kotlin projects and that provides a plugin system that can be used to extend the build process.

2.4.2 Main motivations

Before diving into the development of the frontend plugin, let's briefly summarize the motivations behind the project. As previously mentioned, Aggregate Computing is still a relatively new paradigm, and some studies [ADVC16] already pointed out subtle bugs that can occur when using some of its possible implementations. Collektive is not an exception and, in this particular case, developing a frontend compiler plugin is motivated by three main reasons:

1. **The backend plugin**: since a backend plugin is already present and necessary for the correct functioning of the framework, the frontend plugin can be simply built on top of it, extending its functionalities and ensuring that the user's operations are safe and within the scope of Collektive DSL correct usage principles.

2. **The presence of an internal DSL**: when building a DSL, a developer must choose to make it *internal* to a host language, like Kotlin, or *external*, like Protelis [PBV17], and so take care of the parsing and the compilation process in general. As already said, internal DSLs are typically easier to use and to develop, but they can also be less flexible and more error-prone, because the developer does not have full control over the parsing process and therefore cannot always enforce the constraints that the DSL should have.

[2] `https://gradle.org/`

This is the case with Collektive, and the frontend plugin can help in this regard.

3. **The integration with the other development tools**: developing a frontend plugin integrates very well with the other tools that are already used in most projects, for example:

   - **IDE**: since the IntelliJ IDEA IDE is the most indicated for Kotlin development and it has a very good support for the Kotlin compiler, the frontend plugin can leverage this support to provide real-time feedback for diagnostics and errors it produces, without the need of developing specific IDE plugins.
   - **Build tools**: the Gradle plugin that is used to apply the compiler plugin requires low effort to be developed and needs almost zero modifications when extending the compiler plugin, differently from what would happen in the case of Gradle plugins wrapping external tools (e.g. *Ktlint*, one of the Kotlin linters available, needs a Gradle plugin to be maintained along with the main tool). This means that it can be easily maintained and integrated into other Gradle projects with low effort.
   - **Rest of the pipeline**: the frontend plugin can be developed very closely to the rest of the Collektive framework through submodules, adding dependencies between subprojects in a very clean and maintainable way.

Considering these motivations, we can now proceed with the actual development in the next chapter.

Chapter 3

Frontend plugin development

In this chapter we will present the development process that lead to the creation of the frontend plugin inside the Collektive project.

Structure of this chapter In this chapter, we will first present the general structure of Kotlin K2 checkers as well as the architecture of the static analyzer developed within the plugin, then we will proceed presenting several **patterns** that were detected observing the Collektive DSL codebase. These patterns represent bad or inappropriate use cases of the Collektive DSL that are **not** captured by default. The way these patterns will be presented does not follow the chronological order of their development, but rather an order with an increasing level of complexity. For this purpose, the patterns will be categorized into four main groups, reflecting the design decisions that were taken to approach the related problems, highlighting the pros and cons of each approach.

Note: the features of the Kotlin compiler explained and shown in this chapter are *experimental*, and therefore subject to possible instability and frequent changes. The code snippets and examples provided are based on the current state of the Kotlin compiler at the time of writing.

3.1 Interaction with the Kotlin compiler

As already introduced in section 2.3.1, the Kotlin K2 frontend performs static checks using *checkers* that inspect the FIR and report diagnostics. To interact with this system, the developer can create **extensions**.

3.1.1 Extension mechanism

Kotlin compiler extensions are mechanisms that allow developers to modify various phases of the compilation process, either by analyzing and transforming code at the FIR level or by modifying the backend IR. These extensions enable advanced features such as additional type checks, automated code generation, and optimizations, empowering Kotlin's extensibility. Like with compiler stages, extensions can be either frontend or backend as well: while frontend extensions impact code analysis, syntax resolution, and IDE support, backend extensions act right after the *IR generation + optimization* phase seen in fig. 2.3 of the previous chapter. This distinction makes frontend extensions more suitable for language-level changes and linting rules, whereas backend extensions are primarily used for performance optimizations and bytecode transformations.

K2 introduces multiple frontend extensions, all following a specific naming convention: `Fir[Name]Extension`. In relation to what was previously discussed about checkers, the `FirAdditionalCheckersExtension` is particularly important in this context, since it is a frontend extension that allows developers to register *additional checkers* to run during compilation. These checkers can enforce custom coding rules, report warnings, or even prevent compilation by issuing errors. Moreover, errors and warnings generated by this extension appear in IDEs like IntelliJ IDEA, improving real-time code feedback. This extension is used by Kotlin plugins like *Kotlin Serialization*, which ensures that serialization-related constraints are properly followed, and *Arrow Meta*, which enforces functional programming best practices.

The extension mechanism is also used in the backend, where backend plugins can intervene, but the extension that needs to be used in only one: `Ir Generation Extension`. This extension is invoked after the FIR phase has completed and the

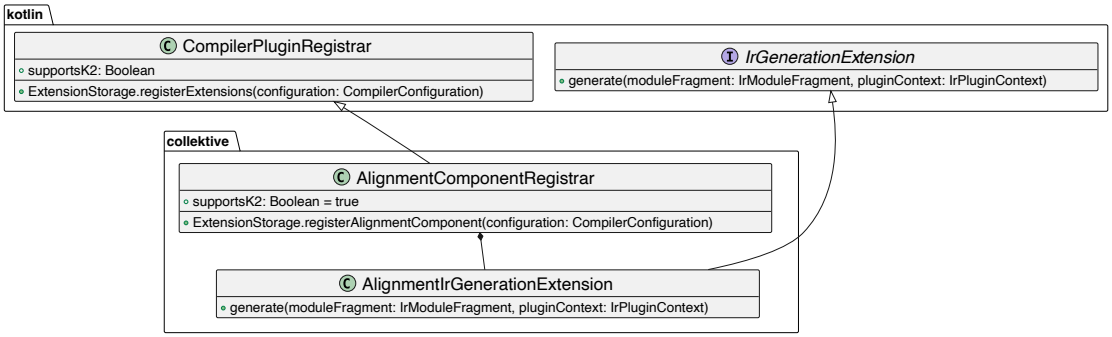

Figure 3.1: Class diagram representing the top-level structure of the Collektive backend compiler plugin. Note: the frontend plugin is not included yet.

IR has been generated. It allows modifications to the IR tree before it is used for bytecode generation. Because IR sits between the frontend and the platform-specific backend, changes made here can impact the generated machine code without altering the high-level source representation — and, for this reason, it does not influence code analysis in IDEs like IntelliJ IDEA. This extension is widely used in performance-critical applications. For instance, *Jetpack Compose* leverages it to transform composable functions into an optimized internal representation. Similarly, *Kotlin Serialization* uses it to generate serialization methods dynamically, ensuring they are both efficient and lightweight. However, modifying IR directly can introduce breaking changes if not handled carefully. Since IR transformations occur at a low level, even minor alterations can lead to unintended consequences, making this extension a powerful but complex tool.

IR generation inside Collektive

Collektive backend plugin uses the `IrGenerationExtension` to perform the operations we already discussed. To register an extension, the developer needs to declare a class that extends `CompilerPluginRegistrar`. Inside that, the developer can register the extension using the `registerExtension` method that depends on the type of extension that is being registered. The core logic of the plugin is then implemented in the extension itself. In the case of the Collektive backend plugin, the class structure is summarized in fig. 3.1.

Finally, to include the plugin, a `META-INF/services` file is needed, where the

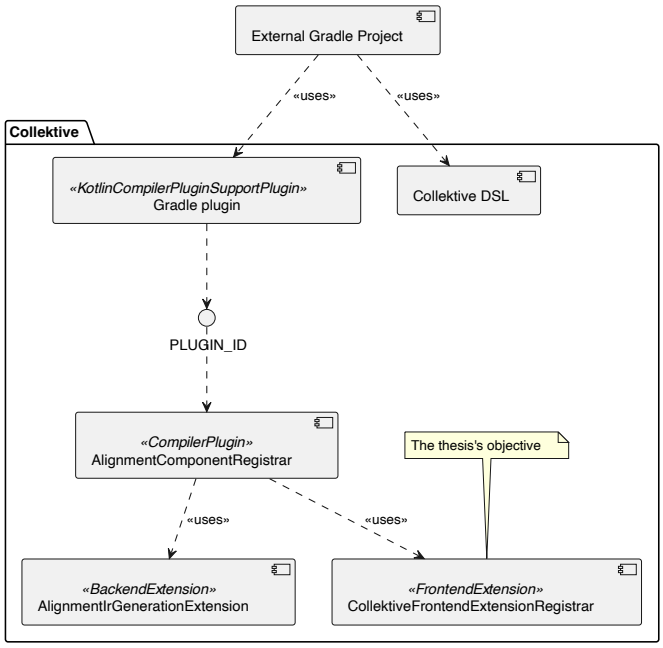

Figure 3.2: Components of the Collektive project after the application of the frontend plugin

fully qualified name of the `AlignmentComponentRegistrar` is written, and the plugin is wrapped in a Gradle plugin that can be applied to the project build[1]. After the application of the frontend plugin also, the structure of the project's components will be the one described in fig. 3.2.

3.2 Static analyzer architecture

The frontend plugin that will be added to the Collektive project represents the static analyzer for the Collektive DSL, target of this thesis. To detect patterns in the code, this plugin needs the already cited checkers, the main components of the static analysis. The architecture of this static analyzer is therefore obtained through a *bottom-up* approach, operating on the compiler internals in order to add new rules to the static analysis. Rules can target different *type of elements* of the

[1]Compiler plugins are, at the time of writing, supported only through the Gradle build system. For this and other reasons that regard the state of the Collektive project, future comments on the build environment will take the use of this tool for granted.

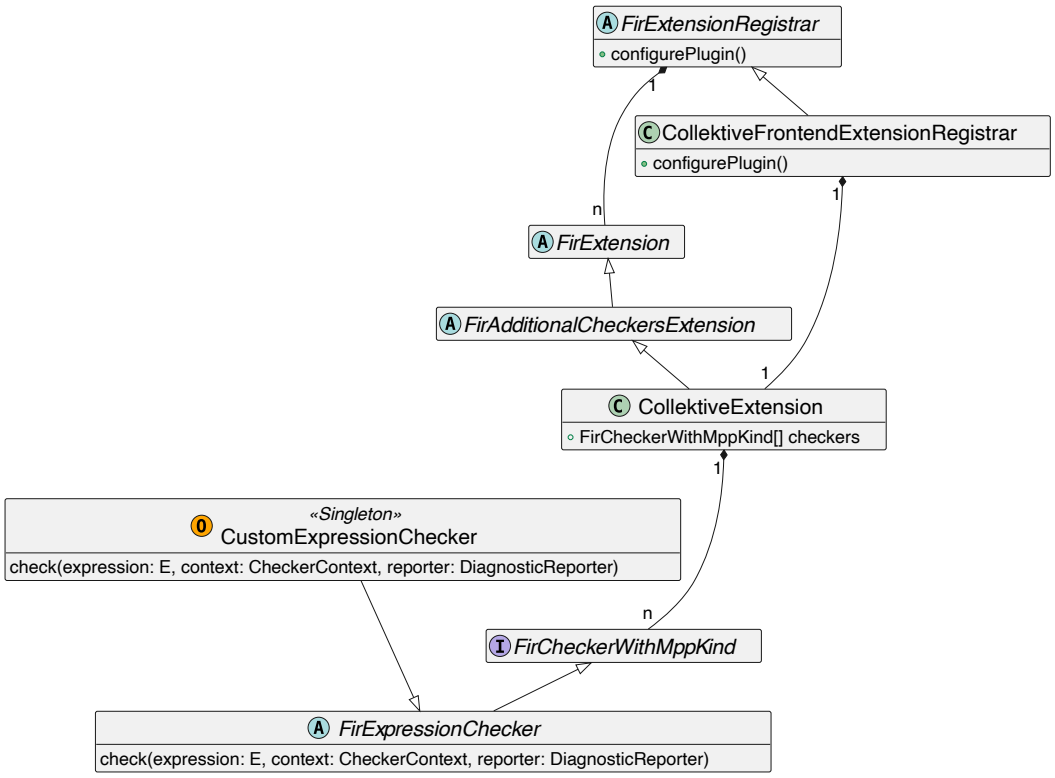

Figure 3.3: Class diagram summarizing the structure of the frontend plugin

FIR tree, for example function calls or property declarations. The targeted type defines the type of the built checker.

