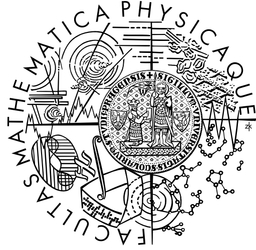

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

Manta C# Scanner

MANTACS

Authors: Bc. Tereza Storzerová, Bc. Lukáš Riedel,
Bc. Jan Joneš, Bc. Jakub Sýkora

Supervisor: RNDr. Pavel Parízek, Ph.D.

Consultant: RNDr. Lukáš Hermann (Manta Tools, s.r.o.)

Faculty of Mathematics and Physics
Charles University

Contents

1 Introduction **2**
1.1 Data Lineage . . . 2
1.2 MANTA Flow . . . 3
1.3 Architecture Outline . . . 4
1.4 Goals . . . 4

2 Symbolic Analysis **5**
2.1 Analysis Sensitivity . . . 5
2.2 Analyses Characteristics . . . 5
2.3 Intermediate Representation . . . 5
2.4 Algorithm Outline . . . 6
2.5 Analysis of Specific Methods . . . 6
2.6 Alias Analysis . . . 6

3 Functional Requirements **7**
3.1 Symbolic Analysis . . . 7
3.2 Inputs . . . 8
3.3 Outputs . . . 9

4 Nonfunctional Requirements **12**
4.1 Integration . . . 12
4.2 Extensibility . . . 12
4.3 Fault Tolerance . . . 12
4.4 Data Integrity . . . 13
4.5 Testing . . . 13
4.6 Continous Integration . . . 14
4.7 Code Quality . . . 14
4.8 Development Tools . . . 14

5 Architecture **15**
5.1 Connector . . . 15
5.2 Dataflow Generator . . . 18

6 Project Execution **19**
6.1 Organization . . . 19
6.2 Roles in Team . . . 19
6.3 Continuous Integration . . . 20
6.4 Future Work . . . 20

Bibliography **21**

1. Introduction

In this project we aim to develop a tool for static analysis of dataflow between C# programs and systems such as databases in the context of data lineage. This chapter explains our motivation for the development of this tool and provides a brief introduction into the data lineage.

1.1 Data Lineage

Nowadays the data volume of many companies (for example in the banking or insurance sector) is very large and keeps growing. This can cause several problems for these companies, as the full control over the data could be impossible. Our motivation is to provide the company a view of data lineage that tracks the flow and transformation of data across their systems. In other words, the data lineage gives an answer to the question of which outputs can be possibly affected by which inputs. The next sections describe specific applications of the data lineage.

Privacy

The strict requirements come directly from customers or a government that wants to protect customers. As an example, a company should be able to clear all customer data and get into a state as if it never knew the customer. Even this seemingly simple task can cause problems if it is difficult to trace how the customer data could be propagated.

Regulations

A regulated pharmaceutical business needs to provide audit information about utilized data due to transparency requirements, medical records protection and manufacturing of drugs. Their data stored in database systems are used in computing in cloud environment with services and deployed applications. The flow of data between these systems, i.e. data lineage, needs to be analyzed as absolute control over the data is required.

Data Management

It is also important for the enterprise itself to be able to see how their data changes are propagated across the system. For example, when a new restriction on some values is done, it should affect all locations where these values may have been stored. It is also practical to know exactly how the program affects the databases or file system in case of a problem (error, exception etc.). The need of such knowledge can be demonstrated on the following examples:

- Reducing data growth in case of mergers and acquisitions of multiple companies. Each of the companies has usually its own data warehouse with lots of valuable data, so it is necessary to consolidate it.
- Selection of stable data to be kept on-site in data warehouses as an alternative to expensive cloud migrations.

Tools for Data Lineage

Necessary parts of dataflow, such as database systems, can be analyzed by many automated tools that provide information about system's data sources, transformations and sinks. Functions deployed in cloud computing and business applications, both written in high-level programming languages, are often analyzed manually. Their possible automated analysis similar to one of the database systems would be beneficial. Developing a tool that performs this analysis is exactly the topic of this software project.

1.2 MANTA Flow

One of the data lineage solutions is a platform called MANTA Flow [1]. For each supported technology in MANTA Flow, there is a scanner that performs an analysis producing dataflow information – data transformations and propagation from data sources to their sinks. Such sinks and sources can be database tables or specific files. The output of MANTA Flow is a data lineage graph where edges represent the dataflow and nodes represent sinks and sources. Figure 1.1 contains a simplified MANTA Flow graph, which represents the following scenario: The program queries user data from a database, modifies them and creates a log file of such activity. The scanner analyzes the dataflow in the program and adds to the graph a directed edge from the node representing a particular database column to the node representing the log file contents. Such connected nodes are then a part of subgraphs representing entities, such as a program or database connection with schema. MANTA Flow then visualizes the nodes and edges, depending on their types.

