FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Lukáš Riedel

Extending Data Lineage Analysis Platform with Support for Dependency Injection Frameworks

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Pavel Parízek, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2021

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the ”Metodický pokyn o etické přípravě vysokoškolských závěrečných prací”.

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the ”Copyright Act”), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs (”software”), I hereby grant the so-called MIT License.

The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and / or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

Copyright (c) 2021 Lukáš Riedel

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

In .... date ....

Author’s signature

I would like to thank everyone who helped me to get through all the difficulties related to distance studies, my supervisor Pavel Parízek, my family, and my colleagues in MANTA in particular, especially Richard Vítek.

Title: Extending Data Lineage Analysis Platform with Support for Dependency Injection Frameworks

Author: Lukáš Riedel

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data lineage forms an important aspect of today's enterprise environment. MANTA Flow is a data lineage analysis platform that already has basic support for analysis of Java programs, provided by one of its components called Bytecode Scanner. Nevertheless, there are very few applications in today's enterprise environment that do not use dependency injection at least in a very limited way. Therefore, we present an extension of Bytecode Scanner in the MANTA Flow platform to support data lineage analysis of dependency injection frameworks as well. The extension is able to process even complex definitions of standard dependency injection containers. Since the dependency injection influences a selection of method call targets, we also provide a description of call graph structure and its modification to support dependency injection. Last, we use this infrastructure to design and implement a plugin into Bytecode Scanner for the Spring Framework, a popular dependency injection framework targeting Java Platform. The plugin has been successfully tested on a small but realistic software system that can read data from a file, transform them, and write them into a database.

Keywords: data lineage, dependency injection, call graph, Spring Framework, static analysis

Contents

1. Introduction

Data lineage forms an important aspect of today's enterprise environment. It describes the flow of data within the application. In other words, it describes relationships between the application's data sources and data sinks. However, these enterprise applications tend to be large, and thus there is a greater need for automation of data lineage across the whole company's system. Tracking of data lineage inside these systems is especially important due to audits and legal reasons, for example, it is necessary to have complete information about the flow of client's data throughout the system and whenever the client asks the system to delete all information about himself, the system should remove all the traces about his existence without any asking. This is where automated data lineage shines because it can scan the whole system's source code, extract information about manipulation with data and associate data accesses to each other. The result is a data lineage graph that is understandable not only for developers but also for other company stakeholders who can easily conclude whether the system satisfies legal obligations. Nevertheless, there are very few applications in today's enterprise environment that do not use dependency injection, or generally any kind of IoC[1] principle, at least in a very limited way. Dependency injection makes programming easy, the code is more readable and dependencies between the system's components are more clear. Therefore, it is needed to support automated data lineage of applications that utilize dependency injection as well.

Dependency injection frameworks with no exception use reflection, which is purely a matter of target language runtime. However, data lineage analyzers are mostly static, which means they should not be able to execute the analyzed code. In other words, the analyzer is technically unable to use features of the dependency injection framework used within the analyzed application. In the thesis, we describe a way how to modify the analyzer so it can analyze the data lineage of applications that use dependency injection frameworks. Particularly, we present an implementation of support for the the Spring Framework [1], a popular dependency injection framework targeting Java Platform, within MANTA Flow [2], the fully automated lineage platform.

1.1 Goals

The main goal of the thesis is to propose a solution for support of the Spring Framework within the MANTA Flow platform. Although the Spring Framework itself has many features, we only focus on its features related to dependency injections, Spring Beans in particular. Since there was no support for dependency injection frameworks in the MANTA Flow platform previously, one of the goals is to analyze an impact on the data lineage analyzer, as well as to design the solution in such a way that it is modular and easily extensible by support for other dependency injection frameworks. Among others, the proposed solution has a great impact on the call graph, one of the core structures of the data lineage analyzer responsible for resolving method calls' targets, which we are

[1] Inversion of Control

supposed to modify in such a way that it can support dependency injection. As a result, the data lineage analyzer should be able to produce a correct data lineage flow graph not only for standard Java applications but also for those which use dependency-injection-related features from the Spring Framework.

1.2 Outline

Chapter 1 is an introduction into the problematics, where we provide a basic view into the problem and what the goals are. In Chapter 2, we describe the MANTA Flow platform, which is the target platform for our solution. Chapter 3 talks about technologies and algorithms that we use in the solution. In Chapter 4, we perform an analysis of the problem and describe requirements on the data lineage. In Chapter 5, we describe an implementation of the plugin for the Spring Framework targeting Bytecode Scanner in the MANTA Flow platform. Chapter 6 describes an implementation of the call graph for bytecode and its extension to support dependency injection plugins. In Chapter 7, we evaluate the solution by discussing the limitations. Last, Chapter 8 concludes the thesis and talks about possible future extensions of the solution.

2. MANTA Flow Platform

The MANTA Flow platform is a commercial product, the unified data lineage platform [2]. The platform supports data lineage analysis of many technologies. For each technology, there is a *scanner* that can perform the analysis, i.e., track data propagations and transformations across the analyzed code from sources to sinks. Among the supported technologies are databases[1], reporting tools[2], data integrations[3] and programming languages. For programming languages, we are particularly interested in Bytecode Scanner, which can perform analysis of Java code, or generally any code compiled into Java bytecode. In the thesis, we have gained inspiration for many problems solving in the novel C# Scanner [3], which is a part of the MANTA Flow platform as well.

2.1 Bytecode Scanner

Bytecode Scanner, like any other scanner in the MANTA Flow platform, is divided into two parts, *Connector* and *Dataflow Generator*. We are particularly interested in Connector only, which is the component that performs the analysis. More precisely, like any other connector, Bytecode Connector is further divided into *Extractor* and *Resolver*. The Extractor prepares input and configuration for the Resolver. The input is a program to be analyzed and some options for the analysis. The Resolver produces the analysis' output, which is made of the graph with dataflows, where a node is some data action endpoint like database or file stream, and there is an oriented edge between two nodes if some data flow between the two nodes. A brief introduction into the analysis is described in Section 2.1.1. In Section 2.1.2, we describe a way how the analysis can be extended. Decomposition of the scanner into modules is visualized in Figure 2.1.

2.1.1 Bytecode Analysis

The analysis of bytecode is *symbolic* and modular, and is realised by Bytecode Resolver. The symbolic analysis deals with *symbolic expressions*. Symbolic expressions create an abstraction over bytecode. For example, the simplest expression is a constant expression, which just holds information about the loaded constant. Another expression can be an expression symbolizing access to a particular element in an array.

Expressions are products of bytecode interpretation. Instead of simulating the value stack in JVM[4], the analyzer maintains a stack of symbolic expressions. For example, if there are two constant expressions and the interpreter is just approaching the instruction `add`, it pops the two expressions from the top of the stack and creates a new binary expression. The binary expression is a simple tree, where the root is the operator and it has two leaves, the operands. Then, this expression is pushed onto the stack of symbolic expressions. The JVM runtime

[1]Microsoft SQL Server, Oracle, PostgreSQL or DB2
[2]Qlik Sense, Microsoft Excel, Tableau or Microsoft SQL Server Integration Services
[3]Alation, Talend or StreamSets
[4]Java Virtual Machine

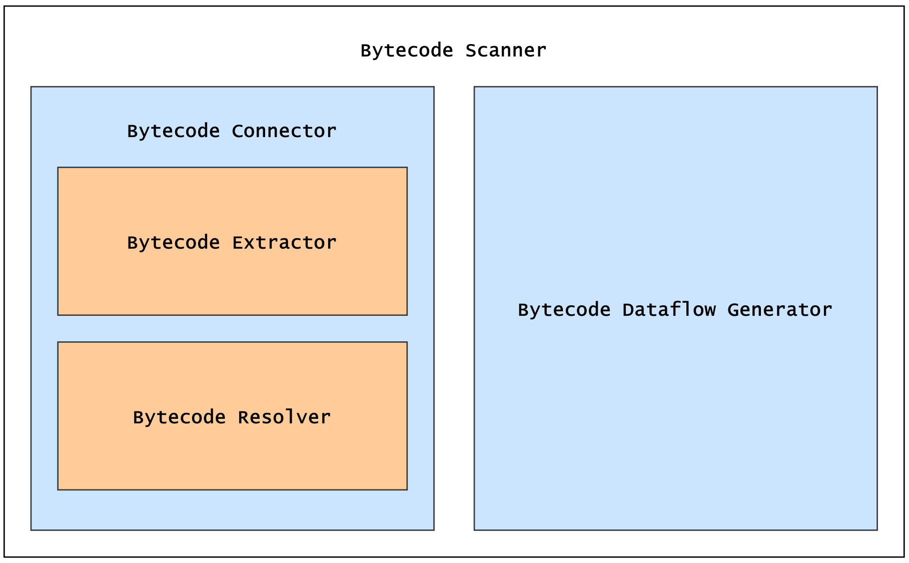


Figure 2.1: Module view of Bytecode Scanner

would push the actual value onto the value stack, which would be a product of the binary operation.

The analysis uses a *worklist* algorithm. It processes methods one by one until the fixed point over dataflows is reached, i.e., there are no new flows. Precisely, a single method is analyzed with respect to an *invocation context*. The context is a particular method and flows for all its arguments, i.e., argument expressions. When the single method analysis is done, the algorithm checks whether the flow data for a particular invocation context has changed. If so, the method, its direct callers, and callees are enqueued into the worklist to be processed in the next iterations. The algorithm terminates when the worklist is empty.

In the context of a single method analysis, the analysis keeps a set of *tracked* expressions. At start, only argument expressions are tracked. Next, if the interpreter encounters some assignment, the expression symbolizing its left-hand side is tracked and flows from the right-hand side of the assignment are *propagated* into the left-hand side expression. Method calls are handled similarly. If some method is invoked with, e.g., the first argument equal to the variable named `x` in the caller method, it is effectively an assignment where the left-hand side of the assignment is the first parameter of the callee method, and the right-hand side of the assignment is the variable named `x`. However, results of such assignments are stored not in the results of the caller method's analysis, but the invocation context of the callee method.

The mentioned method calls and assignments construct a *method summary*, i.e., the flow data for a particular invocation context. Method summaries are then used to get the final result of the analysis. The process of transformation of method summaries into the final graph is not-so-important aspect of our project.

2.1.2 Bytecode Scanner Plugins

The analysis can be extended by plugins, which is the way we shall implement a support for the Spring Framework. Two main concepts of plugins exist:

1. **Dataflow Plugins** – This kind of plugins directly contributes to the results of the analysis. The main idea is to supply some semantics to the handled methods, which the analysis would not be able to analyze with the interpreter. The most basic dataflow plugin in Bytecode Scanner is a plugin for Java Platform. For example, if there is some manipulation with I/O, the plugin familiarizes the analysis with this fact. In our context, methods handled by our extension for the Spring Framework will be those that are responsible for retrieving Spring Beans objects from the application context. Their integration with the core analysis is rather easy, instead of calling the bytecode interpreter for a single method analysis, the semantics provided by a plugin is used to handle a method call instead.

2. **Dependency Injection Plugins** – This kind of plugins indirectly contributes to the results of the analysis. The main idea is to extend a set of methods that could be possibly called by the analyzed method. This is especially important in the Spring Framework's analysis since it is necessary to pass the type information about injected instances into the core analysis. In other words, when obtaining some Spring Bean using an application context, then it is necessary to have the correct type information for it because usually some method will be further called upon the Spring Bean object and it is necessary to determine the target of such a call precisely. This kind of plugins existed on a conceptual level only, with no clear idea of how it should be designed, and there was no support for it previously by any means. Its realization is one of the topics of this thesis.

3. Technologies and Background

We do not start our solution from scratch, we have gained inspiration for many problems solving in several other articles and theses. In Section 3.1, we provide an overview of frameworks that are useful for obtaining information about bytecode, along with their relation to call graph construction. In Section 3.2, we describe several algorithms which can be used to build a call graph. In Section 3.3, we take a look at few dependency injection frameworks and what features they can offer.

3.1 Bytecode Inspection Frameworks

To construct a call graph, we need to obtain information about the types and methods declared in the analyzed program. Even though there is an abstraction in Bytecode Scanner which helps us to surpass the gap between our code and third-party libraries, we need at least a basic idea of what some bytecode inspection frameworks can offer.

T. J. Watson Libraries for Analysis

T. J. Watson Libraries for Analysis framework is better known under its acronym of WALA [4]. The framework provides static analysis capabilities for bytecode mainly. Not only it can parse bytecode and provide information about types and methods declared in the analysis scope, but it also provides a *class hierarchy*, a structure with relations between declared types, and *Pointer Analysis*, a very powerful tool suitable for call graph construction. The Pointer Analysis will be described in greater detail in Section 3.2.

The author of the thesis [5], which deals with static analysis of Java programs built on top of the WALA framework, describes the workflow of call graph construction in great detail. Based on information provided by the author, we can declare that the manipulation with framework seems to be very straightforward and could suit our purposes very well. However, there is one major downside with this framework, being the fact that it can only handle programs written in Java 8 or older. With new versions of Java being rolled out once in a while, this makes the framework technically unusable in a commercial environment where it is expected that Bytecode Scanner should analyze programs targeting newer versions of Java runtime as well. It is also worth mentioning that the call graph which WALA provides does not support dependency injection, the Spring Framework in particular.

ASM

ASM framework [6], similarly to the WALA framework, is a framework for bytecode inspection and analysis. Unlike the WALA framework, the ASM framework is more focused on performance. This fact makes the framework very suitable to be used in compilers, it is utilized in both Groovy and Kotlin compilers. This is also why this framework is well-maintained and supports newer versions of Java, in contrast to the WALA framework. However, the ASM framework has no explicit support for class hierarchy and call graph construction, and thus we would be supposed to implement this functionality on our own.

3.2 Call Graph Algorithms

The call graph is an oriented graph where the set of vertices is made of all methods declared in the analyzed program, and there is an edge between the nodes `A` and `B`, if the method represented by the node `A` calls the method represented by the node `B`. The call graph is a crucial structure in static data lineage analysis, it is necessary to analyze all methods reachable from the entry-point method. Note that the call graph is not necessarily a tree, a cycle can occur when having recursive method calls. Also note that the call graph can consist of multiple connected components, for example, if there are multiple entry-points in the analyzed program. In the thesis, if we talk about the call graph, we always refer to the connected component that contains the entry-point method.

The construction of the call graph seems to be quite straightforward, it could be sufficient to traverse methods' instructions from the entry-point method and whenever some instruction `invoke` is encountered, a new recursive branch of the same algorithm is started from the caller, i.e., the operand of the instruction `invoke`. However, in object-oriented languages, we usually work with interfaces or abstract classes, or generally with virtual methods. Therefore, determining the call targets of the instruction `invoke` is not that straightforward in the static analysis, since it cannot be precisely said of which type the receiver of the method call is. To resolve the problem in the static analysis, we *over-approximate* the results, meaning that we do not declare exactly which method would normally be called at a given call site. We work with a set of methods that could be called at the given call site instead. This set of methods is computed by a *call graph builder*, a component using heuristics to compute call targets. In this section, we present several call graph builders that can help us to build the call graph, each of them using different heuristics.

Pointer Analysis

The Pointer Analysis [7] aims to find all possible heap pointers which can the variable point to. Therefore, it should be easy to find possible call targets, by finding a method with the same name and signature in types which can receiver of the method call point to. However, the implementation of the analysis in terms of sufficient efficiency and reasonable performance is rather hard, although it has been proven that there exists a construction algorithm running in almost linear time [7].

Class Hierarchy Analysis

The Class Hierarchy Analysis [8] is implementation-wise very straightforward. It uses a combination of the statically declared type of receiver of a method call with the class hierarchy to compute a set of possible targets for a particular method call. This principle is considered the best one for heavy over-approximation, however, such kind of over-approximation is not always suitable to produce results that are precise enough. In the context of Java programs, consider some classes located near the root of the language type hierarchy, for instance, the class `java.lang.Object`. The class defines the methods `toString()`, `hashCode()` or `equals(Object)` which almost every class overrides. Therefore, when calling any of these methods, the Class Hierarchy Analysis would return a set of all overriding methods across the analyzed program as possible call targets. Note that the call graph construction algorithm is recursive, and these methods can potentially call many other methods. In other words, with the Class Hierarchy Analysis, the data lineage analysis might produce very imprecise results and would be unreasonably slow in terms of performance.

Rapid Type Analysis

The Rapid Type Analysis [9] uses the Class Hierarchy Analysis in its core. It works exactly the same as the Class Hierarchy Analysis, but filters a set of possible call targets to only those methods, whose declaring types are marked as instantiated. The Rapid Type Analysis itself does not specify how exactly a set of instantiated types should be created. There are several ways of how to create a set of instantiated types, for example, we can mark all types defined in specified application packages as instantiated. The algorithm presented by the authors, however, constructs this set by recursive traversal of methods' instructions, similarly to call graph construction, and marks the type as instantiated if there is the instruction `new` somewhere in the bytecode with an operand equal to that type [9]. The latter way is probably the best way of how to build a set of instantiated types.

3.3 Dependency Injection

Dependency injection is a programming technique that allows developers to specify references between instances of classes, these references are specified declaratively and the process works on the background. These references are called *dependencies*. The process of passing dependencies to instances of classes that they are dependent on is called *injection*. Without dependency injection, dependencies between objects are usually hard-coded, which in result makes the code less readable and less maintainable. Generally, there are several types of dependency injection:

- **Constructor injection** – Dependencies are injected into the instance of the class during its initialization, i.e., in the class' constructor. The dependencies are mapped to the constructor's parameters. The benefit of constructor injection is clear here, the object can become immutable immediately after instantiation.

- **Setter injection** – Dependencies are injected into the instance of the class by calling setter methods. As a general rule, these setter methods usually have a single parameter and they are mapped to respective fields, but it is not a strict requirement. Similarly to constructor injection, in setter injection, dependencies are mapped to parameters of the setter method.

However, classes utilizing setter injection usually cannot be immutable, in contrast to using constructor injection.

- **Field injection** – Dependencies are injected into the instance of the class by directly assigning a particular field with its dependency. This kind of injection is the least recommended one, because of the impossibility to achieve immutability of the class and tight coupling between the class and the dependency injection container.

In Java, there are several dependency injection frameworks that somehow manage the dependent objects. These objects are then called *managed objects*. In this section, we present the most popular ones.

Jakarta Enterprise Edition

Jakarta Enterprise Edition, formerly known as Java Platform Enterprise Edition, is a set of specifications defining features to be used within enterprise applications. In our context, we are particularly interested in Jakarta Contexts and Dependency Injection [10], also known as CDI. The CDI is described by JSR[1] 299, however, it also uses JSR 330 Dependency Injection for Java, which is very simplistic. It defines own few annotations from the package `javax.inject`.

The authors describe the specification as "a powerful set of complementary services that help to improve the structure of application code" [10]. The whole concept is built on top of Jakarta Enterprise Beans, objects managed by the dependency injection framework. Note that the CDI is only a standard, there are several implementations of the standard, including those from Oracle or IBM.

```
@Named("KITT")
@RequestScoped
class Car {
    @Inject
    private Engine engine;
    ...
}

@Named
@RequestScoped
class Engine {
    ...
}
```

Listing 3.1: A basic example of field injection in CDI

In Listing 3.1, a basic example of field injection in the CDI can be observed. Here, the dependency injection framework instantiates the class `Engine` at first. It is therefore considered as an object managed by the framework, or Jakarta Enterprise Bean. The class `Car` is instantiated similarly, however, the class defines a field annotated with the annotation `@Inject`, saying that some dependency should be injected into the field. The framework, therefore, scans a pool of

[1] Java Specification Request

managed objects and finds those which match the field's type. There should always be a single object of such type, which is this case as well. However, it can easily happen that there will be multiple managed objects of the same type, and therefore it is also possible to specify some requirements for the injection. These requirements are usually called *qualifiers*, and only such managed object which satisfies the qualifiers is *qualified to be injected.* The most basic qualifiers matching is by name of Jakarta Enterprise Bean, which is defined by the value of the annotation `@Named`. If the value is empty, the name of the class is considered a default name. Note that constructor and setter injection work similarly, the annotation `@Inject` is simply defined on constructor, setter method respectively. Also note that the annotations `@Named` and `@Inject` are defined by JSR 330.

The example with cars and engines was chosen not only because it is easily understandable for beginners, but it also nicely demonstrates features of dependency injection generally, not only in Jakarta Enterprise Edition. In the car industry, cars are being manufactured concerning some configurations. Without loss of generality, suppose that the only customizable configuration is a selection of the engine. Usually, a car model comes with multiple engines that the customer can choose from. Except for the engine, the car model is always the same. When mapped to dependency injection, there would be multiple Jakarta Enterprise Beans of the type `Engine`, each of them having qualifiers describing the features of the engine. When constructing an instance of the class `Car`, based on defined qualifiers, we can decide which particular managed object of the type `Engine` would be injected. Except for injected instance of the type `Engine`, everything else would remain the same.

Of course, there are many more use cases for dependency injection. In the enterprise environment, dependency injection is especially useful when distinguishing between development, test, or production environment.