Note: a checker does not necessarily correspond to a single rule, but for the sake of clarity and to embrace Single-Responsibility Principle (SRP) the static analyzer will be built enforcing this one-to-one correspondence.

After defining the `CollektiveFrontendExtensionRegistrar` class that wraps the frontend plugin, already introduce in the previous section, the first step is to register the actual extensions to the compiler using the `configurePlugin` method. For this plugin, the extension that will be used is the already cited `Fir Additional Checkers Extension`, which allows registering additional checkers to run during compilation. This extension can contain an arbitrary number of checkers, each of them assigned to different elements of the FIR tree (e.g., expressions, declarations etc.). All the checkers that will be presented are added through this extension. The final class structure is summarized in fig. 3.3.

3.2.1 Adding new rules

To add a new rule, the developer needs to create a new `object` that extends one of the `FirChecker` types, depending on the type of element that will be targeted. The checker will then be added to the frontend extension by adding the singleton of the checker. The checker's internal behavior is defined by its method `check`, which is called by the compiler when the element is encountered in the FIR tree. The method takes a `FirElement` as parameter, which is the element that is being analyzed, and two more objects, the `CheckerContext` and the `DiagnosticReporter`. The first one is used to provide contextual information to the checker, while the second one is used to report diagnostics to the compiler. Inside the check method, the developer can inspect the element and its contextual information using both the `FirElement` and the `CheckerContext` objects. Finally, in case of a positive detection, the `DiagnosticReporter` will be used to report it.

The checker object is added to the frontend extension inside a *set* of checkers corresponding to its type of inspected element. An example of implementation can be found in the first pattern's checker, shown in listing 3.2.

3.3 Adopted workflow

From this point on, the workflow of the plugin development will be as follows:

1. **Pattern detection**: when a new pattern that needs to be captured by the plugin is detected, several examples of Collektive programs expecting a positive or negative diagnostic (i.e., cases in which the diagnostic should be reported and cases where it should not) are written;

2. **Test arrangement**: the examples previously written are used to create a test suite that will be used as certification of the pattern's correct capture, following a Test-Driven Development (TDD) approach;

3. **Checker creation and implementation**: a new checker is created and added to the frontend extension, implementing the logic that will detect and correctly report positives of the pattern's usages;

4. **Adjustments and identification of corner cases (optional)**: during the development of a checker, certain corner cases may emerge that require handling and should be added as *regression tests*. Once these cases are properly identified, the process loops back to step 2, refining and extending the existing checker accordingly.

In this chapter, we will present these steps for each pattern, except for step 2, which will be covered in chapter 4.

Note: Unless explicitly stated otherwise in the code, the snippets presented in the following sections are assumed to be **inside an Aggregate block**. This means they are either within a function that has the `Aggregate` interface as its receiver or inside an `aggregate` entry point block, as these are the designated contexts where the Collektive DSL can be used to specify aggregate behaviors. This choice has been made to enhance brevity and clarity.

3.4 First approach: direct Kotlin API usage

The API that Kotlin provides to interact with the FIR is sufficiently powerful to be used to perform a wide range of operations and checks. The first approach to the development of the checkers was to use this API directly, without the need of any additional constructs or classes. This approach is the most direct, but it was used only for the pattern with the lowest complexity.

3.4.1 Pattern 1: explicit align/dealign

One of the first pattern detected is the *explicit usage of the* `align` *and* `dealign` *functions* of the Collektive DSL. For how the Collektive DSL is structured, it was not possible to prevent the usage of these functions directly through the API's design, so the frontend plugin was put in charge of this task.

Rationale

These functions are not supposed to be used directly by the developer because they are already managed by the backend plugin, which uses them when inspecting call

Listing 3.1: Example of Pattern 1 detection in code

```
1 align(null) // Pattern detected
2
3 dealign() // Pattern detected
4
5 otherAggregateMethod() // Pattern not detected
```

sites of the functions that need alignment, wrapping these calls with the alignment constructs. The direct usage of these functions can lead to inconsistent behavior and unexpected results, since it might also interfere with the already working backend plugin, so the frontend part should prevent the presence of these calls in the code.

Pattern detection

Given two functions `align` and `dealign` available in the `Aggregate` interface, the pattern is satisfied when any of these functions are used explicitly in the code. Listing 3.1 shows an example of this pattern.

Design and Implementation

One of the type of checkers available among FIR checkers is the `Fir Function Call Checker`, which is just a type alias for a `FirExpressionChecker` — that is, a checker that takes care of expressions in the code — typed with a `FirFunctionCall` — i.e., an element of the FIR tree representing a function call in the code. This type of checker can inspect function calls usages, calling the `check` method of the checker with a `FirFunctionCall` as parameter. To performs checks on the function call, we can inspect the properties of this parameter. In this case we only need to **compare the fully qualified name** of the function with the one of the interested functions.

Getting the fully qualified name of the function is not directly supported by the FIR API, but it is possible to build a small utility function that retrieves it (whose implementation is omitted). The final implementation of the checker is shown in listing 3.2: some portions of the code are omitted for brevity.

The `reportOn` method of the `reporter` object is used to report a diagnostic

Listing 3.2: Implementation of the Pattern 1 checker: `ExplicitAlignDealign`

```
object ExplicitAlignDealign : FirFunctionCallChecker(MppCheckerKind.Common) {
    override fun check(
        expression: FirFunctionCall,
        context: CheckerContext,
        reporter: DiagnosticReporter,
    ) {
        val fqnCalleeName = expression.fqName()
        if (fqnCalleeName in FORBIDDEN_FUNCTIONS) {
            reporter.reportOn(
                expression.calleeReference.source,
                FirCollektiveErrors.FORBIDDEN_FUNCTION_CALL,
                fqnCalleeName,
                context,
            )
        }
    }
  // ...
}
```

when the pattern gets detected. The parameter passed to this method determine where the diagnostic is going to be reported (i.e., the position in the code) and the message shown to the user (in these checkers, specified inside an object called `FirCollektiveErrors`). It is possible to see that the checker is quite general, as it can be used to detect the presence of any function call with a specific name.

3.5 Second approach: declarative and modular API

Even though the API available in the checker is flexible and feature-rich, it can appear quite verbose if we need to implement utilities even for relatively simple operations like obtaining the fully qualified name of a function. To simplify the development of the checkers and make them more modular, a small API was developed to make declarative checks on the FIR tree, implemented during Pattern 2 checker development.

3.5.1 Pattern 2: simple aggregate operations in loops

The second pattern that was inspected is the usage of *aggregate operations inside loops*. This pattern is more complex because it reflects cases that are not always

inappropriate usages, but if not properly handled can lead to problems during the *alignment* phase.

Rationale

As presented in section 2.1.2, the domain alignment inside an AC program cannot be statically guaranteed in all cases. There are, however, particular constructs that can lead to misalignment, and in the case of Collektive programs, since it is internal to Kotlin, an example of these constructs are **loops**. When an aggregate construct (i.e., a function call that needs alignment between devices) is called inside a loop (e.g. a `for` loop), the alignment of the computation fails because multiple devices are not able to align to the same "instance" of the call between the iterations. However, the alignment can succeed if the loop contains a *custom alignment operation*, that can be done using the `alignedOn` method, another construct of the DSL, that manually performs domain alignment with manual constraints. This method accepts an anonymous function that will be the subject of the alignment. Again, this behavior cannot be captured through the internal DSL alone, so the frontend plugin is needed to handle this case.

It could seem that this pattern is not overly complex to detect compared to the previous one, but in reality just adding one more small constraint makes the range of possible cases that can be captured following this pattern much broader and more varied. Consider the cases presented in listing 3.3. These corner cases are exceptions to the general rule that we previously stated, in fact:

- In the first case, although, technically, the construct is "present" inside the loop, it is not directly called by the loop itself, but by a nested function. Note that even though this can be seen as a corner case, this is valid Kotlin code and might appear, maybe as slightly modified versions of this one, in real use cases;

- In the second case, we have the required `alignedOn` operation wrapping the Aggregate function call, but it is placed outside the loop, making the alignment operation useless (since the multiple calls inside the loop will still not be aligned);

Listing 3.3: Corner case related to Pattern 2, where the construct is used inside a nested function

```
// Nested function
for(/* loop condition */) {
  fun Aggregate<Int>.nested() {
    neighboring({ 2 * 2 })
  }
}

// AlignedOn outside the loop
alignedOn(/* ... */) {
  for(/* loop condition */) {
    neighboring({ 2 * 2 })
  }
}

// Loop outside the `aggregate` block
for (/* loop condition */) {
  aggregate {
    neighboring({ 2 * 2 })
  }
}
```

- In the third case, the loop is done without alignment, but this is in fact correct because the construct is not iterated by the loop since a new aggregate instance is created at each iteration. This makes each of them like a separate Aggregate program, and the alignment is not necessary.

Many more cases regarding this pattern can be found, but some of these will be presented as another, separated pattern in section 3.7.1, both because it was treated in a separated moment during the development and since the growth in complexity is not negligible. For now, we will deal with simpler cases, described in the following, more precise description.

Pattern detection

Given a construct that loops through various iterations (e.g., a `for` statement, an anonymous function called when cycling collection's elements, like `map`, `forEach` etc.) *inside* an "aggregate" block, the pattern is satisfied when an aggregate construct is used *inside the loop body* satisfying both the following conditions:

1. There is no `alignedOn` operation that wraps the construct inside the loop;

2. The construct is **not** part of a nested static declaration (e.g., a nested function) whose use is not iterated by the loop.

Design and Implementation

The description might still seem incomplete or vague to some extent, and the reason is that some potential cases are still not captured. For now, though, the developed checker will be able to capture the most common cases, and the rest, as we will see, will be integrated when covering the Pattern 5.

One possible way to approach this pattern is reasoning in terms of **containing blocks**: starting from the aggregate function called, we could be able to determine if the pattern is detected or not just by looking at the **sorted list of containing blocks of the function call**. Starting from there, we first inspect if a loop is, in fact, containing the function call, and if so, we check for other elements, like the presence of the `alignedOn`, in the correct order. This procedure is summarized in fig. 3.4.

As we can see from the diagram, the procedure ends immediately if the function name is one of `alignedOn`, `align` or `dealign`, since these are functions that do not require alignment. Thanks to Kotlin nullability system, we can perform these subsequent checks in a declarative way, adding utilities to the API that we previously introduced and making it more extensible for other checks like this.

This API is composed of a set of functions with receiver that can be used to perform common operations on the FIR elements, for example to check if a Checker context is **inside an Aggregate function or block**. Some of these utilities are shown in listing 3.4.

