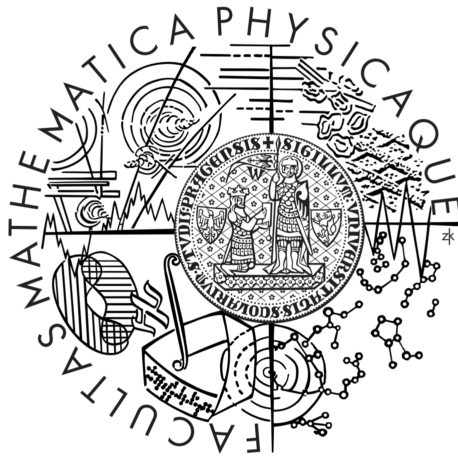
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Kleprlík

Performance and Usability Improvements for Data Lineage Analysis of C# Programs

Department of Distributed and Dependable Systems

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

Prague 2024

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) 2024 Jan Kleprlík
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, sub-license, 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 NON-INFRINGEMENT. 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 express my deepest gratitude to my supervisor, doc. RNDr. Pavel Parízek, Ph.D., for his accurate insights, guidance and encouragement throughout the development of this thesis. I would also like to thank all my colleagues in the Manta company, especially Mgr. Dalibor Zeman, who helped me greatly in the beginning. I would also like to appreciate the strong endurance of my partner, who helped and cared for me in the key moments. Finally, I'm incredibly grateful for my family, who supported me relentlessly throughout all of my studies.

Title: Performance and Usability Improvements for Data Lineage Analysis of C# Programs

Author: Jan Kleprlík

Department: Department of Distributed and Dependable Systems

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

Abstract: The Manta Flow is a highly automated static analysis platform producing data lineage over its input and representing it in a graph. The platform performs analysis on various technologies and programming languages via specialised scanners. One of the scanners performs analysis of C# code, or rather its compiled alternative Common Intermediate Language. While the scanner was already capable of analysing non-trivial scenarios, it lacked in some aspects that held it up from its broader adoption by customers. The main issues are low support for analysis of real-life scenarios such as web applications or embedded code in other technologies, sub-optimal performance and imprecise lineage output. As a part of this thesis, we improved the precision, scalability and performance of the scanner on multiple levels of abstractions, from analysis of the CIL to modifications of core high-level analysis algorithms. We added support for analysis of the ASP.NET web endpoints and enabled the C# scanner to be used as a service for analysis of embedded code in other technologies. We improved the precision of the resulting lineage for existing scenarios by modifying the core algorithms used throughout the analysis and optimized the analysis process to lift its performance.

Keywords: static analysis, data lineage, C#, embedded code

Contents

1 Introduction **3**
1.1 Data Lineage . . . 3
1.2 Goals . . . 4
1.3 Glossary . . . 5
1.4 Outline . . . 6

2 Manta **8**
2.1 Manta Architecture . . . 9
2.2 Scanner Architecture . . . 11
2.2.1 Connector . . . 11
2.2.2 Dataflow generator . . . 12
2.3 Previous Work . . . 14

3 C# Scanner **15**
3.1 C# Language . . . 15
3.2 Architecture . . . 17
3.2.1 Intermediate Code Infrastructure . . . 17
3.2.2 Intermediate Code Simulator . . . 20
3.2.3 Alias Analysis . . . 20
3.2.4 Symbolic Analysis . . . 21
3.2.5 Analysis Result Transformer . . . 26
3.2.6 Analysis Runner . . . 27
3.3 Analysis Life Cycle . . . 27
3.4 Current Limitations . . . 30

4 Requirements and Analysis **32**
4.1 Stakeholder Requirements . . . 32
4.1.1 ASP.NET Endpoints . . . 32
4.1.2 Embedded Code Service . . . 35
4.2 Addressing Technical Debt . . . 41
4.2.1 Unknown Column Support . . . 41
4.2.2 Method Summary . . . 43
4.2.3 Wrong Method Handler Selection . . . 44
4.2.4 Multiple Handlers on Single Call . . . 46
4.2.5 Protocol Buffers . . . 48

5 Design and Implementation **50**
5.1 ASP.NET Endpoints . . . 50
5.1.1 Access Modifiers . . . 51
5.1.2 Type Instantiability . . . 52
5.2 Embedded Code Service . . . 52
5.2.1 Pin Nodes . . . 53
5.2.2 Insight and Outsight . . . 54
5.3 Unknown Database Column Support . . . 54
5.3.1 Database Query Mapping . . . 54

5.4 Method Summary 54
5.4.1 Flow Locations 55
5.5 Method Handler Selection 55
5.5.1 Support Database Query Enumeration 56
5.6 Remove Protocol Buffers 57
5.7 Testing 57
5.8 Other 58

6 User Documentation **60**
6.1 Analysis Execution 60
6.2 Examining the Analysis Result 63

7 Evaluation **66**
7.1 New Supported Use Cases 66
7.2 Regression Testing 67
7.3 Future Work 68

8 Conclusion **74**

Attachments **82**

1. Introduction

In the current environment, data and the movement of some data are present in every organization and corporation. Be it tracking customer purchases, the organization's financial history or personal employee data. With that comes a lot of responsibility and complexity in managing how we transform and move individual pieces from place to place. Understanding these movements helps us not only manage our resources but also often comply with necessary legislation and regulations or understand the complexities of our own systems.

A great example where understanding implemented processes with data, be it simple moving or transformation, proves handy is compliance with various legislation. In the thesis upon which this work builds upon [1], the author brings a very good example. In the European Union, every subject that processes personal data wholly or partly by automated means is required to comply with the *General Data Protection Regulation* (*GDPR*). One of the implications of this regulation is known as the *Right to Be Forgotten* or the *Right for Erasure*. Whenever a person wants all his information deleted from the system, we need to know precisely where the original information was propagated - ignoring other legal details. With complex systems where no single person understands every bit of the system, this is hard to prove or validate. Visualization of how personal data is moved around provides one way to ensure this data can be tracked and properly managed.

1.1 Data Lineage

The process of following data from its source through any form of transformation to its destination is referred to as *data lineage*. Through this process, we are able to track data in complex systems and visualize them in graphs with adjusted filters to suit our use case. In the introduction, we discussed one use case where data lineage is useful. Let's discuss a few more use cases.

The process of migrating systems to cloud infrastructure exceeds its budget in about 75 % of cases and time schedule by 38 %, of which 13 % is by more than three quarters as surveyed by *McKinsey* [2]. Mapping data flow and its further visualization in graphs may help us find errors in our own analysis and see the domain from another perspective. For example, we can double-check whether everything can be migrated to the cloud. Cloud infrastructure may often reside in a different country which means that some additional regulations may apply. Even if we have a professional legal consultant, they may not at first be aware of the data that should not be migrated at all. This is partly one of the reasons the company *Meta* [3] received the biggest fine historically for GDPR violation [4], which would be terminating for many.

Data lineage can also prove very useful for reducing error rates and faster error resolution. If finding duplicity in code is difficult, finding duplicity in data management across complex systems is ambitious. Data lineage may help us with reducing unnecessary data growth and keep things a little simpler. Understanding where new data restrictions will have an influence even before we implement them provides a helpful insight into the restrictions themselves. It also works the other way around. If the end data quality does not meet the standards required with

data lineage, it is much less demanding to find sources that can be enhanced and improve the end data quality as a result.

We can track data lineage in several ways. The most obvious and challenging one is to understand the system operating with data. This approach is highly unscalable, unreproducible, error-prone and creates an eminent bus factor [5]. Usually, it should not be even considered. The natural solution to this is to automate data lineage. Company *impreva* [6] states four options to create lineage:

1. discover patterns,
2. data tagging,
3. use self-contained lineage,
4. static analysis.

The first option of discovering patterns involves analysis of metadata and discovering patterns in between them. For example, columns with the same or very similar names may point to the same data at a different time of its life cycle. It is easy to see that this approach will not provide reliable lineage. The upside is that it can be applied to different technologies without modifications ie. SQL and Oracle.

The second option is commonly used in other industries in various forms. The essence is to mark data with recognizable marks and track its flow through the system. The imminent downside is that this usually requires the system to support data tagging. A similar approach is supposedly used to trace original render leakers in *Apple* as stated by *9to5mac* [7].

Self-contained lineage is an excellent and reliable way to gather lineage. If the technology we use provides a way to view data at all of its points of the life cycle, we immediately get the lineage. However, the technology must be built with this in mind from the ground up. As with the data tagging approach, this method fails to provide any further lineage outside of the technology. Therefore, it is often combined with other tools and approaches. We can see this method work with *Databricks Unity Catalog* [8].

The last approach is to perform static analysis of any arbitrary technology that we need the lineage of. This means reading the metadata and often the source code itself, interpreting it and creating an abstraction with required lineage. At first sight, this is apparently a tedious task to automate since the barrier to getting any lineage at all is very high. On the other hand, this approach is highly customizable and granular. Such analysis is often performed by specialized tools. One such tool is *Manta Flow* [9]. This thesis was developed in cooperation with the company *Manta, an IBM Company* and builds upon its existing infrastructure.

1.2 Goals

In this thesis, we aim to enhance the performance and usability of the C# scanner, which is a part of a larger static analysis tool called *Manta Flow* developed by *Manta, an IBM Company* [9]. The C# scanner is a unit specialised in data flow

analysis of C# programs by inspecting the compiled CIL code. We plan to extend the usability and enhance the performance of this scanner based on the needs and requests of both Manta's customers and internal requests. We will refer to these customers as stakeholders. Many of the goals we set for this thesis that will extend the tool's usability come directly from use cases provided by stakeholders or were discovered as necessary when analyzing stakeholder input. That is because Manta has a highly customer-first oriented development culture. Some crucial decisions may, therefore, appear arbitrary. The goals themselves can be divided into two separate categories — usability improvements and performance enhancements.

Usability Improvements

Usability improvements will enhance the features of the scanner with the aim of making it usable for real-life scenarios provided by stakeholders. Even though the C# scanner is already able to analyse non-trivial programs, it lacks some features in the context of data management and processing systems. We will extend the entry point extraction to web endpoints in order to support ASP.NET applications. This requires supporting more C# language features that are ignored at this point. Another goal is to enable the use of C# scanner as a reusable service that can be called from other technologies on demand. It should analyse embedded code and create a subgraph that can be merged into the graph of the source technology. Numerous stakeholders do not use the C# scanner directly but would like to analyse embedded code inside other Extract-Transform-Load technologies such as SQL Server Integration Services or Azure Data Factory. Note that the goal is not to implement a fully functional service with integration to the source technologies. It is rather to enable the scanner to be used in such a way. Complete integration is outside of the scope of this thesis. Other goals are set by analysis of different inputs by stakeholders that are analysed inaccurately or plain wrong. The goal then is to analyse these inputs and make the analysis more precise.

Performance Enhancements

The performance enhancement aims to enable the use of the C# scanner in the context of both small code snippets embedded in other technologies as well as large programs. Some analyses take tens of hours with obvious improvements that have been validated before in other scanners inside the Manta tool. One such improvement is a modification of flows that are required for the resulting graph but do not bring any value to other parts of the code, yet they are still propagated. For instance, any console output must be found and marked in the graph. Its flow does not have to be propagated further throughout the analysis since no new data can be created from the output.

1.3 Glossary

Action Method

An action method on a controller of a web application is a method that gets called via HTTP request [10].

ADF

Azure Data Factory or ADF is a serverless data integration service [11].

ASP.NET

ASP.NET is a cross-platform open-source framework for the development of web applications and services in the .NET ecosystem [12].

CIL

Common Intermediate Language is an intermediate language defined by the Common Language Infrastructure specification [13]. It is used by the .NET runtime environment for running C# code.

Dependency Injection

Dependency injection is a programming technique to provide resources to objects or functions instead of having them create them [14].

ETL

ETL stands for *Extract, Transform, Load.* It is a data manipulation process in the order of first extracting the data, arbitrary transformations and loading data to a storehouse [15].

Protocol Buffers

Protocol Buffers are language-neutral data format used for serialization of structured data [16]. Its serialized form is not human-readable.

SSIS

SQL Server Integration Services or SSIS is a platform for building data integration and transformation solutions [17].

XML Schema Definition

XML Schema Definition or XSD is a W3C recommendation on the formal description of an XML document. It is most often used for validation and code generation purposes [18].

1.4 Outline

This thesis is split into several chapters. First, we will describe the high-level architecture of Manta and a generic language scanner. After that, we will introduce the C# scanner itself and explain any specifics of this scanner. After a detailed description of the architecture and all individual parts of the scanner, we will connect them in a description of an analysis life cycle to give a good

overview of the analysis. We will pinpoint the limitations of the scanner's current state. Understanding the scanner's intricacies, architecture and limitations, we will analyse individual problems and requirements individually in great detail and propose solutions. Then, we will describe the implemented solution and how to use the Manta Flow tool and the C# scanner in more detail. We will conclude this thesis with an evaluation of our work and a proper conclusion.

2. Manta

This thesis was developed as a project inside the Manta company. Manta originated as an internal project inside *Profinit* [19]. Later, it separated and continued to grow, eventually being bought by an *IBM* [20] in late 2023 [21] and was renamed to *Manta, an IBM company*. We will refer to it only as Manta.

Manta's software provides data lineage on more than 50 [22] technologies ranging from databases, ETL and modelling tools, reporting and analytics software to programming languages. These include *Sqoop*, *Qlik Sense*, *PostgreSQL*, *Power BI*, *Oracle*, *Teradata Database*, *Cobol* and many more. Internally, each technology is handled by its own so-called *scanner* unit. Data lineage can then be integrated into customer pipelines using extensive OpenManta API or visualized in Manta's own graph visualizer as comprehensible graphs. Those graphs can be modified to show different levels of granularity. One example of such a graph can be seen in Figure 2.1. This graph represents a code that reads some value from the standard input and then inserts it into the MSSQL ReportServer database in the `USERCONTACTINFO` table, `USERID` column.

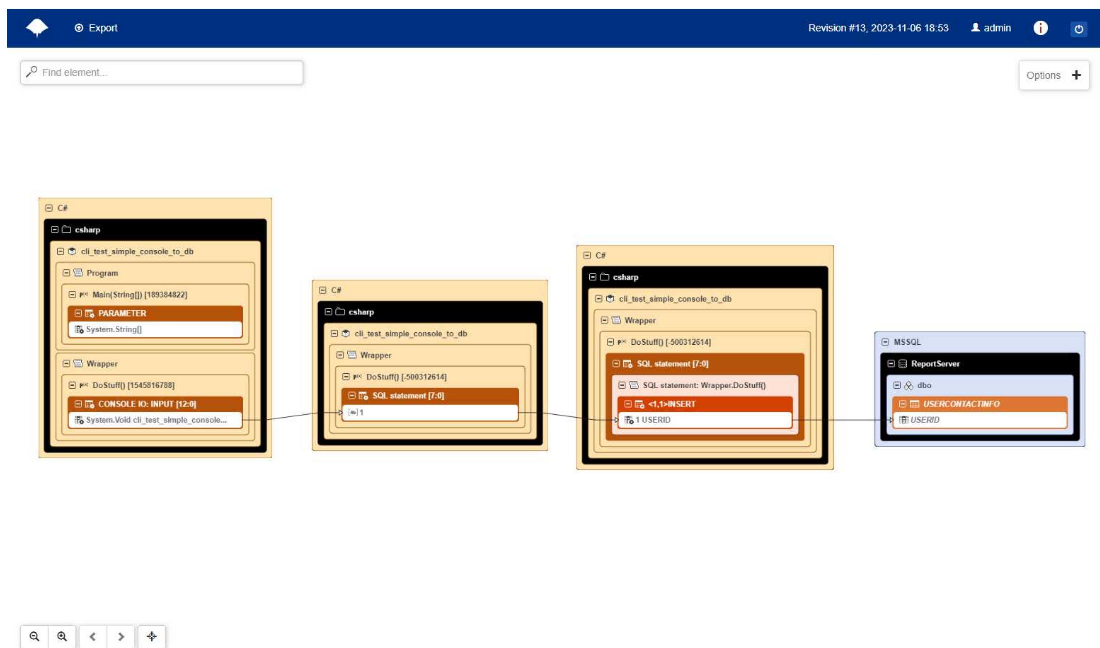


Figure 2.1: Screenshot of a Manta data flow graph viewer with a graph representing data lineage of a sample C# console application, which reads data from the standard input and inserts them into a database.

This chapter will explain the high-level architecture of Manta's technology and its scanners. We will also introduce previous work on the C# part, which is the main focus of this thesis.

2.1 Manta Architecture

The Manta platform consists of three major parts:

- Manta Admin GUI,
- Manta Flow Server,
- Manta Flow CLI.

Manta Admin GUI is a Java server application that provides a graphical and programming interface for managing the Manta platform, such as installation, updating, authorization, configuration and other setup actions. Every responsibility is contained in its own tool, bound together by Admin GUI. In this part, the user sets up individual scanners and launches analysis. The web interface can be seen in Figure 2.2.

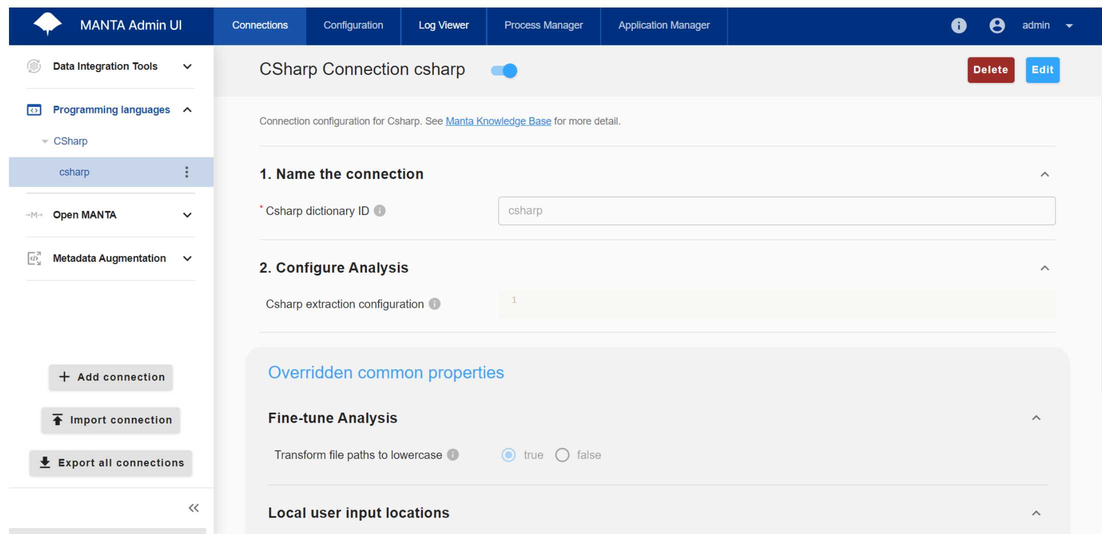


Figure 2.2: Screenshot of the Manta Admin GUI

Manta Flow Server is the second part where users spend most of their time. This part has three important responsibilities. One, it contains the web interface for data flow graph visualization, which can be seen in Figure 2.3. It contains a selector of analysis, a picker for parts of the graph that are supposed to be visualized and lastly, a section with the level of detail to display. Two, it stores all metadata from analysis in the internal data lineage repository. Finally, it exposes programming API for third-party integration with user applications and data pipelines.

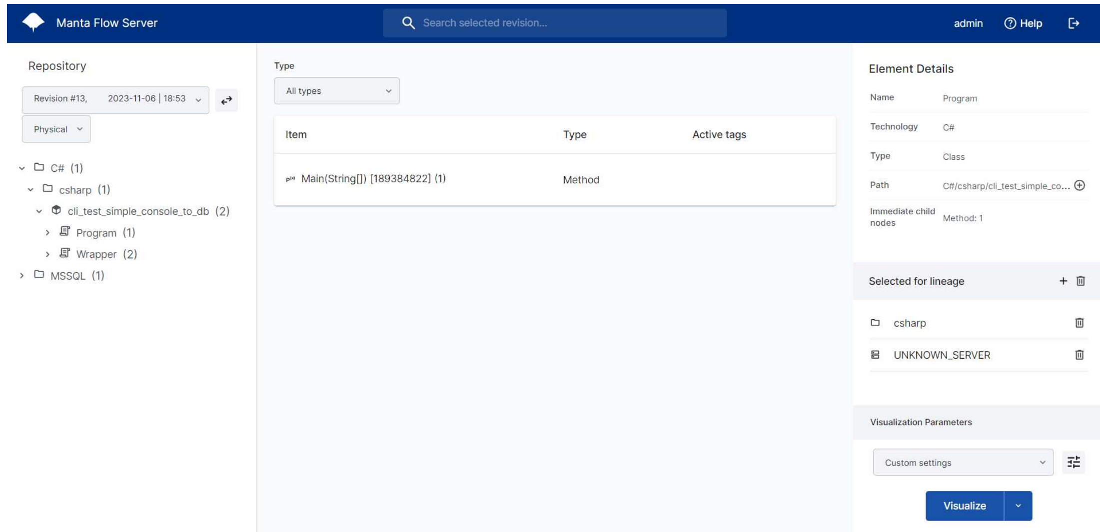


Figure 2.3: Screenshot of a Manta Admin GUI

The last part of the platform is Manta Flow CLI. This part is made of technology-specific scanners. It is a command-line program that performs static analysis over its input and produces data flow. This part interests us the most as the C# scanner resides here. Individual processes are grouped into scenarios which represent a unit of analysis. For example, the whole C# source code analysis is made of an Extractor scenario and a Dataflow scenario. We will discuss these in more detail in the following chapter about C# scanner architecture in Section 3.2. This means that to perform an analysis of a specific technology, we will set up the workflow in the Admin GUI to run relevant scenarios of the technology. Depending on what we want to do with the result, we might add scenarios to visualize data flow directly in the Admin Flow Server.

Figure 2.4 shows how the three components interact and communicate. The Admin GUI provides configuration to the Manta Flow CLI, including paths to source code. Manta Flow CLI executes static analysis via technology-specific scanners and creates an output which is transferred to the Manta Flow Server. There, we can send an update about the finished scenario back to the Admin GUI or visualize the data flow graph. Lastly, users can integrate the graph via exposed API into their applications.