Spring Framework

The Spring Framework [1] provides a comprehensive programming model for enterprise applications targeting Java Platform. It provides not only features related to dependency injection, very similar to Jakarta Enterprise Edition, but also many more features which simplify development, like powerful expression language called SpEL, extensive support for data accesses, or a convenient test environment. However, it is important to say that the Spring Framework is not an implementation of Jakarta Enterprise Edition, but is considered a more lightweight container. In other words, a developer does not need the whole Spring Framework to use only a part of it, for example, he can use Spring Web MVC, a part of the framework suitable for building applications with Model-View-Controller architecture, without Spring Data JDBC, a part of the framework dealing with accesses to the database.

```
@Component
@Qualifier("KITT")
class Car {
    @Autowired
    private Engine engine;
    ...
}

@Component
class Engine {
    ...
}
```

Listing 3.2: An example from Listing 3.1 in the Spring Framework's representation

In Listing 3.2, we introduce exactly the same example as in Listing 3.1, but in Spring Framework's representation. Note that even though the Spring Framework defines its own set of annotations, it also has support for annotations specified by JSR 330. Therefore, if we replaced the annotation `@Autowired` by the annotation `@Inject` and the annotation `@Qualifier` by the annotation `@Named`, it would work exactly the same even in the Spring Framework.

In the Spring Framework, objects managed by the framework are called Spring Beans. We analyze the Spring Framework in bigger detail in Section 4.1.

Spring Boot

Spring Boot [11] is nothing but an extension of the Spring Framework, eliminating boilerplate configurations required to set up the application managed by the Spring Framework. Therefore, the development of applications built on top of the Spring Boot is even faster, more convenient, and more efficient, opposed to using the Spring Framework.

4. Requirements and Analysis

In this chapter, we discuss requirements on the solution, based on which we carefully analyze the problem. As the solution itself can be technically divided into two parts, we do the same in this chapter. First, we introduce requirements on the data lineage analysis of applications that use the Spring Framework, its features related to dependency injection in particular. Then, we analyze a way of how to construct a call graph in the context of dependency injection, where assignments of instances of implementation classes to respective variables of interface types are not hardcoded in the source code. Our proposed solution has the form of a plugin for the data lineage analyzer, which handles the usage of the Spring Framework by the subject application.

4.1 Spring Framework

The Spring Framework has already been briefly introduced in Section 3.3. We have chosen the Spring Framework as the main target for our work on extending the data lineage platform with support for dependency injection. The Spring Framework is probably the most popular dependency injection framework targeting Java Platform. It features all the important aspects of dependency injection, and therefore all the principles described in our thesis apply not only to the Spring Framework but generally to any dependency injection framework targeting any platform.

In this section, we discuss dependency-injection-related features of the Spring Framework in a detail. The text will be structured in such a way, that we first describe a particular Spring Framework's feature, and then we discuss what are the requirements and how to design a support for data lineage analysis of such feature.

The basic entities in the Spring Framework are Spring Beans. From now on, we will refer to them simply as *beans*. The bean is an object managed by the Spring Framework. From our perspective, there are two important views of the beans:

- **Spring Bean Definition** – Definition is a static concept describing properties or configuration of some bean. The framework can instantiate and manage the bean based on the configuration. We describe definitions in Section 4.1.1.

- **Spring Bean Flow Data** – Flow data are specific for our project. Flow data represent the state of some bean. At runtime, such a state is defined by the object which is managed by the framework. The flow data for a particular bean can be constructed based on the definition of the bean. We describe flow data in Section 4.1.2.

4.1.1 Spring Beans Definition

Beans are defined within a configuration of the *application context* which holds information about beans in the program. From the developer's perspective, the application context is a place where he can configure beans to be used within his program. In Section 4.1.1.1, we describe configurable properties of beans. This description is rather abstract, but in Section 4.1.1.2, we describe a way of how to configure the application context which goes along with practical examples of beans configuration.

4.1.1.1 Spring Bean Properties

Every single bean has multiple configurable properties which we discuss in this section. For each property, we describe its meaning. Then, we provide our requirements on the data lineage analysis with respect to the property configuration. Last, we propose an approach of how the data lineage analysis should work, based on the property configuration.

Profile

- **Definition.** A bean can be assigned with a profile name or multiple names which is active in. It can be either a plain profile name, like `myProfile`, or profile expression, like `(thisProfile | thatProfile) & !otherProfile`. If none profile is assigned, the bean is automatically assigned with the profile `default`. Active profiles can be specified at various places, for example, in Maven scripts or at the command line. If no active profile is selected, the profile `default` is considered as the only active one.

- **Requirements.** As mentioned, active profiles can be specified at various places. In the data lineage analysis, it is generally not possible to cover all these places. As a fault-tolerance requirement, the data lineage analysis should not fail if there is some problem with profiles that could not otherwise happen at runtime. It should log a warning instead.

- **Approach.** We introduce a configurable field in Bytecode Scanner configuration where the customer can specify active profiles manually. If the field is empty, all profiles that appear in the application context are considered active ones. This approach has several consequences. The major one being the fact that instead of working with a single bean matching some requirements, we should work with multiple beans. This would normally cause an exception at runtime. Nevertheless, if some problem related to multiple active profiles should occur, a warning should be logged and the customer should be asked to configure the active profiles, to make the data lineage analysis more precise.

Name

- **Definition.** A bean can be assigned with a unique name, sometimes called an identifier. The name can be further used in the program to obtain the object of bean. It is also possible to specify aliases of names.

- **Requirements.** Both simple names and aliases shall be supported as they are one of the most important aspects of bean configuration.

- **Approach.** In our representation of application context, we define a map from name to a set of definitions matching the name. The set is introduced due to already described support for profiles. Aliases are handled similarly, multiple same values can appear in the map, but with a different key.

Type

- **Definition.** Each bean has some type, this type refers to a class type of the object managed by the Spring Framework. The type can be defined either explicitly, or implicitly.
- **Requirements.** Type shall be supported as it is one of the most important aspects of bean configuration and the key aspect of dependency injection itself.
- **Approach.** If defined explicitly, we use the explicitly configured type. Otherwise, we deduce the actual type from the bean configuration. This deduction is based on other properties of a particular bean definition.

Qualifiers

- **Definition.** A qualifier is a mapping from some keys to respective values. A single bean can have none or multiple qualifiers. Qualifiers are used to distinguish between multiple beans of the same type, where differentiation based on name only is not applicable. At a point of injection, qualifiers matching the qualifiers of bean which we would like to inject shall be specified as well. During a process called *finding of appropriate autowire candidates*, only those beans which match qualifiers defined on the injection point are considered.
- **Requirements.** The only qualifier pre-defined by the Spring Framework is a mapping consisting of a single key-value pair, where the user can map the pre-defined key to a string value. However, the Spring Framework also supports custom qualifiers, where the mapping can be customized to contain any key-value pairs, based on the configuration. We shall support both concepts.
- **Approach.** We don't have to distinguish between the pre-defined qualifier and custom ones. We can handle them both in the same way. All we have to do is to parse Spring Framework's representation of qualifiers into a simple map, representing qualifiers. When finding appropriate autowire candidates, the plugin tests these key-value pairs defined on the bean and key-value pairs defined at the point of injection on equality.

Primary

- **Definition.** A bean can be declared as a primary bean of its class type. At a point of injection, if there are multiple beans of the same type, the primary one will be injected, if there is any. By default, beans are non-primary.

- **Requirements.** Primary beans shall be supported, filter of primary beans should be applied wherever applicable.
- **Approach.** When finding appropriate autowire candidates, the found candidates are filtered to primary beans only. If there is no primary bean among the candidates, this filter is not applied.

Scope

- **Definition.** A bean is always assigned with some scope, which refers to a way of how the bean can be obtained from the application. The Spring Framework defines two scopes:
  - Singleton – A single bean definition refers to a single instance, i.e., a single object managed by the Spring Framework. Whenever such bean is accessed, the framework always returns the same instance.
  - Prototype – A single bean definition can refer to multiple instances, i.e., multiple objects managed by the Spring Framework. Whenever such bean is accessed, the framework always creates a new instance and returns it.
- **Requirements.** We claim no requirements on the scope since it technically has no impact on the data lineage of the analyzed application.
- **Approach.** We can parse the scope configuration of bean, however, it has no real usage. Whenever some bean is accessed in the analyzed application, the flow data for such pre-analyzed bean are returned. This refers to prototype scope. To achieve the functionality of singleton scope, after every single method analysis, it would be required to track the bean and propagate new flow data to our representation of application context where the plugin stores flow data of beans. This operation might be performance-wise unfeasible and would slow down the analysis which is already quite computational demanding. The approach is discussed in greater detail in Section 8.2.

Initialization Method Arguments

- **Definition.** Initialization method arguments refer to parameters' values that should be used during bean instantiation, i.e., arguments of the initialization method, like a constructor. The value can be either simple, or it can be a reference to another bean.
- **Requirements.** Initialization method arguments shall be supported, additionally, we need to find appropriate overload of the initialization method based on these arguments' types and their count.
- **Approach.** The process of finding an appropriate overload of the initialization method is described in Section 4.1.1.2 since the process is influenced by other properties of the bean as well. In terms of flow data, the flow data of an initialization method argument has to be correctly injected into the respective parameter. This is described in Section 4.1.2.1.

Factory Method

- **Definition.** Instead of being instantiated by a constructor, a bean can define a factory method that is used to instantiate the bean. The factory method must be declared in the class type of the bean and shall be static. The return value of such a factory method is the instantiated bean.
- **Requirements.** A factory method of the bean shall be supported, however, its appropriate overload should be looked-up correctly.
- **Approach.** Factory methods are similar to constructors, the only difference in static analysis is that the factory method stores flow data of the bean being instantiated into an object which is returned by the method, whilst the constructor stores them into the receiver of the constructor call. The process of finding appropriate overload applies to constructors as well, the appropriate overload is looked-up based on the initialization method arguments and is further described in Section 4.1.1.2.

Factory Spring Bean

- **Definition.** Factory bean refers to abstract factory pattern and goes in conjunction with factory method of the bean. In the Spring Framework, the factory bean can be defined and such bean refers to an object to call factory method upon. The factory method call will then instantiate the defining bean. Note that in such a case, the factory method does not have to be static.
- **Requirements.** Factory beans shall be supported. Since a bean which defines the factory bean does not have to define its class type explicitly, we should support type deduction in these cases.
- **Approach.** The concept of factory beans is important for us especially in terms of searching initialization method and injecting flow data of factory bean into the receiver of the factory method call. The process of flow data injection is described in Section 4.1.2.1. Deduction of class type is rather easy. If there is some factory bean defined, we declare a return type of the factory method as the class type of the defining bean. However, such factory method must be looked-up at first. This process is described in Section 4.1.1.2.

Autowire Mode

- **Definition.** Autowire mode refers to how a bean is instantiated, there are several autowire modes:
  - Default – This is a standard and default in the Spring Framework with no magical autowiring, the initialization method arguments are used to instantiate the bean.
  - By name – Properties of the bean, i.e., setters, will be injected by searching for a bean with the same name. These found beans will be further used to call particular setters with arguments of these beans.

  - By type – Properties of the bean, i.e., setters, will be injected by searching for a bean with the same type. These found beans will be further used to call particular setters with arguments of these beans.
  - Constructor – The bean will be instantiated by a particular constructor, the Spring Framework will search for beans that could be injected into the arguments of the constructor.
  - Autodetect – Autowiring by constructor will be tried at first, if autowiring using constructor cannot be done, autowiring by type is utilized instead.

- **Requirements.** All the autowire modes shall be supported. One special note goes to constructor autowire mode, where a constructor to be used can be either marked by a special annotation, or a single constructor in the class is used instead. If there is neither a single constructor in the class[1], nor a constructor marked by a special annotation, an error should be raised.

- **Approach.** The whole concept of autowire mode is nothing but finding appropriate initialization methods which can initialize bean or setters that should be called immediately after the initialization of the bean. Autowire mode is one of many factors that can influence flow data whose computation is described in Section 4.1.2.1.

Initialization Callbacks

- **Definition.** Initialization callbacks are methods that accept no arguments and have no return value. These methods can be configured to be called immediately after an instantiation of the defining bean.

- **Requirements.** Initialization callbacks shall be supported, they can be specified in several different ways and we shall support all of them:
  - Explicitly by configuring method name of the initialization callback.
  - Implicitly by marking initialization callback method by a special annotation.
  - By implementing the interface `InitializingBean` which defines a single method. This method refers to an initialization callback.

- **Approach.** As the last step in the initialization of the bean, we extend the flow data of bean by flow data computed by these initialization callbacks. The process of computing flow data is described in Section 4.1.2.1.

Destruction Callbacks

- **Definition.** Destruction callbacks are methods that accept no arguments and have no return value. These methods can be configured to be called just before a destruction of the defining bean.

[1]This means more like *at most* one constructor from a developer's perspective, however, a default constructor is always there if not specified otherwise, and therefore we can refer to a single constructor.

- **Requirements.** Destruction callbacks are not supported, we assume that these methods propagate no information which could be important for the data lineage.
- **Approach.** There is no correct way of how to handle destruction callbacks. At runtime, these callbacks are called just before a garbage collector calls a finalizer upon them. However, in the data lineage analysis which is static, there is no garbage collector. The data lineage analysis does not model neither dynamic heap, nor dynamic heap objects. In fact, data accesses are represented by symbolic expressions. Therefore, we are unable to determine a point in the program where the method should be called.

Property

- **Definition.** A property of some bean refers to a setter that should be called during an initialization of the bean. In property definition, the actual argument of the setter invocation is expected to be specified. Similarly to the initialization method argument, either plain string value or reference to another bean can be specified.
- **Requirements.** Properties shall be initialized right after the instantiation of the bean. Properties can be defined either explicitly, or implicitly by marking setter methods by a special annotation. We shall support both.
- **Approach.** First, we need to resolve setter methods based on the names of properties that the bean defines. Then, we can analyze these setters with respect to flow data of the defining bean, as described in Section 4.1.2.1.

Abstract

- **Definition.** A bean can be marked as abstract which means that it should not be managed by the Spring Framework. Beans are non-abstract by default.
- **Requirements.** We shall support abstract beans in terms of parsing these bean definitions.
- **Approach.** No analysis of abstract beans is required, as these are never instantiated by the Spring Framework itself.

Parent bean

- **Definition.** A bean can be assigned with a parent bean which is some abstract bean. The defining bean inherits configuration from its parent, initialization method arguments, initialization callbacks or properties in particular, but also factory method or factory bean. There are two types of configurable fields in bean definition, each having different semantics when inheriting from the parent bean:
  - Single value – Among single values are, e.g., factory method or factory bean. These values can be inherited from parent bean, but only when the defining bean does not override these values.

   - A collection of values – Among collections of values are, e.g., properties or initialization method arguments. The defining bean always inherits these values from its parent.

- **Requirements.** Inheritance of beans shall be supported.

- **Approach.** Inherited values can be treated in the same manner like they were defined by the defining bean.

4.1.1.2 Spring Bean Configuration

Beans can be defined at several places, the most common ones being XML documents and code.

Spring Beans in XML Documents

Application contexts defined by XML documents are the most classic way of how to configure beans.

In Listing 4.1, an empty application context can be observed. First, the element `beans` ia a root element of the XML document. The application context is valid only if the document conforms to XML Schema defined by the root element. Next, we can see definitions of multiple namespaces, the namespaces `util` and `context` namely. One of our requirements is to support not only all the elements in the schema called `beans`, but also some elements defined in the two previously mentioned namespaces. In particular, we are interested in the elements `context:component-scan`, `context:property-placeholder`, `util:map`, `util:set`, `util:properties` and `util:list`. In the example, we can also observe that the profile named `myProfile` is defined, which means that this particular profile is active in all beans defined under the particular element `beans`. Note that the element `beans` can be defined recursively.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..."
       xmlns:xsi="..."
       xmlns:util="..."
       xmlns:context="..."
       xsi:schemaLocation="..."
       profile="myProfile"/>
```

Listing 4.1: An empty application context

A single bean is defined by the element `bean`, as can be seen in Listing 4.2. In this particular example, bean with the name `kitt` will be instantiated in such a way that it will use bean named `carFactory` with the method `makeSuperCar`. The return value of such method is a result of the bean creation, and it is an object of the type `Car`. The method accepts a single parameter of the type `Engine`, which is defined by the element `constructor-arg`. This refers to constructor injection, if no factory method was specified, a constructor matching the method descriptor would be used instead. In data lineage analysis, we do not mind whether we work with constructor or factory method. All we have to do is to find appropriate initialization method. The algorithm of finding such method works as follows:

1. Is there any factory bean defined? If so, consider this particular bean as a reference one. Otherwise, consider the current bean as a reference one.

2. Is there any factory method defined? If so, consider all methods defined in the bean from the previous step matching the name. Otherwise, consider all constructors defined in the current bean.

3. Filter methods from the previous step to those matching parameters' types by comparing them to types of the elements `constructor-arg` with respect to the ordering of the elements.

4. The initialization method is returned.

The value of the element `constructor-arg` can be of multiple types, similar holds for the element `property` in Listing 4.3:

- **Plain string value** – This can be either string literal, property variable placeholder, or SpEL expression. One of our requirements is to support both string literals and property variable placeholders, SpEL expressions are not required to be supported. A support for SpEL expressions is considered a future work and is discussed in Section 8.2.

  Support for string literals is rather straightforward. For property variable placeholders, we introduce a global variable that defines a mapping between placeholders and actual values. This mapping is pre-filled by the customer's provided files with the extension `.properties`.

  The process of resolving string value to a final string works as follows:

  1. Property variable placeholders are expanded iteratively. Note that the value of a placeholder can refer to some other placeholder, and therefore the expansion should work recursively. However, this is rather easy to achieve. If we detect a cycle, we should raise an error that the string could not be evaluated correctly.
  2. SpEL expressions are evaluated iteratively. We don't support SpEL expressions, and therefore we should only log a warning if we encounter some expression that cannot be evaluated.
  3. Final string is returned.

  The Spring Framework also has a support for type conversion. As the most basic example, a simple string literal can be mapped to a parameter of the type `int`. Type conversion should be supported in our project as well. We claim no requirements on type conversion, however, the most common conversions should be supported and it should be easily extensible, so new conversion services can be eventually supplied in the future.

- **Reference to another bean** – A name of referenced bean is provided, as in Listing 4.2.

- **Bean in-place configuration** – This applies not only to the element `bean`, but the elements `util:list`, `util:map`, `util:set`, `util:array` or `util:props` are supported as well.

```
<beans>
  <bean name="kitt"
        class="com.company.Car"
        scope="prototype"
        factory-bean="carFactory"
        factory-method="makeSuperCar">
          <constructor-arg ref="superEngine"/>
  </bean>

  <bean name="carFactory"
        class="com.company.Factory"/>

  <bean name="superEngine"
        class="com.commpany.Engine"
        scope="prototype"/>
</beans>
```

Listing 4.2: A simple example of XML configuration of a bean

Last, in Listing 4.3, we provide an example with parent bean and the element `property`, which refers to setter injection. In this particular example, there are two abstract beans, both have the name `abstractBean`. However, each is defined in a different profile. Note that abstract bean is somewhat different from the abstract class. Abstract bean does not even have to define its type, it is more like a logical concept. Bean with the name `myBean` inherits everything from the specified parent bean, which are properties in this case. The element `property` itself defines a name of the property. It is expected by the Spring Framework that in declaring class, there will be a matching setter, i.e., a method with the signature `void setA(String)`, in this case, a method with the signature `void setB(String)` respectively. The particular setter with the specified argument is then called during bean creation.

Now, let's move back to our design of support for profiles. The bean named `myBean` specifies the name of the parent bean, however, if no profile was active, this bean would technically have two parents. This is where the problem arises, since the class `C` does not define a method with the signature `void setB(String)`, as defined by the bean named `abstractBean` in the profile named `B`. Therefore, it is an invalid configuration. To prevent problems with invalid configuration, due to our design of support for profiles, we always enforce bean to have at most one parent. If more parents due to multiple active profiles should occur, we shall stop the analysis and force the customer to configure profiles.