As shown, we can perform checks on the FIR tree using built-in methods like `containingElements`: we can use this method to check if a `FirElement` is contained inside a function that has a *receiver parameter* of a certain type, in this case the Aggregate one, by inspecting the name of the *Class-like Symbol* (i.e., the symbol referred to a class or similar entities like interfaces) to which it is related. Encapsulating this composed operations in functions makes them still reusable and declarative in the context of domain-specific checkers like the ones we are developing. This greatly empowers the Kotlin FIR API, making it more interesting

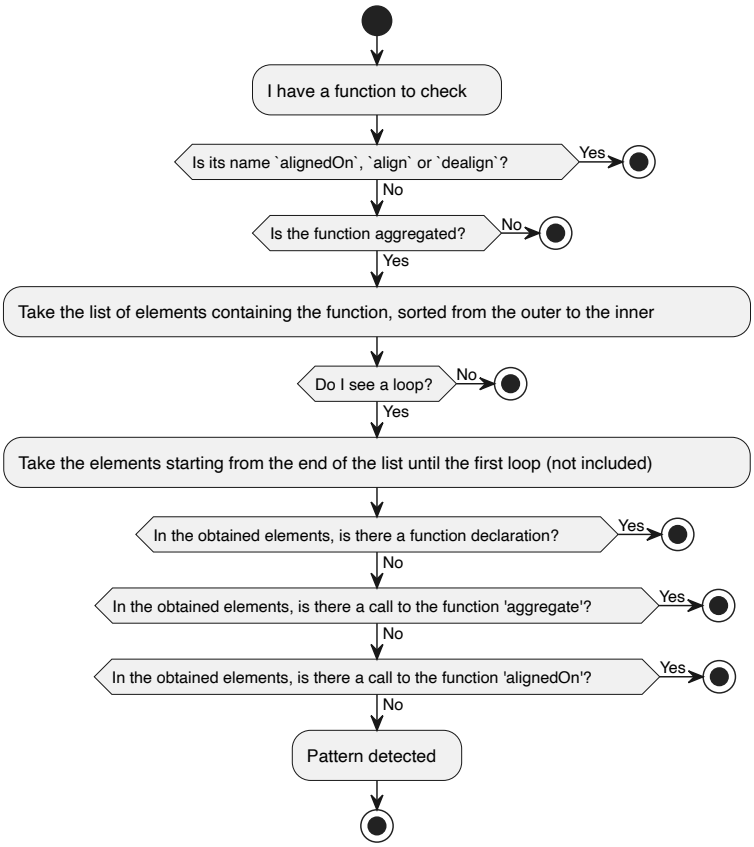

Figure 3.4: Flowchart representing the procedure to detect Pattern 2

Listing 3.4: Some utility functions of the small API developed

```
fun CheckerContext.isInsideAggregateFunction(): Boolean =
    containingElements.any { (it as? FirSimpleFunction)?.receiverParameter?.
        isAggregate(session) == true }

fun FirReceiverParameter.isAggregate(session: FirSession): Boolean =
    typeRef.toClassLikeSymbol(session)?.name?.asString() == AGGREGATE_CLASS_NAME

fun FirFunctionCall.isAggregate(session: FirSession): Boolean {
    val callableSymbol = toResolvedCallableSymbol()
    return callableSymbol?.receiverParameter?.isAggregate(session) == true ||
        callableSymbol?.getContainingClassSymbol()?.name?.asString() ==
            AGGREGATE_CLASS_NAME
}
```

Listing 3.5: Utility functions added to help in the detection of Pattern 2

```
fun CheckerContext.wrappingElementsUntil(
    predicate: (FirElement) -> Boolean,
): List<FirElement>? =
  containingElements
    .takeIf { it.any(predicate) }
    ?.dropLast(1) // the element itself
    ?.takeLastWhile { !predicate(it) }

fun List<FirElement>.discardIfFunctionDeclaration(): List<FirElement>? =
  takeIf { elements -> elements.none { it is FirSimpleFunction } }

fun List<FirElement>.discardIfOutsideAggregateEntryPoint(): List<FirElement>? =
  takeIf { it.none(isFunctionCallsWithName("aggregate")) }

fun isFunctionCallWithName(name: String): ((FirElement) -> Boolean) = {
  it is FirFunctionCall && it.functionName() == name
}
```

also for more complex cases that will be presented later. We first add these small functions shown in listing 3.5 to the API, and then we can finally implement the checker, as shown in listing 3.6. Many portions of the code are omitted for brevity.

The `wrappingElementsUntil` function is used to get the list of elements that wrap a certain element until a certain condition is met. This is used to get the list of containing blocks of the function call, and then check if the pattern is satisfied only by looking at this list. The other functions are used to provide conditions to the previous function, "discarding" the result — i.e., returning null — if the condition is not met. The result shown in the checker of listing 3.6 shows how this API can be used to chain these conditions in a declarative way, leveragin Kotlin's nullability system.

As we can already see, implementing a checker with this approach has several advantages: the code is more readable and modular, and the logic is sufficiently declarative and easier to understand. This approach is also extensible and can be adapted to other patterns that require similar checks. This is, however, a very particular case that can be caught only by looking at containing elements. Once the pattern becomes more complex, adding concepts like *symbol's usage references* or *structured operations with specific requirements*, for example nested anonymous calls or subsequent statements that need to be checked together, this approach quickly becomes less effective and more verbose, as it requires a very effective and carefully designed API that cover as many cases as possible. During the

Listing 3.6: Implementation of the Pattern 2 checker: `NoAlignInsideLoop`

```
object NoAlignInsideLoop : FirFunctionCallChecker(MppCheckerKind.Common) {
  // inside a 'for' or 'while' construct
  private fun CheckerContext.isInsideALoopWithoutAlignedOn(): Boolean =
    wrappingElementsUntil { it is FirWhileLoop }
      ?.discardIfFunctionDeclaration()
      ?.discardIfOutsideAggregateEntryPoint()
      ?.none(isFunctionCallWithName(AggregateFunctionNames.
          ALIGNED_ON_FUNCTION_NAME)) ?: false

  // inside a function like 'forEach' or 'map' of Kotlin Collections
  private fun CheckerContext.isInsideIteratedFunctionWithoutAlignedOn(): Boolean =
    wrappingElementsUntil { it is FirFunctionCall && it.functionName() in
        collectionMembers }
      ?.discardIfFunctionDeclaration()
      ?.discardIfOutsideAggregateEntryPoint()
      ?.none(isFunctionCallWithName(AggregateFunctionNames.
          ALIGNED_ON_FUNCTION_NAME)) ?: false

  // the 'check' method verifies that the function is aggregate and uses these two
      methods...
}
```

development of the rest of the checkers, it appeared natural to cover corner cases and patterns that reflect many possible FIR trees with an approach more suitable to this kind of data structures.

3.6 Third approach: visitor pattern

One of the main design patterns that can be used to traverse a tree-like data structure is the *visitor pattern*. This pattern is particularly useful when the tree structure is complex and the operations that need to be performed on it are varied and not easily encapsulated in a single class. It is no coincidence that many of the available static analysis tools use this kind of approach [LPS+23]. The visitor pattern well fits the context of the Kotlin FIR, since the structure resembles the one of the AST built by the Kotlin parser. Visitors are naturally used during the compilation process even if we don't explicitly define any, since the compiler itself needs to traverse the tree to perform the various operations needed to compile the code. Kotlin provides a set of utilities that can be used to implement Visitors that fit the developer needs and that can be easily integrated into the checkers, in order to explore the FIR structure behind the `FirElements` (i.e., the nodes of the tree)

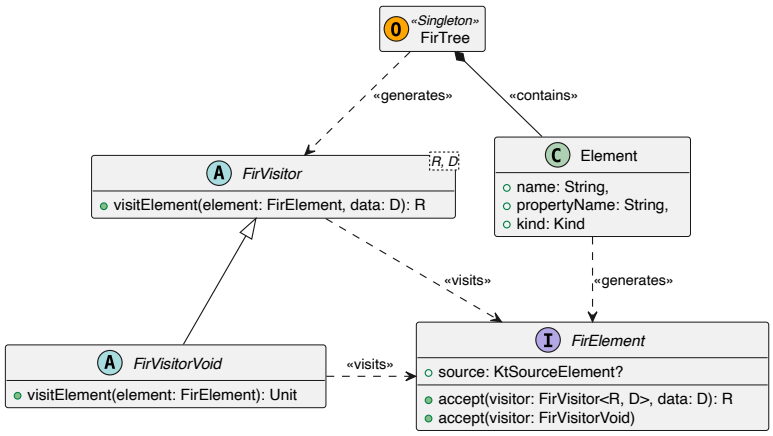

Figure 3.5: Summarized class diagram of the visitors inside the Kotlin compiler

that are passed to the checkers.

Kotlin FIR visitors are automatically generated inside the Kotlin compiler code in order to have a specific method for each type of `FirElement` that can be used to visit that element. All that is left to the developer is to choose which type of visitor to extend in order to implement their own:

- `FirVisitor<R, D>`: a visitor where each visit method accepts a `D` parameter that can be used to pass data between the visit calls, and a return type `R` that can be used to return a result from the visitor;
- `FirVisitorVoid`: a visitor where each visit method accepts only the element to visit and returns nothing. Under the hood is simply a `FirVisitor<Unit, Nothing?>`.

Depending on the specific task assigned to the visitor, the developer will choose one of these two types of visitors and extend it into a new class. Their structure is summarized in the class diagram of fig. 3.5.

3.6.1 Pattern 3: unnecessary Yielding usage

Having introduced the visitor pattern, we can now see it in practice having to deal with the next pattern, whose implementation in the checkers exploits FIR

Listing 3.7: Example of usage of the yielding context in the Collektive DSL, taken from the Collektive documentation

```
// Normal version
exchange(initial = 1) { field ->
  field.map { it + 1 }
}

// Version with yielding context
exchanging(initial = 1) { field ->
  val fieldResult = field.map { it + 1 }
  fieldResult.yielding { fieldResult.map { "return $it" } }
}
```

visitors. This pattern regard the *unnecessary usage of the* `yielding` *contexts* in the Collektive DSL.

Rationale

The `yielding` context is a specific feature inside the Collektive DSL, briefly introduced in section 2.2.1, that allows the developer to call a construct of the DSL, like `exchange` for instance, returning a value different from the result of the construct. To use this feature, instead of using the actual construct, the variant with the `-ing` suffix is used, like `exchanging` in this case: these variants are called **yielding operations**. Essentially, a `YieldingContext<Initial, Return>` is used inside yielding operations to act on an `Initial` value — that can, for example, be exchanged with neighbors — but **return a different value of type** `Return` to the caller, without having to return the same value as for the normal version.

In listing 3.7 we can see an example of the usage of this yielding context, taken from the Collektive documentation. In the example provided, the `exchanging` construct will still perform like the normal one, which sends the results from the evaluation of the provided function to other devices and returns an object of type `Field<Int, Int>`. The entire operation, however, will return another object of type `Field<Int, String>`, resulted from the `map` call inside the yielding action.

As can be observed from listing 3.8, the usage of this feature can be redundant: nothing prevents the developer from using this construct and return the same value as the one been computed and sent to the other devices by the construct, resulting in a useless `yielding` that could be safely substituted with the normal version

Listing 3.8: Example of Pattern 3 detection in code, with an unecessary usage of a yielding context, this time with the `sharing` construct

```
sharing(initial) {
  // ...
  value.yielding { value }
}
```

of the construct. Since it represents an inappropriate usage of these methods, its detection is a task for the static analyzer.

Pattern detection

Given a Collektive *yielding operation* (e.g., `exchanging`, `evolving`, etc.), the pattern is satisfied when the expression returned inside the anonymous function passed as parameter to the `yielding` call is *equivalent* to the one used as its *receiver*, therefore resulting in a redundant usage of the yielding context.

Design and Implementation

As previously introduced, this Pattern's checker will be based on the Visitor pattern. The checker can perform an initial check on the function name to intercept a Collektive yielding operation based on a set of predefined, fully-qualified names. Once one of these constructs is found, the checker will delegate the inspection of the anonymous function to the implemented visitor, that will traverse the small part of the FIR tree that is contained within the construct's parameters (i.e., the anonymous function passed as parameter to the construct) Finally, the visitor will return a boolean value used to report the successful or unsuccessful detection of the pattern.

The implementation of the checker can be seen in listing 3.9. The core part is the visitor method call `containsUnnecessaryYielding`, whose implementation will be briefly discussed in the following.

Visitor implementation

To check if a function call corresponding to a construct with a yielding context matches the pattern, we need to inspect the anonymous function that is passed as

Listing 3.9: Implementation of the Pattern 3 checker: `UnnecessaryYielding`

```
object UnnecessaryYielding : FirFunctionCallChecker(MppCheckerKind.Common) {

  private fun FirFunctionCall.usesAnUnnecessaryYieldingContext(): Boolean =
    with(YieldingUnnecessaryUsageVisitor()) {
      containsUnnecessaryYielding()
    }

  override fun check(
    expression: FirFunctionCall,
    context: CheckerContext,
    reporter: DiagnosticReporter,
  ) {
    if (expression.fqName() in constructs // FQ names of the yielding operations
          && expression.usesAnUnnecessaryYieldingContext()) {
      // report as in the other checkers
    }
  }
}
```

parameter. When the visitor encounters a function call to the `yielding` operation, it will save the **explicit receiver** of the call and then visit the anonymous function that is passed as parameter to `yielding`. When visiting a *return expression* of the anonymous function passed to the yielding construct, the visitor will check if the expression is *equivalent* to the saved receiver of the yielding construct. If so, the visitor will return true, meaning that the pattern is detected, otherwise it will return false.