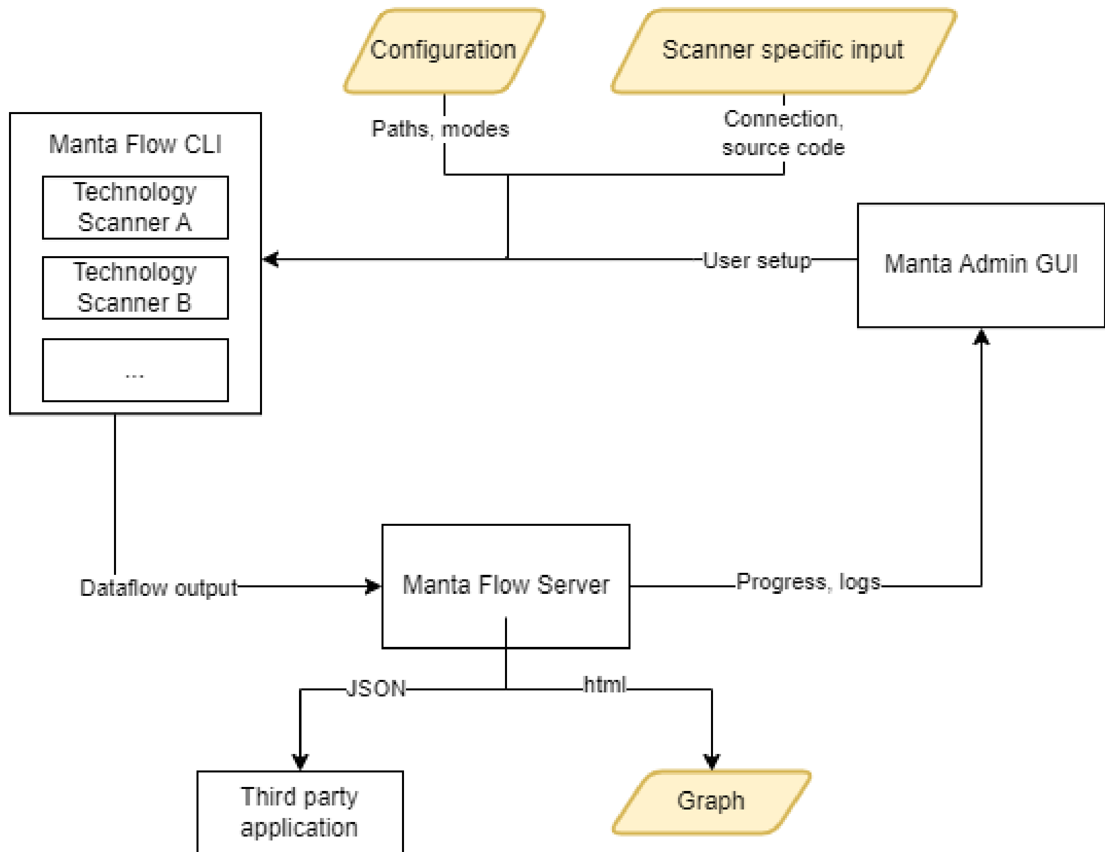


Figure 2.4: Simplified general architecture of Manta consisting of three main modules: Manta Flow Server, Manta Flow CLI and Manta Admin GUI

As we can see, the Manta platform is highly automated in the sense that the only human interaction is needed in the configuration. Every other part of the analysis, except configuration, is done automatically for us.

2.2 Scanner Architecture

Each technology is handled by a so-called Scanner. Such scanners are, for example, SSIS Scanner, Databircks scanner or C# scanner. Scanners differ slightly in the details of their architectures because of the technology specifics. The overall high-level architecture tends to follow the same pattern. With that being said, we will focus on the programming language scanners as that is the theme of this thesis. In this section, we will go through the high-level architecture that tends to be common for all programming language scanners. The scanner is separated into two major components: the *Connector* and the *Dataflow generator*.

2.2.1 Connector

The connector's main task is first to extract important metadata that will be used in the analysis. Its second task is the static analysis itself. These two distinct responsibilities are split up into two separate modules — the Extractor and the

Reader. The connector's goal is to generate a representation of the metadata from which the dataflow generator can create a graph.

Extractor

Extractor is the first phase of static analysis. Its job is to gather required metadata and satisfy pre-conditions of static analysis. That may include connecting to some connection provided in the configuration and extracting any metadata necessary. For example, database schemas, definitions, ETL pipeline metadata, or source code. All these additional resources are stored in a local repository available for the Reader. Note that this implies that any input, including language libraries, that might have to be analyzed must be gathered and copied to the local repository. Even though programs must have exactly one runtime entry point, this is not the case with data lineage. Data may enter the application from various places, such as REST endpoints. One of the responsibilities of the Extractor is, therefore, an entry-point analysis and extraction. Database scanners additionally create a *dictionary*, which is a data structure that contains information about the database's schema and is ready to help answer queries about the structure of the tables and their columns. This is useful for resolving database queries using the *Database Query Service* later in the analysis.

Reader

The second part of the connector is the Reader. Its purpose is to read all provided metadata and create a general model of the input. This model is common for all language scanners with minor deviations. The model is later used to create the graph itself. For programming language scanners, this is also where the actual analysis of the source code happens, and the *worklist* algorithm is used. This algorithm will be explained in greater detail in the next chapter about the C# scanner in Section 3.2.4. The Reader tends to contain the most complex logic, and most of the analysis time is spent there. Therefore, it offers the most room for improvement.

2.2.2 Dataflow generator

The dataflow generator is the second part of any scanner. Its main purpose is to generate a final graph from the connector's output. To accomplish this, it must perform several tasks varying from technology to technology. Filtering unnecessary orphan nodes, validating and verifying connector's output, dealing with not-yet analyzed parts and merging graphs. Unresolved parts include SQL queries or embedded code snippets. Since many technologies use one SQL dialect, it doesn't make sense for each technology to implement its own SQL query resolver. The *Dataflow Query Service* takes care of it. The service analyzes the query and outputs a final graph, which must be merged into the source technology's graph. Merging is performed back again in the dataflow generator. An excellent example is a programming language with a database framework where we write queries. These queries are marked by the language scanner but are resolved later in the dataflow generator. The same may apply to programming

languages, as we may encounter some embedded code snippets throughout the analysis, which are resolved using the *Embedded Code Service*.

In Figure 2.5, we can see an overview of an arbitrary programming language scanner. It is separated into two scenarios which are marked by colored boxes with dashed lines. Even though the extractor is logically part of a connector (marked with a red box) and is specific for each technology, just like the reader is, it usually has a separate scenario — *Extractor Scenario* marked with the dashed blue box. This allows us to launch it independently from the reader and dataflow generator, which share one scenario together — *Dataflow Scenario* marked with the dashed green box. That only makes sense because their common goal is to analyze its output and generate an output graph useful to the end-user. To only run an extractor and gather entry points is a valid scenario. To run only a reader and obtain an abstract model whose representation is only understandable to the dataflow generator is groundless. In practice, the case usually is that both scenarios are run together. After all, the extractor prepares an input for the reader. At the top of the figure is an extractor that takes an input program and input configuration. As a part of the Extractor Scenario, it creates *Exctracted Configuration*, which is then passed to the reader inside of the Dataflow Scenario. The reader creates a model representation of the lineage called *Connector Output*. Finally, the dataflow generator transforms the output, optionally using the embedded code service and dataflow query service, into the final graph representation.

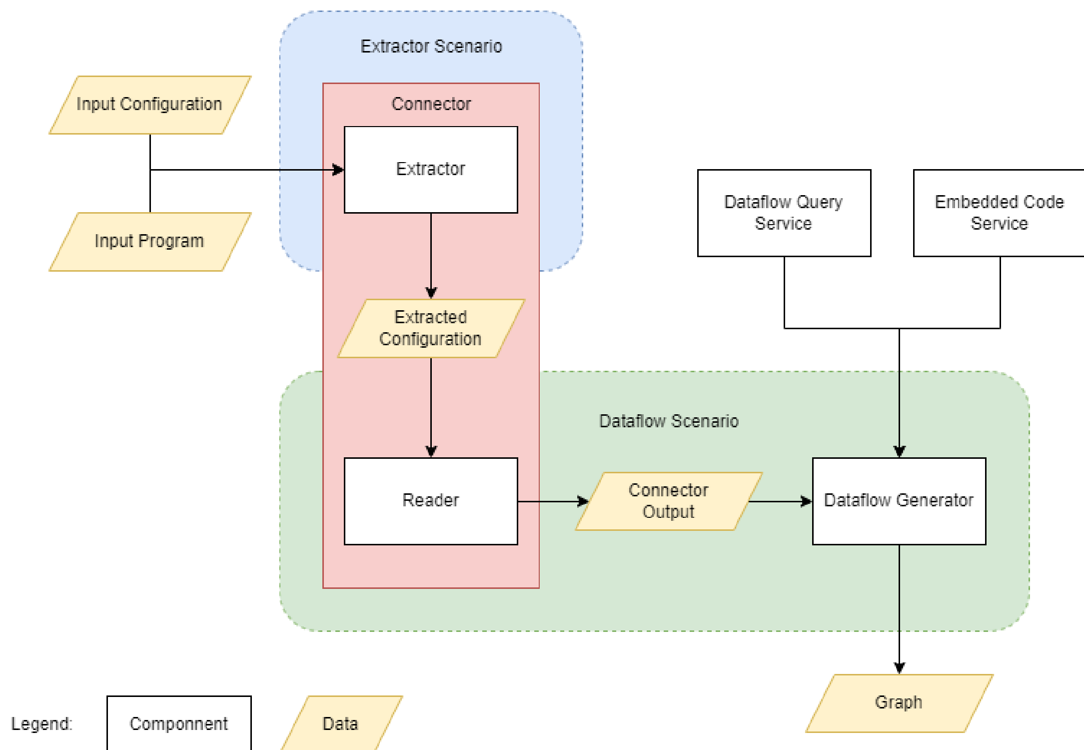


Figure 2.5: Architecture of an arbitrary programming language scanner.

2.3 Previous Work

This thesis builds upon multiple previous projects developed by this thesis supervisor himself - doc. RNDr. Pavel Parízek, Ph.D., and students of Charles University, faculty of Mathematics and Physics, in cooperation with the Manta company, where it is already used in production. Implementation decisions in these projects have implications for our own implementation. Therefore, we will introduce them briefly. The C# scanner started as a software project in 2020 called *Data Lineage Analysis of C# Programs for Manta* [23]. In this project, the foundations of the C# scanner were set, and all the vital parts were implemented. This includes the extractor, the reader and the dataflow generator. The team took heavy inspiration from an existing bytecode scanner, referred to as Java scanner or bytecode scanner interchangeably. Java scanner was first developed by doc. RNDr. Pavel Parízek, Ph.D., later extended by students as a thesis in cooperation with Manta. Many key decisions remained the same, even with the same reasoning. For example, the Java scanner analyses compiled bytecode instead of the text source code. The same applies to the C# scanner, although the bytecode in our domain is called *CIL* or *Common Intermediate Language*. This has multiple advantages which can be explored in the software project documentation. Relevant parts for our thesis will be explained in Chapter 3, dedicated to the C# Scanner.

After successful completion, it continued to be worked on by Manta employees as the scanner was still in an experimental state with highly limited real-life scenario usability. Another project was a thesis by Dalibor Zeman called *Extending Data Lineage Analysis Towards .NET Frameworks* [1]. Zeman's work added extensive feature upgrades demanded by various Manta's customers.

3. C# Scanner

Now that we have established a high-level description of Manta in general and its scanners, we can get into the description of the C# scanner and C# itself. In this chapter, we will first explain the C# language domain, and then we will take a look into the implementation details of the C# scanner that are relevant to our thesis. We will explain in detail the purpose of individual modules active throughout the analysis. In the end, with the knowledge we have gathered, we will shed a little bit of light on the current limitations of the scanner.

3.1 C# Language

First, we will explain the C# language and how it works. It is a high-level programming language whose toolchain and compilers are open-sourced. C# source code is first compiled into an intermediate language called Common Intermediate Language (*CIL*) using the Roslyn compiler. This intermediate language is platform-independent and is usually stored in executable files or dynamically linked libraries. On this platform, the .NET runtime, or more precisely *Common Language Runtime* (*CLR*), then compiles this intermediate language using *RyuJIT* just-in-time compiler to native code for the front-end of the CPU. An important note is that CLR is a stack-based virtual machine. RyuJIT is dynamic and performs several compilation optimizations.

For the purpose of data lineage analysis, we have two options for the C# programming language. We can either inspect high-level C# code, or we can inspect its compiled counterpart CIL. Note that this decision was not made by us but by the authors of the original C# scanner in Manta. Authors have decided on CIL for multiple reasons:

- The CIL rarely introduces changes compared to new syntactic sugar added to the C# language. With every change of the language, we risk the need to modify the scanner in order to support the analysis of code written with new language versions.
- It allows us to use the scanner for other .NET languages that emit CIL, such as F# or Visual Basic .NET.
- It is less memory-demanding than working with abstract syntax tree representation.
- There is already a similar Java bytecode scanner which follows the same principles. That allows us to learn from many design decisions made there and apply the same crucial algorithms.

The code snippet in Listing 3.1 shows an example of a simple C# console application. It is a program that takes input from the user, parses it as an integer, adds another number to it and prints the result into the console. If we take a look at Listing 3.2, we can see selected parts of its emitted CIL as presented by the Rider IDE. We can see that most of the important information for data lineage is preserved. For method calls, we have a `call` opcode with the method signature.

We only don't know the object upon which this method is called. This has to be resolved by runtime. For local variable loading, we have a `ldloc.x` where `x` is a number of the local variable. Metadata about these variables is defined in metadata tables bundled with CIL. However, the instance is again not always known. This is very similar to Java bytecode.

```
1 int a = Int32.Parse(Console.ReadLine() ?? string.Empty);
2 int b = 2;
3 int c = a + b;
4
5 Console.WriteLine($"The result is: {c}");
```

Listing 3.1: Simple C# program

```
1 IL_0000: call         string
    [System.Console]System.Console::ReadLine()
2 IL_0005: dup
3 IL_0006: brtrue.s     IL_000e
4 IL_0008: pop
5 ...
6 IL_0016: ldloc.0      // a
7 IL_0017: ldloc.1      // b
8 IL_0018: add
9 ...
10 IL_0041: call        void
    [System.Console]System.Console::WriteLine(string)
11 IL_0046: nop
12 IL_0047: ret
```

Listing 3.2: CIL of a simple C# program

One exception to the similarity between CIL and Java bytecode is, for example, the differentiation of arithmetical operations on simple number types. In Listing 3.3, we can see a side-to-side comparison of emitted CIL and Java bytecode, respectively, together with high-level code. Apart from indexing from 1 for Java bytecode, we can see that Java bytecode opcodes have an `i` prefix, signalising an instruction on integer operands. CIL doesn't have this prefix. Thus, operand types must be deducted by runtime. With that being said, this does not bother us that much as for data lineage. The concrete type of a simple type is often irrelevant. Further analysis has been done by the authors of the software project, and the base has already been implemented and is working. This short overview and knowledge of C# and CIL should be sufficient for the purpose of this thesis.

```
1 /*
2 IL_0000: ldc.i4.s 21        |       BIPUSH 21
3 IL_0002: stloc.0            |       ISTORE 1
4 IL_0003: ldc.i4.s 21        |       BIPUSH 21
5 IL_0005: stloc.1            |       ISTORE 2
6 IL_0006: ldloc.0            |       ILOAD 1
7 IL_0007: ldloc.1            |       ILOAD 2
8 IL_0008: add                |       IADD
9 IL_0009: stloc.2            |       ISTORE 3
10 */
11 int a = 21;
12 int b = 21;
```

```
13 int c = a + b;
```

Listing 3.3: Integer addition in C#/CIL and Java bytecode.

3.2 Architecture

Like other programming language scanners, C# scanner analysis consists of two scenarios. The *Csharp Extractor Master Scenario* and *Csharp Dataflow Master Scenario*. The core of the C# scanner is written in C# itself. That complicates things a little bit, as the rest of the Manta is written in Java. The interaction between those two parts is solved by launching the C# part as an extra process from the Java application. The implication of this is that any piece of information we want to pass from one part to another must be serialized in Java and read in C#. We can divide parts of the scanner based on what language it is written in. There are three Java wrappers for C# parts:

- `manta-connector-csharp`,
- `manta-connector-csharp-extractor`,
- `manta-connector-csharp-runner`.

Their main purpose is to implement a common interface to be able to integrate the C# scanner into the rest of the platform. The `manta-connector-csharp-extractor` contains data class `ExtractorConfigurationData` with extractor configuration. Everything must be serializable because the extractor implementation itself is in the C# part. Instead of a functional class that would serve as an abstraction which could be used throughout the analysis, the configuration is a simple data class with paths to the input application and paths telling us where to serialize extracted entry points and other analysis configurations. The rest of this module is an abstraction to run the scenario through a spring bean configuration. The `ExtractorScenario` class invokes the `CSharpApplicationRunner` class from the `manta-connector-csharp-runner` module. This helper class prepares common arguments and logging for the C# process and executes it. The same approach is mirrored for the `manta-connector-csharp` module, which is technically the reader part of the scanner as presented in Figure 2.5. This module also contains the serializable `ReaderConfiguration` class. The `ConnectorOutputReader` class is a part of the dataflow scenario, and its purpose is to again invoke the `CSharpApplicationRunner` class and execute the C# process with analysis.

The complete C# implementation of the scanner is grouped into one assembly made out of 23 projects. Each project plays a different role throughout the analysis. We will go through the key projects and explain their roles.

3.2.1 Intermediate Code Infrastructure

In order to be able to statically analyse CIL we have to be able to read it first. The `IntermediateCodeInfrastucture` class is an abstraction above the CIL. Internally, it uses the `Mono.Cecil` library [24], which itself parses metadata tables

and provides access to any type, method or any other definition that was present in the code before compilation. This representation is language-independent and follows the CIL nomenclature. CIL nomenclature is sometimes a little bit different from what programmers are used to at C#. For example, the access modifiers are differentiated differently in CIL than they are in C#. The Microsoft documentation defines the keyword `protected` in C# as: *"The type or member can be accessed only by code in the same class or a class that is derived from it."* [25]. Meanwhile, a protected member compiled into CIL carries the modifier `family` defined in the *ECMA-335* standard [13]. We can observe this behaviour in a member definition and its CIL counterpart in Listing 3.4. The CIL counterpart is written into a readable form by the *Sharplab.io* tool [26].

```
/*
.class public auto ansi beforefieldinit Program
    extends [System.Runtime]System.Object
{
    // Fields
    .field family int32 a
    ...
}
*/
public class Program{
    protected int a;
}
```

Listing 3.4: The C# `protected` access modifier in code and its CIL representation as `family`.

Another example is that top-level classes defined in the C# code as `internal` carry the modifier `private` in CIL. Note that `private` is a valid access modifier in the C# code. Using the CIL nomenclature would allow us to use it without any modification for other .NET languages as well. However, at the start of the development of this scanner, it was decided to internally create an abstraction at the higher level of C# and use the C# nomenclature. Some obtainable information from `Mono.Cecil` is therefore not available in our infrastructure as it was not necessary for C#.

Two important abstractions are the `ClassHiearchy` class and the `CallGraph` class. The class hierarchy allows us to create quick queries for what classes a particular class inherits from or which classes implement a particular interface. These queries are very important when building the call graph. In the CIL, we don't always have enough information to determine which method is being called. That is determined by the runtime. This problem is called a reachability problem. In the C# scanner, it is resolved using the *Rapid Type Analysis* [27]. Suppose class declarations as defined in Listing 3.5. An interface `Animal` with a contract for a method `Speak` and two classes `Dog` and `Cat` implementing this interface.

```
public interface Animal {
    public abstract void Speak();
}

public class Dog : Animal {
    public void Speak() {
        System.Console.WriteLine("Bark");
    }
```

```
9 }
10
11 public class Cat : Animal {
12     public void Speak() {
13         System.Console.WriteLine("Meow");
14     }
15 }
```

Listing 3.5: C# virtual method definitions example.

Now, let's suppose a simple `Main` method where we call the function `GetAnimal` and assign its return value to the variable `animal`. Then, we call the method `Speak` on this variable. This code snippet and its generated CIL can be seen at Listing 3.6. The `GetAnimal`'s method signature is `Animal GetAnimal ()`. This means that the call of method `Speak` is a virtual call whose implementation is determined by the runtime. Note that it doesn't matter if its implementation always returns an instance of the same type. Its CIL must always contain a virtual call with the return type `Animal`.

```
1 /*
2 IL_0000: nop
3 IL_0001: call class Animal Program::GetAnimal()
4 IL_0006: stloc.0
5 IL_0007: ldloc.0
6 IL_0008: callvirt instance void Animal::Speak()
7 IL_000d: nop
8 IL_000e: ret
9 */
10 public class Program{
11     public static void Main(){
12         var animal = GetAnimal();
13
14         animal.Speak();
15     }
16
17     public static Animal GetAnimal(){
18
19             return new Cat();
20     }
21 }
```

Listing 3.6: C# virtual method call example.

To create a call graph for the `Main` method we have to go through all the call instructions in the method's body. On each call instruction, we register the method that is being called. For virtual calls, it is a bit more difficult. We need to take into account the implementation of the method instead of using the interface method. This is where we query the class hierarchy for all implementations of the method. Simple comparing signatures to determine reachability isn't sufficient as the target class names differ. We could consider all the possible implementations for the method. In our case, that would be `Cat.Speak()` and `Dog.Speak()`. This would result in a big portion of false positives as most of the implementations may never even be called in the first place. The C# scanner uses a more advanced *Rapid Type Analysis* algorithm. The improvement is that it tracks instantiated classes and uses them to further reduce the pool of potential implementations.

The only instantiated class in the code Listing 3.6 is `Cat`. Therefore, the only call registered would be of the method `Cat.Speak()`. Note that for the purpose of the analysis, we also have to register the interface call `Animal.Speak()` itself. This is further explained in Section 3.2.4.

3.2.2 Intermediate Code Simulator

Another major project of the scanner is the `IntermediateCodeSimulator` project. Its purpose is fairly simple: to simulate code execution and create abstractions to help us with the analysis. The simulation traverses instructions of every reachable method from the entry-point method and interprets each instruction using a visitor pattern. Because the .NET virtual machine is stack-based, the simulation is also stack-based. When necessary, it creates an expression that makes further analysis easier. There are many different expressions, such as `MethodExpression`, `ArithmeticExpression`, `ConstantExpression` and many others. For instance, the `ConstantExpression` class with its value of 0 is created on instructions that load a fixed value onto a stack (`ldc.i4.0`, `ldc.i4.1`, `ldc.i4.2`, `ldc.i4.3` etc.). The `MethodExpression` class is created on the `ldftn` instruction. These expressions are used, for example, for computing alias analysis.

3.2.3 Alias Analysis

Another problem that has to be solved concerning dataflow analysis is the aliasing of variables. Suppose the code snippet from Listing 3.7. First, an instance of class `A` is created and assigned to the variable `a`. Lambda that writes to the console is assigned to its field `functionCall`. After that, an alias `x` of the variable `a` is created. Then, a new lambda that writes to a file is assigned to the field `functionCall`. Eventually, the `functionCall` is invoked and called through the variable `x`. If alias analysis was omitted, the only detectable dataflow would be the write to the console. The write to the file is assigned to the original variable and only after the creation of the alias. That means that if we do not propagate the assignment of the new lambda into all of the `a`'s aliases, any following call of `x.functionCall()` would only consider the console output. For data lineage, this is unacceptable behaviour.

```
1  public class A
2  {
3      public Action functionCall;
4  }
5
6  public class Program{
7      public static void Main(){
8          A a = new A();
9          a.functionCall = () => Console.WriteLine("FIRST");
10
11         var x = a;
12
13         a.functionCall = () => File.WriteAllText("output.log",
               "SECOND");
14
15         x.functionCall();
16     }
```

```
17 }
```

Listing 3.7: C# snippet with the variable `a` aliased with variable `x`.

The complete alias analysis is implemented inside the `AliasAnalysis` project. The project offers both lazy and eager analysis. Just like in the intermediate code simulator, we use the visitor pattern to go over all instructions, which may create an alias and record it in `AliasCollection`. These instructions are, for example, `st.loc` for assigning into a local variable or `stelem` for assigning into an array.