```
<beans>
  <beans profile="A">
    <bean name="abstractBean" abstract="true">
      <property name="A" value="a"/>
    </bean>
  </beans>

  <beans profile="B">
```

```
    <bean name="abstractBean" abstract="true">
      <property name="B" value="b"/>
    </bean>
  </beans>

  <beans profile="default">
    <bean name="myBean"
          class="com.company.C"
          init-method="myInit"
          parent="abstractBean"/>
  </beans>
</beans>

class C {
    void setA(String a) { ... }
    void myInit() { ... }
}
```

Listing 4.3: A simple example of bean inheritance

Spring Beans in Code

Although definitions of beans in XML documents are considered a gold standard, beans defined in code are gaining more popularity in recent years. They are most commonly configured using Spring Framework's annotations. These annotations have their retention policies set to runtime, and therefore are recognizable by the Spring Framework at runtime during application context initialization.

There are altogether three ways of how to configure beans in code and we should support all of them:

- **Configuration classes** – As the name suggests, these beans are used to configure the application context. These classes must have the annotation `@Configuration`. The two most important annotations except for the annotation `@Configuration` are the following:
  - The annotation `@PropertySource(s)` is an equivalent to the element `context:property-placeholder` in XML documents. Their values refer to locations of files with the extension `.properties` which are used when resolving property variable placeholders. If none property source is specified, all the files with the extension `.properties` in resources are utilized instead. This Spring Framework's behavior is utilized in our solution as well.
  - The annotation `@ComponentScan(s)` is an equivalent to the element `context:component-scan` in XML documents. This refers to a *component scan*, a process during which beans are being looked-up and registered into the context. The process itself is decribed later on in this section.
- **Bean methods** – These methods must be defined in classes annotated by the annotation `@Configuration` and the methods by themselves must be

annotated by the annotation `@Bean`. Return values of these methods are treated as beans. These methods can define parameters. During an initializing call, beans matching types of these parameters will be injected into arguments of the initializing call. This refers to a process of finding appropriate autowire candidates, which is described in Section 4.1.2.1. These methods can be either static or instance. In the case of the instance method, the defining bean annotated by the annotation `@Configuration` is utilized as a receiver of the initializing call.

- **Component classes** – These classes are being discovered during the component scan. Technically, it can be an arbitrary class. When discovered by the component scan, such a class is instantiated with autowire mode set to the constructor. Then, it is considered a bean.

All of these beans defined in code can be annotated by the annotations like `@Scope`, `@Profile`, `@Primary` or `@Qualifier` whose meaning should be obvious. We support all the standardly used annotations defined by the Spring Framework, but also those defined by JSR-330.

In Listing 4.4, there is an example of simple beans configuration in code. We can observe the class `DatabaseConnectionConfiguration` annotated by the annotation `@Configuration`. This class is considered a bean. Next, there is the field named `credentials` in the class, annotated by the annotation `@Autowired`. The annotation says that during the initialization process of the bean named `DatabaseConnectionConfiguration`, some bean of the type `Credentials` with the qualifier `value:myCredentials` will be injected into the field. This refers to field injection. The annotation `@Inject` is an alternative to the annotation `@Autowired`, similarly the annotation `@Resource` where it is possible to also specify a name of the bean to be injected. We shall support all of these annotations. Apart from that, the annotation `@Value` declared on the field named `connectionString` is a stronger variant of autowiring, saying that a specified string literal will be injected into the field. An expansion of property variable placeholders is supported here as well, as explained in Section 4.1.1.2. Similar holds for type conversion.

The method `getConnection(String)` in Listing 4.4 is another example of definition of a bean, by using the annotation `@Bean` in this particular case. Here, we can notice that the annotation `@Value` can also be declared on parameters with the same semantics when defined on fields. If the first parameter was not annotated by the annotation `@Value`, it would work exactly the same as for the fields annotated by the annotation `@Autowired`. However, for parameters, there is no need of specifying neither of dependency injection annotations[2]. Also note that the annotation `@Value` can also be defined on methods. In such a case, its value is used as a default value for all the parameters that are not annotated by the annotation `@Value`. If the bean of the type `Connection` is accessed somewhere in the analyzed program, we shall simulate this whole Spring Framework's machinery to obtain correct flow data as computed by a call of the method `Connections.initialize(String, String, String, String)`.

[2]The annotations `@Autowired`, `@Inject`, or `@Resource`

```
@Configuration
public class DatabaseConnectionConfiguration {

    @Value("${database.connection}")
    private String connectionString;

    @Autowired
    @Qualifier("myCredentials")
    private Credentials credentials;

    @Bean
    public Connection getConnection(
        @Value("${database.schema}") String schema) {
        return Connections.initialize(connectionString,
             credentials.user, credentials.pw, schema);
    }
}
```

Listing 4.4: An example of usage of beans defined in code

Last, in Listing 4.5, we can observe an example of all three types of injection that can be realized within the Spring Framework:

- Field injection refers to fields annotated by the annotation `@Autowired` or similar, as described in the previous paragraphs.
- Setter injection refers to methods annotated by the annotation `@Autowired` or similar, this is equivalent to properties defined in XML documents.
- Constructor injection refers to constructors annotated by the annotation `@Autowired` or similar, autowire mode is set to the constructor.

In this particular case, the class `MyBean` is scanned during the component scan and registered as a bean. All these injections get processed during the initialization of the bean. In our environment, this refers to the computation of flow data for a particular bean which is described in Section 4.1.2.1.

```
@Component
public class MyBean {

    private final MyValue constructorInjection;

    private MyValue setterInjection;

    @Autowired
    private MyValue fieldInjection;

    @Autowired
    public MyBean(MyValue value) {
        this.constructorInjection = value;
    }
    @Autowired
    public void setProperty(MyValue value) {
        this.setterInjection = value;
    }
}
```

Listing 4.5: An example of all types of injections in the Spring Framework

Component Scan

The component scan is basically an iteration through classes in the specified package and its sub-packages with some filtering, the result of which is a set of classes that are further registered as beans. There are several options that can be defined for component scan:

- It is possible to turn off the default filter which is by default turned on. The default filter looks for all classes that defines some of the meta-annotations `@Component`, `@ManagedBean`, or `@Named`.
- An include filter can be specified, as the name suggests, it includes all classes matching the specified condition.
- An exclude filter can be specified, as the name suggests, it excludes all classes matching the specified condition.

The condition in both include and exclude filters is specified using some kind of predicate. However, no arbitrary predicate can be chosen. There are few types of predicates supported by the Spring Framework:

- Regex filter that matches the specified regular expression, the fully qualified name of the tested class is then matched with such a regular expression. We shall support this type of filter.
- Annotation type filter that checks whether the tested class contains the specified annotation as a declared annotation or meta-annotation. We shall support this type of filter.
- Assignable type that checks whether the specified type is assignable from the tested class. We shall support this type of filter.
- AspectJ type filter that uses AspectJ expression to test the class. This type is not required to be supported and is considered a future work.
- Custom type filter implementation that uses custom code segment to test the class. This type is not required to be supported and is considered a future work.

Component scan can be defined in two different ways, by the annotation `@ComponentScan` or by the element `context:component-scan`. In Listing 4.6, an example of the configuration by the annotation `@ComponentScan` can be observed. In Listing 4.7, there is an equivalent example but defined by the element `context:component-scan`. However, the component scan itself has the same semantics in both cases. Therefore only what differs is the actual parsing of the component scan definition.

```
@ComponentScan(
  basePackages = "com.company"
  useDefaultFilters = false,
  includeFilters = {
    @ComponentScan.Filter(
      type = FilterType.ASSIGNABLE_TYPE,
      value = Annotation.class),
    @ComponentScan.Filter(
      type = FilterType.ANNOTATION,
      value = Interface.class)
  }
)
@Configuration
class ConfigurationClass {}
```

Listing 4.6: An example of the annotation `@ComponentScan`

```
<context:component-scan
    base-package="com.company"
    use-default-filters="false">
    <context:include-filter
         type="annotation"
         expression="com.company.Annotation"/>
    <context:include-filter
         type="assignable"
         expression="com.company.Interface"/>
</context:component-scan>
```

Listing 4.7: An example of the element `context:component-scan`

Our requirement is to support both styles of definition by introducing particular parsers. Then, we should be able to scan the analyzed code and register scanned classes as beans, which copies a behavior of the Spring Framework.

4.1.2 Spring Beans Flow Data

Flow data are specific for our project. At runtime, there is a heap and beans are nothing but ones of many objects on the heap. However, the data lineage analysis does not simulate the heap. Actually, data accesses are realised by symbolic expressions. There are three basic types of expressions:

- Local variable expressions – These expressions are always defined within some method and cannot cross method boundaries.

- Argument expressions – In fact, argument expressions are treated in the same manner like local variable expressions.

- Field accesses expressions – Each field access expression consists of a *target object* and the field being accessed. Target object should always refer to another field access expression, local variable expression or argument expression. For example, for the field access `a.b.c`, the target object is the field access `a.b` and the accessed field is the field `c`. Note that a receiver for a method call, i.e., the variable `this`, is realised by the zeroth argument expression. Unlike local variable expressions, field access expressions can cross method boundaries, in such a way that the root target object is replaced by some other root target object. For example, consider the method call `foo(a.b)` where the target object `a` is some local variable defined in caller method. When processing such call, the root target object, which is the local variable `a` in this case, is replaced by the first argument expression for the callee method, and propagated into the call. Therefore, in callee method, the expression `arg1.b` is tracked and contains exactly the same flow data as the field access `a.b` in caller method.

The propagation of flow data across method boundaries is especially important to know in context of beans flow data, since these root target objects are being replaced during injection of bean into some field of argument.

In Listing 4.8, we present a simple example of beans configuration. In the following enumeration, we explain what are those *flow data* for the two beans in the example that the Spring Framework plugin should compute:

- The bean of the type `Injected` contains a single tracked expression:
    1. The field access expression `Injected.value` – The expression is assigned with a console flow which denotes that there is some value which was read from the console.
- The bean of the type `Configuration` contains two tracked expressions:
    1. The field access expression `Configuration.injected.value` – The expression contains exactly the same flow as the field access expression `Injected.value` in the flow data of the bean of the type `Injected`.
    2. The field access expression `Configuration.stream` – The expression is assigned with a file stream flow which denotes that there is some opened file stream. In this particular case, the file stream flow refers to the location `path/to/file.txt`.

```
@Configuration
class Configuration {

    private Injected injected;

    @Value("path/to/file.txt")
    private InputStream stream;

    @Autowired
    void setInjected(Injected Injected) {
        this.Injected = Injected;
    }
}

@Component
class Injected {
    private final String value;

    public Injected() {
        this.value = System.in.readline();
    }
}
```

Listing 4.8: An example of flow data of beans

4.1.2.1 Computing Spring Beans Flow Data

The analysis computes flow data across the program by analyzing methods one by one. For us, the important methods are initialization methods, initialization callbacks or setters defined by some bean. Therefore, we shall be able to identify them and analyze them. However, we are not interested in flow data of the methods as a whole, which usually contain also some local variables which we do not care about at all. We are interested in flow data of analyzed bean only, these are usually field accesses associated with a receiver of the method call, in case of constructors, initialization callbacks or setters, or expressions related to return value of analyzed method, in case of factory methods or methods annotated by the annotation `@Bean`. Therefore, we analyze these important methods in some ordering and we pass the flow data for bean computed by some method to a receiver of the call of the following method, because the following method can refer to flow data computed by the previous method. The ordering is the following:

1. Initialization method – Based on the bean configuration, it can be either constructor, factory method or method annotated by the annotation `@Bean`.
2. Setters – Based on the bean configuration, they can be properties defined in XML documents, or methods defined in declaring type which are annotated by the annotation `@Autowired` or similar.
3. Initialization callbacks – Based on the bean configuration, they can be explicitly configured initialization callbacks, the methods in the declaring type annotated by the annotation `@PostConstruct`, or the method `afterPropertiesSet()` defined by the interface `InitializingBean`.

This refers to initialization process of a single bean. Last, we add flow data which are created due to fields defined in declaring type annotated by the annotation `@Autowired` or similar.

This process is recursive, if some bean references another bean, the referenced bean needs to be analyzed beforehand so it can be injected. References are either explicit, or implicit, the implicit references are being resolved during a process of finding appropriate autowire candidates which is further described in Section 5.2.2. Note that references can be cyclic, and therefore our requirement is to design a solution which should be robust enough so that the initialization process does not end up in an infinite cycle.

It is important to say that neither initialization method, nor initialization callbacks, nor setters appear in the call graph. Therefore, we are supposed to analyze them in advance so we can access them once needed during the program's analysis. We considered two approaches:

- **Modifying call graph by adding auxiliary node for the Spring Framework's initialization** – This approach is based on modifying the already existing call graph. A root of each call graph is the entry-point of the analyzed application, in this approach, we would create an auxiliary node for Spring Framework's initialization. This node would be called before the actual entry-point and we would add an edge between this node and the entry-point, symbolizing that our auxiliary node calls the entry-point. From such node, initialization methods of all defined beans would be called, i.e., initialization methods, initialization callbacks or setters. The advantage here should be clear, if we constructed such a node correctly, everything would be analyzed implicitly within the analysis run, even cyclic dependencies.

  However, construction of such a node might be complicated, in this paragraph, we describe one of the reasons and proposed solutions. Consider some bean named `A` that has a dependency on bean named `B` in its constructor. In such a case, we shall incorporate such dependency into the graph. The first step would be to add an auxiliary edge between the initialization method of the bean named `A` and the initialization method of the bean named `B` into the graph. In the second step, the problem arises, because we shall be able to propagate flow data of the bean named `B` into an argument of the initialization method call of the bean named `A`. In the analysis, this only happens if there is some instruction `invoke` in bytecode. At this point, there are two proposed solutions:

  1. Modify the initialization method of the bean named `A` to call the initialization method of the bean named `B` with correct arguments. As arguments for the call are expected to appear on the operand stack, it would be necessary to push there these operands beforehand. Another step would be to assign the return value of the instruction `invoke` into a particular argument of the initialization method of the bean named `A`. In other words, a non-trivially amount of generated bytecode would be needed.
  2. Modify the analysis somehow so it can propagate flow data not only if it encounters some instruction `invoke`. This could be achieved by introducing some callback into the core analysis algorithm that would do the trick. Compared to the previous proposal, this one is preferred, and should be less invasive.

Either way, this approach would cause that all defined beans would be initialized which might be performance-demanding since not all the defined beans are necessarily used in the analyzed application. Therefore, we would prefer a solution where the initialization of a bean can be performed lazily.

- **Calling the data lineage analyzer as a service to compute flow data for a bean** – For invocation of some bean method[3], we would need some black-box that computes its flow data, analyzing its whole call tree, since the method associated with the invocation can call some other methods, which can call other methods and so on. That sounds much simpler, in contrast to the previous approach. The disadvantage here being the fact that we should do all the propagations manually, e.g., if the bean named `A` has dependency on the bean named `B`, we should call our black-box for the bean named `B` initialization method at first, and then inject results into an appropriate argument in initialization method of the bean named `A`. However, such a disadvantage is quite acceptable and this approach can also be implemented in a lazy manner, and therefore we have decided to choose this approach.

With all that being said, we can present a basic outline of the initialization algorithm in form of a pseudocode in Algorithm 1. The algorithm uses the function GetFlow as a black-box. Its purpose is to analyze a method which is passed into the function call in the first argument, and return computed flow data for the method. However, this function is a subject to a deeper discussion, Section 5.2.1 is dedicated to its problematics.

Algorithm 1 AnalyzeBean

```
    Input: Application context $\mathbb{A}_{\mathbb{C}}$, bean definition $b \in \mathbb{A}_{\mathbb{C}}$
    Output: Flow data $f(b)$ for the bean definition $b$
 1: $f(b) \leftarrow$ GetFlow($b.initMethod$)                    ▷ Analyze initialization method.
 2:
 3: for all $property \in b.properties$ do                       ▷ Analyze properties.
 4:     $f(b) \leftarrow f(b) \cup$ GetFlow($property$)
 5:
 6: for all $initCallback \in b.initCallbacks$ do     ▷ Analyze initialization callbacks.
 7:     $f(b) \leftarrow f(b) \cup$ GetFlow($initCallback$)
 8:
 9: for all $field \in b.fields$ do                                 ▷ Analyze fields.
10:     if $\{@\text{Autowired}, @\text{Inject}, @\text{Resource}, @\text{Value}\} \cap field.annotations \neq \emptyset$ then
11:         $f(b) \leftarrow f(b) \cup$ AnalyzeBean(GetBean($field$))
12:
13: return f(b)
```

Recalling the example in Listing 4.8, the bean of the type `Configuration` is initialized by using Algorithm 1 as follows:

[3]Initialization method, initialization callback or setter

1. Initialize an empty set for flow data.
2. Add flow data computed by the analysis of the constructor into the set. Note that the default constructor is utilized here, and therefore the set remains unchanged.
3. Extend the set with flow data computed by the analysis of the setter method `void setInjected(Injected)`. Note that at this point, the function AnalyzeBean is called recursively for the bean of the type `Injected`.
4. Extend the set with flow data originating from the value of the annotation `@Value`. Note that at this point, a plain string value is considered an in-place bean configuration which is further converted into the flow data representation of the type `InputStream`.
5. Return the computed set. The returned set refers to flow data of the bean of the type `Configuration`.

4.1.2.2 Accessing Spring Bean from Application Code

The whole purpose of managing the flow data of beans is that these flow data will be accessed somewhere in the analyzed program. Flow data of beans by themselves do not contribute to the results of the data lineage analysis unless some piece of code in the analyzed program manipulates with them.

There are several ways of how to access bean object in application code. The most common one is by using the interface `ApplicationContext`. The example with using the interface `ApplicationContext` can be observed in Listing 4.9.

```
public class Program {
    public static void main(String[] args) {
        ApplicationContext context =
           new XmlApplicationContext("app.xml");

        Object myBeanByName =
           context.getBean("myBean");

        MyBean myBeanByClass =
           context.getBean(MyBean.class);
    }
}
```

Listing 4.9: An example of usage of `ApplicationContext` interface

We can see that a bean can be accessed either by its name or by its type. Our goal is to support both these ways. To do that, we have to *semantically describe* these two overloads of the method `BeanFactory::getBean`. This refers to a concept of dataflow plugins as described in Section 2.1.2, which already exists in Bytecode Scanner. Therefore we shall design a dataflow plugin for the Spring Framework. Without a loss of generality, let us describe what happens when the data lineage analyzer encounters a call of the method `BeanFactory::getBean(String)`:

1. The call of the method `BeanFactory::getBean(String)` is encountered by the data lineage analyzer.
2. The data lineage analyzer tries whether some of the registered dataflow plugins can process the method.
3. The dataflow plugin for the Spring Framework can process the method, and therefore is selected to analyze the method.
4. A semantic description of the method `BeanFactory::getBean(String)` extracts flows from the first argument, this refers to the name of the bean being accessed.
5. The semantic description accesses our representation of application context with a query for all beans matching the name.
6. Flow data of returned beans are computed. This refers to a call of the function AnalyzeBean in Algorithm 1.
7. The semantic description associates the computed flow data with a return value of the call of the method `BeanFactory::getBean(String)`.

From the description, it should be clear that a design of semantic descriptions is rather straightforward. In our particular case, all the semantic description should do is to find appropriate beans, compute flow data for them and return these flow data.

4.1.2.3 Evaluating Call Targets on Spring Bean

Recalling the example in Listing 4.9, the methods to obtain beans are designed generically, in other words, their return values do not say us anything about the actual type of the returned bean. Similar holds in the example which can be observed in Listing 4.10, we only know that some bean whose class type implements the interface `MyInterface` will be injected into the field `injectedBean` in the class `ComponentClass`, but we do not know what is the actual type of the injected bean.