A behavior like this can be implemented using the `FirVisitorVoid` class: the listing 3.10 summarizes the visitor's implementation.

The function `isStructurallyEquivalentTo` is a utility function that uses Kotlin *expression rendering* to check if two expressions are equivalent. For brevity purposes, its implementation is omitted. The function `containsUnnecessaryYielding` is also omitted, but its implementation consists only of a call to the visiting methods, returning a boolean variable set during the exploration.

Using the visitor approach to this task revealed itself to be more effective and limited in complexity compared to using the FIR API on the elements directly. However, this still represents a relatively simple case, often solved by common static analysis checks for programming languages in similar situations (e.g., same expression on both sides of an assignment). We will now see how to approach more complex cases with the visitor pattern.

Listing 3.10: Implementation of the Pattern 3 Visitor

```
class YieldingUnnecessaryUsageVisitor : FirVisitorVoid() {
  // ...private fields declarations...

  override fun visitFunctionCall(functionCall: FirFunctionCall) {
    if (functionCall.fqName() == YIELDING_FUNCTION_FQ_NAME) {
      insideYielding = true
      yieldingReceiver = functionCall.explicitReceiver // we save the receiver
      functionCall.argumentList.arguments.forEach(::visitElement) // we visit the
          arguments
      insideYielding = false
      return
    }
    super.visitFunctionCall(functionCall)
  }

  override fun visitReturnExpression(returnExpression: FirReturnExpression) {
    if (insideYielding) { // inside the anonymous function passed to yielding
      containsUnnecessaryYielding = returnExpression
        .result
        .isStructurallyEquivalentTo(yieldingReceiver) // we check if the return
            value is the receiver
      return
    }
    super.visitReturnExpression(returnExpression)
  }
  // ...
}
```

3.6.2 Pattern 4: unnecessary construct usage

In order to prove the effectiveness of the visitor pattern in more complex cases, we will now present a more complex pattern that can be detected using this approach. This pattern regards the *unnecessary usage of constructs* in the Collektive DSL.

Rationale

Kotlin compiler won't, by default, warn the developer when an anonymous function that can have one or more parameters is created without using them in the body (and maybe passed to another function as a parameter). For example, in the code shown in the following snippet:

```
listOf(1, 2, 3).map { 5 }
```

The anonymous function passed to the `map` function is not using the `it` parameter and, in general, this cannot always be interpreted as an error because, depending on the function, the parameter might not be needed. Some Collektive

constructs, however, are designed to work with the parameters of these anonymous functions, since their internal actions work based on that: in many cases, the construct would simply appear useless, and the absence of these parameters could be a sign of **misinterpretation of the construct's behavior** and intended usage as well as an unnecessary network exchange that results in a waste of resources. Consider the `evolve` construct, for example: not using its parameter would mean that the evolution is not taking the field state into consideration, depending on some other values that do not regard the aggregate computation, resulting in a bad and unnecessary usage. In other words, the usage of these constructs without using the parameters can always be considered a misuse of the construct, and the frontend plugin should be able to detect it.

Not all constructs behave in the same way: for example, the `neighboring` and `neighboringViaExchange` constructs are different from the rest. Since these two functions evaluate expressions in neighbors devices, if the expression to evaluate is provided as an anonymous function, this does not accept parameters. In this case, the construct usage is considered unnecessary if the anonymous function has an *empty return* — i.e., a return of type `Unit`. An example of both the cases is shown in listing 3.11. Note: the code shows the usage of *explicit* parameters, but the pattern is valid also when the parameters are used *implicitly* (e.g., using the `it` keyword).

Pattern detection

Given a Collektive construct that accepts an anonymous function as parameter, the pattern is satisfied when the body of the anonymous function does not *use* the parameters that its signature declares *or* when the body of the anonymous function returns `Unit` in absence of accepted parameters (i.e., the `neighboring` and `neighboringViaExchange` constructs).

Design and Implementation

Similarly to the previous pattern, the checker will be implemented with very little logic, exploiting more complex visitors to inspect the pattern matching and, for this reason, its implementation is omitted. In this case, the checker will only detect

Listing 3.11: Examples of Pattern 4 detection in code, with both the cases described

```
// example with anonymous function that takes parameter (pattern detected)
evolve(initial) { value ->
    // but `value` is not used inside the body
    10
}

// example with anonymous function that takes parameter (pattern not detected)
evolve(initial) { value ->
    // `value` is used inside the body
    value + 10
}

// example of a parameter-less anonymous function (pattern detected)
neighboring {
  val example = 0
}

// example of a parameter-less anonymous function (pattern not detected)
val example = 10
neighboring { example }
```

if a function call is one of the Collektive constructs to check. If so, it will delegate the inspection on one of two visitors, whether the construct is one that accept anonymous functions with parameters or not.

The two visitors are the `ConstructCallVisitor` and `EmptyReturnVisitor`. The latter's implementation is not particularly surprising compared to the previous visitor (and, therefore, will be explained without code): it behaves like a default visitor, but when visiting a `FirReturnExpression` element, it checks if the return type is `FirUnitExpression`. If so, the pattern is detected.

The second visitor is more complex: when checking for usages of parameters, we **cannot simply check the return expression of the anonymous function** and see if it uses the parameters, because the parameters can be used in other expressions to create **dependent symbols** — i.e., symbols that depend on and can be used instead of the parameters in the return expression. To solve this problem in a simpler way, the visitor will **not** check if the dependent symbols are used in the return expression, therefore implementing a *symbol marking* system (i.e., a program that marks symbols inside expressions depending on their role, in this case dependent symbol), but instead it will simply consider the pattern as detected once it sees at least one usage of the parameter inside the anonymous

Listing 3.12: Implementation of the Pattern 4 Visitor

```
class ConstructCallVisitor : FirVisitorVoid() {
  private var checkedParametersDeclarations = listOf<FirValueParameterSymbol>()
  private val found = true

  override fun visitAnonymousFunctionExpression(anonymousFunctionExpression:
      FirAnonymousFunctionExpression) {
    if (!nestedAnonymousFunction) { // we don't check the parameters of nested
        anonymous functions
      val anonymousFunction = anonymousFunctionExpression.anonymousFunction
      val parameters = anonymousFunction.valueParameters // usage of FIR API
      checkedParametersDeclarations = parameters.map { it.symbol } // we save the
          symbols to check
      if (checkedParametersDeclarations.isEmpty()) {
        found = false // no parameters -> skip check
        return
      }
      nestedAnonymousFunction = true
    }
    super.visitAnonymousFunctionExpression(anonymousFunctionExpression)
  }

  // Visits (resolved) name references in the code (variables usage for example)
  override fun visitResolvedNamedReference(resolvedNamedReference:
      FirResolvedNamedReference) {
    if (resolvedNamedReference.resolvedSymbol in checkedParametersDeclarations) {
      checkedParametersDeclarations = // we filter out the parameter that are used
        checkedParametersDeclarations.filter { it != resolvedNamedReference.
            resolvedSymbol }
      if (checkedParametersDeclarations.isEmpty()) {
        found = false // if all the parameters have been used, we are done
      }
    }
  }
}
```

function body.

The implementation of the `ConstructCallVisitor` is shown in listing 3.12. The variable `found` starts off with the value `true` and will be set to false when the visitor encounters a usage of all parameters. Its value is the result obtained by the visitor when used externally (i.e., in the checker).

The visitor approach works sufficiently well also in this case: the code is able to perform checks for the pattern in a simpler way than it would have with the FIR API alone. Even though the logic is more complex compared to the previous cases, however, we are not yet dealing with patterns that require to visit **more distant elements** in the FIR tree, for example *visiting the function declaration of a function call* or *linking the usage of a symbol to its declaration.*

Despite these considerations, it is already possible to see some drawbacks of

this approach: the code is already becoming less declarative, having to use *flags* to keep track of the state of the visitor, for example preventing some visit methods to be called in contexts we do not want them to be called (e.g., in the visitor just presented, the case of nested anonymous functions). This approach would clearly benefit from a more structured and modular API to simplify this kind of logic.

3.7 Fourth approach: mixed approach

In order to avoid the possible pitfalls of the visitor pattern that would make this approach less scalable to the increasing complexity of the pattern, we can use a mixed approach that combines the visitor pattern with the small API we previously introduced. The main advantage over other static analysis tools, in fact, is that the tree we are visiting is already enriched with a lot of information that the visitor can use inside its visit methods to perform checks, therefore avoiding the need to visit the tree in a more complex way. Moreover, this information is the same used by the Kotlin compiler to compile the code. We can also create utility functions that call other visitors to extract data useful for the checker or another visitor. Making this structure modular and extensible is key for the success of this approach.

3.7.1 Pattern 5: complex aggregate operations in loops

The case of aggregate operations in loops was already treated in section 3.5.1, but the pattern was not completely covered, (intentionally) presenting the pattern's description in a non-complete way. In particular, a special usage case was omitted, here called the case of **delegated functions**.

Rationale

In this case, the construct is not directly called inside the loop, but it is *delegated to another function that is called inside the loop*. This function is not necessarily a nested function, as in the case already seen in section 3.5.1, but it can be a function that is defined elsewhere in the code: from now on, this will be called the *delegated function* or simply the *delegate*. In order to contain aggregate calls, the delegate must have a parameter of type `Aggregate` to use as a receiver for

Listing 3.13: Examples of Pattern 5 detection in code, with the case of delegated functions

```
// One of the simplest examples of Pattern 5: this should raise a warning
fun delegate(aggregate: Aggregate<Int>) {
  aggregate.evolving(0) { ... } // Pattern detected
}

fun Aggregate<Int>.entry() {
  listOf(1, 2, 3).forEach { delegate(this) }
}

// One, more complex example of Pattern 5: this should NOT raise any warning
fun delegate(aggregate: Aggregate<Int>) {
  fun delegate2() {
    aggregate.alignedOn(0) {
      aggregate.evolving(0) { ... } // Pattern NOT detected
    }
  }
  delegate2()
}

fun Aggregate<Int>.entry() {
  listOf(1, 2, 3).forEach { delegate(this) }
}
```

the aggregate constructs (if, instead the delegate would have an explicit receiver of type `Aggregate`, in fact, the pattern would be trivially detected by the Pattern 2 checker, as the delegate would appear as an aggregate construct itself). When the delegate is called without alignment, the same problem as the one described in section 3.5.1 arises: the body of the delegated function could use the aggregate parameter to perform some computation that requires alignment, resulting in non-safe DSL usage, but the pattern would still not be captured because the delegate is technically not an Aggregate function. Not only the delegate could be called inside a loop, but it could also be part of a chain of function calls that ends with a call inside the loop, and detecting a possibly wrapping `alignedOn` operation could be quite complex, as shown in the corner cases of listing 3.13. The plethora of possible cases that can be captured in the code by this pattern is quite large, and seeing the second example should give an idea of how many possible code variations are related to this pattern, and why this additional constraint has been treated as a separated pattern with respect to Pattern 2.

Listing 3.14: Utility function to check if a function call has an aggregate parameter

```
fun FirFunctionCall.hasAggregateArgument(): Boolean =
  getArgumentsTypes()?.any {
      it.classId == ClassId.topLevel(AGGREGATE_CLASS_FQ_NAME.toFqNameUnsafe())
  } ?: false
```

Pattern detection

Given a looping construct (like the ones already seen for Pattern 2) and a declared function, here called *delegate*, that satisfies two requirements:

1. Its signature accepts at least one parameter of type `Aggregate`;
2. Its body contains an aggregate construct call, different from `alignedOn`.

The pattern is satisfied when *delegate* is called inside the loop or in a chain of other function calls that ends with a call inside the loop, and neither *delegate* nor the functions in the chain contain an `alignedOn` operation wrapping one of the calls.

Design and Implementation

To implement this pattern, a new condition will be added to the checker seen in listing 3.6 that will handle the case of delegated functions. The new condition will perform the same checks as the previous one (i.e., is inside a looping construct, and it is not aligned) but instead of checking if the function call being examined is an aggregate one, it will check if one of its parameters is of type `Aggregate`. To do this, a new utility function is implemented: `hasAggregateParameter`, shown in listing 3.14. This uses the `getArgumentsTypes` and `toFqNameUnsafe` extension methods that have been implemented for the job (and are omitted for brevity).