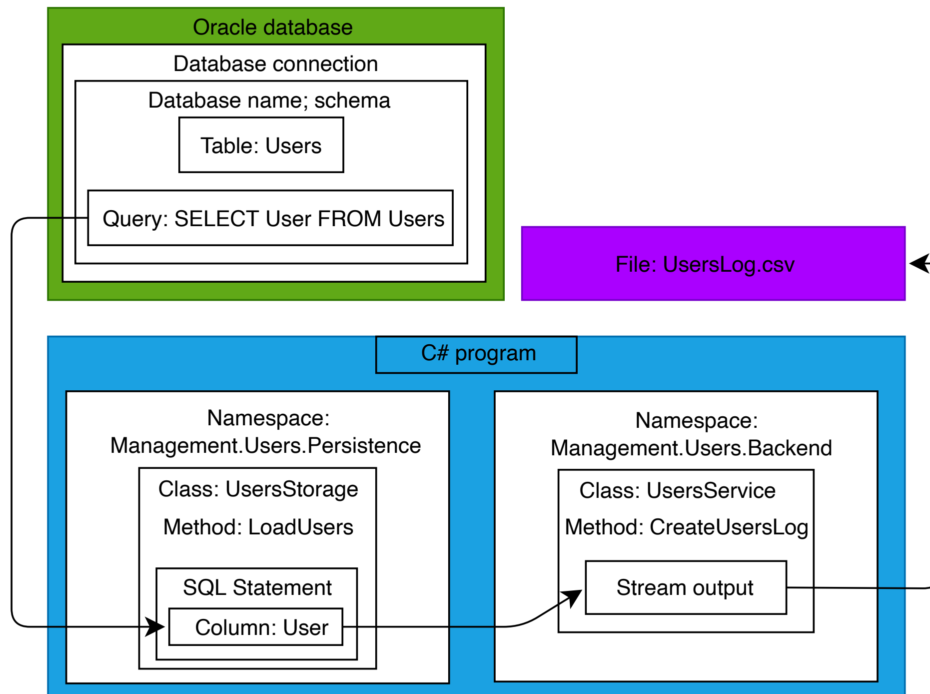

Figure 1.1: Example of a dataflow visualization

In the case of high-level programming languages, MANTA Flow offers a Java scanner. Its dataflow analysis utilizes symbolic analysis designed by Parízek [2], based on Java bytecode inspection. The main goal of this project is to extend the set of available scanners in MANTA Flow by a new scanner for C# programs. As the .NET platform uses CIL as intermediate code for the C# programming language, similar to Java bytecode, the scanner will implement the already mentioned symbolic analysis. Main concepts of the analysis will be briefly explained in the following chapter, as this particular analysis needs to be considered when specifying both functional and nonfunctional requirements.

1.3 Architecture Outline

When specifying requirements, it is important to understand the high-level architecture of the scanner. As the scanner is developed for MANTA, its architecture must follow the general scanner architecture that consists of two main parts:

- **Connector** – runs a dataflow analysis of provided input program
- **Dataflow generator** – implemented in Java, adds connector analysis results to the output graph of MANTA Flow

`Connector` is further divided into five modules, described in Section 5.1.

1.4 Goals

Based on the introduction, we can specify overall goals for the project:

1. Develop the C# scanner that will perform a static dataflow analysis of C# programs:
   (a) The analysis shall be based on the symbolic analysis described in Chapter 2.
   (b) The scanner shall be integrated into the MANTA Flow platform.
2. As the scanner is being developed for a commercial company, MANTA, we shall meet its requirements for development practices and functionality of the scanner.

In the next chapters we propose in further detail the specification of the C# scanner, specify its both functional and nonfunctional requirements and show a brief description of symbolic analysis.

2. Symbolic Analysis

Symbolic analysis is a method of programmatically analyzing the behavior of computer programs with respect to their inputs and outputs, i.e., what data transformations and propagation they perform.

The behavior of a given computer program can be analyzed either statically or dynamically. Dynamic analysis executes the program with various inputs and observes how the data are propagated. This kind of analysis requires lots of test data (inputs) to be useful. Static analysis, on the other hand, inspects the behavior of the program without actually executing it. In order to do that, the program's source code, machine code or intermediate representation can be inspected. In MANTA Flow, all analysis is of the static kind, hence scanners for languages with intermediate code representation perform static analysis, as well. Symbolic analysis is a specific approach to static analysis.

2.1 Analysis Sensitivity

Automatically analyzing the dataflow in full detail of any given program is computationally intractable [3]. Therefore, static analysis only aims to approximate the behavior of a given program. Based on how much they approximate their results, the analyses have different sensitivities (for example, with respect to control flow or calling context) [4, 5].