```
@Component
class ComponentClass {

    @Autowired
    private MyInterface injectedBean;
}

interface MyInterface {
    void doIt();
}

@Component
class MyClass implements MyInterface {
    void doIt() {
        ...
    }
}
```

Listing 4.10: An example of unclear actual type of the injected bean

However, we need to know such information in the data lineage analysis. We shall be able to correctly evaluate a call target when calling some method upon the bean object. For example, when evaluating targets of the method call `injectedBean.doIt()`, the method `MyClass::doIt` shall be returned, because the class `MyClass` is the actual type of the injected bean. Therefore, we need to modify the data lineage analyzer a little bit. Without that, the only evaluated target would be the method `MyInterface::doIt` which could lead to a loss of some important data lineage information.

This refers to a concept of dependency injection plugins as described in Section 2.1.2. In our particular case, our proposed solution is that we design a special flow that marks the point of injection, saying there is some bean in it. In the example, when performing an injection into the field `ComponentClass::injectedBean`, we assign the particular field access expression with a special flow representing the bean `MyClass`. Such a flow can be propagated across assignments freely. Now, let us describe what happens when the data lineage analyzer encounters some method invocation:

1. The method invocation `injectedBean.doIt()` is encountered by the data lineage analyzer.
2. The data lineage analyzer tries whether some of the registered dependency injection plugins can process the method.
3. The dependency injection plugin for the Spring Framework is selected to process the method.
4. The dependency injection plugin extracts flow data from the receiver of the method call.
5. The dependency injection plugin checks whether there is some special flow denoting the presence of some bean in the receiver.
6. If there is no such special flow, the dependency injection plugin cannot process the method. Let us suppose that there is some special flow denoting the presence of the bean.
7. The actual type of the bean is resolved.
8. The dependency injection plugin tries to obtain a declared method with the same signature as the invoked method in the resolved type.
9. If there is such declared method, it is returned as the only possible call target. Otherwise, the dependency injection plugin cannot process the invocation.

From the description, it should be clear that an evaluation of possible call targets should be context-sensitive because we shall be able to obtain flow data for the receiver of a method call. The way this is achieved is described in Section 4.2.4.

4.2 Call Graph

The call graph is a crucial structure for static data lineage program analysis. Whenever the data lineage analyzer encounters some method invocation, a target of such a call must be evaluated correctly. At runtime, the evaluation is resolved by looking up the target in the virtual method table. However, in the static analysis, there is no such table because we do not often know the actual type of a receiver of the method call, due to the missing model of program heap. Therefore, we shall over-approximate targets of method calls. It effectively means that in the data lineage analysis, multiple target methods have to be considered for each instruction `invoke`. In this section, we discuss possible approaches to call graph implementation and its relation to support for the Spring Framework. We consider two variants of the call graph. One of the variants is the context-insensitive call graph, which is described in Section 4.2.2. Its stronger variant, the context-sensitive call graph, is described in Section 4.2.3.

4.2.1 Type Hierarchy

The most important helper structure we need for call graph is type hierarchy, sometimes referred to as class hierarchy. Every single type in Java Platform defines two pieces of information about relations between types:

- A list of super interfaces it implements. In Java code, this is realised by the keyword `implements` in type definition.
- A superclass it extends. In Java code, this is realized by the keyword `extends` in type definition. If there is no such keyword, the class always extends the class `java.lang.Object`, a root of class hierarchy in Java Platform.

However, as we will see in Section 4.2.2.1, this is not enough. To have a complete picture of relations between types, we shall be able to perform the following two queries:

- For an interface type, we shall be able to obtain a list of all reference types that implement the particular interface.
- For a class type, we shall be able to obtain a list of all its subclasses.

Additionally, a requirement on these two queries is that their evaluation should be fast enough since queries for the relations of types form the core of the static analysis. However, it can be observed that these queries are nothing but scanning for types that implement a given interface, extend a given class respectively.

4.2.2 Context-Insensitive Call Graph

The context-insensitive call graph is a graph where the set of vertices is made of all the methods defined in the analyzed program, and there is an edge between the node `A` and the node `B` if the method represented by the node `A` calls the method represented by the node `B` regardless of the arguments' values. In other words, the context-insensitive call graph does not distinguish between the method call `foo(1)` and the method call `foo(2)`.

The context-insensitive call graph is being constructed from the specified entry-point of the analyzed application. In the data lineage analysis, we only care about the graph component which contains the entry-point. The other graph components are not being constructed since they are unreachable from the entry-point, and therefore they are not being analyzed. The algorithm that can construct the context-insensitive call graph is described in Algorithm 2. The algorithm is nothing but a simple depth-first search algorithm using a call graph builder to evaluate call targets, which is described in Section 4.2.2.1.

Algorithm 2 CONSTRUCTCONTEXTINSENSITIVECALLGRAPH

```
    Input: Entry-point $e$ of the application, call graph builder $CGB$
    Output: Context-insensitive call graph $G$ built from $e$
 1: $vertices \leftarrow \emptyset$
 2: $edges \leftarrow \emptyset$
 3: $stack \leftarrow \{e\}$
 4: while $stack \neq \emptyset$ do
 5:     $current \leftarrow$ POP($stack$)
 6:     if $current \in vertices$ then                    ▷ Avoid recursive calls
 7:         continue
 8:     $vertices \leftarrow vertices \cup \{current\}$
 9:     for all $invocation \in$ GETINVOKEINSTRUCTIONS($current$) do
10:         $targets \leftarrow$ EVALUATETARGETS($CGB, invocation$)
11:         for all $target \in targets$ do
12:             $edges \leftarrow edges \cup \{(current, target)\}$
13:             $stack \leftarrow stack \cup \{target\}$
14: return (vertices, edges)
```

Note that since the context-insensitive call graph cannot differentiate between two invocations of the same method but with a different context, it also cannot determine whether the receiver of some method invocation is some bean or not. Similar holds for lambda expressions, which are evaluated dynamically. Therefore, for our project, we need some stronger variant of call graph which could handle contexts of method invocations as well. The call graph that can handle contexts of method invocations is described in Section 4.2.3 and is built on top of the context-insensitive call graph.

4.2.2.1 Rapid Type Analysis

The Rapid Type Analysis is one of many algorithms for building the context-insensitive call graph, as we discussed in Section 3.2. These algorithms are sometimes referred to as call graph builders, and the algorithm itself has a form of the function EVALUATETARGETS from Algorithm 2. From all the call graph builders discussed in Section 3.2, we have chosen Rapid Type Analysis to be used in our project, because the amount of over-approximation is acceptable for the data lineage analysis and the implementation is rather straightforward.

The Rapid Type Analysis uses Class Hierarchy Analysis under the hood. In JVM specification, the Class Hierarchy Analysis is referred to as *method selection* [12]. The method selection is only applicable when a method being selected is expected to be overridden, i.e., it is either interface, or abstract, or a virtual method. For static methods and instance initializers, the method selection is not being used. There can always be a single target of invocation of such a method, the method itself. Otherwise, our modification of method selection uses type hierarchy from Section 4.2.1 as follows:

- If a method being selected is defined by an interface, get all possible overrides of the method in all the implementors, i.e., all implementing interfaces and classes.
- If a method being selected is defined by a class, get all possible overrides of the method in all the subclasses.

Therefore, there is not a single call target in the data lineage analysis, there is a set of call targets. The data lineage analyzer simulates the behavior in such a way that a single instruction `invoke` can call multiple methods, the evaluated targets.

All the Rapid Type Analysis does is that it filters targets computed by the method selection. The Rapid Type Analysis selects only those methods, whose declaring type has been instantiated earlier during symbolic program execution. Therefore, our goal is to construct a set of instantiated types. To do so, there are two approaches:

- Treat all user-defined types in the analyzed application as the instantiated ones.
- Treat all types, which has been instantiated by the instruction `new` somewhere in the symbolic program execution, as the instantiated ones.

Our solution combines both approaches. For the first approach, we require that the implementation will contain an option that, if enabled, causes that all user-defined types in the analyzed application are added into the set of instantiated types. Enabling such option should increase performance during the call graph construction, but make the analysis less precise. Otherwise, Algorithm 3 collects all the instructions `new` in methods reachable from the entry-point method and constructs the set this way. Note that the input parameter CGB in the function ConstructInstantiatedTypesSet is the Rapid Type Analysis itself, which utilizes the set being constructed. The algorithm computes the set until the fixed point over instantiated types is reached, because a new instantiated type may have caused that some set of targets has changed. Therefore, if a new target should occur in the set, it means that we have to process it as well. Such a new target can, however, instantiate some other new type, which we would not know if we did not compute the set of instantiated types to the fixed point.

Algorithm 3 CONSTRUCTINSTANTIATEDTYPESSET

Input: Entry-point e of the application, a set of static initializers $\mathbb{S}$, call graph builder CGB

Output: A set of instantiated types $\mathbb{T}$

```
1:  $\mathbb{T} \leftarrow \emptyset$
2:  $targetsMap \leftarrow \emptyset$
3:  $changed \leftarrow true$
4:  while $changed$ do                              ▷ Repeat until the fixed-point is reached.
5:      $changed \leftarrow false$
6:      $visited \leftarrow \emptyset$
7:      $queue \leftarrow \{e\} \cup \mathbb{S}$          ▷ Enqueue entry-point and static initializers.
8:      while $queue \neq \emptyset$ do
9:          $current \leftarrow$ DEQUEUE($queue$)
10:         $\mathbb{T} \leftarrow \mathbb{T} \cup$ GETINSTRUCTIONSNEW($current$)
11:         for all $invocation \in$ GETINSTRUCTIONSINVOKE($current$) do
12:             $tagets \leftarrow$ EVALUATETARGETS($CGB, invocation$)
13:             if $targetsMap[invocation] \neq targets$ then
14:                 $changed \leftarrow true$                 ▷ The method targets have changed.
15:                 $targetsMap[invocation] \leftarrow targets$
16:             for $target \in targets$ do
17:                 if $visited \cap \{target\} = \emptyset$ then          ▷ Avoid recursive calls.
18:                     $queue \leftarrow queue \cup \{target\}$
19:                 $visited \leftarrow visited \cup \{target\}$
20: return $\mathbb{T}$
```

4.2.3 Context-Sensitive Call Graph

The context-sensitive call graph is a graph where the set of vertices is made of all the methods defined in the analyzed program, and there is an edge between the node `A` and the node `B` if the method represented by the node `A` calls the method represented by the node `B`, concerning the arguments' values. In the context-insensitive call graph, for the two method calls `foo(1)` and `foo(2)`, there is only one edge between the caller and the method `foo`. In the context-sensitive call graph, there are two edges, between the caller and the method call `foo(1)`, the caller and the method call `foo(2)` respectively. For clarity, consider the example in Listing 4.11. A visualization of the context-insensitive call graph can be observed in Figure 4.1. A visualization of the context-sensitive call graph can be observed in Figure 4.2.

The context-sensitive call graph utilizes the context-insensitive call graph in its core. Since the context-sensitive call graph does care about arguments' values, it cannot be pre-computed during the initialization of the data lineage analyzer as the context-insensitive call graph. It is being built on the fly, during the analysis in such a way, that whenever the data lineage analyzer encounters a method invocation, it adds an edge into the graph between invocation context representing the caller, and invocation context representing the callee. However, a single callee can have multiple call targets, and therefore the context-insensitive call graph is utilized here to compute a set of call targets for the callee.

As the data lineage analyzer uses a fixed-point algorithm to compute method summary for a single method, queries for direct callees and direct callers are the most commonly used operations during the analysis. This is also one of the motivations for the context-sensitive call graph. In the rest of this paragraph, without loss of generality, consider the query for direct callees only. The reasoning for the query for direct callers is analogous. With the context-insensitive call graph, the data lineage analyzer does not know which invocation contexts are callees of the invocation context being analyzed. Therefore, it must analyze all invocation contexts related to a callee method. However, with the context-sensitive call graph, we know the exact callee's invocation contexts. In result, the analysis utilizing the context-sensitive call graph should be significantly faster, because a set of callees is expected to be much smaller. Therefore, the analysis should take less time, opposed to utilizing the context-insensitive call graph. Formally, if the set C contains all edges leading to vertices representing invocation contexts of the direct callees and the set M represents the direct callees methods, it always holds that $|C| \leq \sum_{i=0}^{|M|} n_i$, where n_i is a number of recorded invocation contexts for the method M_i. Whilst the $|C|$ is a count of iterations when utilizing the context-sensitive call graph, the $\sum_{i=0}^{|M|} n_i$ is a count of iterations when utilizing the context-insensitive call graph.

However, the main motivation for the context-sensitive call graph is to enable support for the Spring Framework and lambda expressions, which is described in Section 4.2.4.

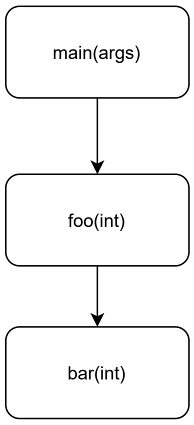


Figure 4.1: A visualization of the context-insensitive call graph for the example in Listing 4.11

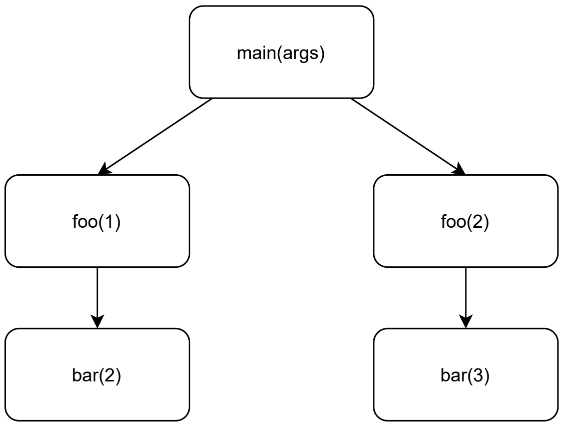


Figure 4.2: A visualization of the context-sensitive call graph for the example in Listing 4.11

```
void main(String[] args) {
    foo(1);
    foo(2);
}

void foo(int n) {
    bar(n + 1);
}

void bar(int n) {
    ...
}
```

Listing 4.11: An example of simple program to demonstrate the difference between the context-insensitive call graph and the context-sensitive call graph

4.2.4 Dependency Injection Plugins

The motivation for dependency injection plugins is the proper support for the Spring Framework. In Listing 4.12, we introduce a basic example of what we aim to achieve. In the example, there are two beans, the classes `SomeComponent` and `OtherComponent`. The both classes implement the interface `MyInterface`. In the method `main`, there is a query into the application context to obtain a bean of the type `MyInterface`, followed by invocation of the method `doIt` upon the bean object. At this point, the context-sensitive call graph evaluates two call target methods for the invocation, the methods `SomeComponent::doIt` and `OtherComponent::doIt`. However, in this basic example, we know that there should be a single target only, the method `SomeComponent::doIt`, because the bean of the type `SomeComponent` was assigned to the local variable variable named `bean` since the bean is marked as primary. Therefore, we should ensure that the context-sensitive call graph follows this behavior, in order to make the analysis more precise.

```
public class Program {
    public static void main(String[] args) {
        ApplicationContext context
            = new XmlApplicationContext("app.xml");
        MyInterface bean
            = context.getBean(MyInterface.class);
        bean.doIt();
    }
}

public interface MyInterface {
    void doIt();
}

@Component
@Primary
public class SomeComponent implements MyInterface {
    ...
}

@Component
public class OtherComponent implements MyInterface {
    ...
}
```

Listing 4.12: An example of a need for dependency injection plugins

Our solution follows a concept of plugins that already exists in Bytecode Scanner, in a form of dataflow plugins as we discussed in Section 2.1.2. Our goal is to add similar support for dependency injection plugins. Since there was already a support for dataflow plugins before, we decided to use almost the same way of modularization, due to already existing infrastructure and to avoid confusion.

Dependency injection plugins are a kind of plugins that extend the context-sensitive call graph. The algorithm of evaluating call targets from Section 4.2.3 is modified as follows for dependency injection plugins:

- Is there any registered dependency injection plugin that can evaluate the call targets? If so, use it to evaluate the call targets.
- Otherwise, use the original algorithm as a fallback. In other words, use context-insensitive call graph, Rapid Type Analysis respectively, to compute a set of the call targets.

If a dependency injection plugin is eligible to evaluate the call targets, it is given the invocation context of the callee at its input. As a result, it produces the set of the call targets. For example, the dependency injection plugin for the Spring Framework is eligible to evaluate the call targets if there is some bean injected into a receiver of the method call. The plugin shall evaluate the set of the call targets in such a way, that it obtains a type of the injected bean and resolves a method with the same method signature in that type. This method is further returned, and it is a result of the dependency injection plugin call.

For the example in Listing 4.12, when evaluating targets for the invocation of the method `doIt`, the plugin knows that the bean of the type `SomeComponent` is in the receiver of the invocation. Therefore, it tries to resolve the method `doIt` in the class `SomeComponent` and such a method is considered the only target of the method call, copying method arguments' values from the original invocation.

Furthermore, the general requirements on the dependency injection plugins are the following:

- The dependency injection plugin shall be enabled only if a framework handled by the plugin is used within the analyzed application.
- It shall be easy to register the dependency injection plugin into Bytecode Scanner, e.g., by using the dependency injection.

The general requirements shall be satisfied implicitly by existing infrastructure in Bytecode Scanner, and that is also the next reason why we opted to use this form of modularization.

Support for dependency injection plugins is further described in Section 6.3 where we also describe support for lambda expressions in Java Platform, which is closely related to dependency injection plugins.

5. Spring Framework Plugin

Before we can immerse ourselves into the implementation of the Spring Framework plugin, a common term for both dataflow and dependency injection plugins for the Spring Framework, we shall have a model for its basic structures prepared. Let us remind that these basic structures are called beans. A bean is an object that is instantiated and managed by IoC container within the Spring Framework. In other words, a bean is simply one of many objects in the application. A model of beans is described in Section 5.1.

From the analysis perspective, beans are especially important because they are usually not included in the basic call graph, they are instantiated implicitly within the Spring Framework, whose code the data lineage analyzer without the Spring Framework plugin does not analyze. Analyzing such code might be performance-demanding and might slow down the analysis. Additionally, the analysis does not support reflection for similar reasons. Therefore, to correctly analyze the application, we shall be able to propagate beans flow data to the correct place somehow. The computation of beans flow data is described in Section 5.2. Propagation of these flow data across the analyzed application is described in Section 5.3.

Listing 5.1 contains an example that we will use in this chapter to explain motivation for the Spring Framework plugin. In Listing 5.1, we can observe that the method `Main::main(String[])` obtains an instance of the class `Executor` at first, followed by a call of the method `Executor::doExecute()` upon the obtained object. This method further calls the method `CompanyData::getAudit()` upon the declared field, and this is the place where the problem arises. Since the field named `companyData` has the annotation `@Autowired`, meaning that it is injected by the Spring Framework, we have no information about its flow data. These flow data are important to correctly analyze the program, since the field named `companyData` can contain important flows, e.g., results of queries of the database or I/O actions.

```
@Configuration
public class ConfigurationClass {

    @Bean
    public static CompanyData companyData() {
        return getCompanyData();
    }

    private static CompanyData getCompanyData() {
        CompanyData companyData = new CompanyData();
        addEmployeesData(companyData);
        addCustomersData(companyData);
        return companyData;
    }

    ...
}
```

```
public class Main {

    public static void main(String[] args) {
        ...
        applicationContext.getBean(Executor.class)
            .doExecute();
    }
}

@Component
public class Executor {

    @Autowired
    private CompanyData companyData;

