Working Papers

ERCIS - European Research Center for Information Systems
Editors: J. Becker, J. vom Brocke, T. Brandt, F. Gieseke, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, J. Varghese

Working Paper No. 40

A Testing Tool Visualizing and Ensuring Data-Flow Coverage

Laura Troost

ISSN 1614-7448

cite as: Laura Troost: A Testing Tool Visualizing and Ensuring Data-Flow Coverage. In: Working Papers, European Research Center for Information Systems No. 40. Eds.: Becker, J. et al. Münster 2024.

Contents

Working Paper Sketch 3

1 Introduction 4

2 Related Work 4

3 Dacite Overview 5

4 Dacite Core 8
4.1 Bytecode Instrumentation 9
4.2 Dynamic Analysis 14
4.3 Analysis during Symbolic Execution 17

5 Language Server Protocol 19
5.1 LSP Message Types 19
5.2 Workflow 21
5.3 Language Server 24
5.4 Language Client 28
5.4.1 Intellij IDEA 28
5.4.2 VS Code 32

6 Conclusion 34
6.1 Summary 34
6.2 Future Work 34

List of Figures

Figure 1: Exemplary screenshot of the Dacite visualization in IntelliJ based on given test. . . 6
Figure 2: Exemplary screenshot of the Dacite visualization for not-covered data flow. 6
Figure 3: Dacite components for IDE integration via LSP. 7
Figure 4: An overview of the Dacite packages and their interdependencies. 8
Figure 5: An overview of the Dacite Core implementation. 9
Figure 6: Example for inter-procedural definition-usage chains (DUCs). 10
Figure 7: An overview of the DUCs implementation in the package `analysis`. Methods are omitted for the sake of clarity [22]. 11
Figure 8: An overview of the analyzer implementations of Dacite in the package `analysis`. All attributes and methods are static, underlines are omitted for the sake of clarity. . . 14
Figure 9: Sequence diagram illustrating the interaction between the user and the components, the client, the language server, and Dacite Core for the dynamic analysis [23]. . . . 21
Figure 10: Sequence diagram illustrating the interaction between the user and the components, the client, the language server, Dacite Core, and Mulib for the symbolic execution and test case generation. 23
Figure 11: An overview of the language server implementation in the packages `ls` and `lsp`. Methods and attributes are omitted for the sake of clarity as well as dependencies and associations to the data structures of `lsp.defUseData` as the majority of the `ls` classes refer to these. 25
Figure 12: An overview of the transformed DUCs implementation for the visualization in the package `lsp.defUseData`. Methods are omitted for the sake of clarity [22] 27
Figure 13: An overview of the language client implementation for IntelliJ in the packages `intellij` and `lsp`. Methods and attributes are omitted for the sake of clarity. 29
Figure 14: Exemplary screenshot of the run line marker in IntelliJ to start the dynamic analysis for Dacite. 31
Figure 15: Exemplary screenshot of the generated driver with run line marker in IntelliJ to start the symbolic analysis for Dacite. 31
Figure 16: An overview of the language client implementation for Visual Studio Code (VS Code) in the package `vscode`. 32
Figure 17: Exemplary screenshot of the code lens in VS Code to start the dynamic analysis for Dacite. 33
Figure 18: Exemplary Screenshot of the tree view and inlay hints of VS Code for the dynamic analysis. 33
Figure 19: Exemplary Screenshot of the tree view and inlay hints of VS Code for the symbolic analysis. 33

Working Paper Sketch

Type

Research Report

Title

A Testing Tool Visualizing and Ensuring Data-Flow Coverage.

Authors

Laura Troost

Abstract

According to different studies, data-flow coverage is more effective in exposing errors than common control-flow metrics. However, there are not many suitable and available data-flow analysis tools. This report illustrates the technical details of the implementation of Dacite (Data-flow Coverage for Imperative Testing). Dacite is an open-source tool able to dynamically derive the covered data flow of a given Java program and its JUnit test cases. Using the Language Server Protocol, it visualizes the data flow within common IDEs. Moreover, Dacite provides feedback about the not-covered data flow and automatically generates test cases for these by using symbolic execution.

Keywords

Data-Flow Analysis, Bytecode Instrumentation, Language Server Protocol

1 Introduction

In this report, the implementation of the Dacite (**Da***ta-flow* **C***overage for* **I***mperative* **Te***sting*) prototype, its architecture, and design decisions are outlined. Dacite is a tool to dynamically identify reachable data flow within a given Java program and its JUnit test set. In contrast to most existing tools that deploy a static analysis, Dacite takes different challenges into account such as aliasing and inter-procedural data flow while considering data flow aspects in detail [22]. Moreover, it provides direct feedback about the data-flow coverage during the test case development through Integrated Development Environment (IDE) visualizations [23]. Furthermore, the symbolic execution engine Mulib [29] is utilized to dynamically traverse the whole program and derive data flow for not covered parts of the program given existing JUnit test cases and automatically generate test cases covering these [24].

This report focuses on the concepts and technical details related to the implementation of Dacite. Hence, the target audience is readers who already have basic knowledge about the concepts of data-flow analysis and Java bytecode (appropriate literature can be found in [20] and [4]). Moreover, Mulib is integrated as an external tool. As this report focuses on Dacite, information about the internal workings of the symbolic execution and Mulib can be found in [24] and [29].

This report is organized as follows. Section 2 relates Dacite to other work concerning the derivation and visualization of data flow. Next, Section 3 gives an overview of the functionalities and architecture of Dacite. In Section 4 the implementation of the core functionality of dynamically deriving data flow is elaborated in detail. Afterward, the implementation of the visualization elements using the Language Server Protocol (LSP) is explained in Section 5. Finally, Section 6 gives a conclusion and an outlook on further work.

2 Related Work

Studies have shown that data-flow coverage is more effective in exposing errors than common control-flow metrics such as branch coverage [8, 10, 18]. However, there are not many suitable and available tools for deriving the data flow for a given Java program [20]. The majority of existing tools utilize a static analysis for assessing the data-flow information, which has severe limitations. It is not possible to statically determine whether data flow is reachable during the execution [20]. However, in the context of test-case generation, it is crucial to know which data flow can be covered by a test case and which cannot be covered. Moreover, information about aliases cannot be accessed statically without techniques such as overapproximation [5].

JaBUTi(Java Bytecode Understanding and Testing) is one available tool to monitor the data-flow coverage [25]. It performs a static analysis based on the Java Virtual Machine (JVM) bytecode to construct a control-flow graph and then enhances it with data-flow information [26]. While it takes inter-procedural data flow into account, it is not able to differentiate the data flow of individual elements of objects and arrays as Dacite does. Furthermore, in contrast to most other tools, JaBUTi provides a graphical visualization of the identified data-flow graph and highlights the corresponding lines in the displayed source code. However, the tool operates as a standalone application that has to be installed additionally and is not integrated into any IDE which decreases its usability. Moreover, for each definition and usage, the whole line is highlighted instead of distinguishing different usages or definitions in one line such as Dacite [25, 26]. On top of that, experiments have shown that JaBUTi often identifies more DUCs (representing the data flow as an association of a variable usage to its latest definition) than the program has and thus, does not work reliably [22].

DFC (Data Flow Coverage) is another tool providing a graphical visualization of the data-flow coverage [2]. It is a plugin integrated into the IDEs Eclipse for statically analyzing the data-flow coverage. However, instead of automatically analyzing the source code, this tool requires user interaction to inquire which methods include object definitions and which methods only include object usages. Hence, the effort of identifying definitions and usages has to be made manually by the user. The identified data flow is visualized by a simple data-flow graph and a list of found DUCs. As this tool utilizes a static analysis, it is constrained by the aforementioned static limitations. Furthermore, it disregards data flow over the boundaries of methods and classes and considers objects and arrays as a whole for the data-flow analysis [2]. Moreover, it is not publicly available [23].

Next to these two tools, there exist several additional static tools lacking a graphical visualization (e.g. DuaF [19]) or analyzing the data-flow information at a more superficial level (e.g. Jazz [15]). To the best of our knowledge, the tool DReaDs (Dynamic Reaching Definitions) [5] is the only existing tool capable of dynamically analyzing data flow at the level of abstraction seen in static tools. However, this tool's data-flow representation is focused on variable definitions and their reachability rather than definition-usage pairs which impedes the comparability to most data-flow tools (e.g. JaBUTi and DFC). Moreover, DReaDs only analyzes the data flow for class state variables, i.e. class fields, and disregards all other defined variables such as local variables defined in methods. Additionally, DReaDs does not provide a graphical visualization and is not publicly available [6, 5, 22].

A graphical visualization is desirable to enhance the comprehensibility of the code coverage results. One prominent example of a tool providing integrated visualizations is the JaCoCo library[1]. It facilitates various control-flow metrics such as branch coverage or line coverage for JVM-based environments but does not provide data-flow coverage metrics. JaCoCo is open-source and provides different IDE plugins e.g. for Eclipse and IntelliJ. Its visualization is a key feature presenting the identified coverage information in a tree-like table and source code annotations similar to Dacite [23]. However, the IDE integrations of JaCoCo have been developed independently without using the Language Server Protocol (LSP) [7].

The LSP is designed to reduce the implementation costs for developing IDE integrations by isolating all IDE-independent functionality into a separate component that can be reused for different IDEs. Initially developed for programming languages, this approach has been extended and repurposed to other domains such as the context of model checking or theorem proving [17]. However, to the best of our knowledge, the LSP has not yet been employed in the area of code coverage [23].