Once this condition is satisfied, the visitor will be called to inspect the delegate function and check if it contains an aggregate construct call or a call to another function that contains an aggregate construct call (possibly in a chain of calls). **Note:** we cannot reuse the existing code for Pattern 2 to check for the presence of the `alignedOn` operation, as its invocation may occur within a different function that is called inside the delegate but is not directly one of the *containing elements*

Listing 3.15: Utility function to get the declaration of a function call

```
@OptIn(SymbolInternals::class)
fun FirFunctionCall.getDeclaration(): FirSimpleFunction? =
  calleeReference.toResolvedFunctionSymbol()?.fir as? FirSimpleFunction
```

of the aggregate construct call. Therefore, the presence of `alignedOn` will be verified within the visitor.

To implement this visitor, we need a way to visit the function declaration associated with the function call that is being visited. Thanks to Kotlin API, this can be done like seen in listing 3.15. The `@OptIn` is necessary in order to access the `fir` field. Once this is done, we need to visit the function declaration and check its body for the pattern presence. In addition, however, we also need to be careful to detect *nested function declarations*, and for this reason a `functionCounter` variable is used to keep track of the nesting level of the function declarations. The other elements we need to visit are the function calls contained in the body of the delegate. The only functions we are interested in are of two types:

1. **Aggregate function calls**: function calls that require alignment — i.e., `alignedOn` operations;
2. **Function calls with aggregate parameter (i.e., delegates)**: this functions could recursively contain dangerous function calls, so are targeted by other checks.

In the first case, we check if the function call is an alignment operation, and if so we accept its anonymous function parameter with the visitor keeping track that we are inside an alignment, so other aggregate operations are considered safe within it. If the aggregate function is not an alignment operation and we are not *inside an alignment or nested function*, the pattern is detected.

In the second case we simply instantiate another visitor and recursively visit the function call in question, checking if the pattern is detected in the delegate.

The implementation just explained is summarized in listing 3.16. Although the code is still short like for the other visitors, in this case we need to take *multiple visiting calls* of the same method and even *recursive visiting* with an another visitor into consideration.

Considerations on this pattern

Unfortunately, some corner cases are not still captured by this checker and visitor: the main reason is that some corner cases that involve the usage of local variables from nested functions, similar to the second case shown in listing 3.13 without alignment present, are difficult to capture with the current approach. Since it is not a common usage case, it was decided to leave it for future works. It is a good opportunity, however, to reason about how what has been developed so far presents, from time to time, some branching conditions in the code or some particular programming "tricks" that appear as fragile or poorly maintainable code. This aspect is related to the "defensive" approach for developing these static checkers: one could think that detecting the pattern is the core of the implementation but, in reality, one of the most important aspects is **being careful not to introduce false positives** [LPS+23] to the static analysis. The cases we are trying to capture, in fact, are actually a very small subset of the possible programs using the Collektive DSL (and Kotlin in general). This is also the reason why adding more complex checks like the one just described has to be taken with much caution, and it is of main importance to add **more tests for cases where the pattern is not detected** than for cases where it is detected. The approach taken for testing this important factor will be discussed in the next chapter: for now, we will move to the final pattern, the one that was considered the most complex to detect, where multiple types of visitors were used together to inspect the code.

3.7.2 Pattern 6: improper Evolve construct usage

The last pattern presented in this chapter involves three of the constructs of the Collektive DSL that we have seen: `evolve`, `neighboring` and `share` (and their variants with the yielding context). Since we have not really explained their behavior in detail, they will be briefly discussed here.

The `evolve` construct is used to "evolve" the value of a device, iteratively updating it with the results of a function that is passed as a parameter. This function accepts also an *initial value*, from which starting the evaluation. The `neighboring` construct is used to evaluate an expression in neighbors, constructing

Listing 3.16: Implementation of the Pattern 5 Visitor

```
class FunctionCallWithAggregateParVisitor() : FirVisitorVoid() {
  // ...

  override fun visitFunctionCall(functionCall: FirFunctionCall) {
    if (functionCall.isAggregate()) { // Case 1
      if (functionCall.fqName() == ALIGNED_ON_FUNCTION_FQ_NAME) {
        insideAlignedOn = true
        functionCall.acceptChildren(this)
        insideAlignedOn = false
      } else if (!isInsideAlignedOnOrNestedFun()) { // utility function omitted
        found = true // Pattern detected
      }
    } else if (functionCall.hasAggregateArgument() && !isInAlignedOnOrNestedFun())
        { // Case 2
      val visitor = FunctionCallWithAggregateParVisitor() // recursion
      found = visitor.visitSuspiciousFunctionCallDeclaration(functionCall)
    }
  }

  // Visit function declaration
  override fun visitSimpleFunction(simpleFunction: FirSimpleFunction) {
    functionCounter++
    simpleFunction.body?.accept(this)
    functionCounter--
  }
}
```

a `Field` object mapping the neighbors to the results of the evaluation. The `share` construct is used to compute a *space-time* evolution of the field, computing an expression over time and sharing it with the neighbors.

Rationale

Since the `evolve` construct too can be used to evolve a field over time, its combination with the `neighboring` construct can behave exactly like a `share`, since the second one can be used to map the *space* aspect. This is not always true, of course, but using the `evolve` construct could easily lead to this inefficient usage of the DSL constructs, especially if the developer is not aware of constructs like `share`. Using this last construct, in fact, would be more efficient and clear in this case, since it would be more explicit in the code that the field is being shared with the neighbors and evolved over time.

This pattern consists specifically in this type of usage of the `evolve` construct. Since the anonymous function passed to it as a parameter must return a value of

Listing 3.17: Example of Pattern 6 detection in code

```
// Evolve + Neighboring substitutable by Share
evolve(initial) { value ->
  val newValue = value + 1
  val field = neighboring(newValue) // use of neighboring
  field.max(0) // field reduction with max
}

// Share
share(initial) { value ->
  val newValue = value.map { it + 1 }
  newValue.max(0) // field reduction with max
}

// Evolve + Neighboring NOT substitutable by Share
evolve(initial) { value ->
  val newValue = value + 1
  val field = neighboring(other) // separated use of neighboring
  newValue // new value of the evolution
}
```

the *same type* of the one gave to it as *initial*, a user could first use the `neighboring` construct to evaluate the expression in neighbors and then perform a *field reduction* operation — e.g., finding the max value in the field — and return that as the result of the anonymous function. Different is the case in which the `neighboring` construct is used to evaluate a *different expression* that **does not depend on the value updated by evolve** and therefore will share a value independently of the one evolved. The cases in which these constructs can be replaced with `share` represent a specific, small sub-portion of their general usage. At the same time, however, they could often appear as the first choice for a developer who is not aware of the `share` construct, since this way of composing the constructs can result being intuitive. It represents, however, a misuse of the constructs, that results in a less efficient and clear code, and therefore it should be detected by the static analyzer.

These examples of usage and the related version with the `share` construct substituted are shown in listing 3.17.

Pattern detection

Given an `evolve` construct, or its variant, that is passed a function that contains a `neighboring` construct, the pattern is satisfied when the expression evaluated

by `neighboring` depends on the value of `evolve` — i.e., the parameter inside the anonymous function — and it is used to compute the value returned by the body of `evolve`.

Design and Implementation

The mixed approach previously introduced will be fully exploited to implement this checker. In order to *trace the dependencies* between the expressions that relate to the evolved value — i.e., the parameter inside the `evolve` anonymous function — we need a system to *mark expressions* that uses this value and then maintains a list of symbols that are dependent on the evolved value. When these symbols are used in other expressions, we then mark these as dependent on the evolved value as well and so on. When we encounter a `neighboring` construct, we can immediately know if the expression evaluated by it depends on the evolved value, and if not the check can immediately end. If the expression depends on the evolved value, instead, we must save the evaluated expression and continue the visit until the end of the anonymous function with the return expression. Obtained the return expression, we must first extract the receiver of the `yielding` construct if we are inside `evolving` and not `evolve`, or simply the expression itself if not. This extra step is necessary for the variants of the construct because the value returned by the `yielding` is not really relevant to the pattern detection, since it will be returned externally as is. Finally, we must check if the return expression is **not** equivalent to the one evaluated by the `neighboring` construct, and if so the pattern is detected.

Since the workflow could appear as not really straightforward, a diagram of the process is shown in fig. 3.6. As for the previous patterns implemented using visitors, the implementation of the checker is quite similar and therefore omitted. The visitor, however, will need a specific section to be treated in detail.

Visitor implementation

The first element the visitor is going to visit is, again, a function call corresponding to a Collektive DSL construct, specifically the `evolve` one and its variant. In this case, the visitor will simply access the construct and look inside its parameters — i.e., the anonymous function. When this is done, the visitor will switch to the

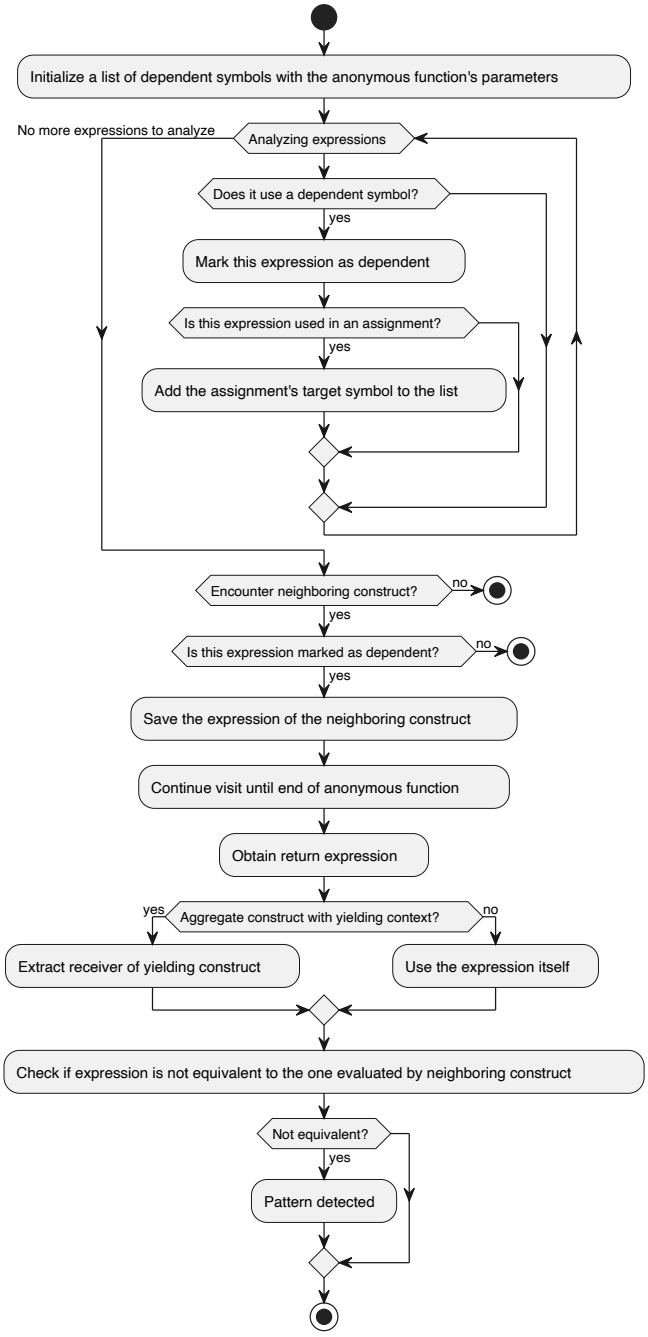

Figure 3.6: Summarized workflow of the Pattern 6 detection

Listing 3.18: Visitor method to visit an anonymous function

```
override fun visitAnonymousFunctionExpression(anonymousFunctionExpression:
    FirAnonymousFunctionExpression) {
  if (nestingLevel == 0) {
    val anonymousFunction = anonymousFunctionExpression.anonymousFunction
    val parameters = anonymousFunction.valueParameters
    parametersDeclarations = parameters.map { it.symbol }
  }
  nestingLevel++
  super.visitAnonymousFunctionExpression(anonymousFunctionExpression)
  nestingLevel--
}
```

Listing 3.19: Visitor methods to implement a symbol and expression marking system

```
override fun visitResolvedNamedReference(ref: FirResolvedNamedReference) {
  if (ref.resolvedSymbol in parametersDeclarations ||
      ref.resolvedSymbol in tracedDependentSymbols
  ) {
    markExpression = true
  }
}

override fun visitProperty(property: FirProperty) {
  if (markExpression) {
    tracedDependentSymbols += property.symbol // add the symbol to the list
    markExpression = false
  }
}
```

visit method of the anonymous function, where first it will check if this function is at **top nesting level** and not a nested function (for the first visit it obviously is) and will access the **function parameters** and add them to a list of *dependent symbols* kept by the visitor. In listing 3.18 this procedure is shown. The visitor will then visit the body of the anonymous function. Every time the visitor encounters a `FirResolvedNameReference`, corresponding to a symbol usage, it will check if the symbol is contained in the list of dependent symbols. If so, the visitor will set a variable `markExpression` to true, meaning that the expression is dependent on the evolved value. In this way, when the visitor encounters a property (i.e., local variable), it will check if the expression is marked as dependent and, if so, it will add the property to the list of dependent symbols. This is shown in listing 3.19. These methods take care of the tracing of the dependent symbols, now it is necessary to approach the core of the visitor: the `neighboring` and return expressions.