    public void doExecute() {
        String audit = companyData.getAudit();
        System.out.print(audit);
    }
}
```

Listing 5.1: Motivational example for the Spring Framework plugin

Therefore, the analyzer shall be able to inject appropriate flow data into the field named `companyData` in the class `Executor` somehow. To do that, an appropriate bean shall be found. This process is explained in Section 5.2.2.

It is necessary to analyze a particular bean, which is a return value of the method `companyData()` defined in the class `ConfigurationClass` in this case. Note that it is not sufficient to analyze a single method, since the method can call other methods as well. Therefore, it is necessary to treat all beans methods[1] similarly like entry-points of the analysis, i.e., analyze their whole call tree and then extract required flow data for this particular entry-point only. This process is explained in Section 5.2.1.

5.1 Spring Beans Model

Our model for beans follows the model defined by the Spring Framework. The Spring Framework defines the interface `BeanDefinition` which is implemented by several classes. Our model is built on top of this interface defined by the Spring Framework. The whole model of beans, described in this section, is visualized in Figure 5.1. We describe in more detail just the most important classes and interfaces from our model.

The most low-level interface of our model is the interface `BeanElement`. The interface is a common interface for all beans entities, most importantly beans and string literals. It defines a single method, which is a getter to obtain an instance of the type `BeanContext`. As the name suggests, the type `BeanContext`

[1]Initialization methods, initialization callbacks, setters etc.

contains all loaded beans in the analyzed program, and it is our representation of the application context. These beans are stored in a simple map, which maps the name of the bean to a set of instances of the class `AbstractBeanDefinition`.

You may have noticed that name of the bean is mapped to a set of instances of the class `AbstractBeanDefinition`, whilst at runtime, the name is always unique. This is because of our design of support for profiles. In general, we over-approximate the results, and this is closely related to bean definitions as well. The customer can specify an active profile in the configuration, by the property named `bytecode.resolver.plugin.springframework.profiles.active`. However, if such property is not set, all profiles are active by default. That is why our solution works with multiple instances rather than just a single instance, the name of bean is one example in many. Active profiles are obtained through the class `Environment` which is our representation of the environment of the application. Not only active profiles are stored in it, but property values are also stored there, including those defined by the system environment. However, the customer shall specify these property values manually, since the analysis of the program is usually executed on a different machine than the application itself.

The abstract class `AbstractBeanDefinition` follows a model from the Spring Framework, however, entities defined by us are more type-safe and suit better the analysis needs. For example, in the analysis, we shall be able to obtain an instance of the class `Method` which refers to the initialization method, being either class' constructor or factory method. Additionally, it is not sufficient to find such method by its name only, we shall be able to find appropriate overload matching initialization method arguments' types. However, the interface `BeanDefinition` in the Spring Framework does not provide us an easy way of how to obtain such method. First, we need to check if there is some factory bean defined. If not, we consider the defining bean as a reference one. Next, we have to check whether there is some factory method defined. If so, we look-up all methods matching the factory method name, which is some string literal, in the class type of the bean. Otherwise, we consider all constructors in the reference bean as potential initialization methods. In the last step, we have to check a number of parameters in all the potential initialization methods and we also have to compare types of parameters.

Another example can be a dependency tree of beans. If a bean defines a factory bean, it must be initialized beforehand. Similarly for beans referenced in initialization method arguments. The Spring Framework does not provide a straightforward way to do so, and therefore we define a method in the abstract class `AbstractBeanDefinition` to do the job. It simply returns a set of all the beans that the defining bean can reference, and these beans have to be analyzed before the analysis of the defining bean can start.

In Section 4.1.1.1, we mentioned that sometimes it is necessary to deduce the actual type of the bean. Again, the Spring Framework does not provide an easy way to provide the actual type of the bean, and therefore we have to implement this functionality by ourselves. First, we have to check whether the class type of the bean is defined explicitly. If so, it is still not sufficient, since we have to check whether the bean defines a factory method. If so, we declare a return type of the factory method as the actual type of the bean, note that it does not necessarily have to be the same type as the explicitly declared one, although it is very likely. Otherwise, if the explicitly declared class type is not set, we check whether the bean defines some factory bean. If it does, then we apply the same procedure of finding the return type of a factory method. If everything fails, which, however, should not, in case of valid configuration, we return the class `java.lang.Object` as the actual type of the bean.

We have already stated that our model defines the interface `BeanElement` as a foundation stone of all the elements that can appear in the bean definition. There is also one more such interface, the interface `InjectableElement` extending the original interface `BeanElement`. As the name suggests, all bean definition elements that can get injected into either parameter or field, implement the interface. The interface defines a single method which determines whether the instance is qualified to be injected into the specified entity, i.e., parameter or field, or not. There are only two direct implementations of the interface. These are the abstract class `AbstractBeanDefinition` for obvious reasons, and the final class `InjectableStringElement` which wraps all the string literals that appear in the application context, basically values of the attribute `value` in XML definitions, or a value of the annotation `@Value` declared in code.

Now, we can move on to the subclasses of the class `AbstractBeanDefinition`. We define the abstract class `AbstractXmlBeanDefinition` for XML definitions and the abstract class `AbstractAnnotatedBeanDefinition` for code definitions. These classes override methods from the class `AbstractBeanDefinition` whose implementations should be common for all the subclasses.

We define two subclasses of the abstract class `AbstractXmlBeanDefinition`:

- The abstract class `AbstractXmlStandardBeanDefinition`, which refers to standard bean definitions in XML documents, and has two subclasses:
  - The final class `GenericXmlBeanDefinition`, which accepts an instance of Spring Framework's class `GenericBeanDefinition` in its constructor argument and implements functionality for, e.g., autowire mode or qualifiers.
  - The final class `BasicXmlBeanDefinition`, which accepts an instance of Spring Framework's interface `BeanDefinition` in its constructor argument. Since Spring Framework's interface `BeanDefinition` does not provide any information about autowire mode or qualifiers, we return a default value, i.e., default autowire mode, empty collection of qualifiers respectively.
- The abstract class `AbstractXmlCollectionBeanDefinition`, which refers to collections defined in the schema `util`. Subclasses of this abstract class are dsecribed in Section 5.1.1.4.

Similarly for the abstract class `AbstractAnnotatedBeanDefinition`, we define two subclasses:

- The final class `ClassAnnotatedBeanDefinition`, representing classes in scope obtained from component scan, i.e., any class having the annotation `@Component` as a meta-annotation, or generally any class if the custom filters have been applied. Such a bean is always set to be autowired by constructor and therefore has no explicit initialization method.

- The final class `MethodAnnotatedBeanDefinition`, representing beans realized by methods annotated by the annotation `@Bean`. Such a bean has autowire mode set to default and the initialization method is a method which defines the annotation.

As a part of bean definition, there are some helper implementation of the interface `BeanElement` which are used within the class `AbstractBeanDefinition`, usually as return types of some particular methods:

- The final class `AutowireCandidateQualifier` is utilized for qualifiers that are defined by the bean.

- The final class `InitializationMethodArgument` represents an argument of the initialization method, i.e., a constructor or factory method if autowire mode is set to the value `default`. It can be easily observed that its value can be an instance either of the class `AbstractBeanDefinition`, or the class `InjectableStringElement`. Note that it shall be a set of these instances instead of a single instance because of our design of support for profiles.

- The final class `PropertyDefinition` is used to represent properties defined in XML definitions. This class is in fact similar to the aforementioned class `InitializationMethodArgument` except that its value is mapped from some string literal, i.e., name of the property.

To make sure a correct class for a particular bean definition will be instantiated, we use the factory method pattern in the class `BeanDefinitionFactory`. Similarly, factory methods are used to obtain qualifiers, property definitions and initialization method arguments. In property definition and initialization method argument, there can be several types of entities, these are resolvable by the class `XmlElementResolver`:

- String literals, the string literal is wrapped by an instance of the class `InjectableStringElement`. These refer to the attribute `value` and the attribute `null` in XML definitions.

- Inner beans, a new instance of the class `AbstractBeanDefinition` is created by using the factory, but not added into the instance of the type `BeanContext` since it cannot be referenced from anywhere else. These refer to the element `bean` in XML definitions. However, also collections from the schema `util` can be specified here, and this applies to the elements `set`, `list`, `map`, `props` and `array` in XML definitions.

- Referenced beans, name of the referenced bean is looked-up in the instance of the type `BeanContext` and returned. We can do this because the model is designed in such a way that methods defined by the class

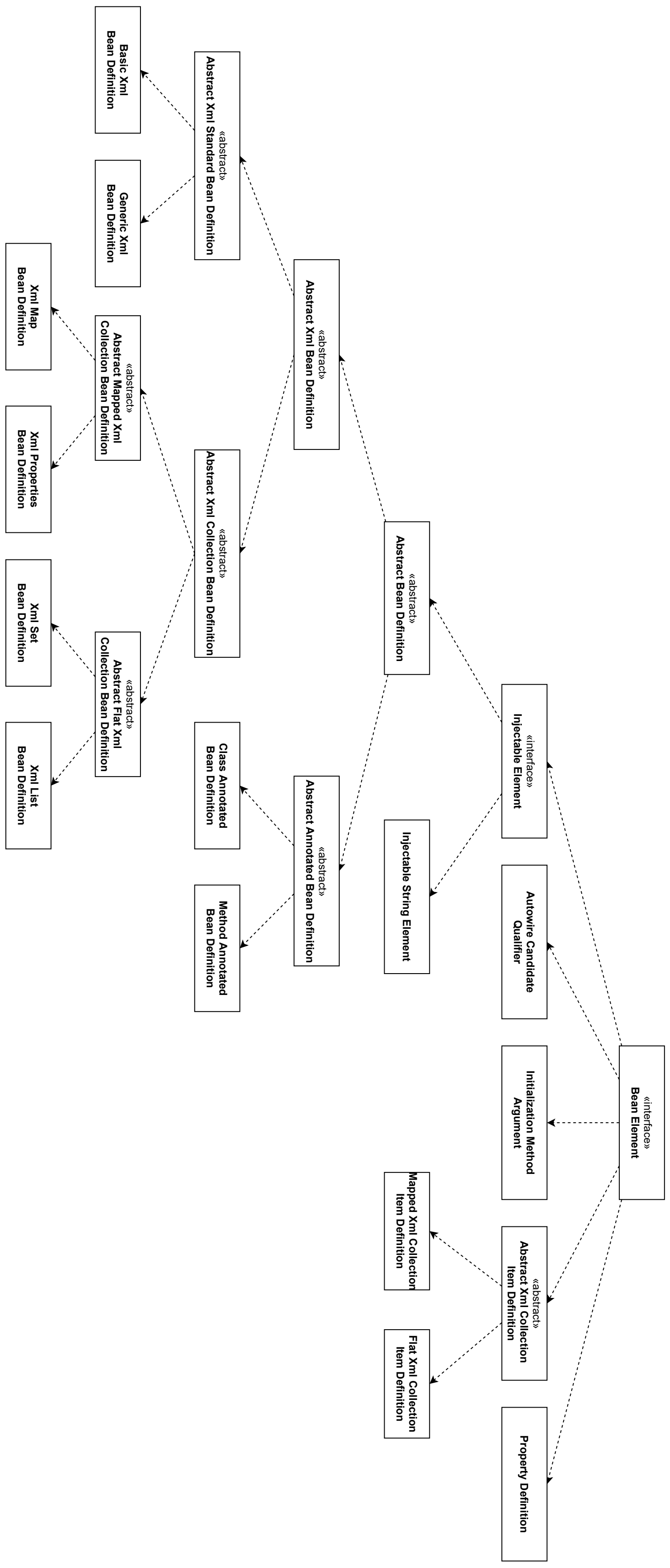

«interface»
Bean Element
«interface»
Injectable Element
Autowire Candidate Qualifier
Initialization Method Argument
«abstract»
Abstract Xml Collection Item Definition
Property Definition
Mapped Xml Collection Item Definition
Flat Xml Collection Item Definition
Injectable String Element
«abstract»
Abstract Bean Definition
«abstract»
Abstract Xml Bean Definition
«abstract»
Abstract Annotated Bean Definition
Class Annotated Bean Definition
Method Annotated Bean Definition
«abstract»
Abstract Xml Standard Bean Definition
Basic Xml Bean Definition
Generic Xml Bean Definition
«abstract»
Abstract Xml Collection Bean Definition
«abstract»
Abstract Mapped Xml Collection Bean Definition
«abstract»
Abstract Flat Xml Collection Bean Definition
Xml Map Bean Definition
Xml Properties Bean Definition
Xml Set Bean Definition
Xml List Bean Definition

Figure 5.1: Simplified UML class diagram of beans model

AbstractBeanDefinition are implemented lazily. Therefore, all bean definitions are parsed and instantiated at first, and only then the methods in the type BeanContext can be safely called. These refer to the attributes ref and idref in XML definitions.

5.1.1 Parsing of Spring Beans

To parse a bean definition, we introduce the interface BeanDefinitionResolver. The interface defines a single method that can parse the definitions into the specified instance of the type BeanContext. There are currently two implementations of the interface BeanDefinitionResolver:

- The final class XmlBeanDefinitionResolver, which can parse definitions from XML documents. The implementation is described in Section 5.1.1.1.
- The final class AnnotatedBeanDefinitionResolver, which can parse definitions from code, based on specified annotations. The implementation is described in Section 5.1.1.2.

5.1.1.1 Parsing of Spring Beans in XML Documents

In order to parse a XML document, we need to have an access to it at first. That is achieved by using an instance of the type AnalysisScope, provided by Bytecode Infrastructure, which provides us such a functionality. Input for Bytecode Scanner is configured using XML file, which can define a list of resources. These resources are simply traversed, files with the extension *.xml are filtered and we further filter these resources to contain the root element named beans only.

Having a list of resources, we can use the class XmlBeanDefinitionReader defined by the Spring Framework, which is utilized by the Spring Framework itself to read bean definitions. This helps us to prevent from bugs that could possibly occur if we decided to implement a custom parser. Additionally, such approach could help us in future releases of the Spring Framework, where could possibly be some changes in the schema. We utilize the class as follows. The Spring Framework stores instances of the class BeanDefinition into an instance of the class BeanDefinitionRegistry at first. Then, our algorithm iterates over the registry and uses the class BeanDefinitionFactory to create our representation of bean definitions. Our representation of bean definitions is realised by the class AbstractBeanDefinition as described in Section 5.1. Instances of the class AbstractBeanDefinition are further stored into the specified instance of the type BeanContext, every bean is assigned with its name. Note that beans defined in XML documents also support aliasing of names, and therefore we should do the same. However, this step is rather simple. Our algorithm requests all aliases for given bean from the instance of Spring Framework's class XmlBeanDefinitionRegistry and saves such information into the instance of the type BeanContext. Last, the algorithm registers all beans defined by component scan, if there are any. The process of component scanning is described in Section 5.1.1.3.

During implementation of the class `XmlBeanDefinitionReader`, we had to face one catch only with the element `<import resource="..."/>`. The problem occurs if the attribute's value is prefixed with the prefix `classpath:`, meaning that the Spring Framework expects such a resource on the classpath. The problem here is the absence of resource on the classpath. During the program execution, the resource should be there, but we must realize that we have classpath of Bytecode Scanner available only, i.e., we cannot access the classpath of the analyzed program. It does not exist at that time in Java Virtual Machine. However, the solution of the problem is rather simple. The Spring Framework uses the class `DefaultBeanDefinitionReader` to parse the XML document. This class defines the protected virtual method `importBeanDefinitionResource(Element)` to process imports. Therefore, we designed our reader, realized by the class `CustomBeanDefinitionReader` which extends the default one, and we have overriden the method with our logic to avoid the problem. If there is some resource with mentioned prefix, our reader simply tries to find appropriate resource in the instance of the type `AnalysisScope` which in fact represents the classpath of the analyzed program. Then, it replaces value of the resource attribute with absolute path to the resource. Note that similar is applicable to other prefixes as well, like the prefixes `file:` or `http:`.

5.1.1.2 Parsing of Spring Beans in Code

Because of the mentioned problem with the classpath in Section 5.1.1.1, we cannot reuse the Spring Framework to parse beans definitions in code which was the case for XML documents. The class `AnnotatedBeanDefinitionReader` provided by the Spring Framework to parse bean definitions, defines a method which accepts an instance of the class `Class<?>` and returns an instance of the class `BeanDefinition` containing parsed data. However, in the analysis, we cannot access instances of the class `Class<?>` being analyzed, we only have an access to instances of the class `Type`. These instances are provided by Bytecode Infrastructure and provide us almost identical information as instances of the class `Class<?>`.

Therefore, we have to parse these beans manually. First, our algorithm iterates through the instance of the type `AnalysisScope` and finds all classes that define the annotation `@Configuration`. The algorithm registers these classes into the instance of the type `BeanContext`. It also checks whether any of these classes define the annotation `@ComponentScan` and if they do, the algorithm stores such information. Then, it registers beans that are defined by methods annotated by the annotation `@Bean` in classes defining the annotation `@Configuration`. Last, it uses information defined by the collected annotations `@ComponentScan` to iterate the scope again, and it registers classes obtained from the component scan as beans. The process of component scanning is described in Section 5.1.1.3.

5.1.1.3 Component Scan

Since the component scan itself has the same semantics for both code definition and XML definition, we can define two classes to implement the common functionality:

- The class FilterBuilder utilizes builder pattern to build filters described in Section 4.1.1.2. For example, the code snippet in Listing 5.2 creates a filter that matches all classes whose names start with the character 'A' and are annotated by the annotation @MyBean.
- The class ComponentScanQueryBuilder accepts a list of base packages in its constructor, and can build a stream of classes with respect to component scan rules and filters that can be specified.

The component scan itself is handled by implementation of the interface BeanDefinitionResolver by utilizing helper classes decribed in this section. More precisely, the parser for both the annotation @ComponentScan and the annotation @ComponentScans is implemented in the class ComponentScanner. For the element component-scan defined in the schema context, the parser is a part of the class XmlBeanDefinitionResolver where we take an advantage of support for namespaces defined within Spring Framework's reader.

```
new FilterBuilder()
  .withRegexMatch("^A.*")
  .withDeclaredMetaAnnotation(MyBean.class)
  .build()
```

Listing 5.2: An example of usage of the class FilterBuilder

5.1.1.4 Collections as Spring Beans

Like almost everything in the Spring Framework, beans with the type of some collection can be defined in both code and XML documents:

- Collections defined in code are usually methods annotated by the annotation @Bean with a return type of some collection. These beans do not require any special handling, their bytecode is analyzed as usual. Therefore, collections defined in the code do not require any special care within the plugin for the Spring Framework.
- Collections in XML documents are realized by the elements list, set, properties, and map from the schema util. Unlike collections defined in code, these beans cannot be processed by Bytecode Scanner by default. Therefore, we introduce a mechanism that can both parse these elements, and convert parsed entities into the representation which Bytecode Scanner expects so they can be further used from the core analysis. We discuss the mechanism in the rest of this section.

In Bytecode Scanner, the most basic dataflow plugin is for Java Platform. Apart from others, it can handle collections as well. Particulary, it can handle both obtaining an element from the collections, as well as inserting an element into the collection. First of all, let us describe a way of how the plugin for Java Platform handles collections. There are two types of collections:

- So-called *flat* collections. Among these collections are, for example, lists or sets. Adding some item into the collection object works as expected, all its flows are wrapped by instances of the class `FlatCollectionItemFlow` and assigned to the expression representing the collection. Obtaining some item of the collection works a little bit unusual. If an element at some index of the list is accessed, the index is partially ignored and all items of the collection are returned instead. Before they are returned, their flows are unwrapped from instances of the class `FlatCollectionItemFlow` at first.

- So-called *mapped* collections. The most common mapped collection is a simple map, but the class `Properties` is also considered a mapped collection. Adding some item into the collection works similarly like for flat collections. A cross product of key flows and value flows is computed, these pairs are wrapped by instances of the class `MappedCollectionItemFlow` and assigned to the expression representing the collection. Obtaining some item of the collection works based on the key being accessed. If it is a constant, only those values assigned to the particular constant are returned, otherwise the whole value set is returned. These values are unwrapped from particular instances of the class `MappedCollectionItemFlow`.

As can be observed in Figure 5.1 representing the model, our infrastructure for collections defined as beans in XML documents is identical when compared to behavior of the plugin for Java Platform. There is the abstract class `AbstractXmlCollectionBeanDefinition` defining the common functionality. It has several subclasses:

- The abstract class `AbstractFlatXmlCollectionBeanDefinition` defines common functionality for all flat collections. There are two subclasses of this class, the final class `XmlListBeanDefinition` and the final class `XmlSetBeanDefinition`.

- The abstract class `AbstractMappedXmlCollectionBeanDefinition` defines common functionality for all mapped collections. There are two subclasses of this class, the final class `XmlMapBeanDefinition` and the final class `XmlPropertiesBeanDefinition`.

The purpose of the class `AbstractXmlCollectionBeanDefinition` should be clear. Its purpose is to provide the analyzer with some instances of the class `AbstractXmlCollectionItemDefinition` representing items of the collection. This class is nothing but a wrapper for a set of instances of the type `InjectableElement` instances that are included in the collection. The particular analyzer should then follow the behavior of the plugin for Java Platform and create flow instances for the items of the collection exactly in the same way, so they can be accessed from other parts of the analysis as well.

5.1.2 Summary of Spring Beans Model

As a summary, we present an overview of the most important classes in the presented model, and also basic examples of bean definitions that are represented by a particular class:

- The class `ClassAnnotatedBeanDefinition` represents beans defined in code in form of classes:
  - The class `@Configuration class MyBean`.
  - The class `@Component class MyBean`, if the default filters are enabled.
  - The class `@Service class MyBean`, if the default filters are enabled.
  - The class `class MyBean implements MyInterface`, if the custom filter for searching bean implementing the interface `MyInterface` has been applied.
- The class `MethodAnnotatedBeanDefinition` represents beans defined in code in form of methods:
  - The method `@Bean MyBean getMyBean()`.
- The class `AbstractXmlStandardBeanDefinition` represents all beans defined by the element `bean`:
  - The element `<bean id="myBean" class="java.lang.Object" />`.
- The class `XmlListBeanDefinition` represents all beans defined by the element `util:list`:
  - The element `<util:list id="myBean" ... />`.
- The class `XmlSetBeanDefinition` represents all beans defined by the element `util:set`:
  - The element `<util:set id="myBean" ... />`.
- The class `XmlMapBeanDefinition` represents all beans defined by the element `util:map`:
  - The element `<util:map id="myBean" ... />`.
- The class `XmlPropertiesBeanDefinition` represents all beans defined by the element `util:properties`:
  - The element `<util:properties id="myBean" ... />`

5.2 Spring Beans Analysis

In this section, we describe a way of how structures from Section 5.1 are utilized to produce flow data for the specified bean. In Section 5.2.1, we describe the algorithm of building a set of flow data for the specified bean in detail. In Section 5.2.2, we provide technical details of how the flow data of some bean are injected into the flow data of another bean. Last, in Section 5.3, we describe a way of how the flow data of beans are utilized in Bytecode Scanner.

5.2.1 Initialization Process of Spring Beans

First, it is important to note that instances of the type `AbstractBeanDefinition` by themselves are not sufficient for the complete analysis of some Spring application. As mentioned in Section 5.1, there is the interface `InjectableElement`, which is implemented not only by the class `AbstractBeanDefinition`, but also implemented by the class `InjectableStringElement`. Therefore, we need to be able to analyze instances of the class `InjectableStringElement` somehow as well. That is why we introduce the interface `InjectableElementAnalyzer` to do the job.

The interface `InjectableElementAnalyzer` defines two methods, it is given some instance of the type `InjectableElement` at input and the implementation of the interface `InjectableElementAnalyzer` can:

- Determine whether it can analyze a given element.
- Analyze the given element and return its flow data. If the element cannot be analyzed, the exception is thrown.

Currently, there are three classes that implement the interface `InjectableElementAnalyzer`:

- The final class `BeanDefinitionAnalyzer`, that can analyze instances of the class `AbstractBeanDefinition`.
- The final class `InjectableStringElementAnalyzer`, that can analyze instances of the class `InjectableStringElement`.
- The final class `CompositeInjectableElementAnalyzer` which involves the two previous analyzers and calls one of them, based on the actual type of the instance at input. As the name suggests, this refers to a composite design pattern.

Injectable String Element Analyzer

The analyzer is realized by the class `InjectableStringElementAnalyzer` and works in three steps:

1. Expansion of property placeholders to actual values — Specific properties defined by the element `context:property-placeholder` in XML documents, or by the annotation `@PropertySource` defined upon some bean with the annotation `@Configuration` can be utilized in expansion. However, if there is no such element or annotation, all properties in scope are utilized instead. Either way, all the properties are stored in an instance of the type `PropertySources` accessible from the class `Environment`. Note that expansion can work recursively and can also detect cycles. If a cycle is detected, the property placeholder is not being expanded. If the value for the given property placeholder is not defined by any of files with the extension `*.properties`, the placeholder is not being expanded as well.
2. Evaluation of SpEL expressions — There is currently no support for SpEL expressions. If some expression is detected, a warning is logged.

3. The evaluated string is returned as instance of the class TypedStringFlow. This flow is further converted by some implementation of the interface TypedStringConverter at the injection point, based on field's or parameter's type, into a particular type. The type conversion is further described in Section 5.2.3.

Bean Definition Analyzer

The analyzer is realized by the class BeanDefinitionAnalyzer. The analyzer is given an instance of the class AbstractBeanDefinition at its input, and it returns an instance of the class ExpressionFlowMap at its output, i.e., flow data for a given bean definition. The returned instance contains computed flow data of the bean object, which means its fields, or potentially a return value. In the analyzed program, when the program tries to access some bean from, e.g., an instance of the type ApplicationContext, an analysis of the particular bean is executed and its flow data are propagated into the desired place, which is usually some variable. The class BeanDefinitionAnalyzer internally uses an instance of the type AnalysisExecutor, which is our black-box to run the analysis as a service. The analyzer works lazily, i.e., flow data for a particular bean are computed and cached once requested. It is also worth mentioning that before analysis of the particular bean, its dependencies shall be analyzed beforehand. This is done before actual analysis of the bean, recursively, using a simple depth-first search algorithm. The recursive algorithm might lead to a potential stack overflow in case of some cyclic dependencies within the beans, and therefore we shall handle cyclic dependencies correctly. This is described in Section 5.2.1.

Within the class BeanDefinitionAnalyzer, there are several handlers to analyze a bean. Their purpose is to decompose the class BeanDefinitionAnalyzer into smaller pieces that can be properly unit-tested. All these handlers implement the interface BeanInitializationHandler and are applied in order when analyzing the bean. The following handlers are implemented, to analyze a general bean:

1. The class AutowiringModeHandler is a dispatcher to other handlers which can process the initial step in the analysis, based on the configured autowire mode:
    - The class DefaultAutowiringHandler is a handler for the classic construction of the bean with no magical autowiring. Specified initialization method arguments are used within this handler.
    - The class NameAutowiringHandler is a handler that can autowire properties based on their names, i.e., call appropriate setters with autowired arguments.
    - The class TypeAutowiringHandler is a handler that can autowire properties based on their types, i.e., call appropriate setters with autowired arguments.
    - The class ConstructorAutowiringHandler is a handler for handling the annotations like @Autowired, @Inject or @Resource defined on constructors. It also handles a case when there is no annotation but only a single constructor in the declaring class.

2. The class `FieldAutowiringHandler` is a handler for the annotations like `@Autowired`, `@Inject` or `@Resource` defined on fields. It scans a class type of the bean and injects flow data into fields that have any of the annotations.

3. The class `ExplicitXmlPropertiesHandler` is a handler that can process properties defined in an XML document, i.e., call appropriate setters with the specified arguments.

4. The class `MethodAutowiringHandler` is a handler for the annotations like `@Autowired`, `@Inject` or `@Resource` defined on methods. It scans a class type of the bean and calls methods that have any of the annotations with appropriate arguments.

5. The class `InitializationCallbackMethodHandler` is a handler to correctly process initialization callbacks. If a class type of the bean implements the interface `InitializingBean` then a method with the signature `afterPropertiesSet()`, methods with the annotation `@PostConstruct`, as well as custom initialization callback methods specified in XML document or defined by the annotation `@Bean`.

As mentioned in Section 5.1, there are some specialized bean definitions that need special care. Therefore, the class `BeanInitializationHandlerDispatcher` has been introduced. It contains a mapping from a subclass type of the class `AbstractBeanDefinition` to the particular handler. If the currently analyzed bean does not conform to any of the specified subclass types in the map, the fallback handler is used. The fallback handler is the list of handlers mentioned above, i.e., the handlers for a general bean. The map contains these entries:

- For instances of the class `XmlListBeanDefinition`, the class `XmlListBeanInitializationHandler` is utilized.

- For instances of the class `XmlSetBeanDefinition`, the class `XmlSetBeanInitializationHandler` is utilized.

- For instances of the class `XmlMapBeanDefinition`, the class `XmlMapBeanInitializationHandler` is utilized.

- For instances of the class `XmlPropertiesBeanDefinition`, the class `XmlPropertiesBeanInitializationHandler` is utilized.

Computing Flow Data of Spring Beans

Whenever some implementation of the interface `BeanInitializationHandler` attempts to analyze some method of a bean, it utilizes an instance of the type `AnalyzeExecutor`. The analysis returns an instance of the class `MethodFlowMap` representing flow data of the particular method, e.g., a constructor. However, such an instance contains flow data for a lot of entities that we are not particularly interested in, these are, e.g., some helper local variables defined within the method. We are interested only in flow data for field accesses, eventually flow data for the return value, if the analyzed method has any.

A completely different story, yet still closely related to the problematics, is injection of flow data of beans into fields and parameters. For both, it is necessary to correctly replace root target objects of accessed fields, so the analysis can work correctly. This corresponds to instances of the class `FieldAccessExpression` which consists of a target object and an instance of the class `Field`, which represents the field itself. In other words, our goal is to find such target object expression that is not of the type `FieldAccessExpression`, i.e., the root one, and replace it. For example, if the tracked expression is the field access `<placeholder>.b.c` and we want to inject flow data associated with the field access into the first argument of some method, our goal is to replace the field access expression `<placeholder>.b.c` with the field access expression `<argument 1>.b.c`. The point why this is necessary is explained in the rest of this section. Also, note that the *placeholder* expression is represented by the class `BeanPlaceholderExpression` and its goal is to unify prefixes of field accesses belonging to a particular bean. Otherwise, a prefix for some field access in a constructor of the bean could be different than a prefix for some field access in an initialization callback of the bean.

Flow data of every single analyzed bean are stored in an instance of the class `MutableBeanFlowMap`. This class extends the class `ExpressionFlowMap` defined in Bytecode Resolver, it is a variant of the class `MutableExpressionFlowMap` used throughout Bytecode Resolver with only one difference. Whilst the class `MutableExpressionFlowMap` contains a method to track arbitrary expression, the class `MutableBeanFlowMap` contains methods to track the particular expression either as an autowired field or as an autowired argument.

First, let's see how the tracking of the autowired field works. At its input, the method is given three parameters:

1. A bean definition of the class being injected into.
2. The field that we want to inject flow data into.
3. The expression to be injected.