3 Dacite Overview

Dacite is an open-source tool[2] for analyzing a Java program and its JUnit test cases for data-flow coverage. Through IDE integrations for IntelliJ IDEA and VS Code, Dacite can be utilized during the test case development to receive direct feedback about the data-flow coverage. For this, Dacite provides different functionalities:

F1: Identifying for a Java program and the given JUnit test cases the executed data flow with a dynamic analysis. Thereby, different challenges of data-flow analysis are addressed with Dacite such as considering variable aliases, i.e. two variables pointing to the same object, or inter-procedural data flow, i.e. data flow over the boundaries of methods and classes. Moreover, array elements and object fields are taken into account in detail so that their respective data flow is inspected instead of simply regarding the arrays and objects as a whole.

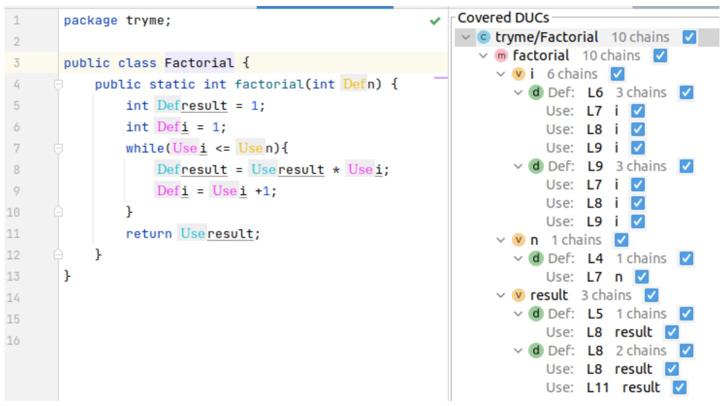
F2: Visualizing the passed data flow within the editor. More precisely, the identified data flow is displayed in a collapsible hierarchical list next to the source code. For enhancing comprehensibility, visualization elements highlighting the data flow within the source code editor can be enabled and disabled by the user for individual data-flow elements (see Figure 1).

[1]https://www.jacoco.org/jacoco
[2]https://github.com/dacite-defuse/DynamicDefUse

F3: Deriving the data flow that has not been passed yet by the given test cases through symbolic execution. For this, Dacite is integrated with the symbolic execution engine Mulib [29]. The idea is that by executing the program symbolically, all paths are systematically traversed, and thus, all reachable data flow is derived. By comparing these with the previously identified passed data flow, it can be assessed which elements were not passed yet.

F4: Visualizing the not-covered data flow in contrast to the passed one within the editor. Analogously to the visualization of passed data flow, the not-covered data flow is visualized as a list next to the source code and with source code highlighting (see Figure 2).

F5: Automatically suggesting JUnit test cases covering the not-covered data flow. By utilizing input-output mappings which are derived from the symbolic execution, JUnit test cases are derived using a test case generator [24] and suggested to the user within the editor.

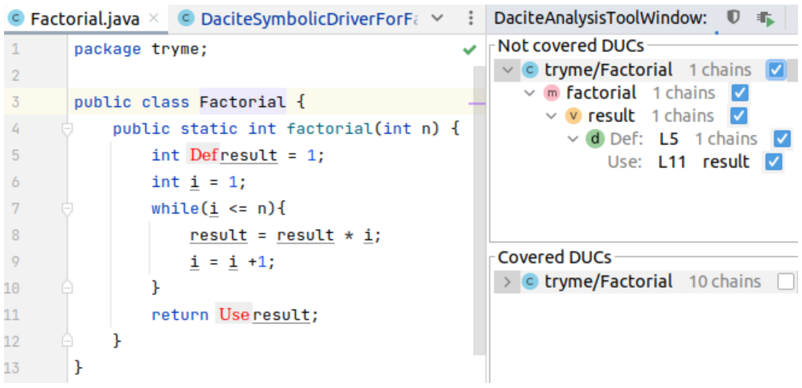


```
package tryme;

public class Factorial {
    public static int factorial(int Defn) {
        int Defresult = 1;
        int Defi = 1;
        while(Usei <= Usen){
            Defresult = Useresult * Usei;
            Defi = Usei +1;
        }
        return Useresult;
    }
}
```

```
Covered DUCs
tryme/Factorial  10 chains ✓
  factorial  10 chains ✓
    i  6 chains ✓
      Def:  L6  3 chains ✓
        Use:  L7  i ✓
        Use:  L8  i ✓
        Use:  L9  i ✓
      Def:  L9  3 chains ✓
        Use:  L7  i ✓
        Use:  L8  i ✓
        Use:  L9  i ✓
    n  1 chains ✓
      Def:  L4  1 chains ✓
        Use:  L7  n ✓
    result  3 chains ✓
      Def:  L5  1 chains ✓
        Use:  L8  result ✓
      Def:  L8  2 chains ✓
        Use:  L8  result ✓
        Use:  L11  result ✓
```

Figure 1: Exemplary screenshot of the Dacite visualization in IntelliJ based on given test.



```
package tryme;

public class Factorial {
    public static int factorial(int n) {
        int Defresult = 1;
        int i = 1;
        while(i <= n){
            result = result * i;
            i = i +1;
        }
        return Useresult;
    }
}
```

```
Not covered DUCs
tryme/Factorial  1 chains ✓
  factorial  1 chains ✓
    result  1 chains ✓
      Def:  L5  1 chains ✓
        Use:  L11  result ✓

Covered DUCs
tryme/Factorial  10 chains ☐
```

Figure 2: Exemplary screenshot of the Dacite visualization for not-covered data flow.

To implement the described functionalities, Dacite consists of different components as shown in Figure 3. The component *Dacite Core* represents the core functionality of executing the dynamic analysis for **F1** and is explained in more detail in Section 4. To facilitate the visualizations for **F2** and **F4**, the LSP is utilized. This approach mitigates the implementation costs of developing integrations for different IDEs by extracting all IDE-independent functionality into a separate component, denoted *Language Server*. The Language Server can then be reused for different IDEs, denoted *Language Clients* by exchanging information with the clients via the established Language Server Protocol (LSP). Figure 3 illustrates these components for Dacite. The Language Server extracts the functionality such as performing the dynamic analysis with Dacite Core or the symbolic execution with Mulib. Dacite provides integrations

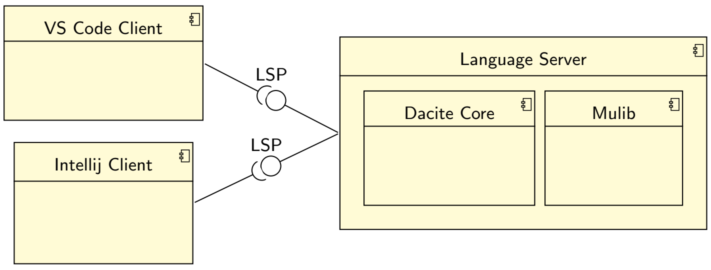


Figure 3: Dacite components for IDE integration via LSP.

for two IDEs, IntelliJ IDEA and VS Code. Therefore, it has two Language Clients, responsible for retrieving the necessary information from the Language Server and visualizing it to the user. This is elaborated in Section 5.

All components are combined within a multi-project Gradle[3] build as visualized in Figure 4 and are implemented in Java except for the VS Code client which was implemented in Typescript. The packages `vscode` and `intellij` represent the implementations for the language clients for VS Code and IntelliJ IDEA respectively. The package `lsp` contains all classes that are needed by both, the language client and the language server, e.g. interfaces for the LSP communication and data structures for the visualization (see Section 5). However, as the client for VS Code is implemented in Typescript as necessary for VS Code the given Java classes of `lsp` cannot be reused. Hence, only the client package `intellij` and the language server implementation in the package `ls` refer to the `lsp` package. Moreover, to be able to initialize the language server, both language clients refer to the `ls` package. The package `core` represents the implementation of Dacite Core of dynamically deriving the passed data flow (see **F1**). As the language server is responsible for starting the dynamic analysis with Dacite Core, it refers to this package. The symbolic execution engine represented by `mulib` is an external package that was integrated into Dacite's functionality. It is imported by Dacite Core for deriving the data flow during the symbolic execution (see **F3**) and the language server for the test case generation (see **F5**).

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

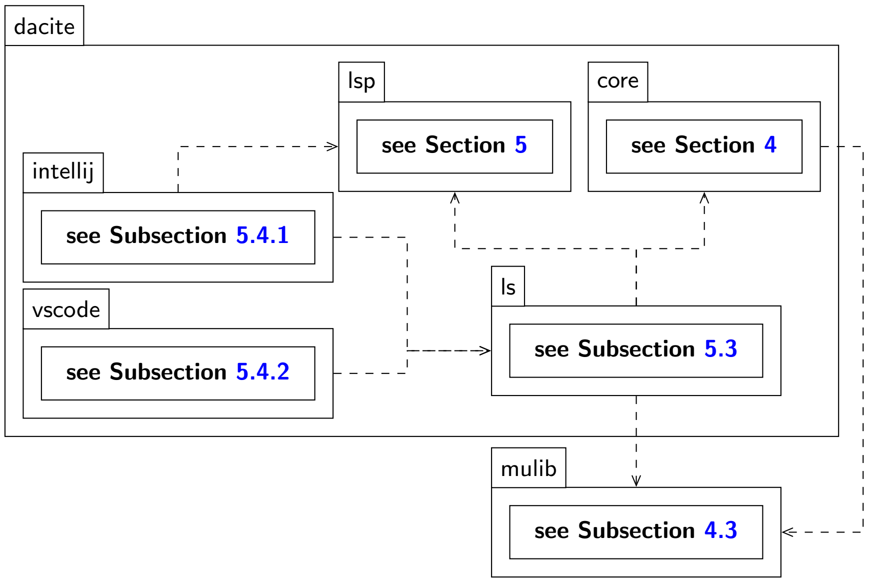


Figure 4: An overview of the Dacite packages and their interdependencies.