3.2.4 Symbolic Analysis

The main core of the analysis, which creates dataflow, is in the `SymbolicAnalysis` project. Precisely, the `StaticAnalyzer` class performs the static analysis and outputs an object of the `ProgramFlowData` class with complete information about the data flow. On the inside, the analyzer first analyzes aliases using the `AliasAnalyzer` class and then performs the static analysis of all reachable methods from the provided entry point method. In brief, the symbolic analysis works with the method descriptors and expressions in the body of this method from the `IntermediateCodeInfrastructure` project, analyses them as necessary with the worklist algorithm and creates data flows associated with these expressions. Let's first discuss the worklist algorithm and how individual methods are analysed, and then we shall explain how exactly any data flow is created.

Worklist Algorithm

One approach to performing static analysis to obtain dataflow information would be to simulate execution, recursing one level deeper on each method call and continuing to analyse there. There are two particular problems with this approach. One, it may lead to deep recursions. Two, this approach explodes very quickly when methods call each other. Another approach is called the *worklist* algorithm and is generally used in various static analysis problems such as liveness or dataflow analysis.

The worklist algorithm is based on the idea of performing only as much necessary work as needed. It starts with a set of items to be analysed called a worklist. Items from the worklist are analysed one after the other completely until there is none left. This means that it is not a set but rather a queue. While analysing, it may happen that analysis of one item may influence other items, even those already analysed. In that case, they are added back to the queue.

In our case, we start with an entry point method. Using the class hierarchy, we gather all reachable methods and add them to the worklist. It is necessary to track the data flow of these methods in their invocation contexts. Each method is invoked in some context that contains currently tracked expressions, flows and other entities that define the method's environment at the time of its invocation. After analysing a single method in its specific invocation context, we have to evaluate whether the data flow has changed. If it did, we must add all callers and callees to the worklist. That is because the caller's dataflow might now change since it may get a different result when processing the call (the invocation context has changed). The same applies to the called methods. The data flow that is considered in callee's call analysis is now different, and it may lead to a different

analysis result. On the other hand, if the data flow has not changed, we don't have to add anything to the worklist and continue with the next method.

Method Handlers

The worklist algorithm processes each method separately. Analysis of methods is performed via so-called method handlers. All handlers implement the common interface `IMethodHandler` with a contract for two methods `CanHandle` and `TryHandle`, which are essential for the worklist algorithm. Both accept a method descriptor from the `IntermediateCodeInfrastructure` project and return a boolean. Each handler operates on a pre-defined set of methods that are hard-coded in the code. For instance, methods `System.String System.Console::ReadLine()` and `System.Int32 System.Console::Read()` are both in the set of methods handled by `InputOutputHandler`. Signatures for these and other relevant methods are directly hard-coded into the source code in their respective handler. When simulating the code of a method, we have access to the complete signature of a method being called. That allows us to hard-code methods by their signature. To make life easier, there is a process to generalize some of the signatures. For example, method `System.Void System.Console::WriteLine(System.String)` is defined with substituted argument type `System.Void System.Console::WriteLine(?)`. This argument is matched later and can be substituted by any type.

There are three special kinds of handlers. The `AnalyzingMethodHanlder`, the `IdentityHandler` and the `SemanticDescriptionsHandler`. The `AnalyzingMethodHanlder` inspects the CIL code of the method and simulates it using the `IntermediateCodeSimulator`. This handler is used for the user-defined methods in which we are looking for the data flow. The `IdentityHandler` is used as a fallback for `AnalyzingMehtodHandler`. Whenever a method is set to be omitted or ignored (usually due to unsupported functionality or to prevent recursion into library functions), we only propagate data flows from arguments to return value when applicable using the `IdentityHandler`. The last is `SemanticDescriptionsHandler`, which handles a set of methods whose calls are very common in any code. Such methods are, for instance, various getters on all collections from the standard library, their constructor methods, string operations or linq operations. The documentation of the initial software project describes this handler as one that *"...handles all methods, which should be analyzed by its semantics and not by its CIL code..."* [23].

The other six method handlers are called plugin handlers. They were developed as a part of an optional plugin and are not vital for the core of the analysis, although the result graphs without those plugins are useless. They can be included or excluded in the analysis as necessary. These plugins include `AdoNetHandler`, `EntityFrameworkCoreHandler` or `InputOutputHandler`. Each handles methods from a specific library or logical part of code. For example, the `AdoNetHandler` operates on methods that expose data access services in the .NET framework. The `InputOutputHandler` handles any file and standard output and input operations such as `System.Console.WriteLine(string)`.

In general, a handler accepts some source flows, performs some well-defined action and propagates them into a sink. These sources and sinks are called endpoints. An endpoint represents the source or target, respectively, of data flows.

They can be arguments, the receiver itself (the instance the method is called on) or return values. Each handler can implement its own special endpoints, such as the `ConsoleOutEndpoint` in the `InputOutputHandler`, to signalize some special, uncommon case. The propagation routine then carries the logic that is necessary to perform for a given method. To see an example, let's take a look at a handler registered for writing into a text writer in Listing 3.8. This handler operates on any method with the signature of `System.Void System.IO.TextWriter::Write(?)` with an arbitrary first argument. It gathers flows from the first argument, which is the content to be written. Transforms it inside the `InputOutputPropagationRoutine.StreamWrite` singleton. Finally, any outgoing flows are then transferred to the receiver.

```
1 handler.AddHandler(Signature.Parse("System.Void
    System.IO.TextWriter::Write(?)"),
2     new IOFlowHandlerBuilder()
3         .AllowCallbacks()
4         .PropagationMode(InputOutputPropagationRoutine
5             .StreamWrite)
6         .PropagateFromArgument(1)
7         .PropagateToReceiver()
8         .Build());
```

Listing 3.8: Definition of a handler for the method `Void TextWriter::Write(?)` that propagates flows from the first argument into the receiver via the `InputOutputPropagationRoutine.StreamWrite` routine.

It is clear that it would be very tedious and unscalable to have a handler for every single library method. Even the example in the Listing 3.8 would not be very helpful as the method `System.IO.Stream::Write` is abstract and must be overridden in every implementation. This is resolved by the process of how handlers for methods are selected. Any encountered method call is first searched by its direct signature as is. If no handler with such a signature is found, the type the method is defined on is queried for ancestors, starting with interfaces and then types. Adequately modified signature is tried to be found with each ancestor type. This process is visualised in the flow graph in Figure 3.1. The terms *unused interface* and *unused type* represent a type that was not used in the algorithm cycle before. In other words, each type is tested only once.

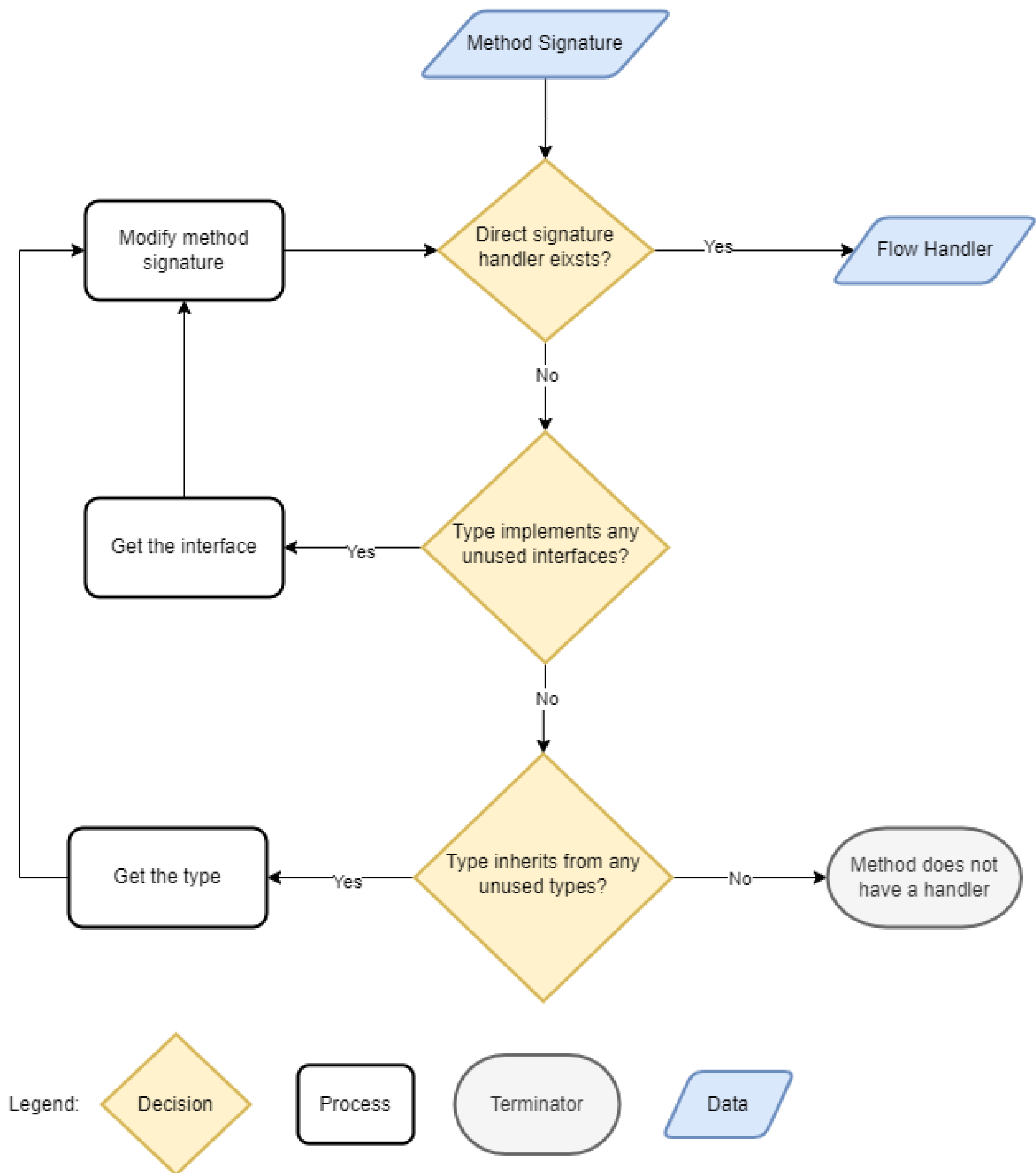


Figure 3.1: A flow graph of handler selection for encountered method. Terms *unused interface* and *unused type* represent a type that was not used in the algorithm.

It is important to mention two details. The process of handler selection is sensitive to what is searched first — implemented interfaces or inherited types. A method in C# can implement multiple interfaces. Therefore, the order in which we process them may also influence the result. Another very important side effect of this handler selection is that the order in which we search for handlers also matters. Theoretically, two plugins may contain handlers for the same interface method, resulting in a preference for the first plugin being processed. This is, however, very unlikely as the purpose of the plugins is to handle methods specific to some library or framework. In other words, they are as specific and specialised as possible.

Data Flow Creation

Now that we understand how the worklist algorithm analyses individual methods, we should discuss what it means that a data flow is created and how exactly that happens. Suppose an arbitrary entry-point method that is analyzed by the `AnalyzingMethodHandler`. On instructions such as the `ldc.i4.0` a `ContstantExpression` is created. This expression is then transformed into a `ConstantFlow` on demand when it's needed for the flow analysis. For simplification, we can imagine that whenever a constant is loaded at some point in the program, a `ConstantFlow` is created and propagated further. In other words, if we get a constant from an argument and we use it to index into a database column, this `ConstantFlow` is tracked and represents an edge from an argument to the database query. There are over forty different kinds of flow classes. Similar to propagation routines and endpoints, plugins can create their own flows as necessary to make abstraction easier.

Other flows are created inside the propagation routines. A good example of this is writing into a file. Suppose a code snippet that opens a file and writes some arbitrary text into it. This C# code snippet can be seen at Listing 3.9 together with selected CIL instructions that are important for this example. Once the `AnalyzingMethodHandler` tries to resolve the first `call` instruction, it finds an `InputOutputHandler` with set `FileStreamPathPropagationRoutine`. We can see the definition of this handler at Listing 3.10. The logic of its propagation routine is to find all possible names from the flows in the zeroth argument and create a new `FilePathFlow`, which is propagated into the return value representing the writer. Later, a `callvirt` instruction is encountered, and a different handler is found. The definition of this handler is shown in the Listing 3.8. That handler's propagation routine finds all possible `FilePathFlow`s, such as the one created earlier, to determine which file is being written to. Then, it will create a `FileStreamFlow` using the information about the file from the `FilePathFlow` and content from the argument endpoint. This completes the complex analysis of writing into a file. As a result, we are left with `FileStreamFlow` with the required information about both the file and the content.

```
1  /*
2  IL_0000: ldstr "demo.txt"
3  IL_0005: call class [System.Runtime]System.IO.StreamWriter
       [System.Runtime]System.IO.File::AppendText(string)
4  IL_000a: stloc.0
5  ...
6  IL_000d: ldstr "sample text"
7  IL_0012: callvirt instance void
       [System.Runtime]System.IO.TextWriter::Write(string)
8  ...
9  */
10 using (System.IO.StreamWriter streamWriter =
       System.IO.File.AppendText("demo.txt"))
11 {
12      streamWriter.Write("sample text");
13 }
```

Listing 3.9: C# code example of writing into a file with corresponding selected CIL instructions.

```
handler.AddHandler(Signature.Parse("System.IO.StreamWriter
    System.IO.File::AppendText(System.String)"),
    new IOFlowHandlerBuilder()
        .PropagationMode(
            InputOutputPropagationRoutine.FileStreamPath)
        .PropagateFromArgument(0)
        .PropagateToReturnValue()
        .Build());
```

Listing 3.10: Definition of a handler for method `StreamWriter File::AppendText(String)` that propagates flows from the first argument into the receiver via the `InputOutputPropagationRoutine.FileStreamPath`.

All flow data is gathered in a complex structure inside of the `ProgramFlowData` class. The `ProgramFlowData` class contains flow data of the whole program. In its field `Methods`, it keeps track of all flows associated with every expression for each part of every method for each of the method's invocation contexts. This structure is visualised in Figure 3.2. It shows a simplified representation structure of how individual data flows are stored for the whole program.

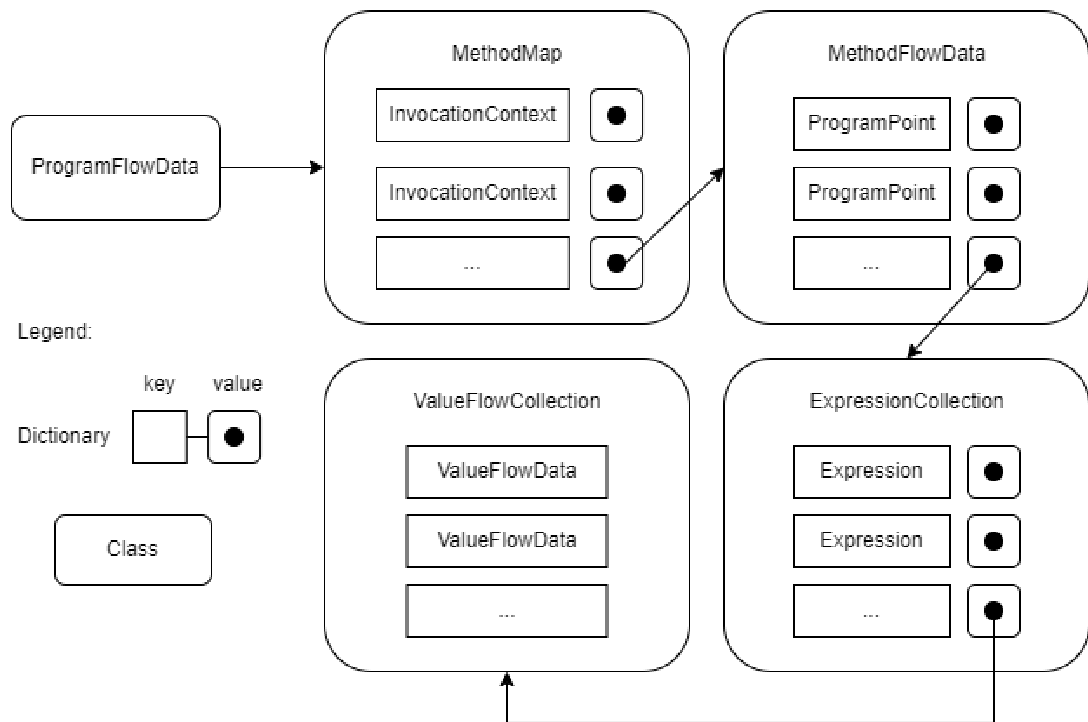


Figure 3.2: Diagram of how `ProgramFlowData` stores data flows for every part of every method.

3.2.5 Analysis Result Transformer

The last important project, which is a little specific to the C# scanner, is the `AnalysisResultsTransformer` project. Its purpose is to process the result of the symbolic analysis and transform it into a serializable graph structure understandable by the `DataflowGenerator` project in Java. The raw output of the

symbolic analysis is the ProgramFlowData class with a rather complex structure. In order to simplify this structure and omit unnecessary data being serialized, the AnalyisisResultsTransformer class was born. Another added benefit is that now the interface is stable, and we can modify the ProgramFlowData class as necessary without worrying about affecting external modules.

Symbol analysis result transformation is handled in the ResultsTransformer class. It uses node separators that are specific for each type of node. The FileStreamNodeSeparator class handles flows corresponding to read and write operations in the program and creates nodes for them. The EntryPointsNodeSeparator class handles all various data flow entry points or exit points. Such as fields, arguments, and return values, but also console input and output. The separator searches for all relevant flows and determines their origin and, if applicable, any incoming edges. Finally, all nodes and edges are gathered in the data class AnalysisResultsGraph, which is the main output of the transformer. The graph is then serialized using protocol buffers, which is necessary to be able to transfer data flow information further to the data flow generator, which is implemented in Java.

3.2.6 Analysis Runner

Analysis Runner is the last project that glues all parts of the analysis together. It prepares the context for the analysis. This includes creating a class hierarchy and a call graph, collecting and registering plugin handlers in the correct order and configuring the overall analysis context based on user configuration. Then, it runs the symbolic analysis itself and transforms the result into a serializable graph.

3.3 Analysis Life Cycle

Now that we understand all the individual parts of the scanner, we can take a look at a simple example and use it to explain the analysis life cycle.

Usecase: User Survey

Let's start with a real-life scenario, which we will simplify for the purpose of this thesis and use to explain the analysis. Suppose we gather potential customer data via a survey where customers fill out forms with details and can sign up for a newsletter. This information is serialized into a flat file. We have an automated job that validates this information for each respondent, anonymizes the personal details, and creates a record in a database used for sending newsletters and a database with the rest of the gathered data. Suppose we also want to inform users who have not completed their profiles and can not join the newsletter. Information about these customers is serialized into a separate file which is then processed by another job. We can see this process on a diagram in Figure 3.3.

The highly simplified code snippet that represents the implementation of this process in Listing 3.11. Implementation details, such as reading user data from file or database queries, are ignored and hidden behind their respective method calls. This code first connects to both databases it will be working with. Then,

it reads the data from the survey, filters incomplete profiles and reports them in another file. The next part is filtering user profiles subscribed to the newsletter and saving them into a special database. Finally, data is anonymised and inserted into another database.

```
1 var NewsletterDB = GetNewsLetterDatabase();
2 var DataDB = GetSurveyDatabase();
3
4 var surveyData = ReadAllUserDataFromFile("survey_data.csv");
5
6 var validatedData = ValidateAndSave(surveyData,
    "incomplete_profiles.csv");
7
8 AggregateNewsletterAndSave(validatedData, NewsletterDB);
9
10 AnonymiseAndSave(validatedData, DataDB);
```

Listing 3.11: Code snippet representing implementation of use case with user survey processing. This code snippet hides implementation details in method calls.

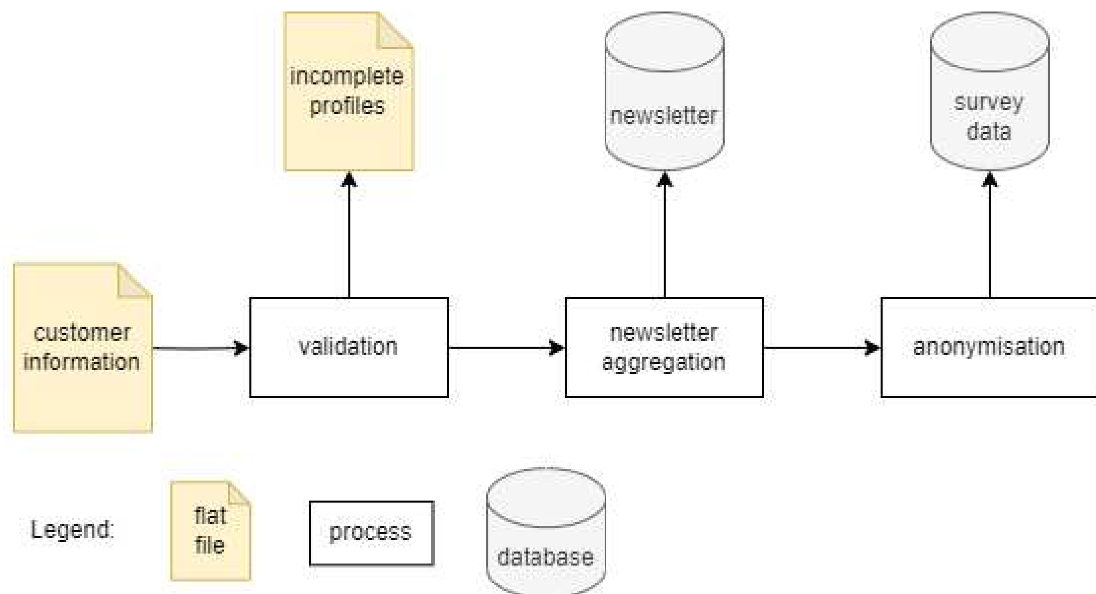


Figure 3.3: User survey processing diagram that validates user data, saves incomplete data into a flat file, saves newsletter to users that opted in and saves anonymised survey data into a database.

We will get a very simple graph if we think about the data lineage in this use case. We can see such a graph in Figure 3.4 generated by the Manta software. It only contains one input file read node with the input file, one node for processing with three output edges that lead into two separate databases and one flat file. There is one more transition block with details about the queries.

Before we dive into the analysis life cycle, it is worth mentioning some interesting insights that are not very obvious at first sight. In general code, information about a database we are operating with is gathered in one place, but operations on this database are performed somewhere else. This is usually the case with

files as well. We must keep track of this information and use it later on action calls upon these databases and files. Even though our diagram in Figure 3.3 has three different processing cells for each row number 6, 8 and 10 of the program in Listing 3.11, the final produced graph has only a single node with three cells and three output edges. That is because, from the data lineage point of view, the place of processing itself is not as important. It may all happen inside one method or in separate calls, as it is in our use case. Data flows into the called method (e.g. `AggregateNewsletterAndSave`), where it gets saved into a database (that is a valid data flow) and then back to the caller for further processing.