```
@Component
class A {
   @Autowired
   public B b;

   @Autowired
   public A(B b) { ... }
}

@Component
class B {
   public C c;
}

```

```
class C {
   public D d;
}
```

Listing 5.3: An example of injecting flow data into the autowired field

Consider the example in Listing 5.3. The expression to be injected is expected to be some instance of the class `FieldAccessExpression` with the field defined in the class `B`, e.g., the field access `<placeholder B>.c.d`. The method which performs the symbolic injection transforms the expression to the field access `<placeholder A>.b.c.d`. The flows of the original expression are assigned to the transformed expression which represents the result of the injection. Alternatively, the original expression can be of the type `InjectableElementExpression`, which is an artificial expression representing return values of factory methods, methods annotated by the annotation `@Bean`, or values originating from the annotation `@Value`. In such a case, this helper expression is without any questions transformed into the expression `<placeholder A>.b`.

Now let's move to the tracking of the autowired arguments. This can be observed in the same example in Listing 5.3, it refers to the constructor injection in the class `A`. It works very similarly, the method is given the following at its input:

1. The parameter we want to inject flow data into.
2. The expression to be injected.

Here, several scenarios can happen:

1. The expression to be injected is of the type `FieldAccessExpression`. Then, it is expected that it shall be some autowired bean. For that reason, the target object of such expression should be some placeholder. Therefore, we replace the expression `<placeholder B>.b.c` with the expression `<argument N>.b.c`, because the expression is tracked as the argument. The number `N` refers to the position of the argument.
2. The expression to be injected is of the type `LocalVariableExpression`. Everything is fine then, no transformation is needed.
3. The expression to be injected is of some other type, e.g., a return value of the bean represented by the type `InjectableElementExpression`. Since the argument is autowired, it is expected to be tracked to the specified argument.

```
@Component
class C {
    private B b;

    public C() {
        this.b = new B(); // OK
        this.b.someField = 3; // OK
        A a = new A();
        a.someField = 4; // NOT INCLUDED
        C self = this;
        self.b.otherField = a.someField; // OK
        B bb = self.b;
        bb.x = 4; // OK
    }
}
```

Listing 5.4: An example of filtering field accesses

With the class `MutableBeanFlowMap` being introduced, we can move on to a description of the actual algorithm which can extract flow data of a bean method[2] from an instance of the class `MethodFlowMap`. The algorithm works in three steps:

1. Collect relevant root target objects for the field accesses[3]:

   (a) If the processed method has no return value and is instance, then consider only those fields whose root target object is an expression representing the zeroth argument of that method, i.e., the receiver. Otherwise, it is a factory method that should have some return value, consider all return value expressions then.

   (b) Find all expressions aliased with any of the collected expressions and consider only those fields whose root target object is any of these expressions.

   Using this approach, we restrict the resulting flow to the flow of the bean only, i.e., there will be no field accesses that may belong to some helper local variables, like in the example in Listing 5.4.

   It could also happen that the root expression is a local variable and serves as an alias for some field access like the local variable named `bb` in the example in Listing 5.4. If this is the case, replace the root expression with that field access and behave as it has never ever been a local variable. If applied to the example, the field access `bb.x` will be expanded to the field access `self.b.x`. With such field access, we can continue in the algorithm mentioned above, i.e., the local variable named `self` is an alias for the expression `this` which yields that the field access can be stored as flow data for the analyzed bean. In the example, the field access `self.b.otherField` is relevant and it contains all flows assigned to the field access `a.someField`. But the field access `a.someField` on its own, however, should not be marked as the one belonging to the bean object.

2. Iterate through the flow data stored in the original instance of the class `MethodFlowMap`:

   (a) During the iteration, check all stored field accesses whether their root target object is in the set, as computed in the previous step. If so, track such an expression in the structure `MutableBeanFlowMap` as bean object, which means to simply replace the root target object of the field

[2]Constructor, factory method, setter, initialization callback etc.

[3]For example, the root target object for the field access `a.b.c` is `a`.

access to an instance of another artificial expression, the expression of the class `BeanPlaceholderExpression`. The expression holds information about an instance of the class `AbstractBeanDefinition` it belongs to. Its main purpose is to give a common prefix for field accesses belonging to a particular bean, provide better exception safety and simplify the debugging.

   (b) Additionally, look for all return value expressions. Assign flows, originally belonging to the collected expressions, to a single instance of the class `InjectableElementExpression` representing the return value of the bean. This is especially necessary for those return types which have special semantics but have no fields. Among these types are, e.g., the class `java.lang.String` or collections.

3. Add information about the bean object. This step is rather simple. Track a particular expression of the type `InjectableElementExpression` and assign it an instance of a flow represented by the class `InjectedBeanFlow`, wrapping the defining instance of the class `AbstractBeanDefinition`. The flow can be then propagated over assignments and method calls freely. This is important in order to correctly resolve all possible call targets, i.e., this is necessary for functionality of dependency injection plugin as described in Section 5.3.2.

The algorithm discussed above is implemented in form of a single method in the class `AbstractBeanInitializationHandler` and is utilized by all the subclasses to store the flow data.

Handling of Cyclic Dependencies

As mentioned in Section 5.2.1, if there are some cyclic dependencies between two or more beans, stack overflow would be reached in case we had not supported the cyclic dependencies. Fortunately, we support them. We show the way the cyclic dependencies are resolved on the example in Listing 5.5.

```
<bean id="A">
  <constructor-arg ref="B"/>
</bean>

<bean id="B">
  <constructor-arg ref="C"/>
</bean>

<bean id="C">
  <constructor-arg ref="A"/>
</bean>
```

Listing 5.5: An example of cyclic dependencies among beans

First, we have to detect a cycle. However, this is rather simple. In our recursive depth-first search algorithm, the algorithm saves a path of references and if it finds the second occurrence of some bean in the chain, it stops the recursion and executes the algorithm for handling of cyclic dependencies. In the example, if the root bean is the bean named `A`, the path would be `A -> B -> C -> A` at the point the recursion would have stopped.

The algorithm for handling cyclic dependencies works as follows. For the chain of length n, analyze n different chains to get as precise results as possible. For example, for the chain `A -> B -> C`, analyze the chain itself at first, then the chain `B -> C -> A`, and the chain `C -> A -> B` at last. This guarantees us that flow data of the analyzed bean, e.g., the bean named `C`, are injected into the bean which references it, e.g., the bean named `B`, and they are at least partially correct.

However, it can be observed that the first chain, i.e., the chain `A -> B -> C` has a little disadvantage. Initially, we don't have any flow data for the bean named `A` which could get injected into the bean named `C`. Note that, in fact, it is the infinite chain `A -> B -> C -> A ->` $\cdots$. Therefore, the algorithm performs several iterations of the analysis. In the second iteration, it already has some flow data for the bean named `A` and therefore it can already inject them, unlike the first iteration. It is assumed that two iterations should be enough to retrieve results which are precise enough and this is also how it is implemented, however, there is actually a constant for debugging purposes if needed any time in the future. An alternative here would be a worklist algorithm, which might be rather time-consuming, but far more precise. The current solution for cyclic dependencies is very experimental, and we may consider replacing it with the worklist algorithm in the future.

It is also worth mentioning, that within the first iteration in the chain `A -> B -> C`, it is important to mark the bean named `C`, i.e., the last element, as it has no flow data. Otherwise, the analysis would enter into an infinite cycle, since the method to obtain flow data would be invoked to obtain flow data of the bean named `C` which would not have existed, and therefore the whole analysis process would start over.

The algorithm is a part of the class `BeanDefinitionAnalyzer`.

5.2.2 Finding Appropriate Autowire Candidates

To simulate a behavior of the Spring Framework, we need to be able to find appropriate autowire candidates for given entities. These two entities are:

- Fields. For any field that has any of the annotations like `@Autowired`, `@Inject`, or `@Resource` defined, we shall be able to find all beans that can be injected into it.
- Parameters. This has multiple use cases, e.g., for methods having the annotation `@Bean` or any of the annotations like `@Autowired`, `@Inject`, or `@Resource`, but the principle is similar to fields.

To find autowire candidates for a given entity, we need an access to an instance of the type `BeanContext` holding information about all loaded beans, as described in Section 5.1.

It is also worth mentioning that we shall preserve the over-approximation principle. In the context of finding autowire candidates, that basically means that there can be multiple candidates for a particular entity. When the Spring Framework tries to find a candidate for autowiring, it always finds a single one or throws an exception. However, we cannot do the same because of several reasons:

- If no autowire candidate has been found, we shall not treat that as an error. The user could provide an incomplete program and that is rather a problem one level up, i.e., in configuration and input providers. Therefore, we log this fact and continue with no flow data injecting into the given entity. It can also happen that the given entity is not required to be autowired, i.e., it can have the attribute `required` set to `false`.
- If multiple autowire candidates have been found, we shall not treat that as an error as well. The reason could be the same as above, however, in this case, the reason which is more likely to cause problems is a design of support for profiles, as described in Section 4.1.1.1. In other words, there can be multiple autowire candidates if multiple profiles are active.

The process of finding appropriate candidates works as follows:

1. Get all bean definitions matching a type of a point of injection, or if the point of injection defines the annotation `@Resource`, use the name defined by the annotation instead.
2. Filter these bean definitions to those matching qualifiers declared upon the point of injection.
3. Are there any primary beans?
    - If so, filter beans collected in the step 2 to those which are primary, and return them.
    - Otherwise, return all beans collected in the step 2.

The algorithm is implemented in the method named `getAutowireCandidates` which has two overloads, i.e., for parameters and fields, and is defined in the class `MutableBeanContext`.

Injection of Values

In this particular case, if we primarily mean values defined by the annotations `@Value`. We treat the annotation as a stronger variant of autowiring. In other words, when checking for the annotation `@Autowired` or others, our algorithm first checks whether a point of injection, i.e., parameter or field, defines the annotation `@Value`. If it does, it adds an instance of the class `InjectableStringElement` representing string literal in the value of the annotation `@Value` into the set of possible autowire candidates. The element is further analyzed in a classic way, in this particular case, by the class `InjectableStringElementAnalyzer`. In other words, we in fact treat these values as beans of the type `java.lang.String` declared in-place.

Note that the annotation `@Value` can be also defined on methods. In such a case, its value serves as a default value for all the methods' parameters. Therefore, when checking for autowire candidates for a particular method parameter, we shall take that into consideration and include the default value as well, if not specified otherwise.

Handling of Qualifiers

Qualifiers in the Spring Framework are useful when identifying a bean using its class type only is insufficient. If this is the case, the bean can be provided a qualifier which is basically a mapping between some keys and values for the keys. Similarly, injection points, i.e., parameters or fields, can be provided with some qualifiers. Qualifiers are mapped to annotation types, but they can also be defined in XML documents. There is already one qualifier prepared in the Spring Framework, the annotation `@Qualifier`. However, custom qualifiers can be defined as well. These custom qualifiers must declare the meta-annotation `@Qualifier`, in order to be considered. Then, when determining whether some bean is qualified to be injected into the point of injection, qualifiers defined on both the candidate bean and the injection point are compared as follows:

1. For each qualifier required by the point of injection, do the following:
    (a) Is there a qualifier defined on the queried bean with the same annotation type?
        i. If so, continue.
        ii. Otherwise, the bean is not qualified to be injected
    (b) Iterate over all the keys defined in the mapping and check their values on equality.
    (c) If any value is not matched, the bean is not qualified to be injected.
2. If we get to this step, it means that the bean is qualified to be injected into the specified point of injection.

The algorithm is implemented in the method named `isQualified` in the class `AbstractBeanDefinition`. It accepts a single argument, which is some annotated entity, i.e., the point of injection. The method is utilized when finding appropriate autowire candidates as described in Section 5.2.2, in the step 2.

```
@Qualifier("Qualified")
@Component
class Qualified extends Base {}

@Component
class NonQualified extends Base {}

abstract class Base {}

@Component
class MyBean {
  public MyBean(@Qualifier("Qualified") Base base) {}
}
```

Listing 5.6: An example of qualified and non-qualified beans

In the example in Listing 5.6, there are technically two beans of the type `Base`, however, only the bean of the type `Qualified` is qualified to be injected into the constructor of the class `MyBean`, because it defines the same qualifier which is defined at the point of injection, which is the first parameter of the constructor in this case. In other words, the method `isQualified` returns `false` for the bean of the type `NonQualified`, and returns `true` for the bean of the type `Qualified`.

5.2.3 Type Conversion

In XML documents or in values of the annotation `@Value`, values can be only specified as plain strings. For example, when specifying some number in XML document, it is in fact treated as an instance of the class `java.lang.String`, although it should rather be treated as an instance of the class `java.lang.Integer`. Therefore, there is a need to support some type of conversion, similar to the one in the Spring Framework. In our case, this will only involve conversion from plain string literals into flow data of the literals, based on the requested type. In most cases, the logic of converters' implementations just copies a behavior as defined by the Java Platform plugin.

We introduce the interface `TypedStringConverter` which defines two methods:

- A method that can decide whether the specified type can be converted by the converter. This specified type usually refers to the type of the injection point, i.e., parameter or field.
- A method that can convert an instance of the class `TypedStringFlow` into a set of flow data. Before the actual injection, this method is invoked and results from the method are injected. Note that instances of the class `TypedStringFlow` are the ones which represent values.

Currently, there is a support for conversion of the following types:

- The class `java.lang.String` which is realized by the final class `StringValueConverter`.
- The class `java.lang.Class` which is realized by the final class `ClassValueConverter`.
- The class `java.util.Properties` which is realized by the final class `PropertiesValueConverter`.
- The classes implementing the interface `java.lang.Iterable` which is realized by the final class `CollectionValueConverter`.
- The classes extending the class `java.lang.Number` and primitive types which is realized by the final class `NumberValueConverter`.

- The class `java.nio.file.Path`, the class `java.io.File` and the class `java.io.InputStream` which is realized by the final class `PathToFileValueConverter`.

The converters mentioned in the previous enumeration are registered within the class `CompositeTypedStringConverter`, which selects appropriate converter based on the actual type at injection point. When implementing a new class which implements the interface `TypedStringConverter`, it is necessary to register its singleton instance of such class into the class `CompositeTypedStringConverter`, so it can be recognized by the Spring Framework plugin.

5.3 Integration with Bytecode Scanner

The solution for the Spring Framework has a form of two plugins into Bytecode Scanner. In Section 5.3.1, we describe a dataflow plugin for the Spring Framework, which is necessary for propagating flow data of beans into the data lineage analysis. In Section 5.3.2, we describe a dependency injection plugin for the Spring Framework, which is necessary for a correct evaluation of targets of invocations upon bean objects.

5.3.1 Spring Framework Dataflow Plugin

The dataflow plugin is realized by the class `SpringFrameworkDataflowPlugin` which extends the abstract class `DataflowPlugin` defined by Bytecode Scanner. The aformentioned abstract class makes the implementation easy, the only we have to do is to implement so-called *propagation modes* which semantically describes a handled method. Our goal is to handle overloads of the method named `getBean` defined in the interface `BeanFactory`. We are particularly interested in the methods `getBean(String)` and `getBean(Class<?>)`.

To do that, we design the abstract class `AbstractGetBeanPropagationMode` which is a realization of propagation mode to handle the overloads of the method named `getBean`. It defines an abstract method that accepts a set of flow data for the argument with information about the queried bean and returns a set of bean definitions. The class extends the abstract class `PropagationMode` defined by Bytecode Scanner and works as follows:

1. Obtain a set of instances of the class `AbstractBeanDefinition` that are queried by the invocation of the method named `getBean`.
2. Compute flow data for these instances as described in Section 5.2, by utilizing the class `BeanDefinitionAnalyzer`.
3. Propagate the computed flow data into a return value of the invocation.

The subclasses of the abstract class `AbstractGetBeanPropagationMode` are the following:

- The final class `ClassGetBeanPropagationMode` which expects a set of instances of the class `LiteralFlow` wrapping a constant of the type `ClassVa-`

lue in its argument. These flows are created in the data lineage analysis whenever the analyzer encounters the statement MyClass.class. The propagation mode collects these types and looks up an instance of the type BeanContext to find appropriate bean definitions. These bean definitions are further returned.

- The final class NameGetBeanPropagationMode which expects a set of instances of the class LiteralFlow wrapping a constant of the type String-Value in its argument. These flows are created in the data lineage analysis whenever the analyzer encounters a string literal defined in bytecode. The propagation mode collects these literals and looks up an instance of the type BeanContext to find appropriate bean definitions. These bean definitions are further returned.

Finally, to make sure that invocations of the overloads of the method named getBean are handled correctly, we need to configure the dataflow plugin, following conventions defined by the abstract class DataflowPlugin. The configuration of the plugin is realized by the XML document, located in resources, named dataflow-plugin-spring-framework-configuration.xml and its content is listed in Listing 5.7.