4 Dacite Core

Given the architecture depicted in Figure 3, this section aims to outline the implementation of the system component Dacite Core that encompasses the core functionality of the dynamic data-flow analysis in more detail.

Dacite utilizes a dynamic approach to analyze the program, i.e deriving the data flow during the execution, to overcome the limitations of static analysis such as the difficulty of distinguishing reachable DUCs or identifying aliases [22]. Figure 5 gives an overview of the components of Dacite Core. To be able to start the dynamic analysis, Dacite provides the class `DaciteLauncher` with a main method as the general starting point. Four arguments need to be passed to the main method of `DaciteLauncher` to start the analysis:

1. option either *dynamic* or *symbolic* indicating which execution should be triggered.
2. the absolute path of the project, necessary for compiling files before the dynamic execution to take local changes to the files into account for the analysis.
3. package name specifying for which package and its subordinate packages the data flow shall be analyzed and consequently, which classes shall be modified to account for the dynamic analysis.
4. class name of a JUnit Class which should be dynamically executed.

Depending on the given arguments, `DaciteLauncher` forwards the call to the `DaciteDynamicExecutor` for the standard use case of dynamically tracking the data flow during the execution of the program and the `DaciteSymbolicExecutor` for the symbolic execution with Mulib to derive DUCs which were not passed by the given test cases yet and to suggest test cases covering them.

To be able to access the data-flow information during the execution, Java instrumentation is utilized with the open-source framework ASM[4] based on the Java Virtual Machine (JVM) bytecode. However, before

[4]https://asm.ow2.io

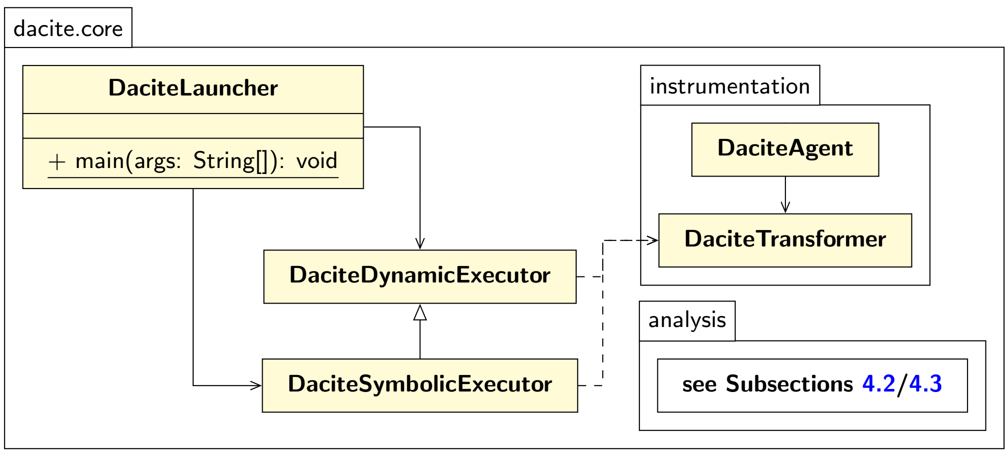


Figure 5: An overview of the Dacite Core implementation.

the classes of the program can be instrumented, it has to be assured that the bytecode files used for the instrumentation are up-to-date. As the Java compiler typically compiles the classes of the program only for execution, local changes to files that were not executed yet would not be considered for the Dacite analysis. Hence, first, the project path and package name are utilized to compile the existing classes into their corresponding up-to-date bytecode files. Afterward, the bytecode instrumentation is performed by automatically adding methods tracking the relevant information for variable definitions and usages to the program based on the bytecode files. The implementation of this bytecode instrumentation with the classes `DaciteAgent` and `DaciteTransformer` (see Figure 5) is elucidated in Subsection 4.1. During the execution of the instrumented code, the methods added by the instrumentation are executed and pass the information to the `DaciteAnalyzer` class of the package `analysis` (see Figure 5) which analyzes the information and derives passed DUCs. This is elaborated in Subsection 4.2. Furthermore, for the symbolic execution, the analysis of Dacite Core has been adapted to be able to deal with symbolic values which is explained in more detail in Subsection 4.3.

4.1 Bytecode Instrumentation

In order to automatically modify the classes of the given program, a Java agent is utilized in the form of the class `DaciteAgent` (see Figure 5). This agent is linked to the start of the program execution via the `DaciteLauncher` with the JVM option `-javaagent`.

With the start of the execution, the Java agent is triggered and invokes a custom subclass `DaciteTrans-former` of the Java-internal `ClassFileTransformer` (see Figure 5). Here, the method `transform` is automatically invoked for every class when it is loaded. Within this method, the method `transform-Bytecode` which takes care of the bytecode instrumentation to facilitate the tracking of the relevant data-flow information is invoked if the given class is located in the specified package. To avoid unnecessary overhead like analyzing internal Java classes and to only focus on the data flow of the program, only the classes defined by the user are instrumented excluding the JUnit test cases as the data flow within a test case does not reflect data-flow coverage of the program and thus, can be neglected [22].

Starting from the bytecode of a class which is given to the `transform` method, ASM is utilized to transform the bytecode into an iteratable structure starting at a `ClassNode`. This `ClassNode` contains different `MethodNodes` representing the methods of this class. Given the first method of a class, instrumentations are added to the beginning of the bytecode pushing the current classname and the classpath as arguments onto the operand stack and calling the `DaciteAnalyzer` method

```
1  public void method1(){
2      // invoking an other
       ↪  method
3      int x = 3;
4      method2(x);
5  }
6  public void method2(int z){
7      System.out.println(z);
8  }
```

(a) Source code

```
1  public method1()V
2      LINENUMBER 3
3      ICONST_3
4      ISTORE 1
5      LINENUMBER 4
6      ALOAD 0
7      ILOAD 1
8      INVOKEVIRTUAL tests/Test.method2 (I)V
9      LINENUMBER 5
10     RETURN
11 public method2(I)V
12     LINENUMBER 7
13     GETSTATIC java/lang/System.out :
       ↪  Ljava/io/PrintStream;
14     ILOAD 1
15     INVOKEVIRTUAL java/io/PrintStream.println
       ↪  (I)V
16     LINENUMBER 8
17     RETURN
```

(b) Java bytecode

Listing 6: Example for inter-procedural DUCs.

`visitSourceCode` with these arguments. This is necessary for analyzing the given information and deriving passed DUCs as elaborated in Subsection 4.2 in more detail.

Iterating the `MethodNodes`, first for each, the specified parameters of the corresponding method are identified [22]. As Dacite facilitates an inter-procedural data-flow analysis, i.e. data flow over the boundary of methods and classes, the method parameter specification represents a potential definition. When the passed parameter values do not refer to variable values but e.g. constants or in case the method was invoked from a class that should not be analyzed, the parameter specification is treated as a new variable definition. In contrast, when a to-be-analyzed class invokes this method with variables given as arguments, these are passed along as parameters to the method so that the corresponding definition of the parameters is identified in the class invoking this method (assuming that this class is contained in the to-be-analyzed package). Listing 6 demonstrates an example for inter-procedural DUCs for which Listing 6a shows the source code which defines two methods while Listing 6b displays the corresponding Java Bytecode for this source code. Within `method1` a variable x is defined (line 3) which is then passed to `method2` as a parameter (line 4). Hence, the definition in line 3 and the usage in line 7 form a DUC although the definition and usage occur in different methods and the variable names are different. When `method2` is called externally or without passing a variable (e.g. `method2(3)`), the specification of the parameter in line 6 represents the definition of z.

To account for the potential definition, the necessary information to be able to identify DUCs has to be passed on to the `DaciteAnalyzer`. In order to associate a definition to a usage, the corresponding variable or element has to be uniquely identified. In JVM bytecode, variables are identified by their index, i.e. the index with which they are stored in the variable table. However, this index is not necessarily unique due to compiler optimization. When a variable is no longer used, the space in the variable table is released, which may lead to the storage of a different variable for the same index. Hence, the value of the variable and the method are used additionally to uniquely identify and associate definitions and usages of variables. To enable users to relate the identified definitions and usages to their program code, the corresponding line number and variable name within the program are stored

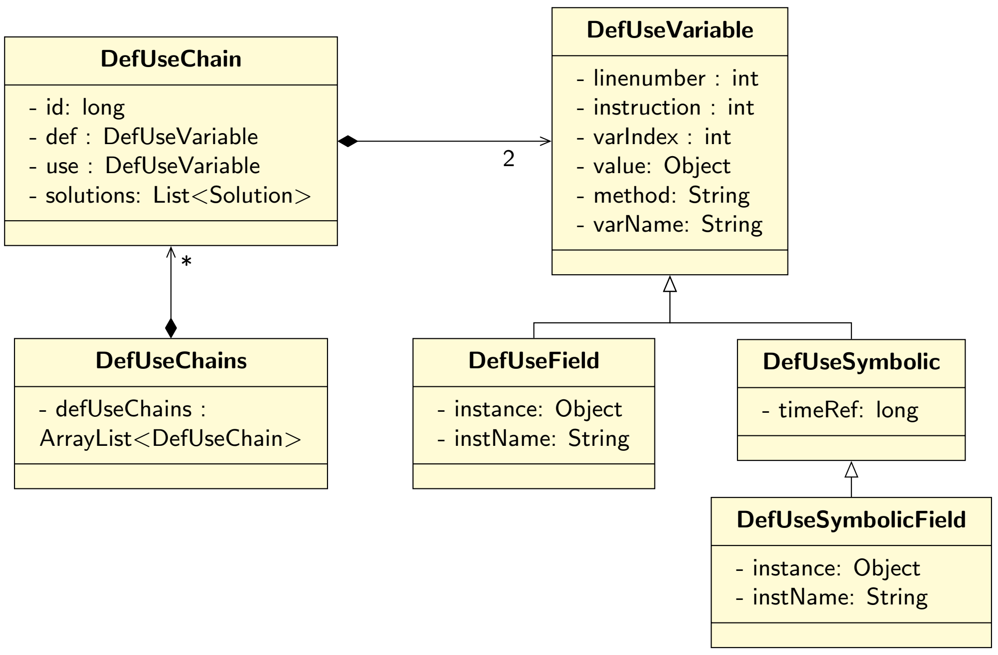


Figure 7: An overview of the DUCs implementation in the package `analysis`. Methods are omitted for the sake of clarity [22].

for each definition or usage [22]. Moreover, to be able to distinguish different definitions and usages in one line, the instruction counter indicates the order of occurrences. The implementation of a DUC within Dacite of the `analysis` package (see Figure 5) can be seen in Figure 7. A `DefUseChain` consists of a definition and usage which are both represented by a `DefUseVariable` with the necessary fields. The other fields `id` and `solutions` are relevant for the test case generation (see Subsection 4.3). A `DefUseField` implements the definition or usage of an object field or array element while `DefUseSymbolic` and `DefUseSymbolicField` are used for the handling of symbolic values during the symbolic execution (see Subsection 4.3).