2.2 Analyses Characteristics

There exist several characteristics describing the sensitivity of the static analysis. *Path sensitive* analyses process different control flow paths separately. *Flow sensitive* analyses take into account the order of instructions in the code. (i.e., while *flow insensitive* analysis is only able to detect that `a` and `b` can point to the same memory location, flow sensitive analysis can determine from which statement this actually occurs.) *Context sensitivity* is the ability to distinguish different calling contexts (i.e., a new analysis is performed for every distinct set of actual method parameters and for different call sites). The symbolic analysis is flow and partially context sensitive.

2.3 Intermediate Representation

As is common for many other static analyses, symbolic analysis works with an intermediate representation of inspected source code. Symbolic analysis uses symbolic representation of program variables and their values. In order to utilize symbolic analysis for C# programs, feasible framework for CIL code inspection must be used.

CIL works with a simple stack of values. For example, if a program wants to work with the expression `array[i]`, its CIL code must load reference to `array` to the stack, get element at position `i` (which pops the array reference, finds

reference to its `i`-th element and pushes that to the stack) and finally dereference the top of the stack to get the actual value of the `i`-th element.

2.4 Algorithm Outline

At start, the worklist (queue of methods to be processed) is created containing all methods reachable from the entrypoint methods. Generally, a method is processed in flow sensitive manner by visiting its symbolic expressions for each execution branch, noticing their effects on analysis (such as value loads and stores) and then merging branches into one analysis result.

As the analysis is context sensitive, several invocation contexts for each method exist. Each invocation context specifies method arguments. In each iteration, a method from the head of the worklist is processed (i.e., its summary is updated). If effects of the method actually change, all callers and callees of that method are added to the worklist along with the current invocation context, because their summaries can be potentially affected by the just computed effects and need to be recomputed. When the worklist is empty, the algorithm terminates.

2.5 Analysis of Specific Methods

Symbolic analysis needs to handle some methods specially, without inspecting their symbolic representation of code.

One group of special methods are framework and library methods, as dataflow effects of such method are not based on CIL instructions processing, but on method semantics instead, explicitly encoded in the scanner. One example could be a sequence of methods creating prepared statement for a database query. In context of scanner, special *plugin* for analysis of specific framework can be developed. Such plugin will intercept analysis of framework's method calls and provide its own semantics of method's dataflow.

When the analysis is implemented for the C# programming language, it must deal with methods in .NET which implement low-level functionality. Many of such CIL methods can be skipped, so that the analysis does not go to the lowest levels which would be computationally demanding. Some other CIL methods can be handled specially because analyzing them generally could give imprecise results. Some of the methods are written in C and compiled to machine code that can't be processed by symbolic analysis.

2.6 Alias Analysis

Symbolic expressions can be aliased meaning that two variables can reference the same object. Symbolic analysis needs to know about these aliases to correctly compute the flow effects on the given piece of code. Symbolic analysis also needs to handle virtual method calls, possibly processing all of the particular virtual methods implementations. Hence, part of the project needs to contain an infrastructure for providing enumeration over several implementations of an interface or an abstract method.

3. Functional Requirements

The C# scanner is developed as one of the scanners for MANTA Flow platform, as mentioned in the Chapter 1. The following section describes requirements for the C# scanner as a part of MANTA Flow, based on the supported technologies in MANTA Flow.

3.1 Symbolic Analysis

As already mentioned in the description of symbolic analysis in Chapter 2, analysis of frameworks and libraries will be provided by plugins. MANTA Flow works with data inputs and outputs. Hence, we need to process semantics of I/O operations and database method calls.

Databases Support

There are several third-party libraries for manipulation with databases. The C# scanner will support ADO .NET [6], which can be considered as the .NET equivalent of JDBC [7], supported by the Java scanner. The plugin for ADO. NET only needs to form queries because their processing is already implemented by other parts of MANTA Flow. We have selected ADO. NET as it is a framework used by one of MANTA's customers.

Constants Analysis

With the analysis of database dataflows comes the requirement for processing string and integer constants. The scanner must know the structure and content of dynamically constructed SQL queries. For example, it is common for programs to concatenate multiple strings into a single SQL query or to substitute parameters in SQL query templates with their actual values. Similarly, we need to analyze numerical values as precisely as possible, since they are used to denote columns and parameters in SQL queries. In particular, we should be able to recognize numeric constants that are written directly in the code. On the other hand, precisely analyzing all numerical operations would be computationally unfeasible.

I/O Support

Common input and output methods used in C# programs will be supported, such as methods of the classes `Console`, `StreamReader`, `StreamWriter`, `FileStream`, and `File`.