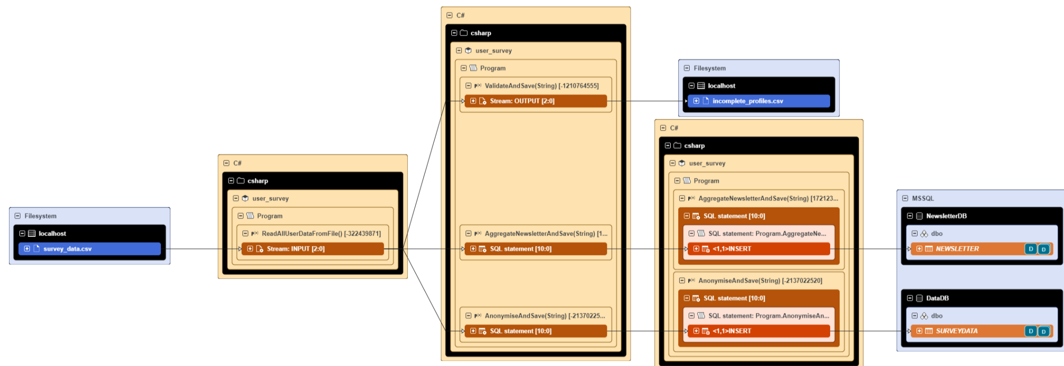


Figure 3.4: Generated data lineage graph of a user survey processing use case 3.11.

The Life Cycle

The input for the analysis is some console application with its source code shown in Listing 3.11. The first part of the analysis is the extractor scenario. That begins in the Java part of the program inside the `manta-connector-csharp-extractor`. The `ExtractorScenario` obtains extractor configuration from a bean and serializes it into a file. Then, it calls the `CSharpApplicationRunner` from the `manta-connector-csharp-runner` package. It sets up arguments for the C# part and invokes it in a different process. From here, we enter the part implemented in C#, and the C# console application `CSharpScanner` is executed. The provided CLI arguments direct the execution further into the extractor part. The individual steps are:

1. create analysis configuration,
2. extract entry points from all input files,
3. serialize analysis and entry point configuration into a file.

The analysis configuration is for the reader, the second scenario. It simply gathers information about excluded namespaces and directories containing input for the analysis. The entry point extraction is also fairly simple. It first creates a class hierarchy from source metadata and filters methods that satisfy some entry point conditions. For example, the console application entry point must be a static public method called `Main`. It must satisfy more conditions enumerated in

the documentation [28]. Once it finds all entry points, they are serialized into a file together with the configuration. This concludes the extractor scenario.

The second scenario to be run is the dataflow scenario. Like the extractor one, this starts in Java inside the `manta-connector-csharp` package, where just CLI arguments are prepared and then in `manta-connector-csharp-runner`, the C# part is invoked in an extra process. The `CSharpScanner` recognizes the dataflow scenario and executes the `Connector`. The steps of the connector are:

1. deserialize configurations and entry points,
2. to create an input for symbolic analysis,
3. prepare static analysis context with class hierarchy and call graph,
4. collect plugin handlers and initialize them,
5. perform alias analysis,
6. using the worklist algorithm it analyses the entry points,
7. transforms the result into a serializable graph and writes it into a file.

The deserialization and creation of input for symbolic analysis happen still inside of the `Connector` class. Then, we move into `SymbolicAnalysis`, where we initialize the class hierarchy and the call graph. After that, method plugin handlers are gathered and initialized based on the provided configuration. We may want to explicitly ignore some, such as handling console input and output. What's left is the analysis itself. Alias analysis is initiated, but since it is evaluated lazily, the work is done later. The provided entry point acts as the beginning of the worklist algorithm. Once the analysis reaches a fixed point, the `AnalysisResultsTransformer` class creates nodes and a graph containing only necessary information for data flow. This graph is serialized, and we return back to the Java part. Here, the data flow generator calls a query service if it is necessary to resolve some database queries. After everything is done, we are left with a graph that can be used in further scenarios for visualisation or export.

3.4 Current Limitations

The current C# scanner is already capable of analyzing non-trivial programs. However, very often, those programs are cherry-picked or fine-tuned. Sometimes, only small scanner fixes are required to serve an edge-case scenario. Sometimes, it requires completely new features. The focus of feature development often changes because the scanner features are developed based on stakeholders' requirements and not by a pre-defined list or specification. This brings in several limitations. One is that what is important to one stakeholder may not be to others. This results in some half-baked solutions. For example, one part of a framework that is required by some stakeholders gets implemented, but the rest of the framework is omitted as it is not deemed necessary. An example of this is a comprehensive refactoring of the dataflow generator performed by the team responsible for the bytecode scanner. This refactoring altered C# scanner behaviour because some

magic constants were previously used to mark lineage from unknown database columns. These constants were removed and replaced with a proper abstraction in the Java part of the dataflow generator. Magic constants were left in the C# scanner, resulting in invalid lineage in some special cases. Another good example is tied to the C# language itself. In most cases, it is not required to differentiate between some language constructs, such as classes and enums. As a result, enums are considered to be a regular class, just like any other.

The entry point extraction is another limitation of the current C# scanner. Currently, only the standard console application entry point is considered. Looking at other entry points, such as web endpoints, is much more interesting for many applications, especially web applications. In other cases, we don't want to analyse a C# program at all but just a snippet of code written in C#, which is a part of other technology. This is called an embedded code. Stakeholders use the possibility of embedding custom code snippets into other technologies fairly often. That means that even though they do not directly use the C# scanner, perhaps some other technology, such as SSIS, they would use the scanner indirectly. C# scanner currently has no capability to analyse any embedded code, let alone be used as an embedded code analyser service from within other scanners.

Another major issue for the C# scanner is its performance. Some inputs take tens of hours to finish the analysis. One of the issues is that unnecessary data is dragged throughout the analysis without any impact. A good example is any output operation. For example, writing into the console output never propagates any data flow further, yet it is now tracked as a flow and checked throughout the whole analysis's life, slowing down the analysis as a result and enforcing bigger memory constraints. There are some known improvements that could be implemented. These improvements were deployed to the bytecode scanner and proved worthy. Other bottlenecks will have to be found and analysed before implementation.

4. Requirements and Analysis

In this chapter, we will dive deeper into individual improvements and issues solved in the scope of this thesis. To do that, we split them into two different categories. One category is for goals and requirements set by stakeholders. The other is set by addressing the technical debt of the codebase. We will analyse them in more detail.

4.1 Stakeholder Requirements

The main source of feature requests and bug reports are stakeholders themselves since Manta has a customer-first ideology.

4.1.1 ASP.NET Endpoints

One of the requirements is to enable the analysis of web applications written in ASP.NET [12]. A generic web application written using the ASP.NET framework is a regular console application with the Main method entry point that is executed. The framework itself takes care of correctly handling the web endpoints. If we think of data flow analysis with the Main method as an entry point, we either get into a very complex analysis of the framework functions or ignore them and do not get any lineage at all. In order to get lineage from the web endpoints, we have to consider them as entry points of the analysis. ASP.NET web APIs can be implemented in two very different ways. We can either go with a controller-based approach where each controller is a class with individual endpoints. This class has to satisfy some special requirements in order to be registered by the framework. The other approach is called *Minimal APIs*. They are simple lambdas or methods. The design hides the host class and focuses on extension methods that take those lambdas or functions as an argument. It is important to note that these two approaches are not equivalent. The minimal APIs do not yet support some capabilities [29], such as but not limited to:

- no built-in support for validation,
- no built-in support for model binding,
- no support for application parts or the application model.

Most of these perform some modifications and configuration of the ASP.NET framework. Details are unimportant for the scope of this thesis. A low-code ASP.NET application with minimal APIs can be seen in Listing 4.1. It consists of an app builder that returns "Hello World!" on GET request to the base_url/HelloWorld. The web endpoint is defined on line 35. In the comment, we can see part of the metadata as visualised by the Rider IDE. This metadata was simplified for the purpose of this example. First, we can see that there is a new class <>c generated by the compiler, as we can see by the custom attribute CompilerGeneratedAttribute and its illegal name. This nested class contains the method <<Main>$>b__0_0. This is our lambda function from

line 36. The term for such methods is *anonymous method*. It also contains the static delegate <>9__0_0, which is assigned the mentioned method on initialization. The metadata snippet also shows part of the CIL of the Main function. Specifically, we can see the opcodes for loading the first local variable (the app), then sequentially loading arguments of the call: the string "\" and the delegate to our compiler-generated lambda function. It is not always the rule that the second argument, the delegate, is a compiler-generated method that can be found in a specific compiler-generated class. That is because we can create a local function manually, and it also gets compiler-generated sugar and name. This function, however, lives inside the class it is defined at. This difference in generated CIL of anonymous methods and local functions exists because of the original intention, backwards compatibility and design as described in this StackOverflow question [30]. To fan the flames, not all compiler-generated functions in the before-mentioned class have to act as web endpoint lambdas.

In conclusion, to support loading the correct method as an entry point in our analysis, we would have to analyze all app.MapGet calls and their equivalents like MapPost. From there, we will look at the last argument and use the delegate's method as an entry point. It is important to note that we do not really care what type of HTTP method is analysed, nor what its name is. Those characteristics are irrelevant to the data lineage.

```
/*
.class nested private sealed auto ansi serializable
   beforefieldinit
  '<>c' extends System.Object
{
    .custom instance void System.Runtime.CompilerServices
      .CompilerGeneratedAttribute::.ctor()
    .field public static initonly class Program/'<>c' '<>9'
    .field public static class System.Func`1<string> '<>9__0_0'

    ...

    .method assembly hidebysig instance string
    '<<Main>$>b__0_0'() cil managed
    {
      ldstr        "Hello World!"
      ret
    }
}

...

.method private hidebysig static void
  '<Main>$'( string[] args) cil managed
{
    ...
    ldloc.1
    ldstr         "/"
    stsfld        class System.Func`1<string>
       Program/'<>c'::'<>9__0_0'
    call          class
       Builder.EndpointRouteBuilderExtensions::MapGet(class
       IEndpointRouteBuilder, string, class System.Delegate)
```

```
30     ...
31 }
32 */
33 var builder = WebApplication.CreateBuilder(args);
34 var app = builder.Build();
35
36 app.MapGet("/HelloWorld", () => "Hello World!");
37
38 app.Run();
```

Listing 4.1: Low-code ASP.NET application with minimal APIs.

The same web API defined using the controller approach is in Listing 4.2. Things are much more simple here. We can see the metadata in the comment. From the Microsoft documentation [31], we know that a controller must extend the `ControllerBase`. For its methods to be considered action methods, they must be public and either follow the naming convention of having the correct HTTP method prefix (GET, POST, PUT, etc.) or have a specific attribute specifying the HTTP method [32], among other requirements for the method itself [33]. We can see this attribute in the code snippet. The important thing is that we can get all this information from the metadata without analysing the CIL as it is with the minimal API approach. Since we need to check all methods to see if they satisfy those conditions, analysing the CIL would nontrivially slow down the extraction process with minimal results, as web APIs are generally uncommon. The minimal API approach is also less common. None of the stakeholders use it. Due to those reasons, it was decided that the extraction of controller-based endpoints was sufficient.

```
1  /*
2  .class public auto ansi beforefieldinit
3    APIWithControllers.Controllers.HelloWorldController
4      extends Microsoft.AspNetCore.Mvc.ControllerBase
5  {
6    .custom instance void
         Microsoft.AspNetCore.Mvc.ApiControllerAttribute::.ctor()
7    .custom instance void
         Microsoft.AspNetCore.Mvc.RouteAttribute::.ctor(string)
8    ...
9    .method public hidebysig instance string
10     Get() cil managed
11   {
12     .custom instance void
          Microsoft.AspNetCore.Mvc.HttpGetAttribute::.ctor()
13     ...
14   }
15 }
16 */
17 [ApiController]
18 [Route("[controller]")]
19 public class HelloWorldController : ControllerBase
20 {
21     [HttpGet]
22     public String Get()
23     {
24         return "Hello World!";
25     }
```

```
26 }
```

Listing 4.2: Low-code ASP.NET controller.

4.1.2 Embedded Code Service

Another requirement that comes directly from stakeholders is using the C# scanner as a service to analyse embedded code in different technologies. A lot of *Extract-Transform-Load* (*ETL*) technologies allow users to define custom operations outside of the standard toolset. Should that be a special kind of transformation or special input or output. Such technologies are, for example, *Azure Data Factory* (*ADF*) or *SQL Server Integration Services* (*SSIS*). Manta already has scanners for both of these technologies. The goal is to extend the C# scanner to be ready to be used as a service from said scanners, analyze C# code snippets, create a graph and report this graph to the source technology (ADF or SSIS). The source technology will then merge the graph with its generated graph. This service should be reusable for multiple embedded code snippets throughout the analysis cycle. There is already a similar service that is used inside of the dataflow generator — Query Service. This service is used to resolve SQL queries. Just like embedded code, SQL queries are used in multiple technologies, and it does not make sense to implement the resolver again every time. There is an existing embedded code service for the Python programming language, which was developed in 2023 as a part of a Master thesis by Michal Jurčo [34]. We can use this service as a starting point for the solution of most problems. The service was designed to be a blueprint usable for other language scanners as well. In fact, the service used a lot of ideas from the query service itself.

We have to ensure a few things for the scanner to be suitable as a service.

- The input for the C# scanner is supported.
- We can get all the necessary information from the source technology for the analysis.
- The expected output can be merged into the source technology's graph.

This translates into several requirements:

- The input for the C# scanner must be compiled C# code — CIL.
- We must be able to detect the entry point statically, i.e. it must not require reflection.
- Any additional input from source technology is serializable.
- The output is serializable.

A decent analysis of ETL technologies can be seen in the master thesis by Michal Jurčo [34]. However, the key points demanded by stakeholders are missing. Let's look closer at three technologies utilizing embedded C# code that are the most relevant for the purpose of C# scanner in terms of demand.

Azure Data Factory

According to the Microsoft documentation, the Azure Data Factory, or ADF, is "*the cloud-based ETL and data integration service that allows you to create data-driven workflows for orchestrating data movement and transforming data at scale.*"[11]. ADF supports so-called custom activities [35]. The activities can actually be any program executable on the target operating system of respective *Azure Batch Pool* nodes. This is partially possible thanks to the generic way data is transferred to the custom activity. Any data or information that could be of use is serialized into a JSON document and must be deserialized in the custom activity. If we want to pass some output from the activity for further processing, it must be serialized into a JSON document again. Standard and error output are saved into a special service inside of the ADF. In the context of the C# scanner, this means that the embedded code is a standard console application without any additional complexities.

SQL Server Integration Service

SQL Server Integration Service, or SSIS, is another platform for data integration and transformation. In the SSIS graphical user interface, users can set up complex pipelines using so-called components. These components serve different purposes that can be used in an integration service.

- Task — Perform a predefined unit of work.
- Connection Manager — Connect to external data sources.
- Log Provider — Logging.
- Enumerator — Iterating over objects.
- Data Flow Component — Sources, transformations or destinations.

All these components were mentioned in the thesis by Michal Jurčo. There was an example of a source component, but some important details were missing. One missing detail is that there are multiple versions of tasks — Data Flow Tasks, Data Preparation Tasks, Workflow Tasks, etc. [36]. Another detail is that users can define custom data flow objects, tasks and script tasks. Data flow objects are the connection managers, log providers, etc. Tasks are actions performed on each row of data. Finally, a script task is an action performed only once per object.

The way these are implemented differs as well. Data flow objects and tasks must implement some special interface or an abstract class in order to be valid and usable [37] [38]. On the other hand, script tasks use specially generated partial classes that are handled internally by the SSIS [39]. That means there is no need to implement a special interface.

In the context of embedded code service, this means we must ensure a correct entry point is extracted. For custom data flow objects and tasks, we must look for classes that implement relevant abstract classes or interfaces and use relevant methods as entry points. Script tasks always reside in the same class, as it is partial implementation. The entry point method is also always the same.

Any incoming and outgoing data is saved in objects accessible via the `Dts` object from the `Microsoft.SqlServer.Dts.Runtime` namespace. For example, any variables are stored in the `Variable` property of `Dts`. In SSIS, they are set up in two different modes: `ReadOnlyVariables` and `ReadWriteVariables`. During analysis, we have to be able to access those variables. A perfect way to handle such methods is to implement a new method handler plugin. In this case, it would be a plugin specific for SSIS operating over methods from the `Microsoft.SqlServer.Dts.Runtime` namespace. If we encounter a getter for a variable, some handler will create a new flow based on the variable. The issue of accessing the values of this kind of data still remains. Should the variable contain a connection to some database, we need access to the value that can be passed to the query service. A similar problem was encountered in the Python embedded code service. We discuss this later in this section.

Microsoft SQL Server

Microsoft SQL Server is a relational database management system. Users can define so-called CLR user-defined functions and types. Types are unimportant from the data flow point of view. They are only an abstraction. Functions further differentiate into scalar-valued functions, table-valued functions and aggregates [40]. This differentiation is solely formal. They first have to be defined in SQL with all necessary details about the assembly and function signature. This completely solves the entry point extraction. The body of the methods is an arbitrary method which can be analysed.

Input

An important note is that all three technologies require the embedded code to be compiled into CIL. This is a crucial detail because the C# scanner can analyse only CIL, not the source code directly. This is a rule, not an exception, because the source technology would have to include a compiler. An entry point is always well-defined and can be analysed statically.

Suppose an example of a simple transformation on an arbitrary database column that propagates the data into two newly created datasets. This could be a custom task in SSIS. The embedded code graph must have one incoming edge from the outside. This edge should come from the source column node. At the embedded code service, we do not have this node nor any information that it even exists. We need to transfer additional information into the service.

Insight and Outsight

Apart from the code to analyze and the graph on output, we may need to transfer some additional information. In the SSIS technology, we have access to variables that may hold crucial information for the analysis. These variables can be passed into the embedded code. This works the other way as well. Some crucial data may be generated in the code and written into variables. In the source technology, we need a way to observe the new data. A very good solution for this in the Python embedded code service is called Insight and Outsight. It was not a part of the thesis by Michal Jurčo, but it is described in great detail. The naming is from the

source technology point of view. Insight provides a way for source technology to look at data coming from the embedded code, whereas outsight provides a way for embedded code service to look at data incoming from the source technology. The implementation in Python embedded code service utilises a kind of observer design pattern where the source technology implements its own insighter and outsighter. These two classes record any interesting operations and materialize them into insight or outsight on demand. The key point in its implementation is that the entire codebase is written in Java. The source technology can easily share an implementation of these observers with the service. The C# scanner is implemented in C#. We can either create an interop between those two languages or force source technology to implement the observer in C# and have it report everything and serialize it. The latter is a much simpler solution and in the context of such a massive product, having the C# scanner itself launched as an external process without interop, it does not make sense to implement any kind of interop.

Output

The output of the embedded code service is the graph itself. Its serialization is possible. Let's again suppose an example from above. Simple transformation on an arbitrary database column that propagates the data into two newly created datasets. Thanks to insight and outsight, we can now carry additional information about the database column. How do you use it to merge the embedded code graph with the source technology graph? We could implement a special node for each technology. Similarly to the insighter and outsigher, this would be the responsibility of the source technology. Such nodes would carry all the necessary information required to successfully connect edges from source technology to the embedded code and back. This approach is naive and hard to maintain. We can instead use the already existing infrastructure of insight and outsight to carry necessary information about edges. Simple dummy node facilitating inputs or outputs of the embedded code graph. This node is called a *pin node*. This abstraction was also described in great detail in the thesis by Michal Jurčo.

Merging of the two graphs is reduced to simple mapping of edges. Output edges from an input pin node to nodes of the embedded code graph. Input edges of all output pin nodes into their respective nodes in the source technology, if they exist. This process is visualised in Figure 4.1. The embedded code graph contains one input pin node with two outgoing edges into two dataset nodes. The source graph, on the other hand, has one output pin node with an incoming edge from a database. These two pin nodes are, in fact, the same. The process is about eliminating the pin node and replacing the edges with their contracted alternatives. The information about what should be pinned where is carried in the insight and outsight and shall be gathered in the method handler plugins for each respective source technology.

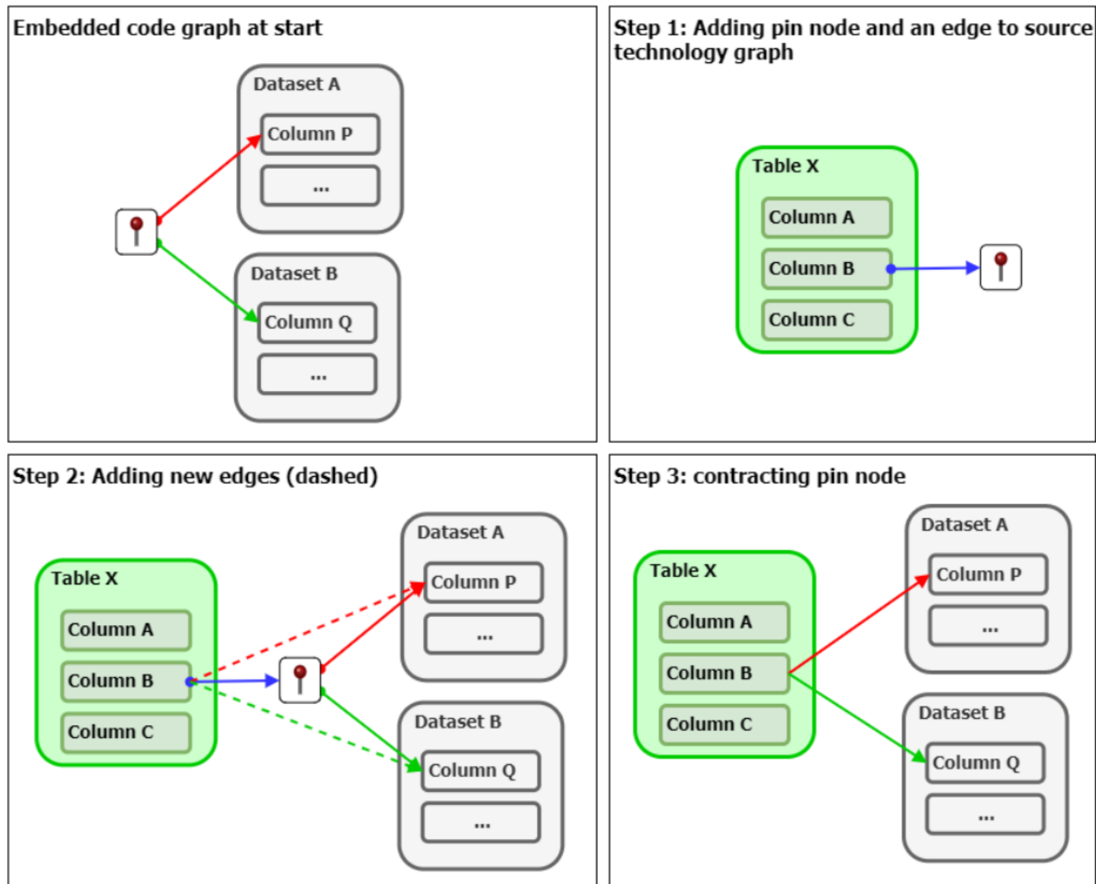


Figure 4.1: Merging of embedded code graph with the source technology graph using pin node. Figure taken from thesis by Michal Jurčo [34].

Conclusion

Based on a deeper analysis of the source technologies that may use the embedded code service, we understand that the design of the current Python embedded code service is sufficient for the C# embedded code service as well. There is one exception to the rule. The service was designed so that each programming language should implement its own version in contrast to one super service operating on all languages (Python, C#, Java, etc.). The thesis states a common interface for the service. This is currently non-existent, presumably by accident, because it is dependent on a part of the Python scanner. With that being said, it is important to state that this is not an issue for the C# embedded code service, as it is written in a completely different programming language and is separated from the rest of the codebase. As a result, all of this infrastructure must be duplicated on the C# part of the codebase.