Listing 3.20: First support visitor to extract the return expression of an anonymous function

```
fun FirAnonymousFunctionExpression.extractReturnExpression(): FirExpression? =
  object : FirVisitorVoid() {
      private var returnExpression: FirExpression? = null
      // ...
      override fun visitReturnExpression(expression: FirReturnExpression) {
        returnExpression = expression.result
      }

    fun extractReturnExpression(): FirExpression? {
      visitElement(this@extractReturnExpression)
      return returnExpression
    }
  }.extractReturnExpression()
```

The first one can be inspected when the visitor encounters a function call, and then check for the fully-qualified name of the construct. In that case, we can extract the expression used as parameter of the function and perform the necessary checks. The `neighboring` construct can be used essentially in two ways:

1. **Direct expression**: for example, a numerical expression like a sum between two numbers;

2. **Computation**: for example, an anonymous function. In that case the field returned by the construct has the "computation" as a result, in the form of a field of functions.

In the first case, we can obtain the first argument of the construct, verify if the expression is *marked* when visiting it and, if so, save the expression into a variable for later. In the second case, we must first visit the anonymous function and obtain the return expression, and then perform the same operations. To do this, a utility function is implemented that **extracts the return expression of an anonymous function using another visitor**. The implementation is shown in listing 3.20. As shown, we create a visitor on the fly and visit the anonymous function, returning the expression found. This is an example of how composing visitors with the FIR API can help detecting a pattern.

Once we have saved the expression evaluated by the `neighboring` construct, a comparison with the return expression of the anonymous function is needed to complete the detection of the pattern. When the visitor encounters the return

Listing 3.21: Second support visitor to extract the receiver of a yielding construct

```
private class YieldingReceiverVisitor : FirVisitorVoid() {
  // ...
  override fun visitFunctionCall(functionCall: FirFunctionCall) {
    if (functionCall.fqName() == YIELDING_FUNCTION_FQ_NAME) {
      returnExpression = functionCall.explicitReceiver
    }
  }
  fun FirReturnExpression.getYieldingReceiver(): FirExpression? {
    visitElement(this)
    return returnExpression
  }
}
```

expression, it must first check if the nesting level is correct and if the expression of the `neighboring` construct if present. If not, the pattern is skipped. Finally, to compare the return expression with the one we previously saved, we can use the `isStructurallyEquivalentTo` utility function already introduced. However, the check is not finished since we must also handle a special case: the yielding context.

In the case of the yielding context, the expression returned by the `yielding` construct is not relevant to the pattern detection, and so its **receiver** must be used instead. To get the receiver, another support visitor is implemented, shown in listing 3.21.

The visitor is used in the visit method of the return expression, performing the final check on the equivalence of the expressions. If these are not equivalent, the pattern is detected. The implementation of this last visit method is shown in listing 3.22.

Final considerations

With this last visitor, all the patterns that were considered for the Collektive DSL static analysis have been implemented. As discussed in this chapter, many approaches were considered through the development, and in some cases mixed together to achieve the desired results. Since the presented patterns were not shown in their order of implementation, it is possible to see some areas of improvement, especially in the first ones implemented using the visitor pattern. Other considerations about this development process as well as other possible ways to implement the checkers will be matter of discussion in the conclusions of chapter 5. First, the

Listing 3.22: Visitor method to visit the return expression

```
override fun visitReturnExpression(returnExpression: FirReturnExpression) {
  super.visitReturnExpression(returnExpression)
  if (neighboringExpression != null && isCorrectNestingLevel()) {
    val expressionToCheck = neighboringExpression as FirExpression
    if (constructNameFQName == EVOLVING_FUNCTION_FQ_NAME) {
      // case with yielding context
      val yieldingReceiver =
        with(YieldingReceiverVisitor()) {
          returnExpression.getYieldingReceiver()
        } ?: return
      isReplaceable = !yieldingReceiver.isStructurallyEquivalentTo(
          expressionToCheck)
    } else {
      isReplaceable = !returnExpression.result.isStructurallyEquivalentTo(
          expressionToCheck)
    }
    return
  }
}
```

testing strategy that was adopted during these checkers' implementation will be presented.

Chapter 4

Evaluation and Testing

When developing tools that need to inspect and modify source code, where developers have freedom to write code in many ways and leveraging the full power of the used programming language, a solid and comprehensive testing strategy is crucial to ensure that the tool is working as expected and that it is able to capture all the cases that it is supposed to capture. If static analysis tools can be tedious to develop, due to the need to cover a plethora of possible cases often with a unified approach, testing static analysis tools must be even more comprehensive and flexible than development, trying not to fall into verbosity and repetition.

In this chapter we will present the testing strategy that was adopted to test the Collektive frontend plugin, starting from the initial approaches and moving into the development of an *ad-hoc* testing framework named *Subjekt*. Several snippets will be shown in this chapter: a basic understanding of the *Kotest* framework, the one used in the project, is required to fully understand them.

4.1 Initial testing approaches

The first approach taken was the simplest one: the tests were written using **static Kotlin sources** as resource files, loading them in the test suites written using the Kotlin testing framework Kotest. The first problem that arose regarded the *compilation process and the check for resulting diagnostics*. To test the checkers, in fact, the code must be compiled programmatically also providing the plugin to

the compiler, and then collect all the produced messages and compare them with the expected ones. Fortunately, a library already exists for this purpose: **Kotlin Compile Testing** by Thilo Schuchort (tschuchortdev on GitHub)[1]. This library provides a concise way to create objects that represent Kotlin source files by passing a string source code, and then compiles them giving a list of compiler plugins or annotation processors that should be used during the compilation process. All of this is done *inheriting the class path* of where the compilation process is running, so the Collektive plugin can be added without any problem also using the corresponding library. Finally, the diagnostics produced are contained inside the `KotlinCompilation.Result` object as a single string that can be easily split and filtered to get the relevant messages and compare them with the expected ones.

With this library, it is now possible to write tests only by reading the resource files that contain the source code that should result or not in a pattern being detected. These testing cases should be the same that were extracted from the codebase during the development of the checkers, as explained in section 3.3. An example of a test suite using this method can be seen in listing 4.1. As we can see, the testing process is quite short and sufficiently clear: `TestAggregateInLoop.kt` is a source file contained in the resource folder and contains a small source code with an aggregate construct used inside a loop, which is the case that should be detected by the checker for Pattern 2.

As one might expect, however, this approach quickly proves to be **not scalable** and **highly repetitive**. For a single pattern, there can be numerous variations in source code where the checker must operate, resulting in many test cases. In most instances, the differences between these cases amount to only a line or two of code. Following this method, the test suite would have required numerous static files, many of which would be nearly identical. Additionally, the Kotest specification would have become excessively verbose, even when leveraging Kotest utility functions like `forAll` to minimize redundancy.

[1]The actual library that was used is a fork of the original one, maintained by Zac Sweers (ZacSweers on GitHub). The switch to the fork was necessary due to some issues encountered during the building process of the project with Kotlin 2.0 which, at the time of writing, is still not supported by the original library. More information is available here: `https://github.com/tschuchortdev/kotlin-compile-testing/issues/411`

Listing 4.1: One of the first developed test for the Pattern 2 checker. It uses static resource files to load the source code to be compiled and checked

```
@OptIn(ExperimentalCompilerApi::class)
class TestAlignRawWarning : FreeSpec({
  "A single aggregate function called inside another one" - {
    val fileName = "TestAggregateInLoop.kt"
    val program = // get text from resource file...
    val sourceFile = SourceFile.kotlin(fileName, program)
    "should compile" - {
      val result = KotlinCompilation().apply {
        sources = listOf(sourceFile)
        compilerPluginRegistrars = listOf(AlignmentComponentRegistrar())
        inheritClassPath = true
      }.compile()
      val expectedWarningMessage = // warning to check...
      result.exitCode shouldBe KotlinCompilation.ExitCode.OK
      result.messages shouldContain expectedWarningMessage
    }
  }
})
```

4.2 Avoiding repetitions through template files

One of the first solutions to the problem of repetition when dealing with textual files very similar to each other is to use **template files**. The approach is the following: for each pattern to test, collapse all the testing cases that are similar to each other into a single file containing placeholders for the parts that change, and then dynamically replace these placeholders with the correct values during the test execution.

4.2.1 Initial template system

The initial template system consisted of Kotlin files containing the source code to be compiled with placeholders like the ones used in classical string formatting — i.e., `%s`. An example is shown in listing 4.2.

Listing 4.2: A example of the templates created for testing multiple cases at once

```
%s
for (j in listOf(1, 2, 3)) {
  %s
  neighboring(0)
  %s
}
%s
```

Then, instead of compiling the string obtained from the resource file directly, replace the placeholders cycling through a list of possible, hard-coded configurations in the test suite and for each perform the correct check on the diagnostics produced. In addition, a small utility was extracted in order to make the process of reading resources, compiling and comparing messages more concise and readable using Kotlin infix methods. The result looks similar to the one shown in listing 4.3. Only a relevant part of the test suite is shown, the rest is omitted for brevity.

Listing 4.3: Part of the test for Pattern 2 revised using template files

```
"A single aggregate function called inside a loop" - {
  val testingProgramTemplate =
    CompileUtils.testingProgramFromResource("TestAggregateInLoop.kt")

  "without a specific alignedOn" - {
    val program = testingProgramTemplate.formatCode("", "", "", "")
    "should produce a warning" - {
        program shouldCompileWith warning(EXPECTED_WARNING_MESSAGE)
    }
  }
  "with a specific alignedOn" - {
    val program = testingProgramTemplate.formatCode("", "alignedOn(0) {", "}", "")
    "should compile without any warning" - {
        program shouldCompileWith noWarning
    }
  }
}
```

4.2.2 Templates flexibility and limitations

Even though this approach gained some flexibility and reduced repetitions on the template files side, it still requires a lot of boilerplate code to be written between each test case. Another step is necessary to make the testing process even more concise: integrating the created utility with a string interpolation mechanism that allows configurable parts without repeated and obscure code for formatting. This allows to write templates that are more clear and readable, because instead of writing placeholders with `%s` we can write variables with names like `%(nameOfTemplateVariable)` and then pass a configurable map through the test code that will replace these variables with the correct values. All of these utilities, contained in the singleton object `CompileUtils`, permit writing tests like the ones

Listing 4.4: Listing 4.3 revised using the new utilities

```
val testingProgramTemplate = // like before
listOf(
  "exampleAggregate" to "exampleAggregate()",
  "neighboring" to "neighboring(0)",
).forEach { (functionName, functionCall) ->
  "using $functionName without a specific alignedOn" - {
    "should produce a warning" - {
      val testingProgram = testingProgramTemplate
          .put("aggregate", functionCall)
      testingProgram shouldCompileWith warning(EXPECTED_WARNING_MESSAGE.format(
          functionName))
    }
  }
  "using $functionName wrapped in a specific alignedOn" - {
    val testingProgram = testingProgramTemplate
      .put("beforeAggregate", "alignedOn(0) {")
      .put("afterAggregate", "}")
    "should compile without any warning" - {
      testingProgram shouldCompileWith noWarning
    }
  }
}
```

shown in listing 4.4. Variables also have default values, making the tests more concise and easier to write.