Given the method parameter, the relevant information passed to the `DaciteAnalyzer` encompasses the value of the parameter, the index where this value is stored in the local variable table, the line number, the current method name, the variable name, and the instruction counter. As the method parameters are stored as the first entries in the local variable table, this information is retrieved by aligning the parameter specifications with the entries in the table. To pass the information to the `DaciteAnalyzer`, new bytecode instructions are added to the code of the method which push the information onto the operand stack and afterward, a method invocation to the corresponding method of the `DaciteAnalyzer` with these parameters is added which retrieves the values of the operand stack as arguments.

Listing 1 shows an excerpt that adds the new bytecode instructions for the method parameters. Therefore, first, a new list of byte instructions is defined in line 1 for which different instruction nodes are added in lines 3-9, e.g. `LdcInsnNode` for pushing a constant onto the operand stack. The method `boxing` (line 3) is used when retrieving the value of a variable of the local variable table. The values pushed onto the stack have to adhere to the type specifications of the called method in line 9 and consequently, to the implementation of the method within the class `DaciteAnalyzer`. For instance, in this case, the method `visitParameter` takes first an Object argument (`Ljava/lang/Object;`) and as a second argument an int (`I`), etc. However, the type of the variable value can differ, e.g. it could be a primitive type such as int or long or an Object. Hence, in order to avoid the implementation of different methods for every possible type of value, the boxing method converts the value based on its type to

```
 1  InsnList methodStart = new InsnList();
 2  ...
 3  boxing(types[parameter], localVariable.index, methodStart, true);
 4  methodStart.add(new LdcInsnNode(localVariable.index));
 5  methodStart.add(new LdcInsnNode(firstLinenumber));
 6  methodStart.add(new LdcInsnNode(classname+"."+mnode.name));
 7  methodStart.add(new LdcInsnNode(localVariable.name));
 8  methodStart.add(new LdcInsnNode(parameter));
 9  methodStart.add(new MethodInsnNode(Opcodes.INVOKESTATIC,
     ↪  "dacite/core/defuse/DefUseAnalyser", "visitParameter",
     ↪  "(Ljava/lang/Object;IILjava/lang/String;Ljava/lang/String;I)V", false));
10  ...
11  insns.insertBefore(firstIns, methodStart);
```

Listing 1: A Java code excerpt inserting bytecode instructions for a method parameter.

its corresponding Object wrapper classes, e.g. int to Integer, so that each variable value pushed on the stack is an instance of the class `Object`. For this, the parameter type is derived from `types` which refers to the array representing the methods parameter types while `parameter` indicates the parameter position e.g. first parameter. Moreover, `localVariable` represents the entry in the local variable table for this parameter and `mnode` is the current `MethodNode` instance. As mentioned before, lines 3 to 9 are used to push the relevant information, i.e. value, index, line number, method name, variable name, and instruction counter to the operand stack. In line 9, the static method `visitParameter` of the `DaciteAnalyzer` is invoked which takes the six previously pushed values as method arguments. In line 11, the list of new byte instructions is added to the beginning of existing method instructions.

After adding the instructions for method parameters, the bytecode instructions of the current method are iterated to find variable definitions and usages for potential DUCs. Based on the different bytecode instructions it can be distinguished whether a variable definition or usage and what type of definition or usage has occurred. Table 1 gives an overview of the different bytecode instructions with their corresponding analyzer methods. There exist four different types of definitions and usages. First, the basic type is loading or storing a variable from the local variable table, e.g. `iload` or `istore` for integer values respectively (see lines 3 and 7 of Listing 6b for variable x), which corresponds to the methods `visitUse` and `visitDef` of `DaciteAnalyzer`. Furthermore, most data-flow tools regard arrays and objects as a whole and thus, associate definitions and usages to form a DUC for with different fields or elements. For instance, considering an object for which field a is defined at some point and field b is used later on, this would be considered a DUC for these tools as both define and use a field of the same object. To allow Dacite to consider DUCs of arrays and objects in more detail, the specific bytecode instructions are differentiated. Hence, another type of bytecode instruction is loading or storing an array element from or into an array, e.g. `iaload` or `iastore` for integer values respectively, which corresponds to the methods `visitArrayUse` and `visitArrayDef` of `DaciteAnalyzer`. For this, next to the information described above and visualized in Figure 7 as `DefUseVariable`, additional information in the form of the instance reference of the array and its name are gathered as well represented as the class `DefUseField`. Analogously, loading and storing object fields corresponds to the bytecode instructions `getfield` and `putfield`, and `getstatic` and `putstatic` for static fields respectively, for which methods such as `visitFieldUse` are added. Another special type of bytecode instruction is the `iinc` instruction. This is utilized for the unary operation incrementing or decrementing a local variable. As this operation includes taking the original value, increasing or decreasing it by a constant, and storing the new value, it incorporates a variable usage as well as a definition, and thus, two calls to `visitUse` and `visitDef` respectively.

To be able to identify inter-procedural data flow, bytecode instructions indicating a new method invocation, e.g. `invokevirtual` for a non-static method invocation as in line 8 of Listing 6b, are tracked by the method `registerInterMethod`. For these methods, available information in the form of the method arguments' values, the invoked method name, and the method name from which this method was invoked are passed to the `DaciteAnalyzer`. This is necessary to be able to relate variables defined in one method and passed on to another as arguments to DUCs. For inter-procedural data flow, the variable name and index typically differ in the method defining the variable and the method using it (see Listing 6). In case more than one argument was given, the method `registerInterMethod` accepts an Object array as an input for the argument values. For this, a helper class `ParameterCollector` was implemented which enables the transformer via static methods to easily collect all values within an array and push it on the operand stack.

Bytecode instruction	**Description**	**Analyzer Method**
`iload, lload, fload, dload, aload`	loading an int, long, float, double or reference variable from the local variable table	`visitUse`
`iaload, laload, faload, daload, aaload`	loading an int, long, float, double or reference value from an array	`visitArrayUse`
`getfield`	loading the field of an object	`visitFieldUse`
`getstatic`	loading the static field of an object	`visitStaticFieldUse`
`istore, lstore, fstore, dstore, astore`	storing an int, long, float, double or reference variable into the local variable table	`visitDef`
`iastore, lastore, fastore, dastore, aastore`	storing an int, long, float, double or reference value into an array	`visitArrayDef`
`putfield`	storing the field of an object	`visitFieldDef`
`putstatic`	storing the static field of an object	`visitStaticFieldDef`
`iinc`	unary operation for incrementing/decrementing values by a constant	`visitUse + visitDef`
`invokevirtual, invokespecial, invokestatic, invokeinterface`	method invocation	`registerInterMethod`

Table 1: Java instrumentation based on bytecode instructions.

Due to the different types of instructions described above, different approaches are necessary to retrieve the relevant information which is passed with the methods to `DaciteAnalyzer`. For instance, the index of the variable can be directly accessed within some instructions, e.g. `istore`, leading to a straightforward invocation similar to Listing 1. In contrast, many instructions, e.g. `iastore` or `getfield`, gather information from already pushed values on the operand stack. To be able to use this information, the relevant data on the operand stack needs to be duplicated to enable its usage as an input to the method call of `DaciteAnalyzer` as well as the original bytecode instruction. There exist bytecode instructions e.g. dup2 to easily duplicate the one or two top values on the operand stack. However, for some instructions more than two values need to be duplicated, e.g. for `iastore` the array reference, the array index, and the variable value are retrieved from the operand stack [16]. In this case, the last three or more values on the operand stack are stored in the local variable table temporarily to be able to push them on the stack again for both the `DaciteAnalyzer` method call and the original instruction call.

After all methods have been iterated and the instrumentation has been completed, the ASM structure is transformed again into a byte array containing the additional methods using the ASM class `ClassWriter`. The result is returned from the `transform` method which is internally forwarded to the JVM and used for the execution of the program.

4.2 Dynamic Analysis

During the execution, the `DaciteAnalyzer` is responsible for collecting and analyzing the passed data-flow information from the instrumentation. The corresponding class is depicted in Figure 8. All fields and methods are static to enable the analysis throughout multiple classes as the bytecode instrumentation is performed for each class individually.