Another thing is that the insight and outsight must be serialized and also duplicated on the Java and C# part of the codebase. This should be an implementation detail of the service. The API of the service should match the one designed for the Python embedded code service.

The design for embedded code services and its interaction with source technology set by Michal Jurčo in his thesis [34] can be seen in Figure 4.2. As discussed,

this design is insufficient for the C# scanner. For the embedded code service, we must create the class `CSharpEmbeddedCodeService`. Its input is the code to be analysed (CIL) and other source-technology-specific information. The service first performs orchestration. This is again specific to each source technology. Note that the orchestration includes running the extractor, which may not be obvious at first. In our case, the two most interesting technologies are ADF and SSIS, as discussed. The orchestration was designed to be performed only before the analysis of the code. Since running the C# scanner requires an interop between two languages and data serialization, post-analysis orchestration (i.e. deserialization) is required. This implies that everything must be performed eagerly. In comparison, the Python embedded code service utilizes exposed API from the source technology because everything is implemented in Java and evaluates as much data as possible lazily.

Each source technology is somehow unique. Some additional helpers and classes must be implemented for a source technology to use the embedded code service. Such helpers are:

1. an insighter,
2. an insight,
3. an outsight,
4. an orchestrator,
5. a configuration.

An insighter is a helper that materializes the insight. Insight should be generated lazily on demand to restrain unnecessary computation. Throughout the analysis, events that alter the insight are collected by the insighter. This is impossible because any interaction between Java and C# is impossible or tedious. An insight must be fully generated and serialized before the CIL code analysis is finished and later deserialized in Java and used as necessary. The outsight is generated by the source technology. The outsight must be serialized as well. The orchestrator's job is to prepare the input for the analysis. It includes running the extractor, serialization and any other additional configuration setup. Configuration for embedded code may differ from the one for regular programs. The only change in the embedded code service design is that there is a second round of orchestration after the connector run. This post-orchestration deserializes insight and makes it available for the source technology.

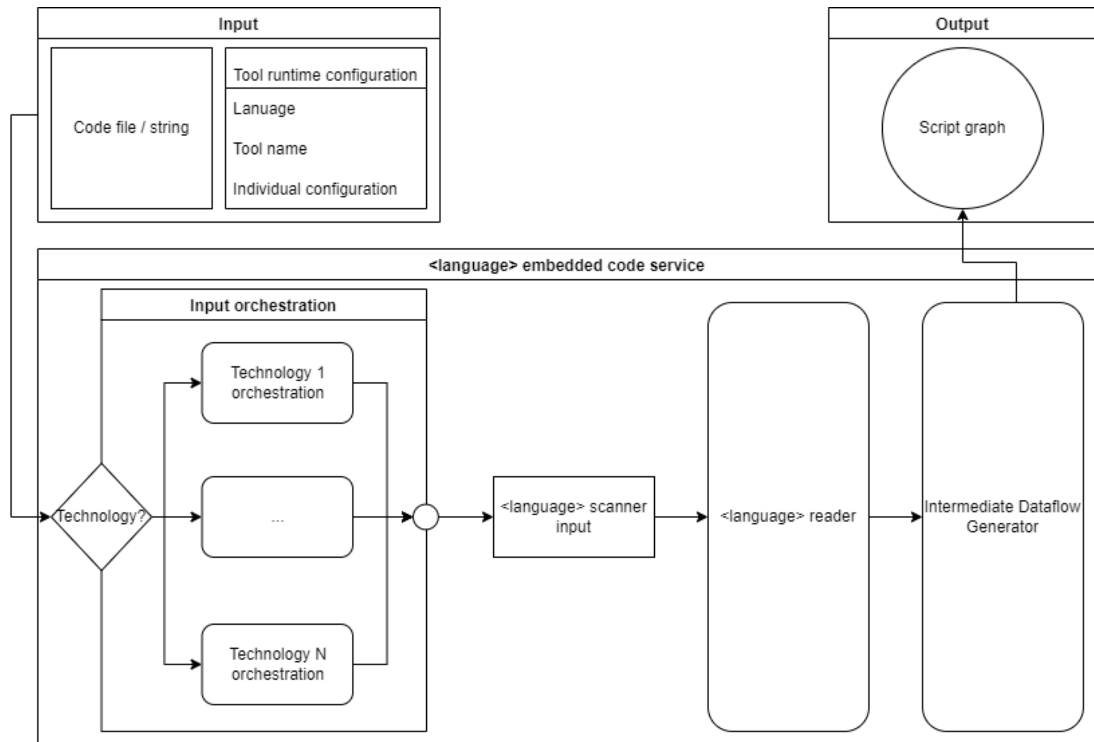


Figure 4.2: General design of an embedded code service.

4.2 Addressing Technical Debt

The second group of goals and requirements comes from the existing technical debt. As stated before, the scanner was developed in iterations, mainly as a university-related project. This resulted in some technical debt that had to be addressed in order to clear the scanner of inefficient approaches and unnecessary technology complexity. Most of the addressed issues were discovered while testing edge case scenarios. Some of them were also encountered in other scanners, and their solutions improved the scanner overall.

4.2.1 Unknown Column Support

As a part of an extensive refactoring performed by other teams using common infrastructure with the C# scanner, the dataflow generator part was modified. This resulted in some scenarios failing completely. One of the reasons is the refactoring of how unknown columns are tracked. In most cases we are unable to tell exactly what is the index or name of the column we are querying. Take the code from Listing 4.3. We access some column `args[1]`. Currently, the scanner differentiates two types of columns: indexed and named, characterised by their index and name, respectively. In case this characteristic is unknown, some special magic value of -1 or an empty string is used as its index and name, respectively. In fact, even if an enumeration of a query or some scalar query is executed, such as `SELECT COUNT(*) FROM USERS`, an indexed column with an index of -1 is created.

```
using (var connection = new SqlConnection(args[0]))
{
    connection.Open();
    var cmd = new SqlCommand("SELECT * FROM USERS", connection);
    var reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        Console.WriteLine($"User: {reader[args[1]]}.");
    }
}
```

Listing 4.3: Query from an unknown named column.

We have to implement an abstraction of an unknown column that will be used in scenarios where the index or name is unknown. This abstraction should follow concepts set by indexed and named columns and shall be used only in cases where the column is unknown. The enumeration of database queries is also a valid reason to use an unknown column. If it is unknown, we have to defensively anticipate any of the possible columns.

It is debatable whether scalar queries should be tracked in the first place. Scalar information from a query of some databases does not propagate any concrete data as it is generally understood. If we look at a graph in Figure 4.3, that would be generated in case of an unknown column query (or enumeration over all columns) as well as an arbitrary scalar query. Suppose both queries are executed, and their result is written into a file. In both cases, the graph contains a node with an SQL statement and an output flow into a stream and file. On the other hand, this information may be vital for some use cases. Even scalar statistics of some data may say a lot about it. We have three different ways to resolve this issue:

1. Use an unknown column for any scalar query.
2. Create a special abstraction for scalar values.
3. Use the named column with the name being the scalar value queried.

The first approach is straightforward and does not require deep changes to the current system. We only have to ensure that any scalar query is eventually represented as an unknown column. This approach neither requires further changes in flow abstractions that are propagated throughout the analysis as they are already supported. The downsight is that the graph information is somewhat inaccurate.

The second approach requires a new abstraction, just like for the unknown column. On one hand, the information would be accurate, but extensive changes affecting other scanners would be required. We would be practically left with the same situation as in the beginning but at a different scanner since they do not support scalar queries but have a different way of representing it.

Finally, the last approach means using existing abstraction for the wrong purpose. Just like now, an indexed column is incorrectly used for unknown columns. Using a named column for scalar values would be incorrect. Similarly to the first approach, not many changes would be necessary.

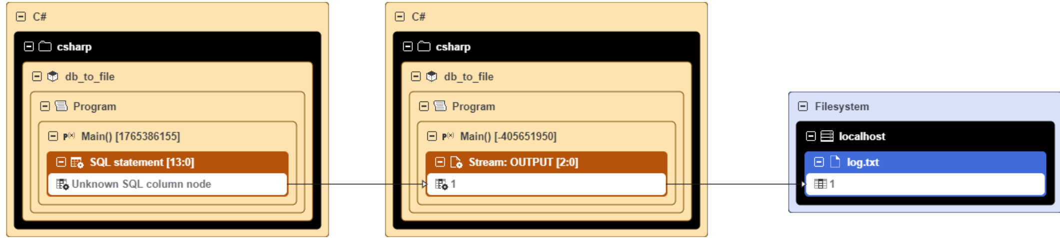


Figure 4.3: Screenshot of a Manta data flow graph with a lineage of a program executing a scalar query and writing the result into a file.

With those constraints in mind, together with the fact that other scanners represent scalar queries as unknown columns as well, it was decided to use the first approach.

4.2.2 Method Summary

A part of this thesis is to enhance the performance of the scanner. We aim to implement several techniques that were, to some extent, already implemented in other scanners. One such improvement is the introduction of the method summary. Throughout the analysis, we encounter a lot of situations that eventually create output nodes. A simple example is `Console.WriteLine()` call. During the analysis, we create a console data flow that indicates the writing operation to the output. We propagate this flow throughout the analysis, validating and checking this flow any time we want to change the flows. The only purpose of this flow is to eventually become a console output node. This flow will not create nor update any other flows because of its nature. On the contrary, the method `Console.ReadLine()` generates a flow that must be propagated further throughout the analysis to map the flow of the data read. The goal is to mark output flows in a summary of the analyzed method, hence the name Method Summary. This summary will carry information about all flows that eventually create output nodes and do not contribute anyhow to the rest of the data flow analysis. Such flows are console output flows and file write flows.

Upon further analysis of the codebase it is discovered that method summary can be used for further improvements in the final part of the analysis. When the result of the symbolic analysis is transformed into a graph within the `ResultsTransformer` class, it is searched for all flows in all methods in all of their invocation contexts. These flows are then filtered one by another by their type and, if applicable, transformed into a node. The method summary can be used as a cache for those interesting flows. We can effectively remove one iteration loop using the method summary. We search in all methods in all of their invocation contexts, which is the minimum we can do to provide complete lineage.

Another insight from further analysis is that determining flow origins can be simplified. Once the flow that will create a node is found (when iterating over all flows of all invocation contexts of all methods in the `ResultsTransformer` class), its origin must be found. That is done by iterating over a map of all callers from the flow's invocation context and then filtering the one caller with the same invocation context as the method being processed in the current iteration of the

transformation. This complex querying and comparison consists of computing multiple hash-codes as everything is saved in dictionary-like data structures. It was discovered that non-trivial time is spent computing those hashes. The improvement is surprisingly straightforward. Since the origin is naturally known at the flow creation, we can use that location instead.

4.2.3 Wrong Method Handler Selection

Method handler selection is another feature deficiency discovered by use case scenario analysis and testing. Suppose the following scenario in Listing 4.4. An interface `IAnimal` and two classes `Dog` and `Cat` implementing said interface. Let's suppose we have three method handler plugins: an animal plugin implemented in the `AnimalPlugin` class, a dog plugin implemented in the `DogPlugin` class and finally, a cat plugin implemented in the `CatPlugin` class. They are registered in order `AnimalPlugin` first, `DogPlugin` second and `CatPlugin` last. Let's also suppose that:

- `AnimalPlugin` has a handler for method `void IAnimal::DoSound()`,
- `DogPlugin` has a handler for method `void Dog::DoSound()`,
- `CatPlugin` has a handler for method `void Cat::DoSound()`.

```
interface IAnimal{
   public void DoSound();
}

public class Dog: IAnimal {
    public void DoSound() {
        Console.WriteLine("Bark");
    }
}

public class Cat: IAnimal {
     public void DoSound() {
         File.WriteAllText("WriteFile.txt", "Meow");
     }
}
```

Listing 4.4: C# example of simple classes implementing an interface.

Currently, the search for method handlers is in a depth-first-search (DFS) manner[1]. If a call of `void Cat::DoSound()` is encountered, it is searched for a direct handler of the method in the plugins in order they are registered. Starting with the `AnimalPlugin`. There is no such handler. Now, instead of searching for direct implementations in other plugins, it is searching for handlers of the interface override[2] of the method. The method is contracted from the `IAnimal` interface. Therefore, we look for `void IAnimal::DoSound()` method handler in the same plugin. There is one in the `AnimalPlugin` class. We use this handler

[1] It is first iterated over all possible overrides of the method before considering the next plugin.

[2] It is not actually an override as the inheritance is in a reverse direction.

and consider this call to be processed. It is easy to see that this is not correct. The desired handler is in the class `CatPlugin`.

This process of searching for a correct handler in plugins is visualised in a sequence diagram in Figure 4.4. We can see three plugins registered in order *PluginA*, *PluginB* and *PluginC*. When searching for a method from class `Class` that implements an interface `Interface` with a contract for a method `Method`, we first look for the class method handler in PluginA. If nothing is found, we look for the interface method handler in PluginA. If nothing is found, we continue searching through other plugins in this manner.

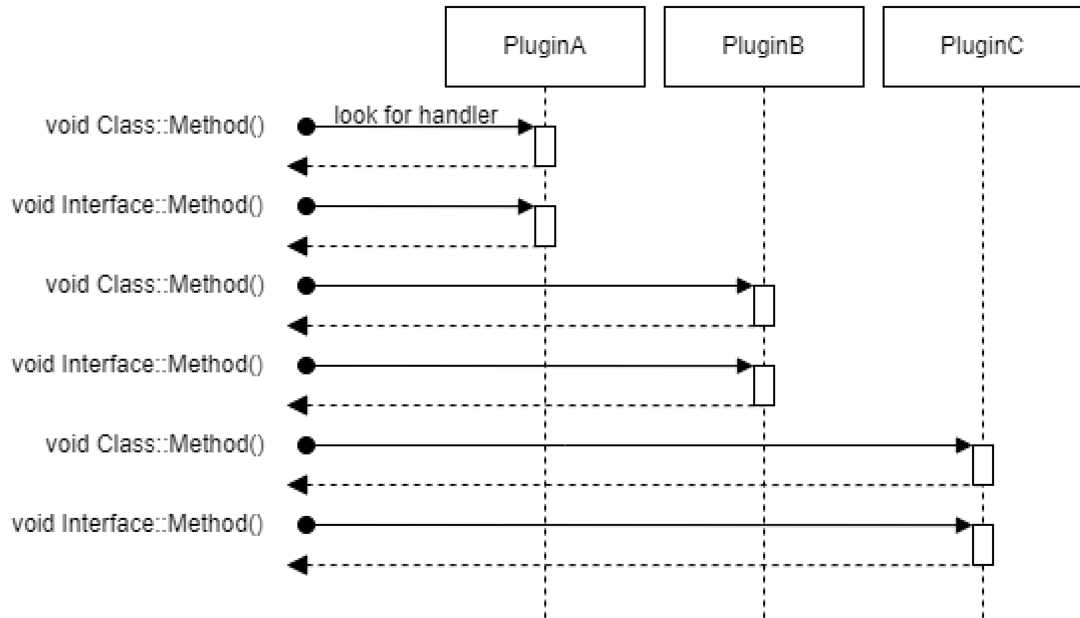


Figure 4.4: Sequence diagram of a search for method handler in a depth-first-search manner.

The correct approach is to search for handlers in a breath-first-search manner. First, iterate through all plugins to look for direct handlers of a method and only then look for overrides [3] of the method. In our example, we would look for a direct handler in the `AnimalPlugin` class, then `DogPlugin` class and finally in the `CatPlugin` class. We would find the correct handler in the last plugin. A generalized example can be seen in Figure 4.5.

There is a concrete application in the scanner. Enumeration is by default handled in the semantic descriptor handler (registered as the first handler), which handles most of the default calls such as `System.Collections.IEnumerator System.Collections.IEnumerable::GetEnumerator()`. This is called in every for-cycle. Database query enumeration overrides this call. The method's signature is `System.Collections.IEnumerator System.Data.Common.DbDataReader::GetEnumerator()`. The handler for this method is registered in the ADO.NET plugin, which is registered only as a second handler. This is a more specific signature, as it is an override, than the one in the semantic descriptor. It

[3] It is an opposite override because we start with overrides and continue with original definitions of the method.

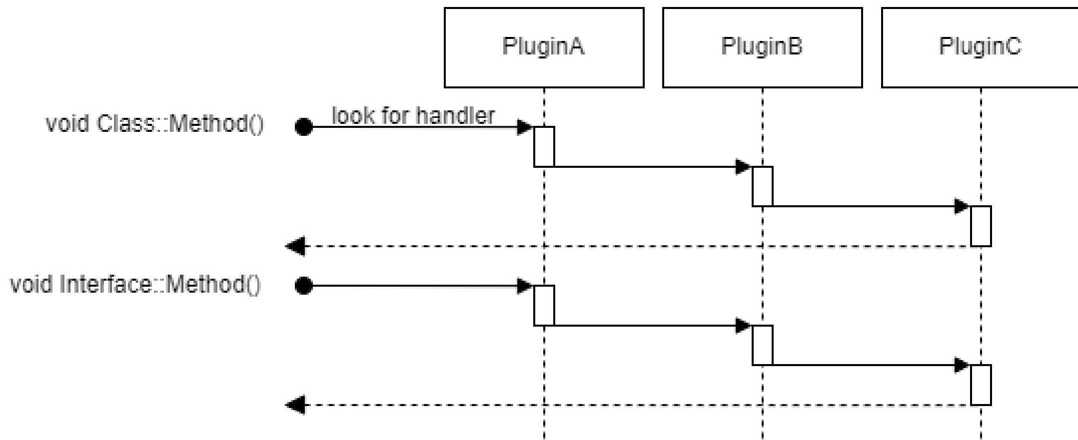


Figure 4.5: Sequence diagram of a search for method handler in a breath-first-search manner.

will never be triggered as this call is always resolved before even considering the second plugin.

It is good to mention that even this approach has its limitations. C# language allows a class to implement multiple interfaces. A class can also implement an interface and inherit from a different class. This means that the order in which method plugin handlers are registered matters. Suppose a class that implements two interfaces contracting a method with the same signature. If there were two different plugins with method handlers for each interface, the one registered first would always get selected. It is easy to see that it is unclear which plugin is correct. Fortunately, this situation should not happen as it is a bad design by nature. Plugin handlers tend to be very specific and always differ, at least in the namespace, as we can see in the example with enumeration.

4.2.4 Multiple Handlers on Single Call

Another issue with handler selection is that multiple handlers are selected and invoked for a single virtual call. Suppose a class hierarchy from the Listing 4.4 again. We have an interface `IAnimal` with a method `doSound` and two classes `Dog` and `Cat` that implement the interface. Suppose two method plugin handlers:

- PluginA handles method `void IAnimal::DoSound()`.
- PluginB handles methods `void Dog::DoSound()` and `void Cat::DoSound()`.

We can see this situation visualised in Figure 4.6. Suppose a code snippet in Listing 4.5. Instances of `Cat` and `Dog` classes are initialized, a random instance is assigned to a variable `a`, and finally, a `DoSound` method is called on this variable. The last call is compiled into a virtual call instruction of a method with signature `System.Void IAnimal::DoSound()`. What handler should be used for this call? This is resolved in two steps.

1. It is searched for all potential method targets of the virtual call. That

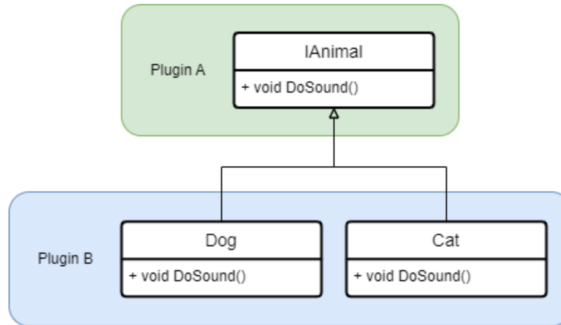


Figure 4.6: Diagram with two classes implementing a common interface and two plugins with handlers for those methods.

means all suitable method overrides must be found using the Rapid-Type analysis (RTA).

2. These method targets are then tested to see if some handler that would handle them exists.

The starting points for the RTA are the type of the virtual call itself and then suitable instantiated types. The RTA searches for overrides in the derivates of the current class.

The type of the virtual call is `IAnimal`. There is a handler in the plugin A. Then, all possible implementations and overrides are analysed. Those are `Cat::DoSound()` and `Dog::DoSound()`. There are handlers in plugin B for them. After this, instantiated types are tested. `Cat` and `Dog`. Their respective handlers from plugin B were already used. This translates into important insights:

1. RTA must consider the type of the virtual call itself.
2. Every suitable handler found along the way of the class hierarchy is used.

```
1 Cat c = new Cat();
2 Dog d = new Dog();
3
4 Animal a = getAnimal(); // returns cat or dog randomly
5
6 a.DoSound();
```

Listing 4.5: Example of a situation where multiple method handlers get selected.

This second point is an invalid behaviour. Only the most specific handler should be used. A real-life example is a virtual call for scalar query execution. Whenever a method `Object Common.DbCommand::ExecuteScalar()` is called on an instance of `SqlClient.SqlCommand`[4] class, two different handlers are invoked. One handler for the method on type `SqlCommand` and a different handler for the method on type `DbCommand`. Note that the class `SqlCommand` is a derivative of the class `DbCommand`. This results in imprecise graphs because only the most specific method should be considered. The solution is only to consider the leaves of the class hierarchy tree in the RTA.

[4]Part of a namespace was ignored for clarity.

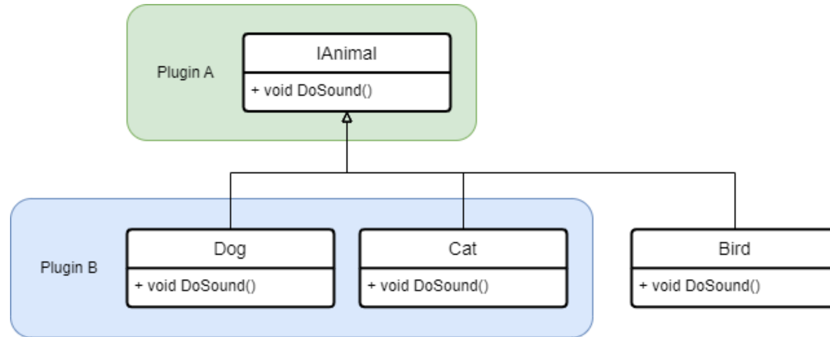


Figure 4.7: Diagram with three classes implementing a common interface and two plugins with handlers for some of those methods.

First, we will prove that the first insight is, in fact, a correct behaviour. Assume only handlers for methods in the leaves of the class hierarchy tree are considered. Assume also a third class `Bird` implementing the `IAnimal` interface. No additional changes in plugins and handlers are made. This situation is in Figure 4.7. The code snippet stays the same as well. The only change is that the `getAnimal()` function now randomly returns any of the three. If the algorithm did not consider the type the call instruction is called on, e.g. `IAnimal` in this case, we would never consider the handler from plugin A. However, that is the most specific handler for the method `Bird::DoSound()`. Since we do not initialize the `Bird` class anywhere, the RTA would never consider its method.

If we look only for handlers of methods in the leaves of the class hierarchy tree [5], we will ensure the correctness of the algorithm while decreasing the load. In theory, this improvement should simplify the analysis quite a bit. We would skip a non-trivial number of handlers. In practice, it may be that this does not happen very often. We usually have handlers only for very specific methods and none for its base classes. We will only search for fewer handlers as there will not be as many method targets. The final amount of handlers will usually be the same. Note that this problem does not occur for non-virtual calls because they only have one implementation to choose from.