Unsupported Features of the Language

Some C# features are too difficult to analyze and they will not be supported at all or only in a limited form. Among such features are:

- reflection and dynamic code generation,
- dynamic language runtime (i.e., `dynamic` variables),
- multi-threading and parallelism,
- exception handling.

Supported Features of the Language

On the other hand, to correctly analyze data lineage, the C# scanner needs to support:

- arrays and other basic collections (lists, dictionaries),
- string and numeric constants, including basic manipulations with them,
- inheritance and generic types with no constrained generic parameter,
- all control flow statements (if-then-else, loops, method calls).

3.2 Inputs

Input data are passed into the scanner together with a configuration file. The symbolic analysis needs to have information about:

- **Entry point methods**. The entry point is considered to be any method where the program execution should start, it does not have to be `Main` necessarily, it can also be a controller's method as route handler.
- **Code to be analyzed**. This can include standalone `.cs` files, `.csproj` projects and compiled assemblies (`.exe,.dll` files)
- **Excluded namespaces**. Some namespaces and classes can be excluded when it is not needed to include them into the analysis (for example the loggers). It is supposed that exclusions will be implemented using regular expressions, so just a part of namespace can be excluded too.

Note that analysis accepts any combination of file types to be analyzed, the scanner will load them all. Source code loading is resolved by compiling the sources into CIL assemblies at runtime. Therefore, all dependencies must be specified, otherwise the code would not be compiled and the scanner might reject such input. This can include even referencing core libraries of .NET runtimes. Handling any compilation errors is out of scope of the project, that means if there is any `.cs` or `.csproj` file at input that cannot be compiled for some reason, the scanner will not continue with the analysis.

3.3 Outputs

The description of MANTA Flow in Section 1.2 implies that the C# scanner implementation needs to map its dataflow sources and sinks to already defined nodes in MANTA Flow and also needs to define its own nodes that can be recognized and visualized by MANTA Flow. The content of the nodes representing the scanner analysis results is determined by business requirements, deciding which analysis results information can be omitted from the MANTA Flow output graph. Based on these requirements and symbolic analysis capabilities with supported plugins, the following dataflow types can be visualized:

- Database queries (SQL)
- File I/O
- Entrypoints

Now we describe the structure of the analysis output and the mentioned dataflow types in further detail.

C# Program

The symbolic analysis of C# program will group its dataflow sink and sources as one subgraph that contains several *program locations*. A program location consists of:

- Namespace name
- Class name
- Method signature
- Dataflow context information

The dataflow context information helps to distinguish location of analysis results by including information such as identification of calling context of a method. A program location is associated with the dataflow sources and sinks taking place at the location. Examples of sources and sinks will be described in the remaining description of the analysis outputs.

Databases

For each detected database statement execution, the scanner will associate the *SQL statement* node with the statement's program location, which is connected with:

- Nodes representing statement parameters
- Nodes representing table column nodes

Both column and parameter nodes are then connected to their data sources or sinks. In order to interpret the executed statement, MANTA Flow utilizes a service which also adds subgraph of the analysed statement directly into the output graph. Such subgraph contains query with subtrees of tables affected by utilized parameters or columns in the result set. Each query is also associated with the specific database connection with its location in the graph.

Consider the following ADO .NET code snippet, which contains a simple `SELECT` statement:

```
1 string queryString = "SELECT UserName FROM management.users WHERE Age > @age";
2 /* Assume that the connection is already created. */
3 SqlCommand cmd = new SqlCommand(queryString, connection);
4 command.Parameters.AddWithValue("@age", 20);
5
6 SqlDataReader reader = command.ExecuteReader();
7 while (reader.Read()) {
8     Console.WriteLine(reader[0]);
9 }
```

Introduced snippet should be analyzed and presented in MANTA Flow graph in a structure similar to one shown in Figure 3.1.