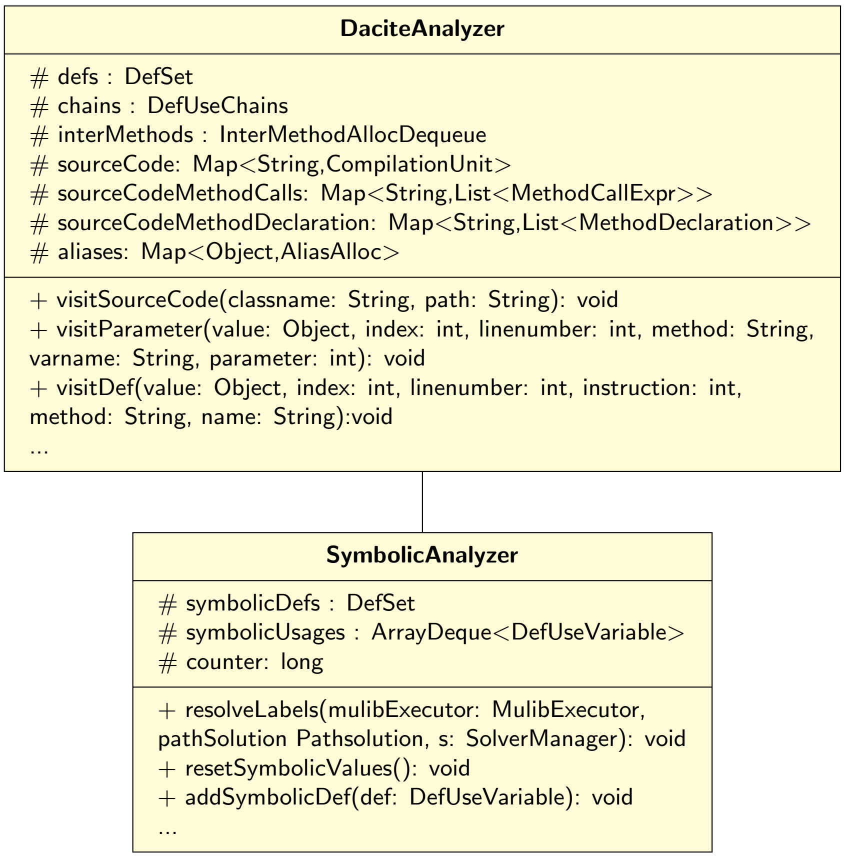

DaciteAnalyzer
# defs : DefSet
# chains : DefUseChains
# interMethods : InterMethodAllocDequeue
# sourceCode: Map<String,CompilationUnit>
# sourceCodeMethodCalls: Map<String,List<MethodCallExpr>>
# sourceCodeMethodDeclaration: Map<String,List<MethodDeclaration>>
# aliases: Map<Object,AliasAlloc>
+ visitSourceCode(classname: String, path: String): void
+ visitParameter(value: Object, index: int, linenumber: int, method: String, varname: String, parameter: int): void
+ visitDef(value: Object, index: int, linenumber: int, instruction: int, method: String, name: String):void
...
SymbolicAnalyzer
# symbolicDefs : DefSet
# symbolicUsages : ArrayDeque<DefUseVariable>
# counter: long
+ resolveLabels(mulibExecutor: MulibExecutor, pathSolution Pathsolution, s: SolverManager): void
+ resetSymbolicValues(): void
+ addSymbolicDef(def: DefUseVariable): void
...

Figure 8: An overview of the analyzer implementations of Dacite in the package `analysis`. All attributes and methods are static, underlines are omitted for the sake of clarity.

As elaborated in the previous subsection, invocations to the `DaciteAnalyzer` methods were added to the inspected program at the appropriate positions during the bytecode instrumentation (see Table 1). Each method invocation then forwards its information to the analyzer which processes the information and stores it in the respective fields.

The analyzer has different fields for collecting the information. The field `defs` collects all variable definitions. A DUC is defined so that on the path from the definition to the usage of a variable there is no other definition. Hence, `defs` is implemented in such a way that the most recent definition on the executed path comes first. This allows for a quick allocation of a variable usage to its most recent definition [22]. A new definition is added for instance when the method `visitDef` is invoked. The second field `chains` comprises all DUCs that were identified until the current execution point. This is represented as `DefUseChains` (see Figure 7). Each time a new usage is registered, the corresponding most recent definition is retrieved and together they are added as a new DUC to `chains` [22].

The other fields are helper fields to allocate inter-procedural data flow and aliases. The field `inter-Methods` contains the allocation of inter-procedural variables from method arguments on the one side to method parameters on the other, for instance mapping the variable x of `method1` to variable z of `method2` in Listing 6. A new allocation is added every time for every argument when a method is invoked with the information from the calling method (with the `DaciteAnalyzer` method `registerInterMethod`). This information includes only the arguments' values and not their variable names or indexes as only this information is available on the operand stack for the method invocation instructions (see Table 1). Hence, this information needs to be complemented with the information gathered from the invoked method.

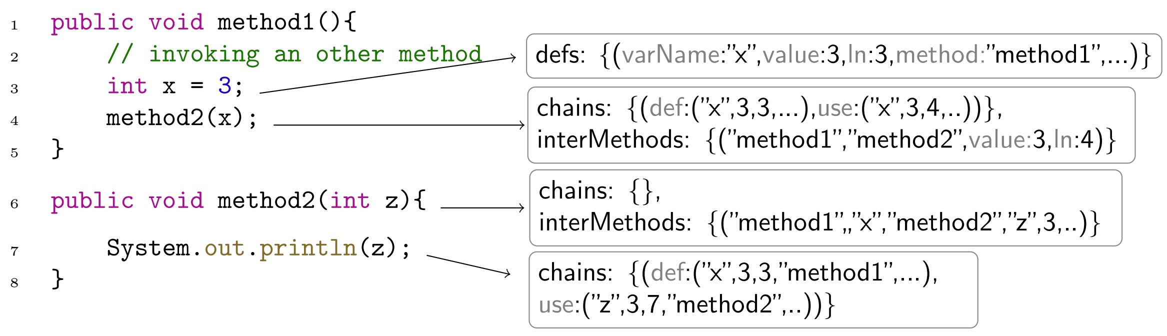
Whenever a new method is entered via `visitParameter`, the procedure `registerParameter` is triggered as shown as Algorithm 1 which handles the data flow of method parameters by either adding a definition for the parameter or adding it to an inter-procedural variable allocation. Therefore, first, `interMethods` is checked whether there exists an entry matching the method name and parameter value indicating that this method was invoked from within the to-be-analyzed package and the parameters may be variables that have been defined elsewhere (lines 2-3). To be able to allocate variable definitions and usages over the boundaries of these methods, the variable name and index of the variable have to be identified on both method sides, e.g. on the calling method side in `method1` for variable x and on the invoked method side in `method2` for variable z in Listing 6a. However, as method invocation instructions do not provide this information, the variable name and index of the calling method (e.g. variable x Listing 6b) have to be gathered with another approach. When a method is invoked e.g. via `invokevirtual`, its arguments are pushed onto the operand stack beforehand. If these arguments entail variables, e.g. x in line 3 of Listing 6a, the arguments are pushed by the bytecode instructions specified in Table 1 depending on the type of variable (e.g. `iload` for the int value of variable x in line 7 in Listing 6b). However, determining which bytecode instructions are responsible for pushing the relevant variable information on the operand stack as input for a method invocation is not always as straightforward as in the example in Listing 6. Pushing one method argument to the operand stack can contain a potentially large number of bytecode instructions e.g. when a calculation or another method invocation has to be made first. Consequently, this is difficult to identify during a dynamic analysis as it would require potentially complex backtracking through previous bytecode instructions.

Algorithm 1 Abstract procedure for handling parameters upon entering a new method

```
1:  function registerParameter(Object $value$,int $index$,int $ln$,String $methodname$,String $varname$,int $parameter$)
2:      for each $alloc$ in interMethods do
3:          if matching allocation for this method exists then
4:              String $name$ = getVariableNameFromSourceCode($methodname$, $alloc$.ln, $parameter$)
5:              DefUseVariable $usage$ = chains.findUse($alloc$.currentMethod, $alloc$.linenumber, $value$, $name$)
6:              if $usage$ != null then        // There exists an inter-procedural allocation for a variable.
7:                  $alloc$.setNewIndex($index$)
8:                  $alloc$.setNewName($varname$)
9:                  $alloc$.setCurrentIndex($usage$.getVariableIndex())
10:                 $alloc$.setCurrentName($usage$.getVariableName())
                    return
11:     registerDef($value$, $index$, $ln$, $methodname$, $varname$) // If no allocation exists, the parameter is a variable definition.
```

Hence, for this, the `sourceCode` field is utilized which collects a map of class names and their corresponding `CompilationUnit` of the `JavaParser`[5] library. This library provides an efficient source code analysis transforming Java Code into an Abstract Syntax Tree (AST). Each time a method in a new class is executed, `visitSourceCode` is invoked and the source code of this class is statically parsed with the `JavaParser` obtaining the AST of the code. The result is saved in the map for further processing. This information from a static AST parsing analysis can easily enhance the dynamic analysis in two scenarios. First, by providing the variable name for a method invocation as described previously. So whenever `visitParameter` is called and there exists an entry of `interMethods` matching the method

[5] https://javaparser.org



```
1 public void method1(){
2     // invoking an other method
3     int x = 3;
4     method2(x);
5 }
6 public void method2(int z){
7     System.out.println(z);
8 }
```

defs: {(varName:"x",value:3,ln:3,method:"method1",...)}

chains: {(def:("x",3,3,...),use:("x",3,4,..))},
interMethods: {("method1","method2",value:3,ln:4)}

chains: {},
interMethods: {("method1",,"x","method2","z",3,..)}

chains: {(def:("x",3,3,"method1",...),
use:("z",3,7,"method2",..))}

Listing 2: Simplified example how DUCs are derived with the `DaciteAnalyzer`.

name and parameter value, the method invocation at the corresponding line number is tracked within the AST and the corresponding argument String representing the potential variable name at the given argument position (`parameter`) returned (line 4 of Algorithm 1). In the example of Listing 6a, this would include identifying the AST element for the method invocation `method(x)` in line 4 and retrieving the String value for the method argument, in this case `x`. To reduce the overhead of analyzing the `CompilationUnit` for method calls every time, all method calls for a class are cached within the field `sourceCodeMethodCalls`. After extracting the argument name, it is then further utilized with the other information to check whether there exists a variable usage within the field `chains` identified when pushing the variable value to the operand stack as input for the method invocation (e.g. line 7 in Listing 6b) represented by line 5 of Algorithm 1. If there exists a DUC with a variable usage for this argument in `chains` indicating that there is an inter-procedural data flow for this variable, then this DUC is removed from `chains`. For inter-procedural data flow usages at method invocations only pass the variables to the new method and thus, are not regarded usages at this point. However, when regarding method invocations to external methods in not to-be-analyzed packages these usages remain within the field `chains` (e.g. for the usage in line 7 in Listing 6. In case of an inter-procedural data flow, the corresponding allocation of `interMethods` is complemented by the information such as the variable name and index for both sides, the calling method and the invoked method, to be able to match future usages to the inter-procedural definition (lines 6-10 of Algorithm 1). If no allocation exists for this parameter, e.g. because the method argument was not a variable or the method invocation was not within the to-be-analyzed package, then the parameter specification is treated as a definition (line 11 of Algorithm 1).