4.2.5 Protocol Buffers

The last improvement is the replacement of Protocol Buffers in the C# scanner with better-suited technology. Protocol Buffers are used to serialize output from the C# part of the scanner to be deserialized in the dataflow generator in the Java part. This improvement was mainly implemented by other team members in the Manta company. However, it plays a key role in other improvements and is worth mentioning here to shed light on the reasoning behind them.

The reasons for replacing Protocol Buffers with a different format are:

1. Protocol Buffers are another library that needs to be maintained in both Java and C#.

2. Protocol Buffers are not a text readable format.

[5] In case there is no direct handler for the method in the leaf, we still have to traverse up in the class hierarchy in breath-first-search order to find the handler.

3. Due to the nature of the C# scanner, Protocol buffers do not have a single format definition but duplicated definitions in Java and C#.

4. Due to the illegibility of the format, it is difficult to create integration tests of the entire C# Scanner.

Protocol Buffers bring unreasonable complexity with very little or even no added value compared to some more traditional formats. The XML format was selected as a replacement. Even though Protocol Buffers are supposed to be faster and more lightweight [16] than XML serialization, the disadvantages outweigh the advantages. The key advantages are:

1. XML is human-readable.

2. XML is easy to test.

3. There is an existing and tested deserialization functionality in the Java part of the scanner that can be reused.

4. There is an existing XML schema for our use case.

5. There is a solid XML serialization library in the C# standard library.

The listed advantages of the XML format solve all issues with the Protocol Buffers. Using any other format does not make sense because of its wide use in other scanners and the already existing and tested implementation of the serializer.

The change from Protocol Buffers to XML format enabled further testing automatisation. Until now, integration testing of the C# part of the scanner was very tedious. Other automated tests that would verify details of the analysis output are complex and also often fail to report minor changes that alter the final graph. One way to test this automatically is to write complex conditions on the analysis output structure. Such tests are very hard to read and orient in. Another approach is to diff-check the serialized output. This is possible with Protocol Buffers, but it is nearly impossible to debug failed tests. The XML format brings a good compromise to this problem. The readability of the format makes it easy to understand what may be wrong with the output. Minor changes that alter the final graph are easy to catch, and the tests are easy to update. The expected output is updated or replaced with a new one completely. Catching errors earlier in the process also makes testing the whole analysis process easier. It is a common practice to test even small changes manually, which is very time-consuming. Automatic integration testing of the scanner, or at least its part, also greatly reduces errors caught in manual testing.

5. Design and Implementation

In this chapter, we will explain the concrete design and implementation details for each feature and improvement to the C# scanner. We will explain important details that help to understand the overall solution and how individual changes influence and complete each other, contributing to the common goal of this thesis. This chapter will partially mirror and extend the previous chapter, where we explained the requirements set by stakeholders and internal needs.

5.1 ASP.NET Endpoints

During its scenario, the extractor processor gathers all implementations of the `IEntrypointExtractor` interface via dependency injection. There are two classes implementing said interface. The `AbstractMethodExtractor` class is an abstraction for common code, and the `MainMethodExtractor` class extends it and implements the abstract method `bool IsMatchingMethod(IMethodDescriptor)`. Its implementation returns true if the provided method is the main entry point method of a console application. False otherwise.

To implement another entry point extractor, we had to create a class that implements the interface. We simply extended the `AbstractMethodExtractor` class and implemented `IsMatchingMethod` in such a way that it returns true if the provided method is an action method of some controller. To validate the method, we had to check if the method resides in a controller and if it is a valid action method. Conditions for a class to be a controller are:

1. It is public.
2. It is instantiable.
3. It does not carry `NonController` attribute.
4. It is labelled as a controller, which translates to either of:
   (a) it carries the attribute `Controller` or,
   (b) itself or any of its parent classes carry `Controller` suffix.

The decision algorithm of whether a class is a controller can be seen in Figure 5.1. First, we have to test the existence of the `NonController` attribute. Then, we can check the existence of the `Controller` attribute. If we do not have an answer yet, we can start searching through parent classes until we either reach one that has the `Controller` suffix or it does not have a parent class. Note that we had to check the existence of attribute `NonController` first because it overwrites the `Controller` suffix. Once we verify that the method resides inside of a controller, we can check whether it is an action method. The action method must satisfy the following conditions:

1. It is public.
2. It is not an extension method.

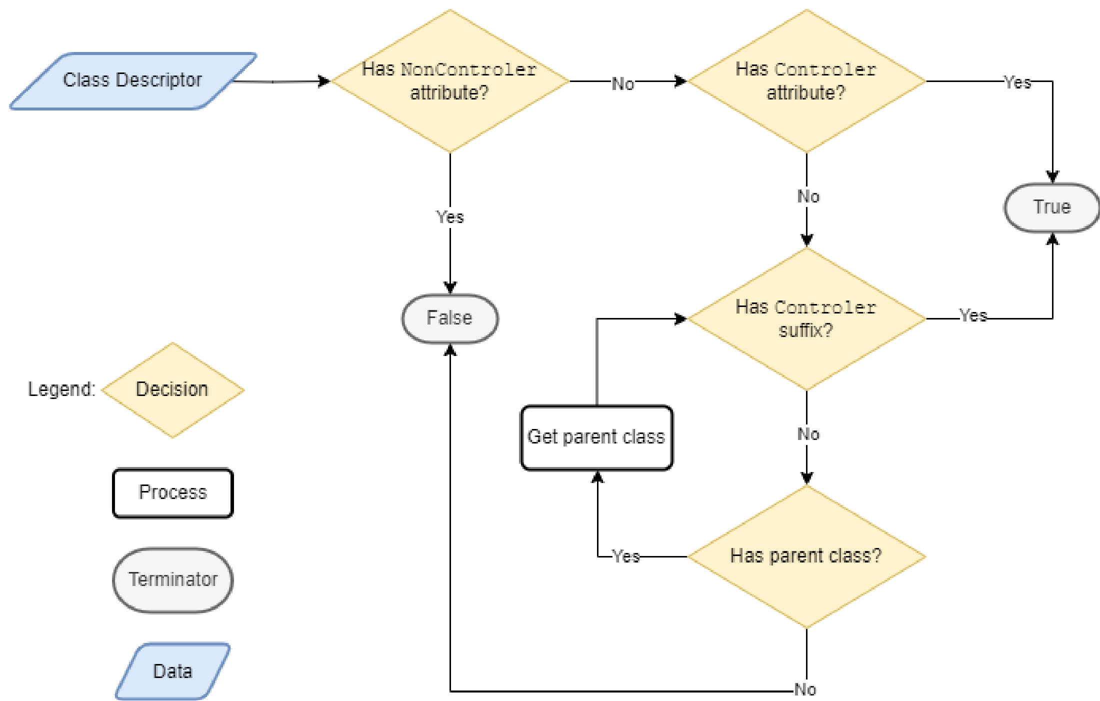


Figure 5.1: Diagram of an algorithm to decide whether a class is a controller.

3. It is not a property getter or setter method.
4. It is not a constructor.
5. It does not have generic parameters.
6. It does not have reference variables (`ref` or `out`).
7. It does not have `NonAction` attribute.

To test the action method, we used a simple logical conjunction over all of the conditions. Note that testing for extension methods can be done with staticness. The infrastructure does not yet support access modifiers or observing type instantiability.

5.1.1 Access Modifiers

Access modifiers are missing throughout the whole infrastructure. We had to define accessibility modifier properties in the `IFieldDescriptor` interface, `IMethodDescriptor` interface and `ITypeDescriptor` interface. The Mono.Cecil library offers API with information about the access modifiers. Its naming convention follows the CIL nomenclature. The C# scanner follows the C# nomenclature even though it could be used for other dotnet languages. Some access modifiers have two keywords (`private protected` or `protected internal`) but logically mean one thing. They are represented by one property. There is one extra access modifier in CIL that does not have a direct variant in the C# language. For completeness, it was implemented into the infrastructure, and its name stayed

C#	CIL
`private`	`NestedPrivate`
`public`	`Public`
`NotPublic`	`NotPublic`
`internal`	`NestedAssembly`
`protected`	`NestedFamily`
`private protected`	`NestedFamilyAndAssembly`
`protected internal`	`NestedFamilyOrAssembly`

Table 5.1: Mapping of access modifiers of types from C# to CIL nomenclature.

C#	CIL
`private`	`Private`
`public`	`Public`
`internal`	`Assembly`
`protected`	`Family`
`private protected`	`FamilyAndAssembly`
`protected internal`	`FamilyOrAssembly`

Table 5.2: Mapping of access modifiers of methods and fields from C# to CIL nomenclature.

unchanged. Mapping from between the nomenclatures for types can be seen in Table 5.1. Mapping for methods and fields is in Table 5.2.

5.1.2 Type Instantiability

Type instantiability (staticness) is another missing piece of the existing infrastructure. There is no single keyword in CIL that would state that a class is static. Instead, static classes are compiled into classes that are abstract and sealed at the same time. That is an otherwise illogical and illegal combination. Note that structures can not be static. We used Mono.Cecil for obtaining this data and incorporated it in our infrastructure.

5.2 Embedded Code Service

Embedded code service is specialised for every programming language. The redesign of the C# scanner to be used as a service can be split into four steps.

1. Decouple extractor and connector runners from scenarios.
2. Add embedded code service orchestrator.
3. Add pin node support.
4. Add insight and outsight.

The first step is to decouple the logic of running the extraction or analysis from the scenario. It was tightly coupled. Because the main analysis and extraction happens inside the C# part of the scanner, the code itself was already

fairly stateless and could be very easily reused for the service without any major changes. However, the analysis orchestration logic, such as the serialization of analysis settings, was implemented inside of the `ExtractorScenario` class and `ConnectorOutputReader` class, respectively. This was separated and decoupled into its own stateless classes `Extractor` and `Connector`. These can be reused from general scenarios for analysing C# applications as well as from other scanners as a part of the service.

As stated in the previous chapter, the source technology must provide a few helper class implementations to use the service. These include:

1. an insighter,
2. an insight,
3. an outsight,
4. an orchestrator,
5. a configuration.

Insighter's only job is to collect events and eventually create an insight. The insighter does not affect the data flow in any way. It records events that require exact data values. For data lineage, we are only interested in the flow of general data, not its value. The insighter closely cooperates with propagation routines designed for it. In the original design, the sole existence of the insighter was to create an immutable insight in the end. Since C# serializes the insight at the end of the analysis, making it virtually immutable, the need for an insighter perishes. To reduce the complexity, we only provided an interface for the insight that every source technology must implement. Thanks to this common interface `Insight`, we can enforce serializability. The same applies to the interface `Outsight`. The orchestrator is also specific for each technology and must implement the common interface `Orchestrator`. This interface has three methods. The method `preOrchestrate` is called before the analysis. Its purpose is to prepare the input (copy input files), set up the configuration and serialize everything for the analysis. The method `postOrchestrate` is called after the analysis. It is supposed to deserialize any output of the analysis that may be required by the source technology. The last method is `getInsight`, which is supposed to deserialise the insight and return it.

5.2.1 Pin Nodes

The third step is adding a pin node support. The dataflow generator already supports pin nodes. It works with intermediate nodes. These are managed inside of the `manta-lib-dataflow-model` library. We had to create a new type for the pin node there and provide its implementation in the scanner. Finally, we had to ensure that all beans that require a reference to the pin node have it.

The implementation inside of the scanner was fairly straightforward and follows already-set standards. The flow from the pins is registered inside of the new `InputPinFlow` and `OutputPinFlow` classes. These flows will be generated inside of propagation routines specialised for the concrete source technology. The

flows are registered inside of the `MethodSummary` class for the same reason as console input and output, as described in the previous chapter. These flows are processed in the results transformer. Two new node separator methods were implemented for them. One creates input pins, and the other outputs pins. The class `ExternallEntryPoint` represents the pin nodes. It is a specialisation of the `EntryPoint` class since it is exactly that.

5.2.2 Insight and Outsight

The last step is the integration of insight and outsight to the scanner. Similarly to pin flows, it will be used in propagation routines specific to the source technology. Both insight and outsight are specialised for the technology. We must determine what technology invoked the service and initialize it accordingly. The source technology is stated inside the configuration. We provided an interface `Inisght` and `Outsight` that both extend `IXmlSerializable`. This way, we enforce serializability and hide it as an implementation detail.

5.3 Unknown Database Column Support

Unknown database queries now use the class `UnknownDataReaderFlow`. This is used for database query enumeration and scalar queries. It implements the common abstract class `ValueFlowData`. The only information it contains is the command itself and its invocation context. It complements the generic class `DataReaderFlow<T>` used for indexable and named columns. This new flow is generated in the scalar execution routine inside of the `ExecuteScalar-PropagationRoutine` class. The other use is in the routines dedicated to the enumeration of a database query.

Unknown data flows are transformed in the results transformer to new nodes represented by the `UnknownTableColumnNode` class. This correctly mirrors the refactored implementation in the dataflow generator in Java. These nodes are assigned to their correct database statements for further processing by the query service in the dataflow generator.

5.3.1 Database Query Mapping

Another issue introduced by the dataflow generator refactoring that was not mentioned before was a new representation of command types. The nomenclature in Java does not match the one from C#. Namely, a `SELECT` command is represented as `GET` command in the generator and is expected by the query service. A simple mapper was a sufficient solution to this problem.

5.4 Method Summary

The complete information about the data flows of a method is gathered inside of the `MethodFlowData` class. This information was moved into the new `MethodSummary` class. It keeps track of all flows that must be propagated further

into callers and callees. Newly, it separates flows that do not have to be propagated in sets. These are used in the results separator to transform the result into a graph more efficiently. Tracked flows are in the following classes:

- `ConsoleFlow` both for input and output,
- `FileStreamFlow` both for input and output,
- `UnknownDataReaderFlow` both for scalar queries and query enumeration,
- `DataReaderFlow<int>` for indexed columns,
- `DataReaderFlow<string>` for named columns,
- `InputPinFlow`,
- `OutputPinFlow`.

Any propagation routine that generated these flows had to be modified. Output flows are marked in the summary and not propagated further. The input flows are propagated. Even though all propagation routines that generate these flows are encapsulated in a common abstraction, this change required fine testing.

5.4.1 Flow Locations

The results transformer gathers information about the flow's origin. This information is available at the flow creation, but the implementation incorrectly registers the method's zeroeth instruction. For example, when processing the `call` instruction of the `System.Void System.Console::Write(?)` method, its origin is noted as the zeroeth instruction of said method. The point where the `call` instruction was called should be registered instead. To change this was fairly simple. All places where flows are created must be checked and corrected. However, it also requires fine testing that exposes additional implementation mistakes, such as wrong handler selection, described in the following section.

5.5 Method Handler Selection

Handler selection was reworked in two ways:

1. search for the correct handler in a breath-first search manner,
2. select only the most specific handler for each method call.

In Figure 5.2, we can see a diagram of how a search for a method handler is now performed. Input is a method of some type. We then proceed to traverse the class hierarchy in a breadth-first search order. Checking the base class first and then the implemented interfaces. We construct a method signature with the given type and check for handlers in all registered plugins. If no suitable handler is found before trying all classes in the hierarchy, there is no handler for the method. Visualisation of the order in which types are tried can be seen in Figure 5.3. Suppose we have a method from class `B` and three handlers registered in order from one to three. Numbers next to the class represent the order in which

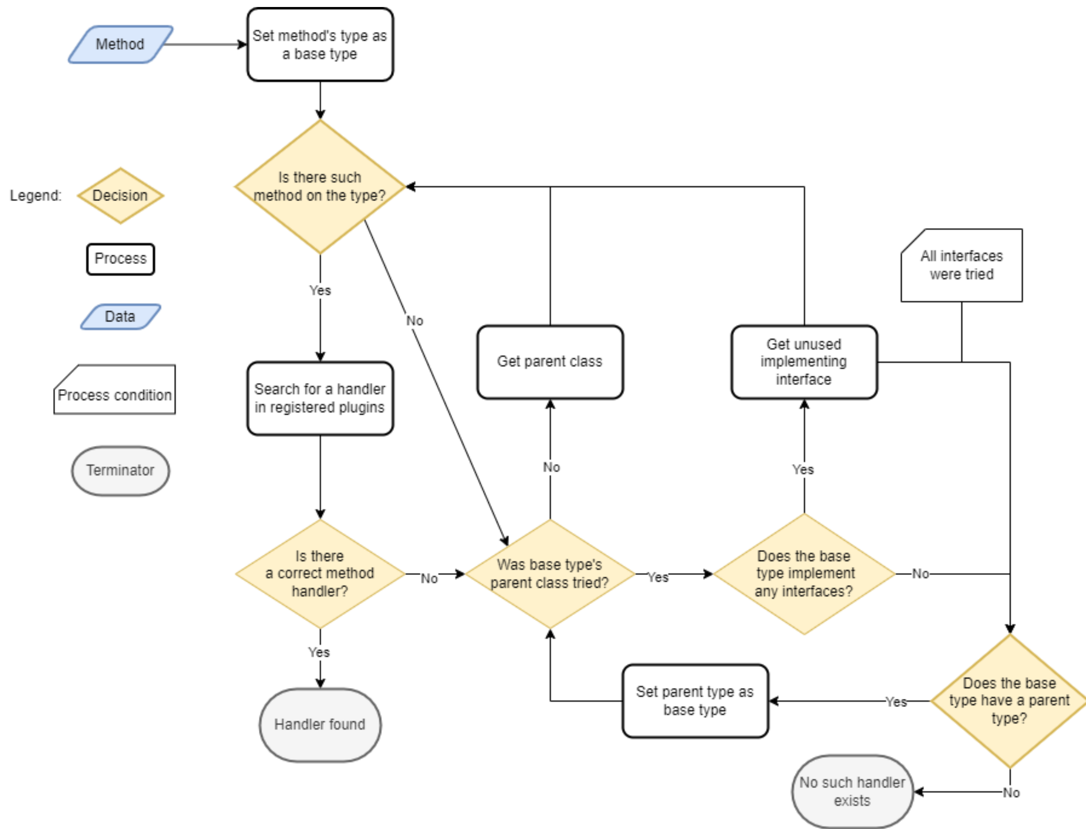


Figure 5.2: Handler selection algorithm diagram.

they will be tested. Starting with the method in class `B` from the first handler to the third. If no handler is suitable, the parent class `A` of the current class `B` is tried. Then, any interface that the class `B` implements. After that, we try the parent class of class `A` (if it exists) and its interface implementations (if any). This algorithm was implemented in the `MethodHandlerView` class inside of the `TryHandleImpl` method where the handler selection takes place.

Possible method targets are selected on the `callvirt` and other instructions invoking virtual methods before searching for the handler. This search must only pick one method target as described in the previous Section 4.2.4. This algorithm was implemented in the class `DynamicContextSensitiveCallGraph` in the method `GetPossibleTargets`. It gathers all possible runtime instances of the type the method is called on using RTA. For each instance, it considers exactly one method target in the leaf of the class hierarchy tree where classes implement the method. The handler search algorithm described above is applied for each target method found.

5.5.1 Support Database Query Enumeration

A direct result of the introduction of the method summary, fix of the flow origin locations and the subsequent method handler search algorithm was an invalid database query enumeration. It was briefly described in the Section 4.2.4. Query enumeration in C# used to be done by enumerating the result via the `get_Current()` method. This method is generally handled by the `SemanticDes-`

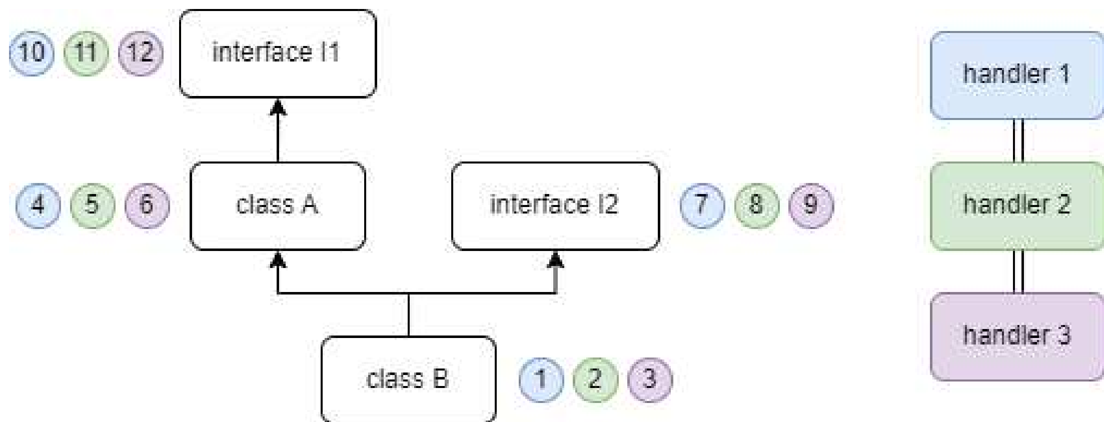


Figure 5.3: Enumeration of order in which classes are tested for handler existence. Starting in class `B` with handlers registered in order from 1 to 3. Numbers in circles represent the resulting order.

`criptionsHandler`. Enumeration of the database was previously resolved by gathering all callers of the `get_Current()` method (regardless of the type this method is implemented in) and later checking if there is any `CommandFlow` instance originating from the same program point. The new approach tracks the `GetEnumerator()` method on the `DbDataReader` class. This particular method of this type or any of its overrides is always called before query enumeration. Thanks to this trick, we are able to track such calls and not mistake them for any other enumeration without having to compare flow origins.

5.6 Remove Protocol Buffers

Protocol Buffers were replaced with XML serialization. Serialization and deserialization on the Java part were already implemented for other scanners. Thanks to already existing *XML Schema Definition* describing the model used in Java, we only had to create the same model in C#. A tool called `xsd`, which is a part of Visual Studio 2022, can generate such models for us. That makes any future changes easier to process because any changes in Java still must be mirrored in C#. The mapper was implemented in the `IntermediateModel.Xml` namespace inside of the `Connector` project. Protocol Buffers were also used for the serialization of the configuration generated in the extractor. The remaining configuration files were already serialized to JSON, and introducing another XML schema would increase the maintenance necessary. The only requirement for configuration files is to make them readable. They are only accessed from the C# code; thus, using JSON was a valid option in this case.

5.7 Testing

The introduction of XML as a means to perform analysis opened doors for more semi-automatic testing of the whole scanner. The scanner has a decent amount of automated tests that test part of the dataflow scenario. These tests perform

the analysis and then compare the analysis result with expected results that are saved in an XML. This XML is custom for testing purposes and does not follow any schema. In fact, these XMLs were generated by running the analysis and dumping the results into the XML. This means that a lot of accidental mistakes were generated this way and are part of the test. This resulted in passing tests even though the implementation was incorrect. A lot of manual testing was required to check that every known use case was functional, even after minor changes. Manual testing included generating the graphs completely and manually observing them in the Manta graph visualizer. Thanks to the serialization of the analysis output into an XML, we were able to automate part of the process. We can run the complete C# analysis, including the extraction, and compare the generated XMLs automatically with their functional versions. XML allows us to observe the differences directly, which is not the case with Protocol Buffers. This way, the complete C# part of the scanner can be automatically tested using simple scripts. These scripts can be found in the `cli-tests` folder inside of the C# solution. The `run_and_check_serialized_output.bat` script performs five steps to generate the output XML.