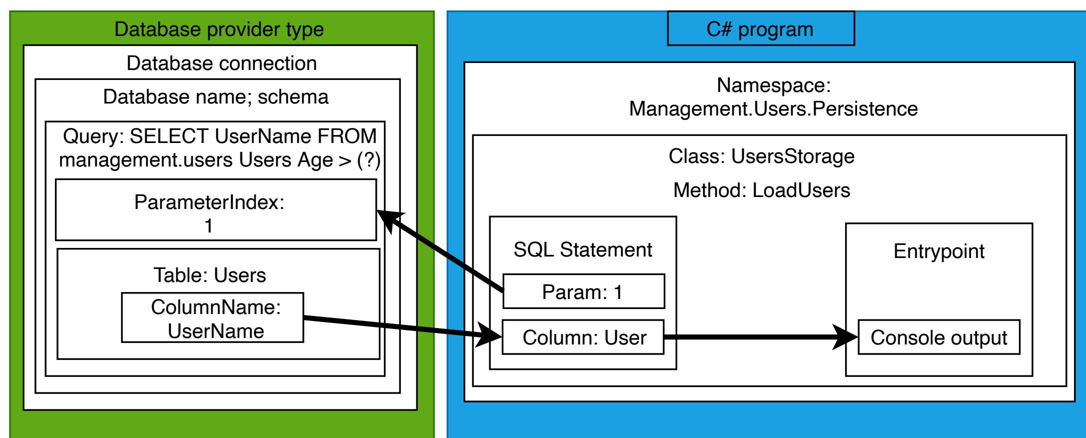

Figure 3.1: Example of database query dataflow output

Entrypoints

The scanner output will also show dataflows where sinks and sources are common program input and output constructs such as entry method arguments or console I/O – we call them *entrypoints*.

Following the dataflow example in the code snippet of the previous section, the output graph would contain a node representing the entrypoint dataflow, where the sink is the call of `Console.WriteLine`. An oriented edge from query result column then leads to the entrypoint node as the dataflow.

The following list provides examples of supported entrypoints:

- Console I/O - methods such as `Console.WriteLine` and `Console.ReadLine`
- Inputs - method arguments, instance fields and static fields
- Outputs - return values

File I/O

The scanner will contain a plugin for analyzing file input and output streams, as described in Section 3.1. This specific type of dataflow source and sinks has a unique visualization in the output graphs. Each file I/O node will contain information whether the node represents a dataflow sink equal to the file output stream or a dataflow source equal to the file input stream. In order to provide detailed information about writes and reads from CSV files, as a business requirement, the node or its subtrees can possibly contain detailed information about affected CSV columns.

4. Nonfunctional Requirements

The following chapter contains specification of non-functional requirements impacting the scanner execution, architecture and development process.

4.1 Integration

As described in Section 1.2, the scanner will be integrated into the MANTA Flow platform. This integration will be based on the already described output graph of MANTA Flow and also on the CLI tool, which will be described in this chapter.

Manta Dataflow CLI

The architecture of the MANTA Flow platform is decomposed into the server and client parts. Manta dataflow CLI is a tool that runs scanner analyses in separated units called scenarios. For each scanner, the scenario in the CLI tool configures the scanner environment and prepares its prerequisities. The scanner scenario has a step that adds new nodes to the MANTA Flow graph as its analysis result. The CLI tool is implemented in Java and loads configurations via Spring boot XML files that contain beans that register scanner output classes. The classes inherit from base classes that are used as a representation of scenarios in the CLI tool algorithm. The implementation of the C# scanner needs to follow these integration rules in order to be run from the MANTA dataflow CLI, hence part of the scanner must be implemented in Java.

4.2 Extensibility

Initially, the C# scanner cannot support all commonly used .NET libraries, and therefore will only support ADO .NET and System.IO as described in Chapter 3. The scanner needs to be extensible enough, so that support for other libraries can be added in the future. Section 6.4 describes possible extensions that could be implemented out of the scope of this project.

4.3 Fault Tolerance

The scanner shall recover from exceptional states with minimal impact on the analysis results and log such states at the warning level. Expected impacts and handling of each error shall be documented. It is expected by the MANTA Flow platform that execution will not halt due to any error. We provide some examples, how fault cases should be handled in certain situation, as a general guideline:

- Method processing – If there is a method call in the analyzed program that the scanner is unable to process, it skips such method and resolves the method as an *identity* – a method that has no instructions in its body.

- Column mapping – when a `SELECT` statement which selects several columns is analyzed, the dataflow target is matched by the column name. Due to imprecise analysis, the column name could be analyzed incorrectly. In such case, if the column name matches no column of a table, all table columns are mapped to the dataflow target.

4.4 Data Integrity

Design of data entities that capture analysis results shall be type safe – the results shall be covered by an enumeration of expected states in order to assure logical integrity. The context under which the scanner operates shall be always well-defined. In order to conform to expected behavior mentioned in Section 4.3, incomplete analysis results shall be propagated into the resulting MANTA Flow graph, and the design of data entities shall also contain enumeration over incomplete states.

4.5 Testing

We plan to develop tests of two kinds: unit tests and integration tests.

Unit Testing

Each project shall be covered by unit tests. Unit testing should be primarily used to discover bugs during execution of the code parts, but should also help to design code by utilizing tools for mocking and should partially work as code documentation.

Integration Testing

Integration tests shall exist for testing of the entire scanner and its modules. Generally, there are several approaches to testing the integration of multiple modules of the system. We aim to use bottom-up testing, since it suits our architecture well. At first, the lowest level module is tested. Then, if all the tests pass, the module is used to test the next level module until the module at the top of the hierarchy is tested. For this purpose, several test scenarios will be created.