```xml
<Package name="org.springframework.beans.factory">
    <Type name="BeanFactory">
        <Method name="getBean"
                      returnType="java.lang.Object">
            <Argument position="0"
                      type="java.lang.Class"/>
            <Propagation mode="getBeanByClass"
                         from="arg0"
                         to="returnValue"/>
        </Method>

        <Method name="getBean"
                returnType="java.lang.Object">
            <Argument position="0"
                      type="java.lang.String"/>
            <Propagation mode="getBeanByName"
                         from="arg0"
                         to="returnValue"/>
        </Method>
    </Type>
</Package>
```

Listing 5.7: A configuration of the dataflow plugin for the Spring Framework

5.3.2 Spring Framework Dependency Injection Plugin

The dependency injection plugin is realized by the class SpringFrameworkDependencyInjectionPlugin which extends the abstract class DependencyInjectionPlugin described in Section 6.3. The plugin is implemented in such a way that every time some bean is analyzed by the class `BeanDefinitionAnalyzer` from Section 5.2.1, one additional helper instance of the class `InjectedBeanFlow` is assigned to its flows. This corresponds to the step 3 of the algorithm in Section 5.2.1. The flow works as any other flow, e.g., if the particular bean gets injected into some parameter or field, the flow is propagated into the particular entity as well.

The dependency injection plugin for the Spring Framework works in such a way that before every call of some method, it checks flows assigned to the receiver of the invocation. If there is some instance of the class `InjectedBeanFlow`, the type of the bean that got injected into the receiver is extracted and a particular overridden method is found if there is such. This way, the algorithm constructs a set of methods that would be called at runtime, and this set is utilized within the context-sensitive call graph described in Section 4.2.3.

6. Call Graph

In this chapter, we discuss technical details of the construction of the call graph which was introduced in Section 3.2. In Section 6.1, we provide details on implementation of Rapid Type Analysis. In Section 6.2, we describe an implementation of both context-insensitive and context-sensitive call graphs. Section 6.3 describes an extension of context-sensitive call graph with dependency injection plugins. Last, Section 6.4 is focused on lambda expressions, as support for them is closely connected to dependency injection plugins.

6.1 Rapid Type Analysis

As we discussed in Section 4.2.2.1, the Rapid Type Analysis filters result from method selection to compute possible invocation targets of a particular method. The filtering is based on a set of instantiated types. Precisely, it filters invocation targets to those, whose declaring type has been instantiated. Our major goal is to construct the set of instantiated types. To do that, we need to recursively traverse reachable methods in the analyzed program and collect types instantiated by the instruction `new`. As we are traversing the bytecode of the analyzed program from the entry-point method, we need to deal with bytecode instructions that can invoke a method. There are several instructions that can invoke a method in bytecode:

- The instruction `invokestatic`. This instruction is utilized to invoke static methods. There can be only a single call target of the method, being the method itself.
- The instruction `invokespecial`. This instruction is utilized to invoke an instance method without a look-up of the virtual method table. This instruction is usually utilized to call a constructor of the class. Similarly to the instruction `invokestatic`, there can be only a single call target of the method, being the method itself.
- The instruction `invokevirtual`. This instruction is utilized to invoke a virtual method. At runtime, the virtual method table is utilized to find the target method. However, in the data lineage analysis, we over-approximate the results by a set of methods that could possibly be called there, as we usually do not know the exact type of the receiver. Such a set is being computed by the Rapid Type Analysis.
- The instruction `invokeinterface`. This instruction is utilized to invoke a method defined by the interface. From our perspective, we can treat this instruction in the same way as the instruction `invokevirtual`.
- The instruction `invokedynamic`. This instruction provides a mechanism for binding a method invocation to its target method at runtime [13]. As we cannot process dynamic invocation in the data lineage analysis which is static, we only care about its arguments. Precisely, we care about the

arguments of the type `java.invoke.MethodHandle`. These arguments basically provide references to methods, along with their invocation modes, which are passed to the bootstrap method of the instruction. For purposes of the Rapid Type Analysis, we extract these references and treat them as they were invoked at the point of the particular instruction `invokedynamic`. Note that our goal is to traverse reachable methods, and thus this approach is perfectly fine for the Rapid Type Analysis.

Our implementation of Algorithm 3 from Section 4.2.2.1 which can construct the set of the instantiated types is located in the class `RtaMethodInvocationTargetSelector.Builder`. The implementation utilizes a builder pattern to build an instance of the class `RtaMethodInvocationTargetSelector`, which is the realization of the Rapid Type Analysis. Moreover, the builder class features the following methods to customize the behavior of the building algorithm:

- The method `setIncludeAllApplicationTypes` accepts a single argument of the type `boolean` and decides whether all application types should be marked as instantiated or not. Setting this flag to `true` may speed up the construction of the Rapid Type Analysis significantly, but can make the analysis less precise.
- The method `setIncludeLibraryMethods` accepts a single argument of the type `boolean` and decides whether all library methods should be traversed as well or not. Setting this flag to `false` may speed up the construction of the Rapid Type Analysis significantly, but can make the analysis less precise, especially in terms of processing library callbacks.
- The method `setIncludeStaticInitializers` accepts a single argument of the type `boolean` and decides whether static initializers should be traversed as well or not. Since the static initializers can never be invoked directly, we shall incorporate them into the construction of the Rapid Type Analysis as well.

Additionally, the builder class features two other methods, the methods named `addEntryPoint` and `withEntryPoints`, that can set the entry-point methods of the analyzed program from which the Rapid Type Analysis should be constructed.

6.2 Call Graph Variants

First of all, to implement either of the variants of call graph, that means context-insensitive and context-sensitive call graph, it is necessary to have the type hierarchy as discussed in Section 4.2.1. The implementation is located in the class `StandardTypeHierarchy` and is rather simple. Except for its straightforward description in Section 4.2.1, it utilizes a little optimization with the use of modifiers in Java Platform. If the type being inspected for its relations with other types in the analyzed program is marked as `package-private`, it only makes sense to scan the declaring package, since no other type can extend the particular type, implement it respectively.

The class `ContextInsensitiveCallGraph` utilizes the Rapid Type Analysis from Section 6.1 to implement the context-insensitive call graph. Respectively, the class `ContextInsensitiveCallGraphBuilder` implements a builder pattern which in the method named `build` runs the implementation of Algorithm 2 from Section 4.2.2 to build the graph. The graph is represented by the class `ContextInsensitiveCallGraph`. However, keep in mind that invocations caused by the instruction `invokedynamic` do not appear in the context-insensitive call graph. The most common usage of the instruction is for lambda expressions. A way of how they are handled is described in Section 6.4.

The most important for us is the context-sensitive call graph which is implemented in the class `ContextSensitiveCallGraph`. Its mutable counterpart is located in the class `MutableContextSensitiveCallGraph` which extends the read-only variant in the class `ContextSensitiveCallGraph`. For us, only the mutable variant is important. However, in later stages of the data lineage analysis, there is a need of finding callers for the specified invocation, callees respectively. At that phase, the call graph shall be modifiable no longer, and that is also the reason for designing the read-only variant. The mutable variant features a method that can add a relation between the specified caller and the specified callee. This method is called whenever the data lineage analyzer encounters some instruction `invoke` and corresponds to adding an edge into the graph. The context-sensitive call graph accepts an instance of the class `ContextInsensitiveCallGraph` in its constructor which is being utilized to obtain possible call targets for the virtual method invocation.

6.3 Dependency Injection Plugins

A dependency injection plugin is an extension of the context-sensitive call graph. It is realized by the abstract class `DependencyInjectionPlugin` which extends the class `ResolverPlugin` defined by Bytecode Scanner. Moreover, it implements the interface `CallHandler` which is the actual realization of extension into the context-sensitive call graph. The interface defines the following methods:

- A method to obtain all callers for the specified instance of the class `InvocationContext`.
- A method to obtain all callees for the specified instance of the class `InvocationContext`.
- A method to obtain possible call targets for the specified instance of the class `InvocationContext`. This is meant, e.g., if there is some injected object in the receiver, then the method should return all overriding methods matching types injected into the receiver. However, the definition of the method is generic enough so this might not be the only usage of it. It should return an empty set in case the handler cannot process the invocation.

Whilst the first two methods are optional and return empty set by default, the third method is the most important one so the whole concept can work. Additionally, an additional helper method for a check of supported invocation modes is present as well, however, the method forms rather some sort of runtime check to assure the configuration is correct.

The interface `CallHandler` is further utilized by the class `ContextSensitive-CallGraph` which provides information about the method invocations. This is also how it is assured that the correct methods will be analyzed by the core analysis algorithm. Precisely, within the class `ContextSensitiveCallGraph`, the class `DependencyInjectionPluginManager` is utilized. This class implements a composite pattern for the class `DependencyInjectionPlugin`. During a search for possible call targets in the context-sensitive call graph, the dependency injection plugins are being tried in order, by calling the method to obtain possible call targets defined by the interface `CallHandler`. If the method returns a non-empty set, the resulting set forms a result of the context-sensitive call graph query for possible call targets. Otherwise, if none of the dependency injection plugins returns a non-empty set, results from the context-insensitive call graph are utilized instead.

There is currently only one implementation of the class `DependencyInject-ionPlugin`, being the dependency injection plugin for the Spring Framework from Section 5.3.2. On top of that, there is one additional implementation of the interface `CallHandler`, utilized for handling lambda expressions. The way of how they are handled is described in Section 6.4.

6.4 Handling of Lambda Expressions

Lambda expressions need special care in the data lineage analysis. The reason behind that is a fact that they do not appear in the context-insensitive call graph, since they can be propagated over assignments and method calls which are context-sensitive. Therefore, our goal is to extend support in context-sensitive call graph to support lambda expressions as well.

Our solution for lambda expressions relies on the expression represented by the class `InvokeDynamicExpression`. In short, the expression is created whenever the analyzer processes the instruction `invokedynamic`. It contains information about lambda to be invoked which could be technically treated as a lambda object.

In analysis, when processing the expression `InvokeDynamicExpression`, it is always associated with an instance of the class `LambdaFlow`, if applicable, i.e., when the handler method is equal to the method named `metafactory` defined in the class `java.lang.invoke.LambdaMetaFactory`. The flow basically wraps the expression but also contains flow data of so-called *captured arguments*. These are the arguments that are not present in the signature of the method defined by the functional interface method representing the lambda but are present in the actual lambda method to be invoked. These arguments are usually some fields or local variables defined in the method where the lambda is defined. In the example in Listing 6.1, in the lambda expression, there is one captured argument, being the variable named `one`.

The instance of the class `LambdaFlow` contains all the following:

- The actual lambda method, so-called *body method*, along with its invocation dispatch mode.

- Expressions representing captured arguments in the method call which created the flow.

- Flows for expressions representing captured arguments in the method which created the flow.
- An instance of the class `InvocationContext` for the method call which created the flow. This is necessary for the propagation of captured arguments back once the lambda is processed.

The presented flow can be propagated, as any other flows, over assignments and method calls which the lambda expression does as well at runtime.

```
void foo() {
    int one = 1;
    Function<Integer, Integer> plusOne =
        number -> number + one;
}
```

Listing 6.1: An example of lambda expression with a captured argument

Lambda Call Handler

The place where instances of the class `LambdaFlow` are being processed is the class `LambdaCallHandler`, a class implementing the interface `CallHandler`. This is an example of another usage of the interface `CallHandler` besides dependency injection plugins.

```
boolean myMethod() {
    Predicate<String> hasMoreThanTenCharacters =
        str -> str.length() > 10;

    return hasMoreThanTenCharacters
        .test("master␣thesis");
}
```

Listing 6.2: A simple example of lambda expression

```
myMethod()Z
 L0
  INVOKEDYNAMIC myMethod()
   Ljava/util/function/Predicate; [
    // handle kind 0x6 : INVOKESTATIC
    java/lang/invoke/LambdaMetafactory.metafactory(...)
     Ljava/lang/invoke/CallSite;
    // arguments:
    (Ljava/lang/Object;)Z,
    // handle kind 0x6 : INVOKESTATIC
    lambda$myMethod$3(Ljava/lang/String;)Z,
    (Ljava/lang/String;)Z
  ]
  ASTORE 1
 L1
  ALOAD 1
```