Listing 2 illustrates a simplified example of how the DUCs are derived with the data structures of `DaciteAnalyzer`. First, during the dynamic analysis a definition is identified at line 3. Therefor, a corresponding entry is added to the field `defs` of the of `DaciteAnalyzer`. Afterwards, a usage is identified in line 4. For this, the last definition is retrieved from `defs` (line 3) and together, the definition and usage is saved within `chains`. Moreover, in line 4 a method invocation to a method within the to-be-analyzed packages takes place. Hence, the `registerInterMethod` call is invoked which adds an allocation to `interMethods` with the available information. When this invoked method is entered in line 6, the procedure of Algorithm 1 is performed. As a corresponding entry in `interMethods` exists for this inter-procedural data flow, the variable name of the passed variable is retrieved with the AST (see line 4 in Algorithm 1), in this case the name x (line 4 in Listing 2). With this name, the usage constituting the variable passing between the methods in line 4 is identified within `chains`. To account for the inter-procedural data-flow, the corresponding DUC (def:("x",3,3,...),use:("x",3,4,..) see line 4) is removed from chains as this parameter passing to `method2` is not considered a DUC for inter-procedural data flow. Furthermore, the entry in `interMethods` is augmented with the variable information at `method1` and `method2`. Finally, a usage is identified in line 7 and the corresponding definition retrieved via the `interMethods` entry. For this inter-procedural DUC from line 3 to line 7 in Listing 2, an entry is stored within `chains`.

The second use case for the AST analysis is the identification of the line number for the parameter specification. As it can be seen in Listing 6b, within bytecode the line number is specified for the start of a method's content. However, the line number of the method specification in the source code (e.g. line 1 of Listing 6a) cannot be exactly identified within bytecode as between the first bytecode instruction of a method and the method specification can lay an arbitrary number of lines e.g. with source code comments (see Listing 6). However, as a method and the corresponding parameter specification is a potential definition and to enable users to relate this data flow information to the program source code and enhance the comprehensibility, the exact line number is essential. Thus, whenever a new parameter specification is invoked (`visitParameter`), the corresponding `MethodDeclaration` is retrieved from the AST and its line number is retrieved. Analogously to the method calls, all method declarations for a class are cached within the field `sourceCodeMethodDeclaration` to reduce the overhead of extracting the declaration for every parameter from the `CompilationUnit`.

Finally, the field `aliases` collects variable aliases pointing to the same object instance. For each object for which at least one alias exists, `aliases` keeps track of the different variables pointing to this instance in the form of variable name, index, and method name. So whenever a variable definition for an object is registered, it is checked whether there already exists an alias allocation or if not, there exists a different variable defining the same object instance. For the latter, a new variable alias allocation is added to `aliases`. When a new variable usage for an object is registered and its corresponding definition is retrieved, it is checked if there exists an alias allocation. If that is the case, the most recent definition of the aliases is determined and compared to the variable definition to identify the last definition of the object for the DUC.

After the execution when all passed DUCs have been identified, the result is parsed and stored in an XML file `coveredDUC.xml` in the project directory. This can then be further processed e.g. for the integrated visualization via LSP.

4.3 Analysis during Symbolic Execution

As depicted in Figure 5, there are two types of Dacite executions: the standard use case of dynamically tracking the data flow during the execution of the program and the symbolic execution and analysis to derive DUCs that were not passed by the given test cases yet and to suggest test cases covering them. For the latter, the symbolic execution engine Mulib is utilized as an external library. As the description of Mulib's internal workings and symbolic execution is out of scope for this report, corresponding literature can be found in [29, 30].

To combine the symbolic execution of Mulib with the dynamic analysis of Dacite Core, the instrumented bytecode files containing the additional methods sending data flow information to the analyzer of Dacite are forwarded to Mulib. This way, during the symbolic execution for all executed paths data flow information is collected within the analyzer, and DUCs are derived. However, some adaptions to the Dacite analysis process had to be made to enable this tracking of DUCs during the symbolic execution, i.e. the handling of values that do not have a concrete value yet but are symbolic and only contain a set of constraints e.g. $x > 1$.

As symbolic values within Mulib are represented by subclasses of Object classes (for more detail see [29]), the bytecode instrumentation does not have to be adapted as it assumes variable values are Objects (see Subsection 4.1). However, deriving DUCs from symbolic values requires adaptations. When dealing with symbolic values, it is not directly possible to identify if two values are the same as they have no concrete value and thus, cannot be compared. Consequently, assigning a variable usage to its last definition is also not always directly possible as it is not clear whether the definition and usage concern the same value and thus, correspond to the same variable [24]. This poses the challenge that not all definitions and usages can be directly compared and thus, some usages cannot be assigned to the corresponding definition immediately. The DUCs for those symbolic values that cannot be compared can only be derived at a later point when the symbolic values have been resolved to concrete ones. Hence, a different handling has to be implemented for such values denoted here as `potential DUCs`. For this, a second analyzer class `SymbolicAnalyzer` is utilized to collect the information for the symbolic values. Therefore, instead of directly relating a usage to its definition as for concrete values (see Subsection 4.2), symbolic usages and definitions are collected separately with the fields `symbolicDefs` and `symbolicUsages`. Only when a leaf node of the symbolic execution is reached, Mulib is able to provide concrete values for the symbolic variables. It then invokes the method `resolveLabels` of the `SymbolicAnalyzer` to resolve the symbolic values for the found definitions and usages so that they can be compared and DUCs derived [24].

Reaching the leaf node of the symbolic execution implies that an execution path has been completed. To map a usage to its last definition retrospectively, an indicator of time is necessary to identify which definition was executed before the usage. Therefore, the field `timeRef` is introduced for symbolic variables in `DefUseSymbolic` (see Figure 7). It represents the sequence of definitions and usages by increasing the field `counter` of `SymbolicAnalyzer` whenever a definition or usage is encountered. When a definition or usage is defined, `timeRef` is set to the current value of the `counter`. With this, the last definition can be identified and the generated DUCs stored within `chains` of `DaciteAnalyzer`.

Furthermore, for all DUCs derived at this execution path, a `Solution` object including the input and output values for this path (for more detail see [29]) is added to account for the test case generation later on (see `solutions` in Figure 7). So for each DUC, a set of `Solution` objects can be derived as a DUC can be covered by more than one combination of input and output values and its corresponding execution path. Moreover, vice versa, for each `Solution` object it can also be derived which DUCs are covered on its path to reduce the solutions and consequently the test cases to a minimal set covering all identified DUCs. The result of the analysis is saved as an XML list of derived DUCs and `Solution` objects to be utilized for further purposes such as the processing for the visualization.

5 Language Server Protocol

To facilitate the comprehensibility of the identified data flow as explained in Section 4, a graphical visualization of the output of Dacite is needed. In order to provide direct feedback for the considered test cases and the program and support the user in the test development process without the overhead of installing a separate tool, integration of the visualization into the development environment is desirable. However, because of market segmentation, there does not exist one mainly used environment but a variety of IDEs that are utilized by different users (e.g., Visual Studio Code vs. IntelliJ). The Language Server Protocol (LSP) is an approach to reduce the implementation costs for developing a Dacite integration and visualization for different IDEs separately by extracting IDE-independent functionalities such as the dynamic analysis from Dacite Core (see Section 4) from the editor into a separate component, the *language server*. This way, this centralized implementation can be reused for different *language clients* [23].

This section aims to explain the implementation of the Dacite integration with the LSP in more detail. Therefore, first, in Subsection 5.1 the utilized LSP message types are introduced. Afterward, Subsection 5.2 elaborates on the interactions between the user, language client, and language server. Next, Subsection 5.3 elucidates the implementation of the language server while Subsection 5.4 describes the language clients, i.e. the editor integrations for IntelliJ IDEA and VS Code, in more detail.

5.1 LSP Message Types

The LSP offers a remote procedure call protocol JSON-RPC [11] for communicating between the language client and server which are separate processes exchanging messages based on standard input and standard output of the data interchange format JavaScript Object Notation (JSON) [21, 23]. Therefore, the LSP specification [14] provides a set of standard message types that are supported by existing language clients and thus, offer a basic functionality. Hence, to decrease the implementation costs of IDE integrations using the LSP, it is desirable to utilize as many standard message types as possible to rely on existing language client implementations.