The test data that we aim to use are called *test targets*. Test targets are small C# programs that should test some functionality of the scanner, e.g., querying the database. For convenience, a test framework shall be created. Such a framework should provide an interface for

- gathering test targets in the file system along with their expected results,

- passing the gathered data into test methods.

Having such a framework has multiple benefits, one of them being a fact that a new integration test can be added without the need of modifying the project with tests, so that testing is automated. Additionally, the framework should speed-up the process of finding bugs in production on the customer side, since bottom-up testing can help to find the bug very easily, and customer data can be treated in the same way as test targets.

4.6 Continous Integration

In order to make the development process easier, a server with automated builds shall be used. Builds shall be triggered for each update of the solution in the central git repository. The team will follow the GitFlow [8] git branching model. That means that individual parts of the code will be developed in the *feature branches* and, when complete, they will be merged into the *develop branch*. The *master* branch is prepared for deployement of release versions. The builds for each branch type will contain different steps:

- Feature branches and pull requests should build the solution, run unit tests and verify code quality.
- Builds on the develop branch shall have the same steps as builds on feature branches and will additionally include time-demanding steps such as integration tests.

4.7 Code Quality

The project team shall utilize a tool that allows code reviews and approvals of included changes to the development branch from team members. The reviewer should be able to see quite easily whether a specific build has passed or not.

The team shall utilize a code quality analyzer that detects language specific programming errors.

4.8 Development Tools

The project team shall utilize development tools given or approved by MANTA, that conforms requirements related to code quality and continuous integration. The team will then provide feedback on the configuration and usage of the utilized development tools, which can be then used by other teams in MANTA in the future.

5. Architecture

The architecture of the C# Scanner is separated into two main parts – `Connector` and `Dataflow generator`. The high-level architecture is visualized in Figure 5.1. The `Dataflow generator` adds nodes to the MANTA Flow output graph as described in Section 1.2. The `Connector` processes the scanner input and performs symbolic analysis, that provides the analysis results to the `Dataflow generator`. Besides these two parts, an integration project for the MANTA Flow CLI with configurations must be developed as mentioned in Section 4.1. This separation follows the general architecture of the MANTA Flow scanners and suits additional problems present in the C# scanner, which will be discussed further in this chapter. Let us describe two main parts in further detail.

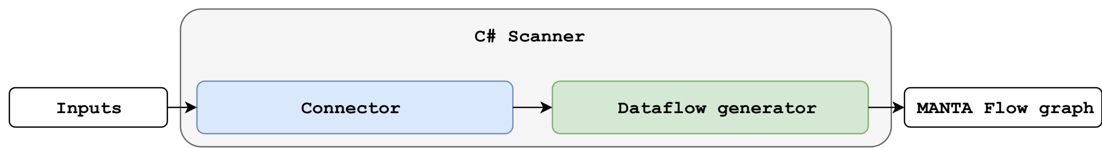

Figure 5.1: High-level architecture of the C# Scanner

5.1 Connector

The `Connector` receives the input C# program and performs symbolic analysis. It consists of the `Extractor` and `Reader` components, as shown in Figure 5.2. The `Extractor` creates a configuration based on the input program – this includes finding entry points, creating scope and common exclusions. The `Reader` then processes the input program and the extracted configuration. It consists of five smaller components. The `Intermediate code simulator` performs a symbolic interpretation of the CIL instructions of a method, based on information from the code provided by the `Intermediate code infrastructure`. With plugins and both infrastructure modules, the `Symbolic analysis` module then performs the actual dataflow analysis. Finally, the `Analysis results transformer` creates a graph of analysis results, which is the final product of the `Connector`. The following sections describe the internal parts in further detail.

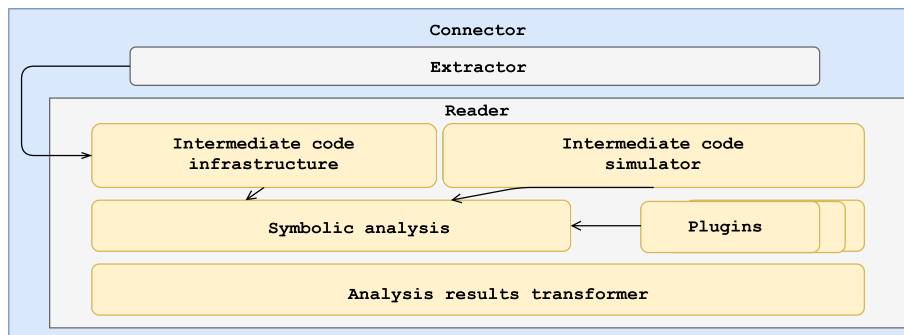

Figure 5.2: High-level architecture of `Connector`

Extractor

The `Extractor` shall receive inputs as described in Section 3.2 and transform them into input structures used by `Reader`. Extractor run is not mandatory as it is only included to simplify the configuration process. Although the configuration can be passed into `Reader` manually, it is strongly recommended to let `Extractor` create the configuration. When `Connector` is integrated into MANTA Flow, the `Extractor` is always utilized.