1. Clean any temporary files from previous failed runs.
2. Build and move input program to expected locations.
3. Run extractor with correctly set configuration.
4. Run connector with correctly set configuration.
5. Clean the input and any temporary files.

Note that manual testing is still required to test the dataflow generator as a part of the scanner run.

5.8 Other

In this section, we will describe other minor changes made to the scanner.

Enum Differentiation

One of the existing invalid infrastructure implementations was enum identification. The library Mono.Cecil was correctly used to determine whether a type is an enum. Or, in other words, extends the `System.Enum` type. This was only done for classes and not structures, where it was hardcoded that the type could not be an enum. It should be the other way around. A class can never be an enum, but a structure can.

Try-catch Resolving

Another infrastructure deficiency was the invalid handling of exceptions. Previously, the code simulator checked whether it was inside of a catch clause (to handle it correctly) using the Mono.Cecil library. It used an API to get exception handlers of the method. This call does not differentiate between `catch` and

`finally` clauses. This resulted in endless stack growth by adding an exception object onto the stack on entering the `finally` clause. We fixed the filtering of the exception clauses by their handler type.

Configurable Logging

Logging used to be set strictly to one level of detail. This has many issues. It logs a lot of data, which may not be desired on every analysis run. Some logged data may be sensitive to the customer, even though that should not happen intentionally. Logging level configuration was added to the configuration, which is set on warning level by default. Logging into a file was also turned off on non-release versions of Manta to avoid dumping sensitive information and creating a vulnerability.

6. User Documentation

In this chapter, we will explain how to execute the analysis of a C# program via the Manta Admin UI web interface. Then, we will explain the lineage in the resulting graph. This chapter is not intended as a manual for the Manta platform and is by no means an exhaustive guide to the Manta platform. Its content is limited to using the C# scanner and only briefly explains some features of the Manta platform that allow us to observe the results of an analysis. For a detailed manual, see the official Manta documentation.

Access to the Manta installer and license is available only to Manta customers and Manta employees. This is a major obstacle for the reader. For that reason, this documentation is accompanied by images of the platform to give readers without access to the platform and the license key at least an idea about the platform.

6.1 Analysis Execution

If we want to run an analysis of some technology in Manta, we have to set up the scanner first.

Setting Up the Analysis

Once you log in to the *Manta Admin UI*, click the *Add collection* button on the main page and select C# under *Programming languages*. You will see a connection setup page. Figure 6.1 shows this page with marked settings. The settings (4) - (7) are shared across all C# scanners unless redefined.

(1) The first required setting is the name of the connection that has to be unique for a given technology.

(2) The second required setting defines a directory where all input, output and temporary data will be stored throughout the analysis.

(3) In the third box, we can set up custom settings for the extractor in the form of JSON. This is not required because the default settings are generated on each analysis run.

(4) This toggle ensures that even if two files use different letter cases, their names will be unified as all lowercase text and connected in the graph.

(5) The fifth setting defines where the input for the analysis is placed. Its default location is set to the input directory under `csharp` inside a directory whose name is defined in setting (2).

(6) Second to last setting defines the location of a temporary directory.

(7) The last setting defines the directory where the extractor output is placed. Its default value is the same as setting (6).

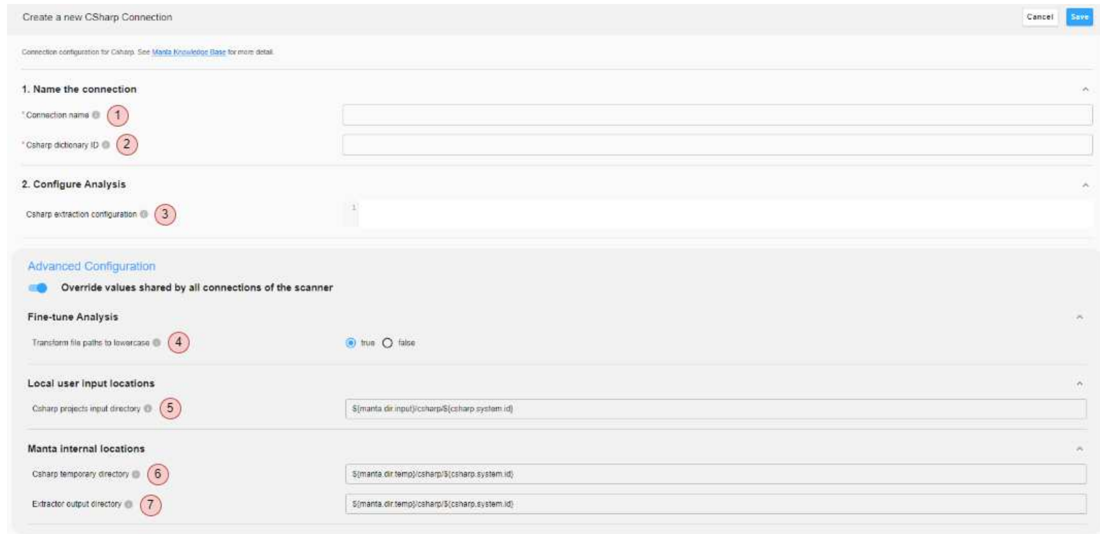


Figure 6.1: Page with C# connection configuration setup.

After creating the connection, you can go to the *Process Manager* window and select *New Workflow*. We should see a workflow configuration setup like in Figure 6.2.

1. This section shows all available scenarios of connected scanners in section (1a). Scenarios can be dragged and dropped into (1b) to apply them in this workflow.

2. The only required textbox is for the workflow's name.

3. This textbox is for a detailed workflow description.

4. Revision type defines whether a revision's major or minor version is created. All metadata and its lineage are grouped in a revision, which is their snapshot in time. Major revisions are snapshots of all selected connections at the same point in time. Minor revisions build upon the last major revision. They add (not update) new lineage and metadata to the revision. This may lead to inaccurate lineage.

5. Advanced settings allow us to create more complex workflows. If disabled, essential scenarios are automatically injected into the workflow and displayed in the designer (1b). When enabled, several extra scenarios are displayed in the menu (1a) and none are automatically injected into the workflow.

6. This textbox states the limit for parallel processes running simultaneously. However, this setting is irrelevant for the C# scanner since it can not run in parallel.

7. In the last section, we can define environment variables with additional runtime settings, which can be seen in Table 6.1.

Value	Description
`csharp.input.exclusions`	List of namespaces to exclude from the analysis. Namespaces can be written as a regular expression or the substring the namespace starts with e.g. *System, Microsoft.*
`csharp.connector.plugins`	Names of plugins that are enabled e.g. *Main, InputOutput, AdoNet.*
`csharp.logging.dir`	Directory where logs should be saved e.g. *./logs.*
`csharp.logging.level`	Level of logging e.g. *Info.*

Table 6.1: Table of available environment variables and their descriptions.

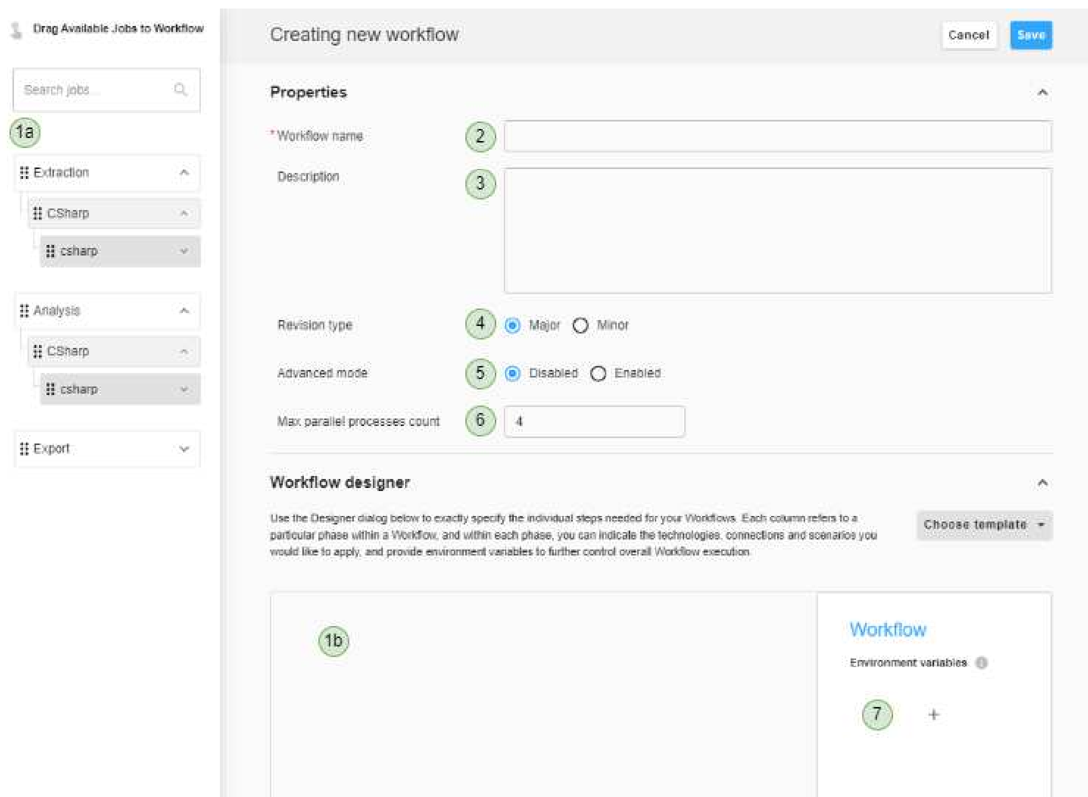


Figure 6.2: Process manager page with workflow configuration setup.

Running the Analysis

To run the analysis, you must prepare the input in a zip folder named *input*. The archive should contain all compiled sources to analyze. Select your workflow in the process manager window, and click the *Execute this Workflow* button at the top left-hand corner. Upload your zip folder with the input and start the analysis. The workflow run should be displayed in the *Workflow queue*. We should see it in the *Workflow history* after it is finished, as displayed in Figure 6.3. We can see detailed information about each scenario grouped by the technology and the connection of the scenario. The information includes return

Figure 6.3: Displayed information about a finished workflow run.

code, approximate execution time and logs. The workflow and scenarios can obtain several states throughout its execution, such as *RUNNING*, *SUCCESS*, *FAILED* or *TERMINATED*.

6.2 Examining the Analysis Result

To see the analysis result graph, we have to go to the *Manta Flow Viewer*. This page can be seen in Figure 6.4. On the left side, we can see a revision selector. Under the selector, there is a set of lineage nodes that can be selected for visualisation. First, select a node to visualize in the area marked with (1) in Figure 6.4. If you select any node in the hierarchy, all subnodes are automatically included. You must confirm the selection by clicking the plus button marked with (2). It should appear in the *Selected for lineage* area marked with (3). Press the *Visualize* button below to see the lineage graph between selected nodes.

Figure 6.4: Manta Flow Viewer with selected revision of a C# code analysis.

In Figure 6.5, we can see a graph generated by the analysis of the source code in Listing 6.1. The graph is generated at the highest level of detail with indirect edges visible. Each node in the graph has a specific colour set based on the node's technology. Database and file nodes are blue. Nodes from the C# application are beige. Each node is made of sub-nodes representing the hierarchy for corresponding technology. Each node contains a black sub-node marking the selected node in the flow viewer. For example, the blue node on the left-hand side is made of six nodes in total. The nodes, in order from the outermost, represent the database technology, concrete database, database schema, table and finally, the column. The C# nodes follow the same hierarchy adjusted to C#. The technology, connection ID (specified during the configuration), namespace, class, method and finally, concrete data flows.

We can see two types of edges: blue dashed and black full lines. Black edges represent direct data flow. For example, we can see that there flows data from the parameter of the `Main(String[])` method to the first parameter[1] of both SQL statements. In the code, we can review this on lines 12 and 23, where the parameter value is added to the command. The value comes from the argument of the `Test(int)` method. This method is called on line 31, with the corresponding parameter value being the argument of the `Main` method. The blue dashed edges represent indirect data flow. Indirect data flow affects the flow, but the data is somehow processed and not used directly. From the graph, we can see that the first parameter of the SQL statement indirectly affects the columns that are processed further. We can check this in the code as well. Both queries have a `WHERE` clause that is parametrized. Hence, the first parameter affects the resulting columns.

```
1  namespace cli_test_basic;
2
3  class Program
4  {
5      private const string FileName = "output.txt";
6      public static string Test(int type)
7      {
8          using (var connection = new SqlConnection("Data Source =
               some.datasource.com; Initial Catalog =
               BIReportServer; User ID =
               someUserId;Password=somePassword"))
9          {
10             connection.Open();
11             var cmd = new SqlCommand("SELECT * FROM dbo.Users
                   WHERE UserType=@type", connection);
12             cmd.Parameters.AddWithValue("@type", type);
13             var reader = cmd.ExecuteReader();
14             while (reader.Read())
15             {
16                 var name = reader["UserName"].ToString();
17                 var id = reader["UserId"].ToString();
18                 Console.WriteLine($"Found user {name} with id
                       {id}.");
19             }
20             reader.Close();
21
```

[1] To see that it is a parameter, click on the node and read its detailed description.

```
22            var cmdCount = new SqlCommand("SELECT COUNT(*) FROM
                dbo.Users WHERE UserType=@type", connection);
23            cmdCount.Parameters.AddWithValue("@type", type);
24
25            return cmdCount.ExecuteScalar().ToString();
26        }
27    }
28
29    public static void Main(string[] args)
30    {
31        File.WriteAllText(FileName, Test(int.Parse(args[0])));
32    }
33 }
```

Listing 6.1: C# program that executes a database query and writes columns *UserName* and *UserId* to standard output. Then executes a scalar query whose output is written into a file called *output.txt*.

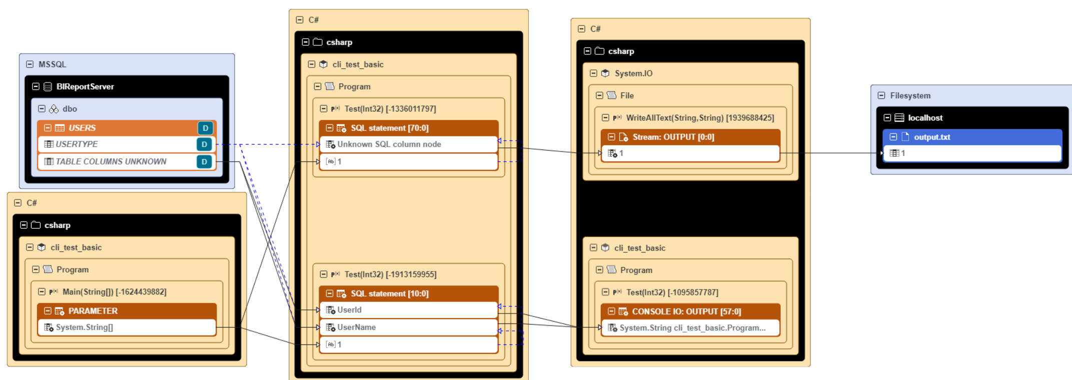


Figure 6.5: Data lineage graph of the C# program in Listing 6.1 a C# code analysis.

7. Evaluation

In this chapter, we will look at the results of the changes described in this thesis. We will showcase examples of code that was impossible to analyse before, or its analysis resulted in inaccurate lineage. The graphs presented in this chapter were all generated by the Manta software. The code will be provided for some use cases where it is important to display nuances leading to different graphs. In other cases, the code logic will be described in text or can be found in the attachment to this thesis. We will also verify that there was no regression in the already functional use cases and propose possible future work on the C# scanner.

7.1 New Supported Use Cases

Firstly, we will introduce new supported use cases. In general, all new features are covered by unit tests as well as end-to-end manual tests, which are the only ones that test the integration of the C# scanner with the rest of the platform.

ASP.NET Endpoint Extraction

ASP.NET applications are now supported. Extraction of endpoints is well covered with unit tests as well as integration tests. Integration tests focus on the extraction of entry points rather than the data flow analysis. In Figure 7.1, we can see a graph of a simple ASP.NET application with one controller called `WeatherForecastController` that contains all possible types of actions (`GET`, `DELETE`, `HEAD`, `OPTIONS`, `PATCH`, `POST`, `PUT`). This graph contains little to no lineage. There is a little bit of lineage in `POST` and `PUT`, where we return the data from the argument. It is important to mention that this graph is a result of a single scenario run where all entry points (including the `Main` method) were found and then analysed separately and merged into a single graph.

Unknown Column Support

Another major new use case is the support of unknown columns. Figures 7.2 and 7.3 show graphs with an unknown column. In Figure 7.2, we have a graph of an application that executes a simple `SELECT * FROM dbo.Users WHERE UserType =@type` query. The Figure shows that the database column `UserType` indirectly affects the result and that we are querying from some unknown column because of the `*`. More importantly, we query this result with the `Main` method argument. Because this piece of data is not statically known, we do not know exactly what the index or name is. Compare this with the graph in Figure 7.4 where the index of the column being queried is 56.

Figure 7.3 contains unknown columns that were created by scalar queries. In this case, there was the `SELECT MAX(*) FROM dbo.Users WHERE UserType= @type` query with the parameter being substituted with an input from the console. We can see that the above-mentioned graphs are very similar, as expected from the analysis.

Embedded Code Service

The last major feature improvement was enabling the use of C# scanner as an embedded code service. This feature is fully implemented and finished, including insight, outsight and pin nodes, but its full capability is limited by source technologies using the service. Currently, no technology is actively using C# scanner as a service. All of its individual parts were tested, and its integral part does not affect the run of classic run scenarios in any way, thanks to the good design choices.

7.2 Regression Testing

All existing functionality is covered by unit tests, end-to-end manual tests and newly added automated integration tests. Modification of the existing algorithm is also covered by new test cases that were missing. By the nature of some of the tests, they had to be updated to comply with the new design. We will show graphs declaring a functional solution for selected problems.

Database Enumeration

In Figure 7.5, we can see two graphs of the same C# application. This application iterates over all arguments of the `Main` method and prints them into the console. Then, it executes the database query of `SELECT UserName FROM dbo.Users` and outputs its first result into the console. We can see the code in Listing 7.1. The top graph shows a case where the enumeration call is handled by two method handlers. One correctly propagates data from the arguments, but the second is mistaking this enumeration with a database enumeration. This is resolved in the bottom graph with the correct handler selection algorithm.

```
1  var cmd = new SqlCommand("SELECT UserName FROM dbo.Users");
2
3  using (var connection = new SqlConnection("Data Source =
       some.datasource.com; Initial Catalog = BIReportServer; User
       ID = someUserId;Password=somePassword"))
4  {
5      connection.Open();
6
7      // This enum should not be connected to the db
8      foreach (var arg in args)
9      {
10         Console.WriteLine(arg);
11     }
12
13     cmd.Connection = connection;
14     Console.WriteLine(cmd.ExecuteReader()[0].ToString());
15 }
```

Listing 7.1: C# application with argument enumeration and basic database query.

Others

Other changes implemented as a part of our solution improved the overall lineage precision while preserving correctness. We can see this in Figure 7.6 with two graphs of the same application. The application executes three database queries and writes the first column into the console. The code can be seen in Listing 7.2. The top graph was generated before our solution was implemented. Note that it contains a lot of unnecessary nodes that do not exist in the source code. It should contain at least three different database statements, but they are merged into one. The bottom graph correctly shows only nodes that are in the source code, and it also contains five database statements. There are five statements because the second statement (line 7 in Listing 7.2) is created by string concatenation of three parts, which is handled as three possible strings by the analysis.

```
1  private static string _contacts = "SELECT * FROM
       dbo.UserContactInfo";
2  static void Main(string[] args)
3  {
4      List<SqlCommand> commands = new List<SqlCommand>()
5  {
6      new SqlCommand("SELECT * FROM dbo.Users"),
7      new SqlCommand($"SELECT {args[0]} FROM dbo.Users"),
8      new SqlCommand(_contacts)
9  };
10
11     using (var connection = new SqlConnection("Data Source =
           some.datasource.com; Initial Catalog = BIReportServer;
           User ID = someUserId;Password=somePassword"))
12     {
13         connection.Open();
14
15         foreach (var command in commands)
16         {
17             command.Connection = connection;
18             using (var reader = command.ExecuteReader())
19             {
20                 while (reader.Read())
21                 {
22                     Console.WriteLine(reader[0]);
23                 }
24                 reader.Close();
25             }
26         }
27     }
28 }
```

Listing 7.2: C# application that executes three database commands and writes the first column to the console.

7.3 Future Work

The scanner can already analyse non-trivial applications, but the ability to analyse production code varies from stakeholder to stakeholder. There are still a lot of frameworks that the scanner could support, but whether they should be sup-

ported depends mainly on the current stakeholder needs. While support of new frameworks may be very beneficial and not the most difficult task, the codebase is covered by a few unfinished or imprecise algorithms that may be even easier to implement and provide a huge benefit in the long term. Such improvements include:

- Analysis of backward jumps.
- Incomplete implementation of routines in the Entity Framework handler.
- Named parameter node support.

Backwards jumps in the CIL are not properly recognized at some edge cases during the analysis, which may result in slightly imprecise graphs that are missing some lineage. While the support for the Entity Framework is working well and provides the most important lineage, it can provide more information with indirect data flows that are currently missing. Finally, all parameters are now represented with their index in the query, even though we can statically determine the parameter's name in the query. The dataflow generator already supports named parameters, but the implementation in the scanner is missing.

Another future work not directly connected to the C# scanner is using the C# scanner as an embedded code service from other technologies. With the scanner being fully equipped to support this scenario, it should easily broaden the use spectrum. To use the service from other technology, there is going to be a need to implement some specific method extractors and handlers for the technology. These changes should be of a smaller scale and fit the capability of the scanner perfectly.