The test methodology not only gained in readability, but it is much more scalable and can be used to test many more cases adding only a few lines of code. The obtained result, however, is still far from optimal: the testing code still depends too much on the structure of the template files, exploiting hard-coded names of variables inside the template files and in the test suite. The problem still relies on the fact that **the testing resource does not encapsulate the possible values of its variables**, making it more flexible indeed but also less maintainable and more error-prone. Without using a system more similar to a *template engine*, it would be difficult to build the testing process without occurring in scale issues, especially for more complex patterns discussed in chapter 3 where the number of possible cases to test is much higher than the ones presented in the examples. Even in that case, however, the testing process would have to interact with the templates in order to determine values of its variable parts, also for **identifying the expected outcomes** of the resulting source code's compilation. In other words, there still would be too much coupling between the testing code and resource files.

4.3 Code generation: a small DSL leveraging Kotlin Poet

A possible approach to solve the problem of coupling between the testing code and the resource files is **code generation done directly inside tests**. Instead of using external files, a developer could create an appropriate configuration in the tests to capture a multitude of cases, declaratively specifying how the source code should be structured and therefore knowing at prior what the expected diagnostics should be. The task is certainly not an easy one, since generating multiple source codes in a declarative way using the same programming language as the target one is not trivial. Thankfully, an already existing and well-known library can be used for the generation: **Kotlin Poet** by Square Inc[2].

Kotlin Poet is a library with a Java and Kotlin API to generate Kotlin sources. Its main use cases are generating code for annotation processors or interacting with metadata files, and it is inspired by another library by the same authors, Java Poet. The library is quite powerful and allows generating code in a declarative way, using a fluent API to create complex structures with few lines of code. One way to use it is through a *builder* pattern, where methods are chained to create the desired structure. This particular way of generating code can reveal itself as very useful for our purpose, since it can be manipulated to build a DSL that facilitates some operations for the developer.

4.3.1 General structure and usage

To make the generation of code more concise and readable, a DSL has been created using **extension methods to the Kotlin Poet builders**. Encapsulating the generation of common code and frequent and repeated operations inside these methods, allows for a completely customizable output with a few lines of code, using constructs tailored for our testing cases. Take the extensions shown in listing 4.5 for example: in this case we create utilities to build programs with blocks and specific wrapping functions, like the `alignedOn` seen in section 3.5.1 during the explanation of Pattern 2. This way, the developer can pass other extension

[2] `https://square.github.io/kotlinpoet/`

Listing 4.5: Example of Kotlin Poet extensions used to generate code

```
fun FunSpec.Builder.block(
  header: String = "",
  content: FunSpec.Builder.() -> FunSpec.Builder,
): FunSpec.Builder =
  beginControlFlow(header)
    .content()
    .endControlFlow()

fun FunSpec.Builder.alignedOn(content: FunSpec.Builder.() -> FunSpec.Builder):
    FunSpec.Builder =
  block("alignedOn(0)", content)
```

methods — e.g., parameter `content` visible in listing 4.5 — to these DSL-like keywords to highly customize the final output, therefore maintaining Kotlin Poet benefits but also making it more directed to our needs.

Finally, it is possible to rewrite the tests, using no external resource file and reusing the same pieces of code that do not change across the several checks. A small snippet of how this small DSL can be used in tests is shown in listing 4.6.

Listing 4.6: Example of a test suite using the Kotlin Poet extensions

```
// creates a file with the needed imports and an example Aggregate function
val sourceFile = simpleTestingFileWithAggregate()
// create a function with `Aggregate` receiver of type Int
val startingFunction = simpleAggregateFunction(INT)

forAll(testedAggregateFunctions) { functionCall ->

  "using $functionCall wrapped in a specific alignedOn" - {
    val generated =
      startingFunction + {  // customization of a common function
        loop {
          alignedOnS {
            functionCall
          }
        }
      }
    "should compile without any warning" - {
      sourceFile withFunction generated shouldCompileWith noWarning
    }
  }
// ...
```

Imagining various checks to be done for Pattern 2 — the one the examples of this chapter are based on — this approach greatly reduces the amount of boilerplate

code, intervening only inside the function containing the call to the aggregate and loop constructs, and then customizing it with other parts, like the `alignedOn` construct that wraps the tested function call. At last, the expected diagnostics are formulated starting from the case that is being generated then tested and compared with the one produced by the compilation thanks to the Kotlin Compile Testing library already cited. This last operation is encapsulated inside the small utility library already implemented for the previous testing approaches, adding also the customized function to the source code: this summarizes what is done at line 18 of listing 4.6.

4.3.2 Considerations

Considering the context of this project, the approach of code generation seems to be the most flexible and scalable solution to the problem at hand: even though it was not considered worth the effort to integrate a system like this in the first stages of the development, patterns examined further in the thesis showed how many possible cases required a proper a more complete testing strategy without falling into verbosity and repetition (e.g., Pattern 5 and Pattern 6). Nevertheless, more complex test cases, where more than just a couple of lines of code change, still require considerable effort to create Kotlin sources subject to compilation. With the small DSL created, the approach just described tends to become less readable and harder to interpret, to the point that in cases where there isn't enough common code, it becomes easier to write the code directly as a single string.

It seems that the problem still regards static resource files in a sense: even though common parts of the code are reused and customized with the DSL, the "variability" of general-purpose languages like Kotlin makes it difficult to reduce possible Kotlin sources to something like templates or customizable parts of code. For these reasons, this small DSL was not integrated into the Collektive project, but it was used as a starting point to develop a more complete and flexible testing framework that could be used to test the Collektive frontend plugin in a more comprehensive way.

4.4 Custom testing framework: Subjekt

Subjekt[3] is a tool for generating textual results starting from a configuration in YAML or JSON format. It can be used to generate multiple results from permutations of parameters defined in the configuration, easily creating many variations of the same output. This tool started as a small utility to generate multiple Kotlin sources from a starting configuration and then integrate them inside Collektive test projects. Later in the development process, it was decided to expand its capabilities to directly generate test cases for the Collektive plugin from inside the test suites, adding only the configurations as resources.

The development of Subjekt was mainly motivated by the need to expand the number of testing cases, since the testing strategy adopted so far was not scalable enough to cover sufficient cases to ensure the correctness of the checkers. After the code generation strategy was considered, it appeared clear that a more flexible and readable approach was needed to make this approach feasible as a testing method. The main goal was to extract common *notions* of constructs or structures (e.g., loop constructs) that were repeated across the testing cases, notions that could be easily understood and that could assume several "aspects", equally important to the testing process and whose *permutations* with other variable structures were needed to be tested as well. Since this was not possible without an ad-hoc tool, Subjekt development started.

Subjekt is a small Kotlin multiplatform library available on *Maven Central*, *npmjs* and GitHub packages repositories. In this section we will present the main benefits of using a tool like this for testing static analysis projects like the Collektive frontend plugin.

4.4.1 Main ideas behind the framework

The main inspiration behind Subjekt is the testing strategy used by the *Scala* compiler[4] (re-invented on a smaller scale). Essentially, the core idea is to set up a suite of *test cases* having expected positive or negative outcomes. In the Scala compiler, source code and expected outcomes are separated into *.scala* and *.check*

[3] `https://github.com/mini-roostico/subjekt`
[4] `https://github.com/scala/scala3/tree/main/tests`

files, respectively. Subjekt follows a similar approach, but linking together source and expected outputs. The developer can write a configuration file in YAML or JSON format reflecting a **suite** of test **subjects** that are related to a similar pattern to be tested. As for Scala, every testing case has a unique identifier, and in the case of Subjekt can be extracted from the result of the generation to refer to a specific test case in the suite.

4.4.2 Core structure of Subjekt

In Subjekt, each configuration file defines a **Suite**, composed by a list of **Subjects**. Each Subject is essentially a map with string keys and values, where the latter contain **Expressions**: strings with expressions are called **Resolvables**. The expressions are strings that are parsed to generate a multitude of results: *Subjekt* will extract these results and produce a *set* of outputs considering all the permutations of the expressions contained in the subjects. Special delimiters `${{` and `}}` are used in order to extract expressions from strings, inspired by the GitHub actions expressions syntax[5]. The diagram in fig. 4.1 shows the class diagram of the general concepts of the library.

The main entities that produce permutations are **Parameters** and **Macros**. Both of them can have multiple values to substitute in the expressions where they are used, but the difference is that the former's values do not contain other expressions, while the latter's ones are actually other Resolvables and can accept arguments to substitute in their expressions, similar to a function that returns multiple values, leading to an exponential growth of the number of results (for example, when other Parameters are passed to the Macro, making Subjekt produce permutations of the passed Parameter's values and with Macro's ones). When a Resolvable needs to be "resolved", the library will determine all the possible **Contexts** that can be generated from the Parameters and Macros. Each Context will contain one "fixed" value for each of them — i.e., **DefinedParameter** and **DefinedMacro** — and will therefore correspond to one of the total permutations. The Resolvable will finally produce one **Instance** for each of the Contexts.

[5] `https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/evaluate-expressions-in-workflows-and-actions`

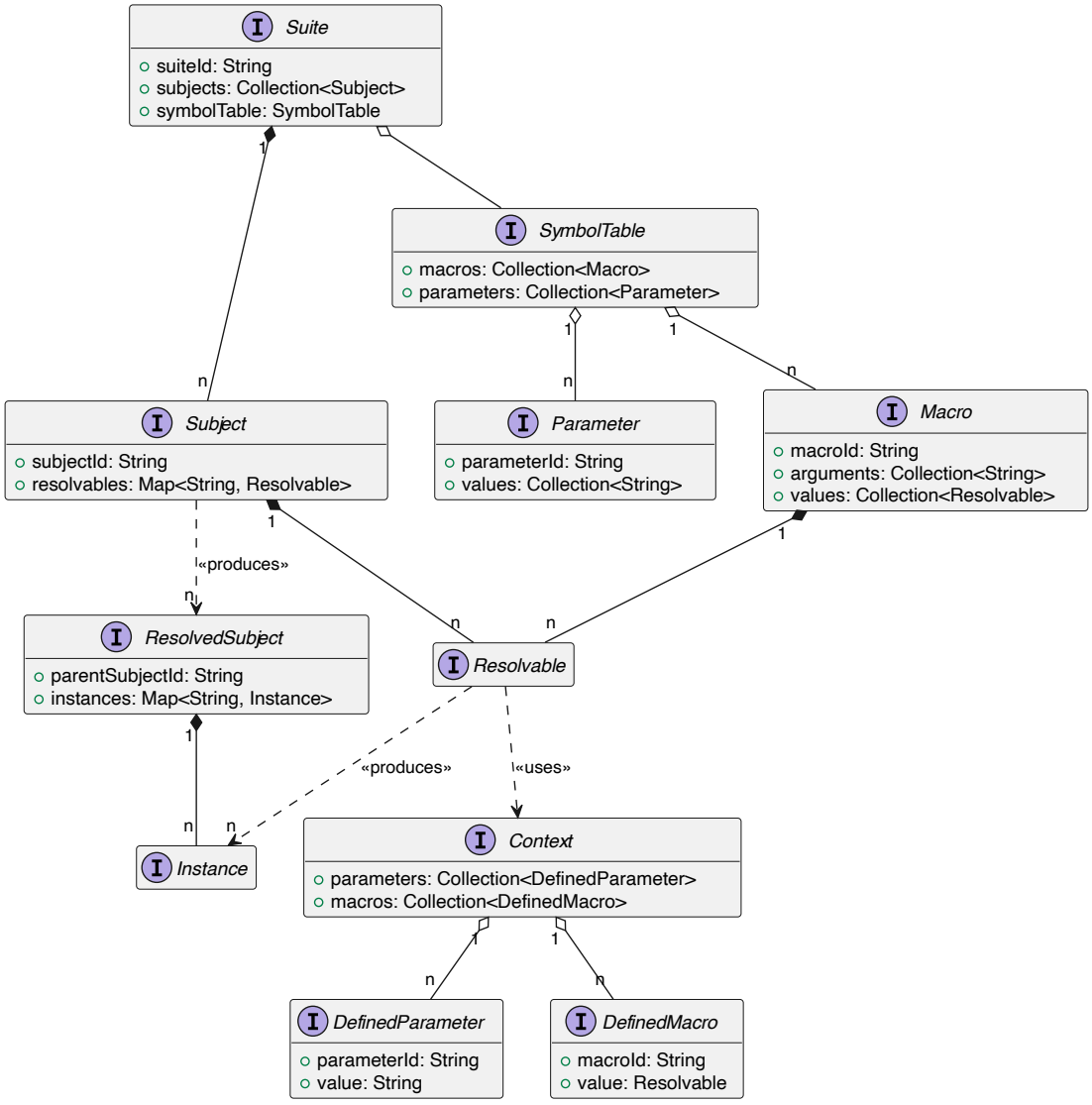

Figure 4.1: Top-level class diagram of the Subjekt library

The final results are the **ResolvedSubjects**. Each of these is essentially one of the generated outputs, and can be configured to contain one of the source codes to be compiled and the expected diagnostics to be checked. The output is actually a *set* of these objects to avoid testing the same case multiple times in case of collisions, due to the permutations of the expressions.