Intermediate Code Infrastructure

`Intermediate code infrastructure` processes assemblies and provides information about program structure for the purpose of symbolic analysis:

- **Class Hierarchy** - Used to get, e.g., all implementing classes (structures, interfaces) of given interface or all derived classes of a given class.
- **Call Graph** - Given an entry point method, we must be able to get all possible method calls reachable from the entry point.

To deal with CIL code, the module internally uses the Mono.Cecil framework [9], which was chosen as the most suitable option (compared with Roslyn, for example). One of the goals of this module is to provide wrapper classes for Mono.Cecil, as well to extend functionality of Mono.Cecil. This can include:

- Caching data about the intermediate code, so that these caches are inside wrappers and the developer does not have to think about whether some data are cached or not.
- Hiding unnecessary functionality of Mono.Cecil, as it might be complex and poorly documented, to prevent confusion.

Additionally, this module should simplify usage of different wrapped third party libraries, if needed, so that the replacement has no impact on other modules.

Intermediate Code Simulator

`Intermediate code simulator` is a layer between the stack machine code of the analyzed program and the `Symbolic analysis`. Its purpose is to provide an abstraction of the code (information about CIL code metadata and instructions) and make the analysis easier to implement.

Symbolic Expressions

As the C# scanner is processing the stack machine code, it is easier to create the simulator layer, because it only has to work with a stack of *symbolic expressions* – an abstraction created over the CIL instructions. Imagine having two numeric expressions on the stack and the simulator just approaching the `add` instruction. The only thing it does is that it pops two expressions from the stack, evaluates addition upon them and pushes the result back onto the stack. These expressions should be passed into the symbolic analysis. `Intermediate code`

simulator provides such information to Symbolic analysis only when some important action from the analysis point of the view is happening. The important actions are, for example, method call and load or store of some variable.

Symbolic Analysis

As the first step of the Symbolic Analysis, it collects information about aliases among expressions in the analyzed program. Basically, the analysis needs to know which expressions can be aliases of each other in order to share dataflow information among them.

Symbolic analysis uses Intermediate code simulator to simulate the execution of a given CIL code in methods. This simulation works with symbolic expressions instead of the concrete variable stack maintained by .NET runtime. The analysis itself is implemented using the worklist algorithm described in Section 2.4.

During its execution, the analysis computes the side effects and dataflow information for each expression used inside the analyzed method. It also has to propagate dataflow information among aliased expressions and across method call boundaries. Some methods will be handled externally by Plugins.

Plugins

Plugins handle the methods that are not resolved by Symbolic Analysis. These are methods that the scanner does not want to analyze and methods that are treated in a special way. In most cases, plugins provide a partial analysis of methods from database frameworks (e.g. ADO. NET) that produce and consume data. To analyze parts of methods bodies, the plugins may need to cooperate with the Symbolic Analysis component.

In this project, we aim to design an interface which shall be general enough to allow implementation of plugin for any library or framework. Plugins for I/O and ADO. NET shall be developed within this project, while additional plugins are subject of the future work (based on customer's requirements).

Analysis Results Transformer

After the work of Symbolic Analysis is completed, the Transformer retrieves the dataflow information of the methods and their invocation contexts. Based on these information, it creates a graph structure whose nodes represent program points (actions) and edges accurately represent the dataflows between them.

5.2 Dataflow Generator

`Dataflow generator` receives the analysis results graph from `Connector` and transforms it into the output graph of MANTA Flow with the node types described in Section 3.3. `Dataflow generator` puts results of `Connector` into the context of data lineage of an entire technology stack with an option to resolve some result dependencies as described in the following text.

Interoperability

Dataflow generator must be implemented in Java as the `Scanner` output is expected to be received from an extensions of base classes of MANTA Flow CLI, as mentioned in Section 4.1. Since `Connector` will be implemented in C#, interaction between those two components must be solved. The interoperability is implemented by using remote procedure calls - gRPC [10] with Protocol Buffers messages [11]. This solution impose less difficulties during development as developers work only with generated code for both *data transfer objects* and remote procedure calls. Protocol buffer message definition in *.proto* file works also as *canonical data model*. Implementation of gRPC does not require any installations on customer's machines. As native .NET interoperoperability with Java should be provided in future in one of .NET releases, each component is designed in way which allows easy change of interoperability implementation as it is provided in separate module.