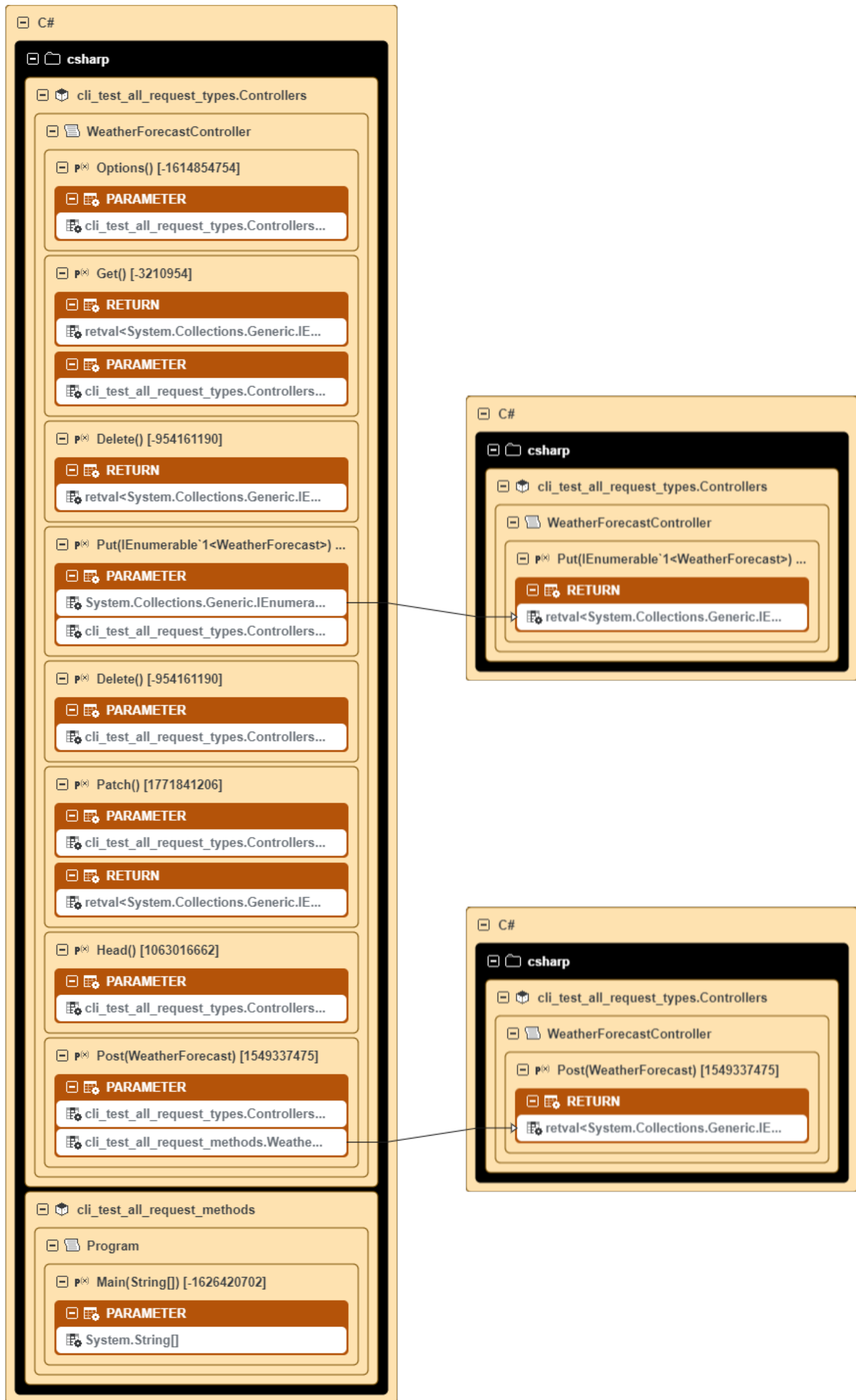


Figure 7.1: Graph of a simple ASP.NET application with all possible controller actions.

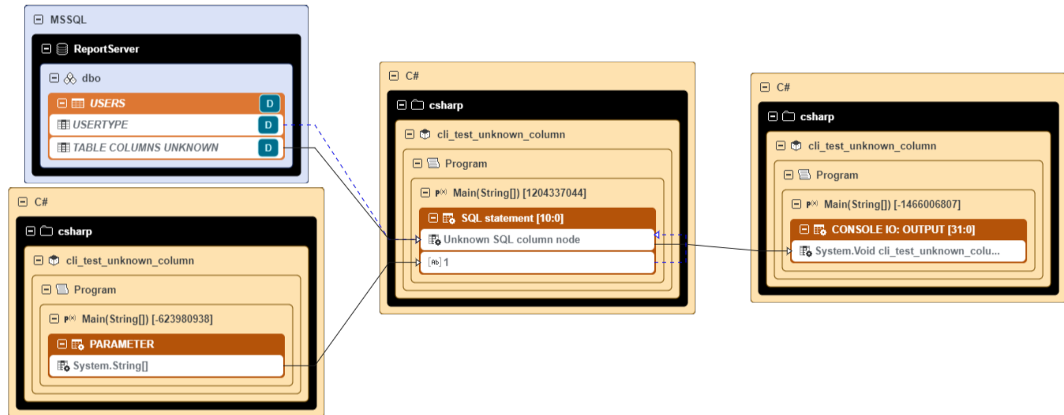


Figure 7.2: Graph of C# program querying an unknown column in the database.

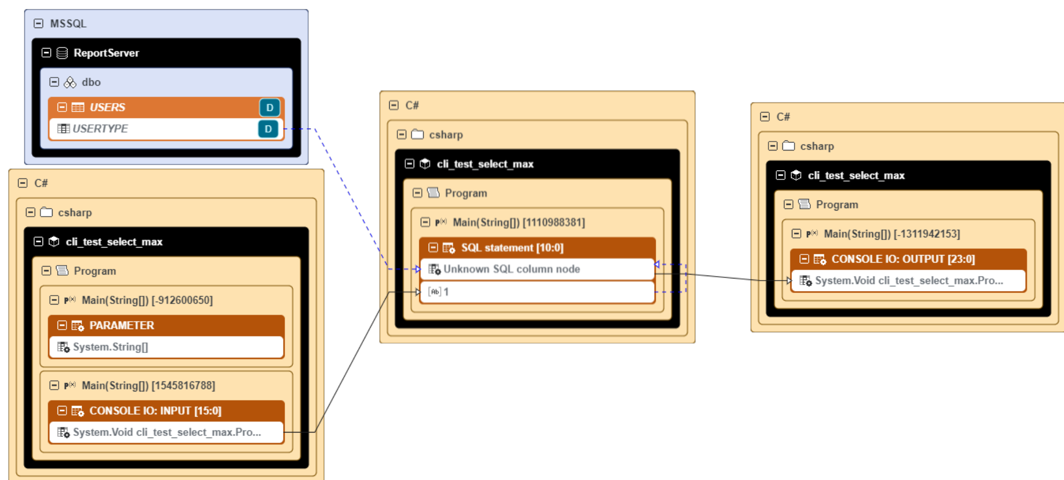


Figure 7.3: Graph of C# program executing a scalar query (`MAX`).

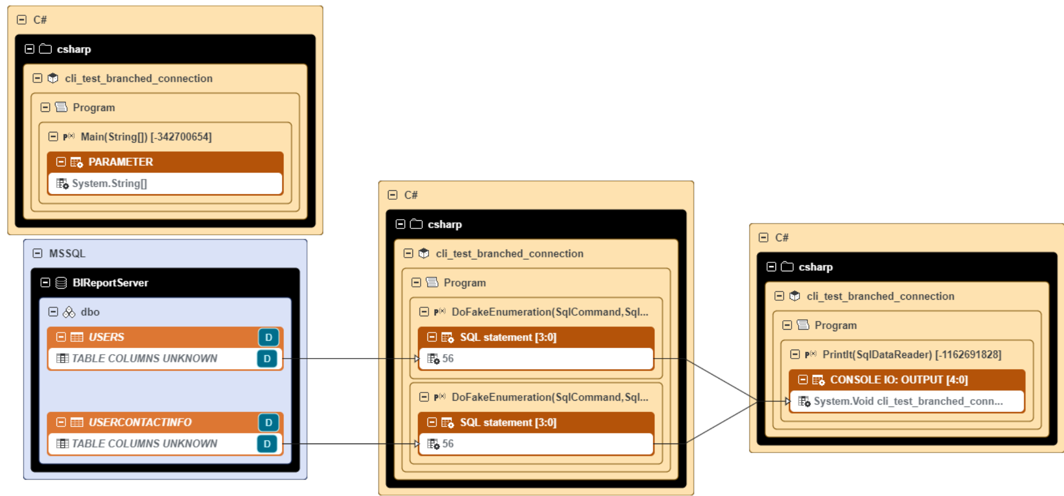
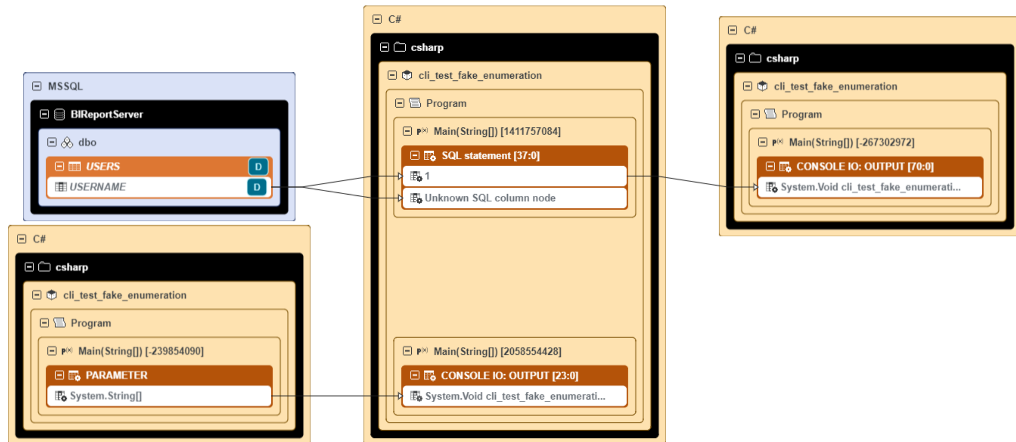


Figure 7.4: Graph of C# program that decides what table it will query in runtime via branch.



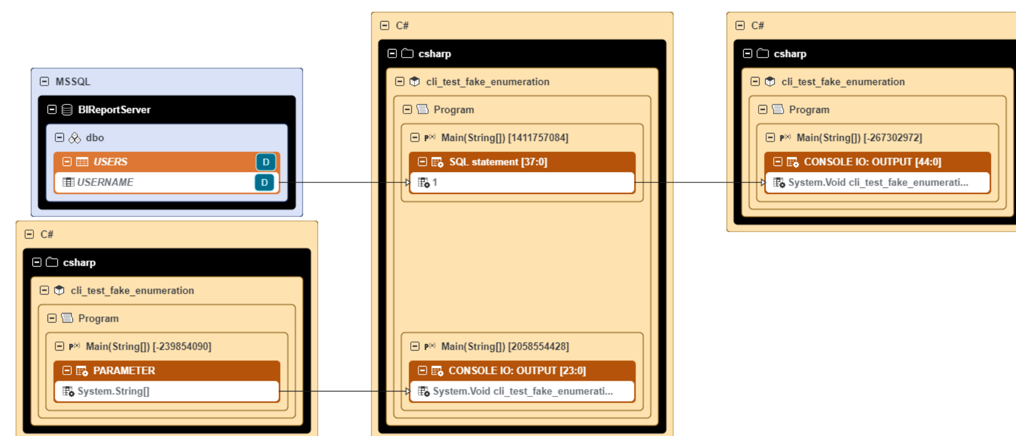


Figure 7.5: Two graphs of the same C# program that enumerates arguments and outputs them into the console and executes database query and writes its first column into the console. The top graph shows an invalid unknown column because of the enumeration. The bottom graph correctly does not contain an unknown column.

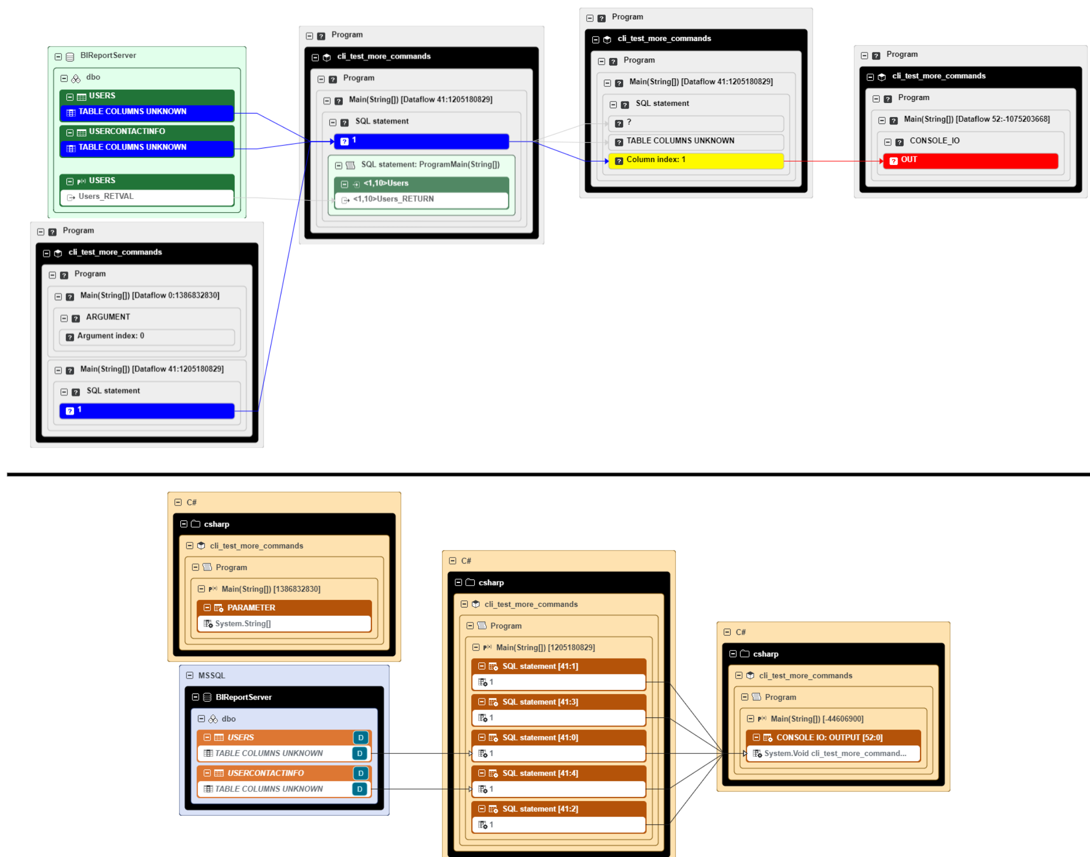


Figure 7.6: Two graphs of the same C# program that executes three database commands and writes the first column into the console.

8. Conclusion

In this thesis, we have successfully improved the lineage precision and correctness of the already existing C# scanner, which is a part of the Manta platform software.

The scanner supports the analysis of ASP.NET web applications, including its web endpoints. It can be used as an independent embedded code service for analysing code snippets for external scanner technologies according to set standards across the Manta platform. With that being said, to be used as a service, a few details specific to the technology using it may still have to be implemented. However, the complete infrastructure for that is already available. Our solution introduced numerous concepts on various levels of abstraction, from low-level CIL code analysis to high-level algorithms that increase the efficiency of the scanner and lineage precision by modifying the fundamental parts of the analysis algorithm. Last but not least, we have reduced the technical debt of the scanner in relation to the rest of the platform and semi-automated the testing process of the scanner.

In the previous chapter, we have shown that all vital parts are fully functional, and all existing scenarios are supported. Scenarios that provided incomplete or inaccurate lineage were improved, and new scenarios were well-tested.

The implementation design follows a set standard held across the Manta platform and should be easily extendable with little knowledge of the scanner insights. This includes the extensions needed for the use of the embedded code service.

Bibliography

[1] Dalibor Zeman. Extending Data Lineage Analysis Towards .NET Frameworks, Master thesis, Charles University, 2021.

[2] Cloud-migration opportunity: Business value grows, but missteps abound. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/cloud-migration-opportunity-business-value-grows-but-missteps-abound.

[3] Meta. https://about.meta.com.

[4] 1.2 billion euro fine for Facebook as a result of EDPB binding decisionT. https://edpb.europa.eu/news/news/2023/12-billion-euro-fine-facebook-result-edpb-binding-decision_en.

[5] Wikipedia page about Bus factor. https://en.wikipedia.org/wiki/Bus_factor.

[6] Impreva - Data Lineage. https://www.imperva.com/learn/data-security/data-lineage.

[7] How Apple catches leakers: From color changes to comma placement. https://9to5mac.com/2023/05/11/how-apple-catches-leakers.

[8] Databricks Unity Catalog product page. https://www.databricks.com/product/unity-catalog.

[9] Manta, an IBM Company. https://manta.io/.

[10] Definition of a controller action. https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs.

[11] Azure Data Factory documentation. https://azure.microsoft.com/en-us/products/data-factory.

[12] ASP.NET framework documentation. https://dotnet.microsoft.com/en-us/apps/aspnet.

[13] ECMA-335 standard of the Common Language Infrastructure. https://ecma-international.org/publications-and-standards/standards/ecma-335.

[14] Wikipedia page about dependency injection. https://en.wikipedia.org/wiki/Dependency_injection.

[15] Wikipedia page about Extract, Transform, Load (ETL). https://en.wikipedia.org/wiki/Extract,_transform,_load.

[16] Wikipedia page about Protocol Buffers. https://en.wikipedia.org/wiki/Protocol_Buffers.

[17] SQL Server Integration Services documentation. https://learn.microsoft.com/en-us/sql/integration-services/sql-server-integration-services.

[18] Wikipedia page about dependency XML Schema. https://en.wikipedia.org/wiki/XML_Schema.

[19] Profinit. https://profinit.eu/.

[20] IBM. https://www.ibm.com.

[21] IBM acuquires Manta Software Inc. https://newsroom.ibm.com/IBM-acquires-Manta-Software-Inc-to-complement-data-and-AI-governance-capabilities.

[22] Supported technologies by Manta. https://manta.io/supported-scanners.

[23] Data Lineage Analysis of C# Programs for Manta, Software project, Charles University. https://www.ksi.mff.cuni.cz/sw-projekty/zadani/mantacs.pdf.

[24] Mono Cecil library. https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil.

[25] C# Access Modifiers documentation. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers.

[26] Sharplab. https://sharplab.io.

[27] Rapid Type Analysis. https://courses.cs.washington.edu/courses/cse501/04wi/papers/bacon-oopsla96.pdf.

[28] C# Main method documentation. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line.

[29] Microsoft documentation containing a comparison of minimal API and controller actions. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/apis.

[30] StackOverflow question about the difference between CIL of anonymous and local function. https://stackoverflow.com/questions/45337983/why-local-functions-generate-il-different-from-anonymous-methods-and-lambda-expr.

[31] Microsoft documentation of ASP.NET controllers. https://learn.microsoft.com/en-us/aspnet/core/web-api.

[32] Microsoft documentation of controller action requirements. https://learn.microsoft.com/en-us/aspnet/core/web-api.

[33] Microsoft tutorial for implementing ASP.NET controller actions. https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/creating-an-action-cs.

[34] Jurčo Michal. Data Lineage Analysis Service for Embedded Code, Master thesis, Charles University, 2023.

[35] Azure Data Factory custom activity documentation. https://learn.microsoft.com/en-us/azure/data-factory/transform-data-using-custom-activity.

[36] Documentation of SSIS tasks. https://learn.microsoft.com/en-us/sql/integration-services/control-flow/integration-services-tasks.

[37] Documentation for developing custom objects for integration services. https://learn.microsoft.com/en-us/sql/integration-services/extending-packages-custom-objects/developing-custom-objects-for-integration-services.

[38] Documentation for developing custom tasks. https://learn.microsoft.com/en-us/sql/integration-services/extending-packages-custom-objects/task/developing-a-custom-task.

[39] Documentation for developing script tasks. https://learn.microsoft.com/en-us/sql/integration-services/control-flow/script-task.

[40] Documentation for developing CLR user-defined functions. https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-user-defined-functions.

List of Figures

2.1 Screenshot of a Manta data flow graph viewer with a graph representing data lineage of a sample C# console application, which reads data from the standard input and inserts them into a database. 8
2.2 Screenshot of the Manta Admin GUI 9
2.3 Screenshot of a Manta Admin GUI 10
2.4 Simplified general architecture of Manta consisting of three main modules: Manta Flow Server, Manta Flow CLI and Manta Admin GUI 11
2.5 Architecture of an arbitrary programming language scanner. . . . 13

3.1 A flow graph of handler selection for encountered method. Terms *unused interface* and *unused type* represent a type that was not used in the algorithm. 24
3.2 Diagram of how `ProgramFlowData` stores data flows for every part of every method. 26
3.3 User survey processing diagram that validates user data, saves incomplete data into a flat file, saves newsletter to users that opted in and saves anonymised survey data into a database. 28
3.4 Generated data lineage graph of a user survey processing use case 3.11. 29

4.1 Merging of embedded code graph with the source technology graph using pin node. Figure taken from thesis by Michal Jurčo [34]. . . 39
4.2 General design of an embedded code service. 41
4.3 Screenshot of a Manta data flow graph with a lineage of a program executing a scalar query and writing the result into a file. 43
4.4 Sequence diagram of a search for method handler in a depth-first-search manner. 45
4.5 Sequence diagram of a search for method handler in a breath-first-search manner. 46
4.6 Diagram with two classes implementing a common interface and two plugins with handlers for those methods. 47
4.7 Diagram with three classes implementing a common interface and two plugins with handlers for some of those methods. 48

5.1 Diagram of an algorithm to decide whether a class is a controller. 51
5.2 Handler selection algorithm diagram. 56
5.3 Enumeration of order in which classes are tested for handler existence. Starting in class `B` with handlers registered in order from 1 to 3. Numbers in circles represent the resulting order. 57

6.1 Page with C# connection configuration setup. 61
6.2 Process manager page with workflow configuration setup. 62
6.3 Displayed information about a finished workflow run. 63
6.4 Manta Flow Viewer with selected revision of a C# code analysis. 63
6.5 Data lineage graph of the C# program in Listing 6.1 a C# code analysis. 65

7.1 Graph of a simple ASP.NET application with all possible controller actions. 70
7.2 Graph of C# program querying an unknown column in the database. 71
7.3 Graph of C# program executing a scalar query (`MAX`). 71
7.4 Graph of C# program that decides what table it will query in runtime via branch. 72
7.5 Two graphs of the same C# program that enumerates arguments and outputs them into the console and executes database query and writes its first column into the console. The top graph shows an invalid unknown column because of the enumeration. The bottom graph correctly does not contain an unknown column. 72
7.6 Two graphs of the same C# program that executes three database commands and writes the first column into the console. 73

List of Tables

5.1 Mapping of access modifiers of types from C# to CIL nomenclature. 52
5.2 Mapping of access modifiers of methods and fields from C# to CIL nomenclature. 52

6.1 Table of available environment variables and their descriptions. . . 62

Listings

3.1 Simple C# program 16
3.2 CIL of a simple C# program 16
3.3 Integer addition in C#/CIL and Java bytecode. 16
3.4 The C# `protected` access modifier in code and its CIL representation as `family`. 18
3.5 C# virtual method definitions example. 18
3.6 C# virtual method call example. 19
3.7 C# snippet with the variable `a` aliased with variable `x`. 20
3.8 Definition of a handler for the method `Void TextWriter::Write(?)` that propagates flows from the first argument into the receiver via the `InputOutputPropagationRoutine.StreamWrite` routine. . . 23
3.9 C# code example of writing into a file with corresponding selected CIL instructions. 25
3.10 Definition of a handler for method `StreamWriter File::AppendText(String)` that propagates flows from the first argument into the receiver via the `InputOutputPropagationRoutine.FileStreamPath`. 26
3.11 Code snippet representing implementation of use case with user survey processing. This code snippet hides implementation details in method calls. 28
4.1 Low-code ASP.NET application with minimal APIs. 33
4.2 Low-code ASP.NET controller. 34
4.3 Query from an unknown named column. 41
4.4 C# example of simple classes implementing an interface. 44
4.5 Example of a situation where multiple method handlers get selected. 47
6.1 C# program that executes a database query and writes columns *UserName* and *UserId* to standard output. Then executes a scalar query whose output is written into a file called *output.txt*. 64
7.1 C# application with argument enumeration and basic database query. 67
7.2 C# application that executes three database commands and writes the first column to the console. 68

Attachments

The attachment of this thesis consists of two subdirectories and the thesis text in a PDF format. The directory hierarchy can be seen below in a tree structure.

Subdirectory *src* contains source code, which was either created or modified during the implementation of the thesis solution. Sources are grouped by the programming language and further by the project they belong to. The subdirectory *Others* contains scripts used for semi-automated scanner testing.

Subdirectory *tex* contains all LaTeX sources of this thesis, including all used figures.