4.4.3 Final usage inside tests

As previously introduced, Subjekt can be configured using YAML or JSON format. To use it inside Collektive in order to produce testing cases, a resource file will be created for each pattern to be tested, containing the entirety of cases for that pattern. In order to test them, we will create a Kotest specification for each Subjekt configuration, that gets loaded using the Subjekt library and then executed comparing the diagnostics produced by the compilation with the expected ones.

Ultimately, the Kotlin Compile Testing library was abandoned in favor of a custom solution that uses a utility singleton named `CollektiveK2JVMCompiler` that automatically compiles a list of source files returning the diagnostics produced via a `MessageCollector` and using the Collektive compiler plugin.

Listing 4.7: Example of a Subjekt configuration file used for Pattern 2 testing.

```
name: "Invalid iteration of aggregate calls"
# ...

subjects:
  - name: Iteration${{ prettify(AGGREGATE, loop(AGGREGATE)) }}
    code: |-
      fun Aggregate<Int>.entry() {
        ${{ loop(AGGREGATE) }}
      }
    outcomes:
      warning: |-
        Aggregate function '${{ AGGREGATE }}' has been called...
      # ...
```

In listing 4.7 a small example of a reduced Subjekt configuration used for the tests of Pattern 2 is shown. In this configuration `loop`, `prettify` and `AGGREGATE` are respectively two Macros and one Parameter that are omitted for brevity. The first one wraps the argument with several types of iteration constructs, for example a `for` loop or a `map` method call on a list. The second is a Macro exposed by

Listing 4.8: Example of a test suite using Subjekt

```
val testSubjects =
  subjekt {
    addSource("src/test/resources/subjekt/IterationWithAggregate.yaml")
  }.toTempFiles() // creates a map of names -> temporary source files

forAll(testedAggregateFunctions) { functionCall ->
  forAll(formsOfIteration) { iteration, iterationDescription ->
    "using $functionCall in $iterationDescription without alignedOn" - {
      // uses a utility function to retrieve the source from 'testSubjects'
      val code = getProgramFromCase("Iteration")

      "should compile producing a warning" - {
        code shouldCompileWith
          warning(
              expectedWarning(functionName),
          )
      }
    }
  // ...
```

Subjekt in all of its configurations and that is used to concatenate any number of arguments into a camel case, special characters-free formatted string. The last one is a Parameter that contains a list of aggregate constructs to be tested in the source code (e.g., `neighboring`).

The configuration file is then loaded inside the test specification, and the subjects are resolved and executed. The expected diagnostics are then compared with the ones produced by the compilation, and the test is considered successful if they match. In listing 4.8 a small snippet of the test suite using Subjekt is shown.

As we can see, the test suite is more concise and readable compared to previous approaches, and can be easily expanded to test more cases by adding more configurations to the resource file. Moreover, inside the Kotest specification the developer does not have to worry about the details of the source being compiled or the diagnostics produced and, with an appropriate implementation of the check, it is relatively easy to immediately see the source code that caused an eventual failure and the expected diagnostics that were produced or not, leading to an efficient test-driven development cycle.

Subjekt is still a small library that can be expanded and improved in order to make the creation of tests even more automatized and with less repetitions, but its approach represents a possible solution to the problem of testing static analysis

tools that was faced during the development of the Collektive frontend plugin for this thesis.

Chapter 5

Conclusions

In this chapter we will present the conclusions of the work done during the thesis, starting from the main contributions and the results obtained, and then moving to the discussion of the limitations and possible future works that can be done to improve the Collektive frontend plugin as well as static analysis tools via compiler plugins in general.

5.1 Opportunities of Compiler plugins

Compiler plugins could represent a very powerful tool for developers that want to customize a specific "environment" within a framework written in a general purpose language. If brought to a sufficient degree of integration, they can greatly enhance *internal* DSLs capabilities, breaking some remaining barriers between the domain and the host language. If added to an automatically integrated static analysis tool like a frontend compiler plugin, it can also help developers to avoid common obstacles to the adoption of new tools and "languages" in their projects, speeding up the learning process of the new constructs and spreading new paradigms' usage.

In this thesis, the development of the Collektive frontend plugin showed an example of how this can be approached in a real use case, and how the interaction with the compiler can happen in a seamless way, potentially leading to libraries made specifically to build Kotlin compiler plugins enhancing the already existing,

still limited and experimental, K2 frontend API. Through the development of the plugin, it was interesting to see how spotting new patterns to be caught with static analysis immediately corresponded to their implementation directly in the same tool used to enable the target DSL. This helped understand just how much bringing static analysis towards the tool that is being analyzed can be beneficial, since it allowed to immediately spot bad usages inside already existing code, therefore leading to the formulation of new patterns to be detected and building a virtuos cycle of improvement of the whole framework. Moreover, such a development could exploit this opportunity to delegate bad usages of the DSL to the compiler, making the developer's life easier and the codebase more robust, without having to rely on "programming tricks" or forced limitations of the language to avoid bad usages. Finally, using a unified approach to the static analysis tool development, in this case the frontend compiler plugin, can greatly facilitates the inclusion of such a tool in the development environment, making it interact with the IDE and build system automatically and without the need of additional IDE plugins or other types of necessary integrations, often developed by third parties and not always up-to-date with the latest versions of the framework.

In order to show a solid development process of such a plugin, covering the full spectrum of the cycle, notable effort was put into building an effective testing strategy that could be used to test also a bigger number of patterns even more complex than the ones presented in this thesis. This led to the exploration of different testing approaches, each one with its own pros and cons, and finally to a more scalable and flexible method, accompanied by the small testing framework Subjekt, developed exactly for this purpose.

5.2 Approaching meta-level analysis

In this thesis, multiple techniques have been employed to build checkers capable of identifying increasingly complex patterns, ultimately relying on the widely used *visitor pattern*. This choice was necessary to align with Kotlin's FIR API, which, while offering extensive capabilities, remains closely tied to the compiler's internal structures. However, this strong coupling highlights a broader challenge: the lack of a higher-level abstraction for static analysis within compiler plugins. As demon-

strated, performing pattern-based checks requires directly traversing and manipulating the tree, making the implementation more intricate and less maintainable. This underlines the importance of elevating compiler plugins to a **meta-level**, where analysis can be expressed through a specialized meta-language rather than being deeply intertwined with compiler internals. Such an approach, similar to what other static analysis tools already provide, enhances at least three aspects of the analysis tool:

- **expressiveness**: meta-level languages already exists in order to capture complex patterns and relationships between code constructs, and can be used to express the analysis in a more concise and readable way;
- **maintainability**: by decoupling the analysis logic from the compiler-level representation, it becomes easier to adapt to language evolution and ensure more scalable and reliable checks;
- **modularity**: the analysis can be split into smaller, more manageable components, focusing on specific aspects of the pattern being checked.

Thus, while the FIR API enables powerful static analysis, the need for a higher-level abstraction remains crucial to making these techniques more accessible, flexible, and efficient in the long term.

5.3 Future works

At the moment of writing, the Kotlin API that enables static analysis in frontend plugins like the one seen in this thesis is still highly experimental, therefore subject to changes and improvements. Developing a meta-level language tool for this API could be a possible future work, but it would require a more stable and mature version of the API to be effective. Despite this, working on a unified language similar to the one used in other static analysis tools could be a good starting point, maybe based on *tree-based pattern matching* techniques or other more advanced methods. This could allow compiler plugins to become easier to use and more accessible to developers, leveraging the full power of this type of system.

The proposed static analysis also has a lot of room for improvements: the static analyzer could switch to a more **rule-based** approach, allowing for a declarative definition of the patterns, that could be recognized *by-name*. In other words, this would consist of a *top-down* approach where the developer defines the rules that the code must follow, without starting from the compiler in order to implement the rule itself. One possible way to refactor the current analyzer's architecture would be to introduce a **rule engine** that could be used to define the patterns to be detected, using a general set of checkers to run the rules on the codebase. Another possible strategy could be a *code generation* one, where the checkers are created and added to the compiler plugin automatically starting from a set of rules defined by the developer, maybe using another compiler plugin made for the job.

More for the static analysis side, allowing compiler plugins to use configuration files already used by other tools like *Detekt* would strengthen the idea of the unified approach even more, making the compiler plugin approach an interesting choice as a standard for static analysis tools on the Kotlin platform.

Another possible future work could involve the testing method, expanding the automatized approach taken with the Subjekt framework to be more inclusive and flexible, allowing the developer to test more complex patterns with less effort, maybe even in a way that is tied to the meta-level language used for the checks. Finally, the development of the Collektive frontend plugin could be expanded to include more patterns and more complex checks, maybe arose after a more in-depth usage of the DSL in a real project, in order to make the plugin more complete and effective in spotting bad usages of the available constructs.

The Kotlin ecosystem is one of the most vibrant and fast-growing, thanks to its flexibility and the wide range of platforms it can target. Compiler plugins are a powerful tool that can be used to enhance the development process and the quality of the code produced, bringing paradigms that struggle to be adopted into the mainstream. Empowering developers to use these tools more seamlessly across this vast landscape of platforms and projects could unlock their true hidden potential, turning them into a powerful asset rather than a niche solution.

Bibliography

[ACD+24] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. The exchange calculus (XC): A functional programming language design for distributed collective systems. *J. Syst. Softw.*, 210:111976, 2024.

[ADVC16] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. Run-time management of computation domains in field calculus. In Sameh Elnikety, Peter R. Lewis, and Christian Müller-Schloer, editors, *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), Augsburg, Germany, September 12-16, 2016*, pages 192–197. IEEE, 2016.

[Aud20] Giorgio Audrito. FCPP: an efficient and extensible field calculus framework. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020*, pages 153–159. IEEE, 2020.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[CV16] Roberto Casadei and Mirko Viroli. Towards aggregate programming in scala. In *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*, page 5. ACM, 2016.

[DVPB15] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computa-

tional fields. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2015.

[EBO+15] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 50–60. ACM, 2015.

[LPS+23] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimäki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.*, 198:111575, 2023.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[PBV17] Danilo Pianini, Jacob Beal, and Mirko Viroli. Practical aggregate programming with protelis. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 391–392. IEEE Computer Society, 2017.

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 247–260. ACM, 2008.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.

[Tho21] Patrick Thomson. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *ACM Queue*, 19(4):29–41, 2021.

[VAB+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2):16:1–16:28, 2018.

[VBD+19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.*, 109, 2019.

[ZSO+17] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In Jesús M. González-Barahona, Abram Hindle, and Lin Tan, editors, *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 334–344. IEEE Computer Society, 2017.

Acknowledgements

I would like to sincerely thank my advisor, Prof. Danilo Pianini, for guiding and supporting me throughout this work. I especially thank him for proposing a thesis topic that aligned as closely as possible with my interests and for helping me learn many new things along the way. I also extend my thanks to my co-advisors, Nicolas Farabegoli and Angela Cortecchia, for their helpful discussions and for proposing interesting ideas that contributed to the project.

Finally, I am very grateful to my friends and family for their constant support throughout my master degree. Their encouragement has been essential in completing this journey.