```
LDC "master␣thesis"
INVOKEINTERFACE java/util/function/Predicate.test
 (Ljava/lang/Object;)Z (itf)
IRETURN
```

Listing 6.3: A bytecode of the example in Listing 6.2

Now, consider the example in Listing 6.2 along with its simplified bytecode in Listing 6.3. In the example, the lambda expression object is created by using the instruction `invokedynamic` at `L0`. In the analysis, an instance of the class `LambdaFlow` is assigned to the particular expression representing the dynamic invocation, i.e., the expression of the type `InvokeDynamicExpression`. The flow gets further propagated over the assignment into the expression representing the first local variable, which is achieved by the instruction `ASTORE 1` and is realized by the local variable named `hasMoreThanTenCharacters`.

When processing the method call at `L1`, i.e., the instruction `invokeinterface`, the flow is present in the receiver. This is achieved by the instruction `ALOAD 1`. Thus, everything the class `LambdaCallHandler` has to do is to check all instances of the type `LambdaFlow` in the receiver expression and then create instances of the class `LambdaInvocationContext` based on the collected flows. In other words, our approach totally neglects the call of the method named `test` defined by the functional interface, but it replaces the call with the actual method representing the lambda expression. Therefore, instead of adding an edge between the invocation of the method named `myMethod` and the invocation of the method named `test` into the context-sensitive call graph, the edge between the invocation of the method named `myMethod` and the invocation of the method named `lambda$myMethod$3` is added instead. Note that captured arguments shall be included in each instance of the `LambdaInvocationContext` if any. In our example, there are no captured arguments.

Within the class `LambdaCallHandler`, it is also necessary to mark the method call which created the instance of the class `LambdaFlow` as a caller of the particular instance of the class `LambdaInvocationContext`. This is necessary because if the lambda method causes any side effects on captured arguments, their flows must be propagated back into the particular place, i.e., the method which created the lambda expression by using the instruction `invokedynamic`. This behavior is not ensured implicitly by the worklist algorithm, since the lambda method does not necessarily have to modify the flow data of the actual caller. It can simply modify just captured arguments which the caller technically does not know about. Note that the other direction, i.e., marking the instance of the class `LambdaInvocationContext` as a callee of the method invocation which created the instance of the class `LambdaFlow`, is handled implicitly. If the flow of captured arguments has changed, a new instance of the class `LambdaFlow` is created and this flow is further propagated into the point of actual invocation.

Furthermore, we also had to modify the class `MethodAnalysisVisitor` to reflect our support for lambda expressions. This class implements a visitor pattern and its instance is given to the bytecode interpreter at its input. We made two changes in its behavior:

- The changes in the method named `visitInvokeDynamicInstruction` – As mentioned above, instances of the class `InvokeDynamicExpression` are associated with instances of the class `LambdaFlow` in this method. Then, the captured arguments must be propagated back to the method flow at the same point. All instances of the class `LambdaInvocationContext` corresponding to the specified instance of the class `LambdaFlow` are collected and the arguments are propagated back. However, this is relatively easy to achieve, since we can create a mapping between expression prefixes of the captured argument in the owner method and the captured argument in the actual lambda method.

- The changes in the method named `visitInvokeInstruction` – When invoking lambda expression, the arguments shall be matched when propagating data out of the method call. This effectively means that we have to skip captured arguments at the beginning, and the receiver of the functional interface's method as well.

7. Evaluation

The solution presented in the thesis is able to analyze a majority of applications utilizing the Spring Framework. The implementation of the plugin for the Spring Framework includes both unit and integration tests and meets general criteria on code coverage. To test the plugin properly, we have also implemented two realistic test examples, on top of unit and integration tests. The tests are presented in Section 7.1, along with their data lineage graphs, i.e., results of the analysis. However, the discussed solution has several limitations. The limitations are discussed in Section 7.2. The structure of the project is described in Appendix B.

7.1 Test Data

The plugin is carefully tested by both unit and integration tests. These tests are included in the folder named `test` in the module. We use the Spring Framework as a basic infrastructure for testing since we can re-use the beans from the production environment, and in the testing environment we provide the framework with beans relevant for testing, usually the test data only. The test data for the integration tests are included in the module named `manta-testing-bytecode-resolver-plugin-spring-framework`. During the build of the module with the plugin, the testing module is compiled into the file `*.jar` and copied into the module with the plugin. In tests, the compiled file with basic test examples is loaded into an instance of the type `AnalysisScope` which is further used in the integration tests.

Moreover, for end-to-end testing, we have implemented two realistic non-trivial examples which test the integration of the plugin with Bytecode Scanner as a whole. These tests are intended to be run on top of the MANTA Flow platform which can produce a graph with dataflows.

The example manta-testing-bytecode-ecdc-etl

This example is a Spring application that takes advantage of open data provided in the comma-separated values format by the European Centre for Disease Prevention and Control. It uses two data sources, the first one to fetch daily reports of cases provided by member countries of the European Union on COVID-19 pandemic, the second one to fetch daily reports of vaccination progress among the European Union.

Here we describe how the application works. First, the values are downloaded from the Internet using standard API provided by the Java Platform. Then, the program parses lines in the file one by one into a list of intermediate values. Furthermore, these intermediate values, i.e., daily reports, are reduced to cumulative reports in the specified time horizon. This is done for both reports of cases, and reports of vaccinations. Last, cumulative reports are written into the standard output. Similarly, these reports are updated in the schema in MANTA's test database. This whole machinery is configured and executed by the Spring Framework, testing the most commonly used features in the Spring Framework plugin.

As a result of the program's analysis, it is expected that there will be a data flow between the data source for reports of cases, the console, and the database. Similarly, there will be a data flow between the data source for reports of vaccinations, the console, and the database. In Figure 7.1, there is the output graph with all edges forming the corresponding data flow path highlighted by the red color leading from the reports of cases. Similarly, in Figure 7.2, there is the output graph with all edges forming the corresponding data flow path highlighted by the red color leading from the reports of vaccinations. We can see that data from both the files are stored in the database using a single SQL statement.

The example manta-testing-bytecode-ecdc-queries

This example is a Spring application that utilizes the results computed by the example `manta-testing-bytecode-ecdc-etl` in such a way that it queries the same database. Precisely, it queries two tables, one with cumulative reports, one with data for countries. Then, for each cumulative report, it finds information about a country that the report belongs to. These data are written into the standard output in a human-readable format. Similarly, the example is configured and executed by the Spring Framework, testing the most commonly used features in the Spring Framework plugin.

As a result of the program's analysis, it is expected that there will be a data flow between the two tables, and the console. The result is visualized in Figure 7.3.

7.2 Limitations

In this section, we discuss the major limitations of the Spring Framework plugin. These are considered a future work at the same time, and therefore we also present a basic outline of the solution for them. On top of that, as the solution is an extension of Bytecode Scanner, it inherits its limitations as well. Among these limitations is memory consumption, which can be quite large and depends on the size of the analyzed program. Next, the analysis can sometimes get unreasonably slow. On the other hand, Bytecode Scanner is one of the latest scanners in the MANTA Flow platform, and therefore there is plenty of space for optimizations. Optimizations of Bytecode Scanner itself may influence the performance of the plugin for the Spring Framework since it uses the analyzer as a service to analyze beans defined in the analyzed program.

Scope of Spring Beans

In the current implementation, whenever some bean is accessed in the analyzed application, the flow data for such pre-analyzed bean are returned. However, this refers just to prototype scope. To achieve the functionality of singleton scope, after every single method analysis, it would be required to track the bean and propagate new flow data to our representation of application context where we store flow data of beans. Since a vast majority of beans have a singleton scope, the improvement of handling scopes of beans is desirable.

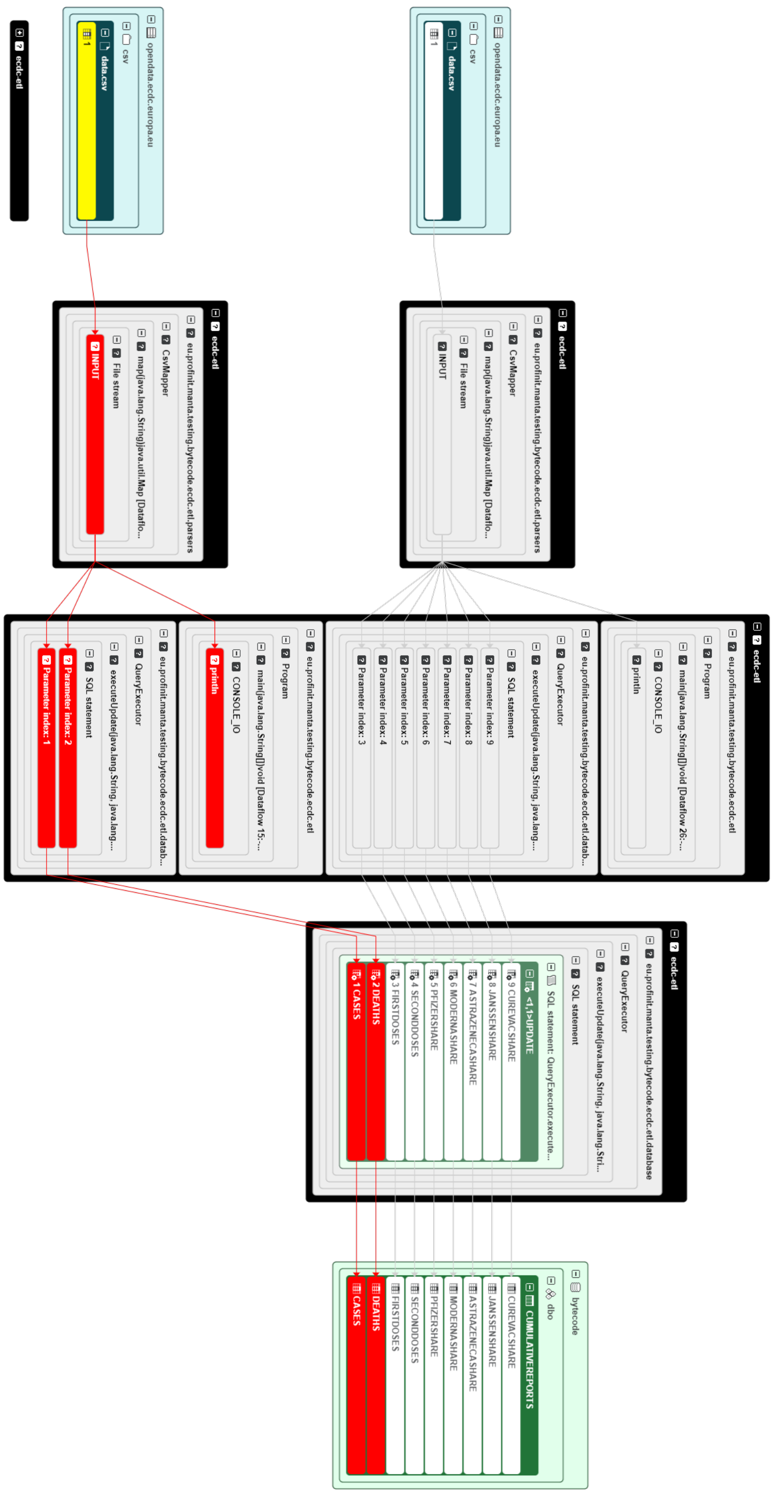


Figure 7.1: A graph of the ECDC ETL example with highlighted edge leading from the reports of cases

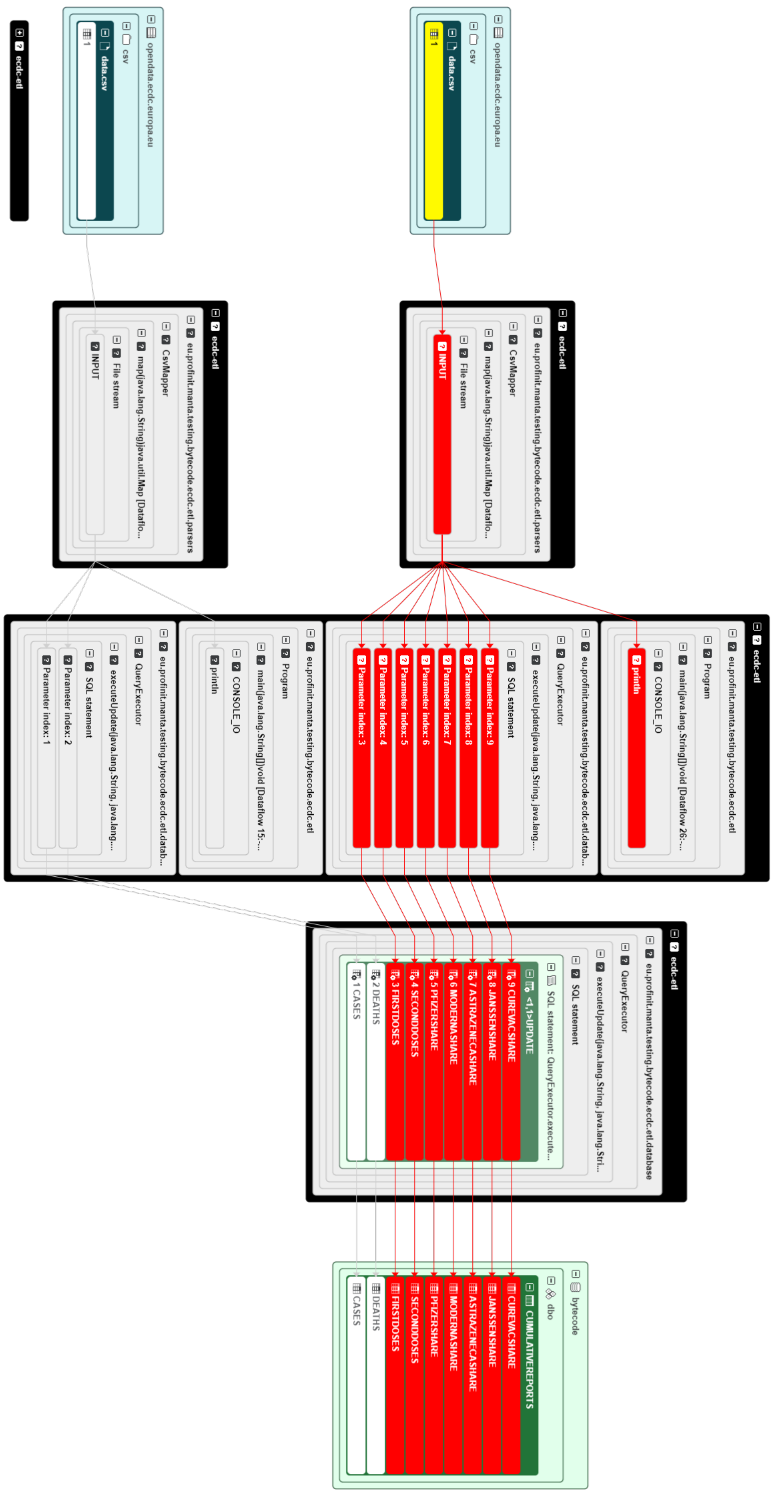
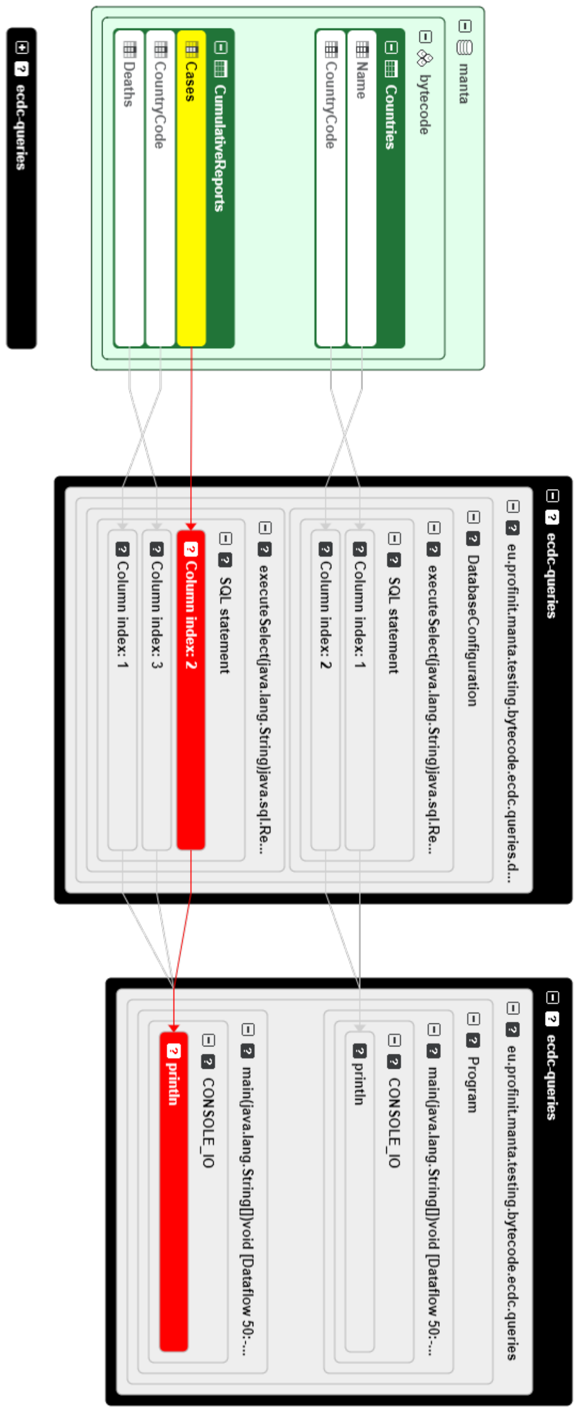


Figure 7.2: A graph of the ECDC ETL example with highlighted edge leading from the reports of vaccinations



manta
bytecode
Countries
Name
CountryCode
CumulativeReports
Cases
CountryCode
Deaths
ecdc-queries
ecdc-queries
eu.profinit.manta.testing.bytecode.ecdc.queries.d...
DatabaseConfiguration
executeSelect(java.lang.String)java.sql.Re...
SQL statement
Column index: 1
Column index: 2
executeSelect(java.lang.String)java.sql.Re...
SQL statement
Column index: 2
Column index: 3
Column index: 1
ecdc-queries
eu.profinit.manta.testing.bytecode.ecdc.queries
Program
main(java.lang.String[])void [Dataflow 50:...
CONSOLE_IO
println
main(java.lang.String[])void [Dataflow 50:...
CONSOLE_IO
println

Figure 7.3: A graph of the ECDC QUERIES example

In the core analysis, there is the class StaticFieldFlowPropagatingMethod-HandlerDecorator which implements the interface MethodHandler and a decorator pattern. This class decorates the core analysis algorithm for a single method analysis. It is responsible for the propagation of flows of static fields into the method call before the analysis, and after the analysis, it propagates the flows out to some global variable. In the plugin for the Spring Framework, we might need similar decorator which would propagate flow data of the analyzed bean out, in case it has a singleton scope. Whether the analyzed method is called upon some bean receiver or not, would be determined based on instances of the class InjectedBeanFlow in the receiver, which denotes that there is the bean in the receiver of the method invocation.

Support for SpEL Expressions

Spring Expression Language is an important aspect of the Spring Framework. However, in the current implementation, whenever there is some SpEL expression, we only log a warning that we currently cannot process these expressions. The proposed solution for SpEL expressions should take an advantage of the class SpelCompiler defined by the Spring Framework. Note that our goal is not to evaluate such an expression, but to analyze it instead. The mentioned class should be able to compile the expression into bytecode. Then, we can run our bytecode analysis onto the compiled expression. This way, we should be able to analyze even complex expressions and propagate data lineage information correctly. Alternatively, we might present our interpreter of SpEL expressions, however, such solution might be unnecessarily over-complicated.

Support for Generic Arguments in Autowiring

The current implementation does not support autowiring of generic types because of the lack of support for generic arguments in Bytecode Infrastructure. In other words, if the point of injection is of the type List<C> and there are beans of the types List<C>, List<D>, and List available, all of these beans will get injected. However, the goal here is to inject the bean of the type List<C> only, which would make the analysis far more precise. The proposed solution should influence the process of finding appropriate autowire candidates only from Section 5.2.2. However, based on the implementation of the support for generic arguments in Bytecode Infrastructure, it is also possible that no changes would be required, except for the support for collections originating from elements in the schema util where it would be necessary to match generic arguments manually.

Support for Type Conversion in Autowiring

The current support for type conversion from Section 5.2.3 only supports converting string literals into particular types. The goal here is to extend the support for type conversion to support not only string literals. However, this is partially influenced by the support for generic arguments in Bytecode Infrastructure. For example, if the point of injection is of the type List<T> and there is no explicit bean of the type List<T>, all beans of the type T should get injected. Similar applies to a concept utilizing the interface Provider<T>. Note that support for this kind of type conversion should influence the process of finding appropriate autowire candidates only from Section 5.2.2.

8. Conclusion

In the thesis, we presented a way of how the data lineage analyzer can be extended with support for dependency injection frameworks. To demonstrate the practical usefulness of the proposed concept, we applied the concept on dependency-injection-related features of the Spring Framework. Next, we implemented the plugin for the Spring Framework for Bytecode Scanner, a data lineage analyzer for programs written in Java bytecode in the MANTA Flow platform. As the dependency injection is focused on invocations of methods that are processed implicitly by the dependency injection framework, we also had to present several modifications of the call graph, to support dependency injection as well. The most important modification is a switch from context-insensitive call graph to context-sensitive call graph, which can support dependency injection plugins.

8.1 Related Works

The thesis covers two topics, each of them has different works that are less or more related to it:

- The topic of call graph construction deals with the problem of how to construct a call graph along with dependency injection frameworks incorporated into it. The C# Scanner [3] describes a way of how the call graph can be constructed, and we have gained a lot of inspiration there. However, it has no support for dependency injection frameworks, and therefore we have presented a way of how to extend it to support dependency injection frameworks as well.

- The topic of the data lineage analysis of dependency-injection-related features in the Spring Framework is unique for us since it targets the MANTA Flow platform. There was no support for the Spring Framework in the MANTA Flow platform previously. The work related to the topic is the Spring Framework [1] itself, as we have to simulate its behavior in many aspects. We could not take an advantage of the framework itself, because it is a runtime matter, and our analysis is static.

8.2 Future Plans

In the future, we plan to improve our support for the Spring Framework with support for:

- Scope of beans.
- SpEL language.
- Generic arguments in autowiring.
- Type conversion in autowiring.
- Spring Boot.

The highest priority has the plugin for Spring Boot as its popularity has increased recently. Since Spring Boot is built on top of the Spring Framework, it could use the plugin for the Spring Framework introduced in this thesis in its core. Additionally, as future work, we may focus on features of the Spring Framework not related to dependency injection.

A. User Documentation

To run the analysis, it is necessary to have the MANTA Flow platform installed. The platform is a commercially licensed product and is not a part of the thesis. Running of the analysis is fairly easy, all the user has to do is to configure inputs for a particular scanner. For us, only Bytecode Scanner is important, however, if the analyzed program accesses the database, it is also necessary to configure scanners for particular database providers, e.g., MSSQL Scanner or Oracle Scanner, to have complete information about the data lineage. If the database scanners are not configured, the tool, however, is still able to produce results of the bytecode analysis, but the database nodes are not matched.

The configuration for Bytecode Scanner is located in the file `cli/scenarios/manta-dataflow-cli/etc/bytecode/template.properties`. Actually, the file is only a template and it is expected to be copied into the same folder, changing its name to whatever the user wants to, usually a name of the analyzed program. It contains the following properties which are documented in greater detail in the template file itself:

- `bytecode.system.id` – An identifier of the analyzed program, this value is irrelevant for the analysis. The identifier is only shown in a visualizer.
- `bytecode.system.application.path` – A filesystem path to the analyzed application, usually a file with the suffix `*.jar`. For example, `/home/user/manta-testing-bytecode-ecdc-etl-1.3-SNAPSHOT-spring-boot.jar`.
- `bytecode.system.application.basePackage` – A base Java package of the analyzed application. For example, `eu.profinit`.
- `bytecode.system.java.standardLibrary.path` – A filesystem path to Java Platform standard libraries. For example, `C:/Program Files/Java/jre1.8.0_251`.
- `bytecode.system.plugin.springframework.profiles.active` – A configuration of active Spring profiles, actually a comma-separated list.

Once the input is configured, it is necessary to run the analysis. It can be done in the following steps:

1. Run the flow server by executing the file `server/bin/startup.bat`.
2. Run the analysis by executing the file `cli/scenarios/manta-dataflow-cli/bin/_run.bat`. This can take some time.
3. View the results visualized in the graph at the address `localhost:8080/manta-dataflow-server/viewer`.

Note that the tutorial has been provided for scripts with the suffix `*.bat` only, however, each such script file has its Unix counterpart with the suffix `*.sh`.

B. Project Structure

Bytecode Scanner, as any other scanner in the MANTA Flow platform, utilizes Maven scripts for builds and the Spring Framework for dependency injection, and so do we. Within the scanner, the code is distributed into multiple modules. Particularly, each plugin has its own module. Our plugin for the Spring Framework is implemented within the module named `manta-connector-bytecode-resolver-plugin-spring-framework` and has the following package hierarchy:

- The package `beans` contains structures and algorithms useful for parsing definitions of beans.
- The package `expression` contains artificial expressions that we had to introduce.
- The package `flow` contains algorithms that can analyze bean definitions, as well as subclasses of the class `Flow` that we need to obtain flow data of some bean.
- The package `propagation` contains propagation modes that can propagate flow data of beans into the core analysis.

Moreover, as it was necessary to modify already existing modules in Bytecode Scanner, in the attached code, we also include code fragments from the following modules:

- The module `manta-connector-bytecode-infrastructure` contains an implementation of type hierarchy. Generally speaking, the module is a representation of JVM for static analysis purposes.
- The module `manta-connector-bytecode-resolver-analysis` contains an implementation of call graph, i.e., Rapid Type Analysis, context-insensitive call graph, context-sensitive call graph and solution for lambda expressions. On top of that, it also includes classes that enable to call the analysis as a service, particularly the interface `AnalysisExecutor` and its subclasses.
- The module `manta-connector-bytecode-resolver-plugin-core` defines structures and basic infrastructure for all the dependency injection plugins. All modules representing the plugins are expected to reference this module.
- The module `manta-testing-bytecode-resolver-plugin-spring--framework` contains very basic test examples for the Spring Framework plugin.

Bibliography

[1] Spring Framework Documentation. `https://docs.spring.io/spring-framework/docs/current/reference/html/`.

[2] MANTA Flow Platform. `https://getmanta.com/about-the-manta-platform/`.

[3] Lukáš Riedel, Tereza Storzerová, Jan Joneš, and Jakub Sýkora. MANTA C# Scanner. 2020.

[4] T. J. Watson Libraries for Analysis. `https://github.com/wala/WALA`.

[5] Martin Mečiar. Static data flow analysis for Java programs. *Master thesis, Charles University*, 2019.

[6] ASM. `https://asm.ow2.io/`.

[7] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery.

[8] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 77–101, Berlin, Heidelberg, 1995. Springer-Verlag.

[9] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 324–341, New York, NY, USA, 1996. Association for Computing Machinery.

[10] Jakarta Contexts and Dependency Injection Specification. `https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.pdf`.

[11] Spring Boot Documentation. `https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/`.

[12] JVM Method Selection. `https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-5.html#jvms-5.4.6`.

[13] K. Ali, X. Lai, Z. Luo, O. Lhotak, J. Dolby, and F. Tip. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering, available online (early access), doi: 10.1109/TSE.2019.2956925*, (01).