Before deciding which message types can be utilized, it has to be specified how the data flow shall be visualized. To comprehensibly visualize the process of Dacite, three user interface elements are necessary:

1. The user requires some form of additional interaction with the IDE to start a Dacite process e.g. the dynamic or symbolic analysis. This can be in the form of an executable link or button [23].
2. The list of identified DUCs should be displayed to the user to give an overview of how many and which DUCs were identified. This list should be sorted by the respective class, method, and variable the DUC was defined for and expandable to increase the comprehensibility e.g. for large programs with many DUCs [23].
3. Next to the list of DUCs, the DUCs should be highlighted within the code editor to enable the user to directly relate the data-flow information with the source code. To avoid convoluted highlighting, the user can enable and disable the DUC highlighting based on the list of DUCs, e.g. only highlighting one DUC. Moreover, to make the highlighting more understandable, different variables should be highlighted in different colors to facilitate visual differentiation [23].

To implement the described user interface elements for Dacite using the LSP and the communication between the language server and client, different LSP standard messages can be utilized:

- `initialize`: Communication is initiated when the user opens a Java file in the IDE, i.e. the client, which starts the server process. After starting the server, the client sends an `initialize` request to the server to exchange its capabilities. With these, the client and server specify which protocol features they support such that the subsequent communication is tailored to the features implemented by both [14, 23].

- synchronization messages(`didOpen`, `didChange`, `didClose`): One task of the server is to perform code analysis e.g. for deriving the positions for code annotations. Therefore, it caches the file content. Consequently, this has to be synchronized with the client via the dedicated notifications `didOpen`, `didChange`, and `didClose` sent from the client whenever a Java file is opened, changed, or closed. Within this notification, the client sends the file's Uniform Resource Identifier (URI) and content to the server [14, 23].

- `codeLens`: A code lens describes positions in the editor from which actions or command executions can be triggered e.g. by providing a link or button. A client can request code lenses by sending a corresponding request to the server containing the file's URI. The server determines if and where the file contains a code lens and returns the position with the underlying command which should be triggered upon interaction. For Dacite, this can be utilized for the first user interface element described above triggering the dynamic analysis and the symbolic analysis [14, 23].

- `executeCommand`: When a command is triggered e.g. when a user clicks on a code lens link, the client sends the request `executeCommand` with the respective identifier to the server which triggers an internal process such as the dynamic analysis or test case generation [14, 23].

- `inlayHint`: Annotations rendered in place with the source code in the IDE editor are called inlay hints in the LSP. A client can obtain such hints using an `inlayHint` request specifying a file's URI and a range within the source code. The server determines the exact positions where the annotation should be displayed and returns a list of inlay hints containing a label and position. This can be utilized for the third user interface element as described previously [14, 23].

- `workspaceEdit`: A workspace edit represents changes to the workspace of the IDE sent from the server to the client. This can either be changing a file's content or creating, renaming, or deleting files. For Dacite, this is utilized to create new files for the symbolic driver and test cases and write its content (see Subsection 5.3) [14].

Next to the message types provided by the LSP standard as described above, Dacite makes use of custom extensions to provide a more comprehensible data-flow visualization. First, inlay hints by the LSP are not styled by default. However, distinguishing different DUCs by color would improve usability. Hence, Dacite defines the LSP extension `InlayHintDecoration`. By sending an `InlayHintDecoration` request from the client to the server with an inlay hint position, the server will return the corresponding color and font style with which the inlay hint should be displayed.

Moreover, to visualize the second user interface element, i.e. the list of identified DUCs, Dacite utilizes the tree view protocol specified by [12]. This protocol specifies protocol messages enabling the client to render tree views (see Figures 1 and 2). Although this protocol is not part of the LSP standard at the time of writing, there exist implementations for the tree view protocol at the server as well as the client side making the adoption into the Dacite integrations easier [12]. A tree view consists of a set of *tree view nodes*. Every node can be identified using a node URI and may have child nodes. Additionally, a *tree view command* can be associated with a node enabling the client to perform actions based on user interactions with the node e.g. by clicking on the element [23]. The tree view protocol specified different message types [12]. For Dacite, the message type `treeViewChildren` is the most relevant. With this, clients can request tree view nodes belonging to a given parent node by sending the request specifying the parent node URI to the server. In order to request a root node, the node URI is left empty. The server responds with a list of tree view nodes. Subsequently, the client can request the children of all newly obtained nodes and thus, iteratively render the tree view [12, 23].

5.2 Workflow

After introducing the different LSP message types used for Dacite, this section explains how they are utilized for Dacite's interaction workflow. As visualized in Figure 3, the Dacite prototype consists of different components interacting with each other:

- **Language Client**: represents the integration into the respective IDE and interacts with the user via its graphical interface. For Dacite, two IDE integrations are available, Intellij IDEA and VS Code (see Subsection 5.4).
- **Language Server**: component for IDE-independent functionalities. For Dacite, this encompasses all internal functionalities such as analyzing the program, transforming data into the visualization structure, and determining exact source code positions for annotations (see Subsection 5.3).
- **Dacite Core**: performs the dynamic analysis to derive DUCs and interacts with Mulib to derive not-covered DUCs (see Section 4).
- **Mulib**: performs the symbolic execution to derive not covered DUCs (see Subsection 4.3) and the test case generation and reduction (see Subsection 5.3).

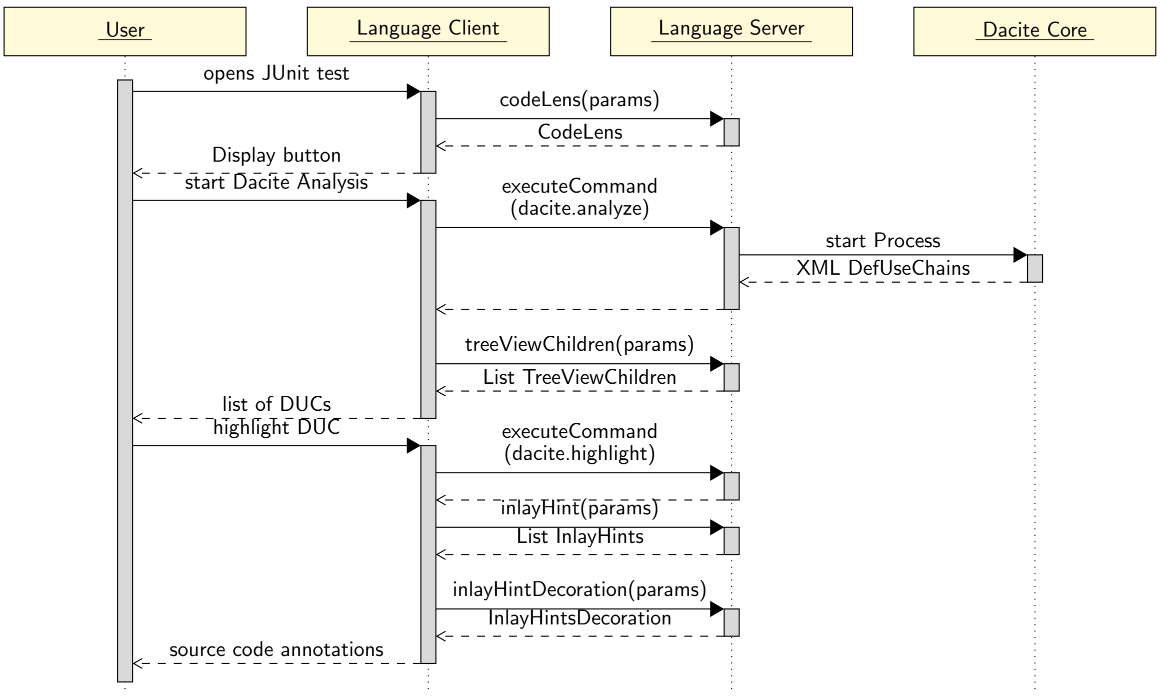


Figure 9: Sequence diagram illustrating the interaction between the user and the components, the client, the language server, and Dacite Core for the dynamic analysis [23].

Figure 9 illustrates the communication based on the user interaction between the different components for the identification and visualization of passed DUCs given an initial test set (see **F1** and **F2** in Section 3). As the symbolic execution engine Mulib is not needed for this, it is omitted for the sake of clarity in this figure. The LSP lifecycle and synchronization messages between the client and server (e.g., the initialization upon the start of the IDE) are also omitted for the sake of clarity. Each time the user opens a Java file in the IDE, the language client sends a `codeLens` request to the language server. This analyzes whether the corresponding file contains a JUnit[6] test which is needed to start the dynamic analysis of Dacite. If this is the case, a code lens is returned containing the display position at the start of the JUnit class for which it is displayed as an interaction for the user (e.g., in the form of an executable link or button). When the user triggers this interaction (e.g., by pressing the button), the client sends an `executeCommand` request for the Dacite analysis to the language server. This starts the instrumentation and analysis process of Dacite Core for the JUnit test cases and the corresponding class files (see Section 4). It returns an XML file containing all identified DUCs as `DefUseChains` (cf. Figure 7). This output is stored and transformed by the language server to account for the tree-like structure grouping all chains based on their class, method, variable, and definition (see Figures 1 and 2 on the right side). This way, when the client requests the tree view children for the root node afterward, the language server returns subsequently the grouped DUCs as tree nodes. These are displayed by the client to the user [23].

Thereafter, the user can interact with the visualized list of DUCs, e.g. by clicking on a tree view node or checking a box, to enable the source code annotations for all related DUCs. For instance, when clicking on the method in the tree view, all related DUCs that were defined in this method are highlighted within the editor. So when the user triggers this source code annotation, the client sends an `executeCommand` request to the language server specifying which tree view node was clicked. The language server internally saves this information, so that when the client sends the `inlayHints` request for the document subsequently, the language server returns the position of all DUCs which should be annotated in the editor. After the request for inlay hints, the client requests for each inlay hint the corresponding decoration containing the color with the custom request `inlayHintDecoration`. With this information, the annotations are rendered within the source code editor where they are visible to the user [23].