Transformations

The transformation process between graphs also includes usage of services provided by MANTA Flow which solves some dependencies of the analysis results. For example, database queries are linked to database connection by several attributes. `DataflowQueryService`, used in multiple MANTA Scanners, resolves the database connection and performs the query as it knows a schema based on the connection. The service then creates the query as a part of the MANTA Flow output graph and provides query results to the `Dataflow generator` in order to connect C# program nodes and the query nodes.

6. Project Execution

This chapter specifies the organization of work on this project and provides guidelines for the development process.

6.1 Organization

The project is conducted in close cooperation with the MANTA company and therefore the work organization is based on the company guidelines. Weekly status meetings take place, where work plan is reviewed and refined. Next to those meetings, special meetings focused on the design and architecture of the C# scanner take place. Important design and architecture decisions are documented in Confluence pages [12]. The work is tracked and reported in JIRA system [13] and the team communicates via Slack [14].

6.2 Roles in Team

The team consists of four members. Based on the described requirements and architecture, the following roles are assigned to team members:

- Jakub Sýkora
  - Development tools
  - Development environment
  - MANTA Flow integration
  - Dataflow generator
- Lukáš Riedel
  - Intermediate code infrastructure
  - Intermediate code simulator
  - Integration testing
- Jan Joneš
  - Symbolic analysis
  - Design of analysis plugins
- Tereza Storzerová
  - Analysis results transformation
  - Implementation of analysis plugins

Several interfaces have been already designed, so that each team member can work independently of the others in a particular way.

6.3 Continuous Integration

Based on the development requirements defined in Section 4.8, the team received the following tools:

- Jenkins server [15] for continuous integration
- Gitea [16], a locally hosted open-source git repository that allows to create pull requests with code review comments as demanded by the code quality requirements defined in Section 4.7

In the early stages of the project, integration between Jenkins and Gitea was established via Gitea plugin [17] and multibranch pipelines plugin [18]. Each commit and pull request carries information whether the build passed or failed. This information is visible from Gitea and contains links to the build in Jenkins.

The integration is based on webhooks managed by mentioned plugins, where the multibranch pipeline plugin manages mappings of Jenkins jobs to repository branches. For each branch update such as pull request or commit, a build in the mapped job is triggered. The content of the build is set via `Jenkinsfile` stored in the branch, which enables to treat continous integration configuration as a part of the repository and code. To enable easy configuration of a Jenkins machine, several steps are run in Docker containers [19]. With Docker containers, developers can easily run the build steps on their development machines. Based on the continuous integration requirements defined in Section 4.6, the builds involve the following steps:

- Feature branches and pull requests
  1. Solution build (when warnings are produced, the build fails)
  2. Unit tests run (when a test fails, the whole build fails)
  3. Code quality checks with SonarLint [20] (when warnings from code quality packages are produced, the whole build fails)
- Develop branch
  1. Documentation validation (report on missing or invalid documentation generated by)
  2. Integration tests run (when a test fails, the whole build fails)

6.4 Future Work

As mentioned in Section 4.2, the C# scanner supports analysis extensions for .NET libraries and frameworks. Several analysis plugins for common .NET frameworks and libraries such as ASP .NET, Entity Framework or Dapper could be developed in future. Next to the common frameworks and libraries, codebases of several MANTA customers also involve specific libraries that require development of respective extensions.

Bibliography

[1] MANTA Flow.
https://getmanta.com/scanners-and-integrations/tech-summary/.

[2] Pavel Parízek. Hybrid analysis for partial order reduction of programs with arrays. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 291–310, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 42:230–265, 1936.

[4] Pavel Parízek. BUBEN: Automated Library Abstractions Enabling Scalable Bug Detection for Large Programs with I/O and Complex Environment, In Proceedings of ATVA 2019, LNCS 11781.

[5] Richard Eliáš. Analyzing Data Lineage in Database Frameworks. Master's thesis, D3S, MFF UK, Prague, Czech Republic, 2019.

[6] ADO .NET.
https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/.

[7] JDBC.
https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html.

[8] GitFlow.
https://datasift.github.io/gitflow/IntroducingGitFlow.html.

[9] Mono.Cecil.
https://github.com/jbevain/cecil.

[10] gRPC.
https://grpc.io.

[11] Protocol Buffers.
https://developers.google.com/protocol-buffers.

[12] Confuence.
https://www.atlassian.com/software/confluence.

[13] Jira.
https://www.atlassian.com/software/jira.

[14] Slack.
https://www.atlassian.com/software/jira.

[15] Jenkins.
https://jenkins.io.

[16] Gitea.
https://gitea.io/en-us/.

[17] Gitea plugin.
https://github.com/jenkinsci/gitea-plugin.

[18] Multibranch pipeline plugin.
https://jenkins.io/doc/book/pipeline/multibranch/.

[19] Docker.
https://docs.docker.com/engine/docker-overview/.

[20] Sonarlint.
https://www.sonarlint.org.