After deriving the DUCs that are covered by the existing JUnit test cases via the Dacite integration, the user can further interact with Dacite to derive those DUCs that are not covered yet, visualize them, and derive test cases covering those DUCs (see **F3**, **F4** and **F5** in Section 3). Figure 10 illustrates the interaction between the user and the components of this process. First, to be able to symbolically execute a program with Mulib, a symbolic driver method is required which defines the method that shall be symbolically executed (denoted Method Under Test (MUT)) and specifies the potentially symbolic input values for this method [29]. However, developing such a method requires domain knowledge about the methods of Mulib and its symbolic execution. To facilitate users without this knowledge to derive which DUCs are not covered by the given JUnit test cases, this driver method is automatically generated by Dacite and suggested to the user (see Subsection 5.3 and Figure 15). Hence, to start the process, the user has an additional button within the IDE window for Dacite. Upon pressing this button, the client sends an `executeCommand` request to the language server triggering the generation of the symbolic driver needed for the symbolic execution. Afterward, the generated String is passed from the server to the client with the `workspaceEdit` request resulting in the creation of a new file with the given name and content. For this new file, the client sends a `codeLens` request to the language server which analyzes whether the given file corresponds to the naming standard of the symbolic driver. If this is the case, a corresponding code lens is returned and displayed along with the new file to the user. When the user triggers the code lens to start the symbolic analysis, the language client sends an `executeCommand` request to the language server which starts the symbolic execution of Dacite Core (see Subsection 4.3). Dacite Core in turn instruments the to-be-analyzed classes and starts the symbolic execution with the engine Mulib. During the execution with Mulib, it sends passed DUC information and input-output-mappings as `Solution` objects to Dacite Core which derives the corresponding DUCs. After the symbolic execution terminates the identified DUCs and solutions are saved as an XML file. These DUCs are read by the language server and compared to the DUCs derived from the previous dynamic analysis (see Figure 9) to identify the DUCs not covered by the given test cases. These are saved internally and transformed into the tree view structure grouping all chains based on their class, method, variable, and definition. When the client then requests the `treeViewChildren` subsequently, the language server returns the grouped DUC which the client renders and displays to the user similar to the process of the covered DUCs. Additionally, the highlighting of DUCs is similar to the previous

[6]https://junit.org

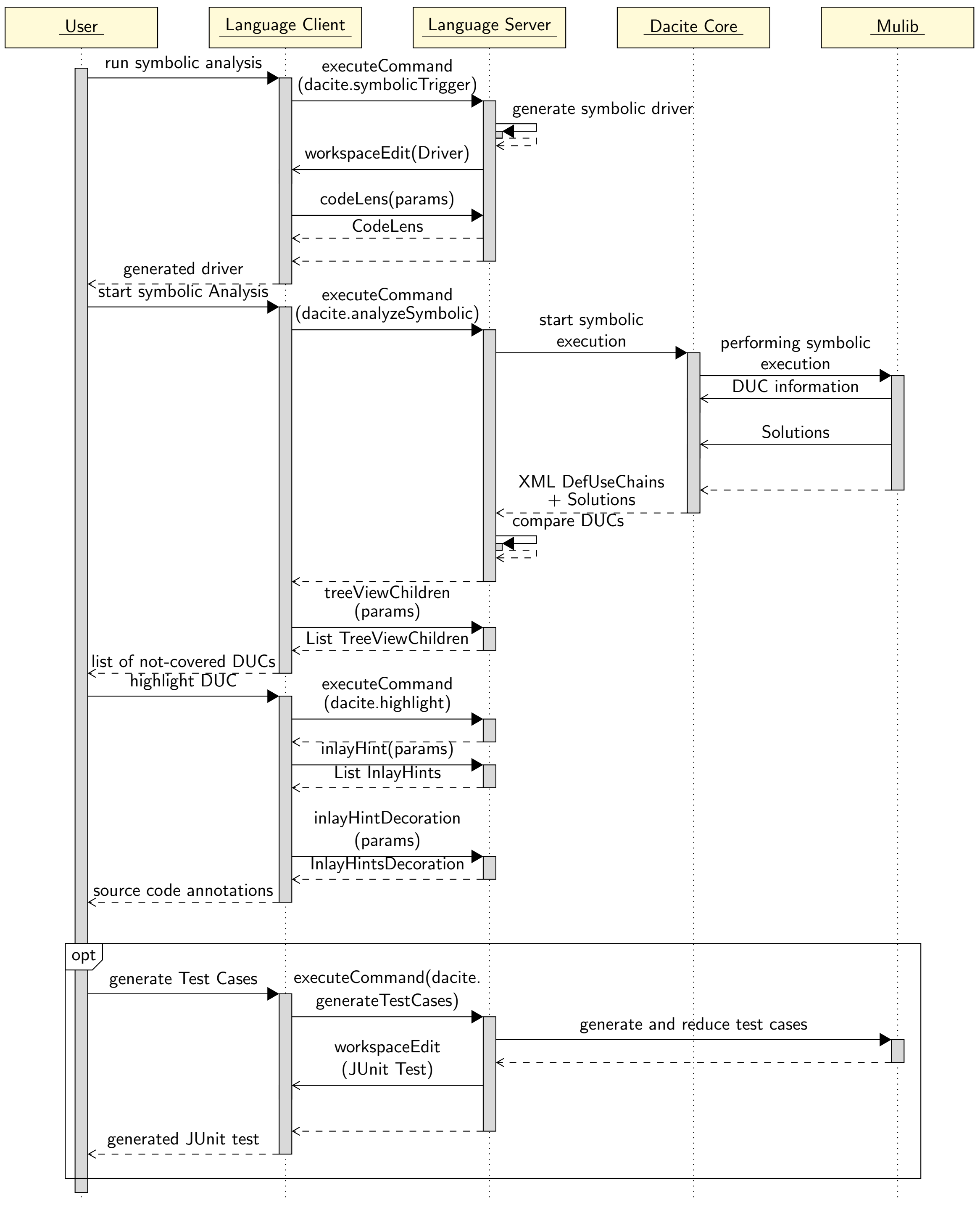


Figure 10: Sequence diagram illustrating the interaction between the user and the components, the client, the language server, Dacite Core, and Mulib for the symbolic execution and test case generation.

process (see Figure 9) with the requests executeCommand, inlayHint, and inlayHintDecoration. Whereas the identified covered DUCs are displayed in different colors except red depending on the variable, the not-covered DUCs are highlighted in red within the editor to enable a straightforward distinction.

After the display of DUCs that were not covered yet, the user has the opportunity to utilize Dacite to derive test cases covering these chains. Therefore, an additional button is displayed within the IDE tool window for Dacite. When the user presses this button, the client sends an executeCommand request to the language server which triggers the test case generation process. For this, the language server first identifies the distinct set of input-output mappings as Solution objects representing the test cases. The result is forwarded to Mulib to reduce the test cases based on covering all possible DUCs (the *all-uses* criteria based on [9]). Afterward, a string representing a JUnit class is generated based on the remaining test cases. This is sent from the server to the client with the workspaceEdit command which results in the creation of a new file with the given name and content which is displayed to the user.

5.3 Language Server

The language server extracting the IDE-independent features into a separate component was implemented in Java based on the lsp4j[7] library providing Java bindings for the LSP. This way, the work needed for realizing standard protocol messages could be reduced to implementing appropriate interfaces. Additionally, custom interfaces were registered to account for the required LSP extensions mentioned in Subsection 5.1. All details needed to enable the communication based on the JSON-RPC protocol are handled by the lsp4j library (e.g., serializing and deserializing JSON) [23].

Figure 11 illustrates the implementation of the language server and the corresponding interfaces. As explained in Section 3, the package lsp contains classes required for the LSP on both the language client and server side while the package ls entails all classes necessary for the language server. The language server is started as a separate process by the language client via the main method of the ServerLauncher. The launcher creates a new object of the class DaciteLanguageServer and connects it to the interface DaciteExtendedLanguageClient via the lsp4j library to enable the exchange of messages. Similarly, an interface for the language server DaciteExtendedLanguageServer exists which is utilized to connect the client to the server on the client side. The DaciteLanguageServer has two attributes, the DaciteTextDocumentService and the DaciteWorkspaceService, which implement the given interfaces of the lsp4j library and are responsible for handling the processing of the LSP messages in the domain *textdocument* (e.g. didOpen or inlayHint) and *workspace* (e.g. executeCommand) which are relevant for the Dacite exchange (see Figures 9 and 10).

Hence, the DaciteWorkspaceService is responsible for implementing two message types, the executeCommand and the workspaceEdit. As the latter is a notification from the language server to the client and thus, the processing is done on the client side, the existing lsp4j implementation can be utilized. Consequently, only the LSP message type executeCommand needs to be implemented by Dacite to account for the custom commands e.g. dacite.analyze. For this, the command identifier and parameter are forwarded to the class CommandRegistry which distinguishes the following commands:

dacite.analyze: This command triggers the dynamic analysis of Dacite (see Figure 9) as a new Java process. To invoke the dynamic process of Dacite Core, the corresponding arguments need to be specified: project path, package name, and class name (see Section 4). In order to derive these values, the class CodeAnalyzer is utilized which encapsulates the source code parsing and analysis with the JavaParser library. Therefore, the file's content is given to the CodeAnalyzer with the TextDocumentItemProvider which maps and stores for each file the URI (given as a parameter to the command) and its corresponding content. With this automatic analysis, the package name to define the scope of the dynamic analysis is inferred from the triggered JUnit class. However, the classes the

[7]https://github.com/eclipse/lsp4j

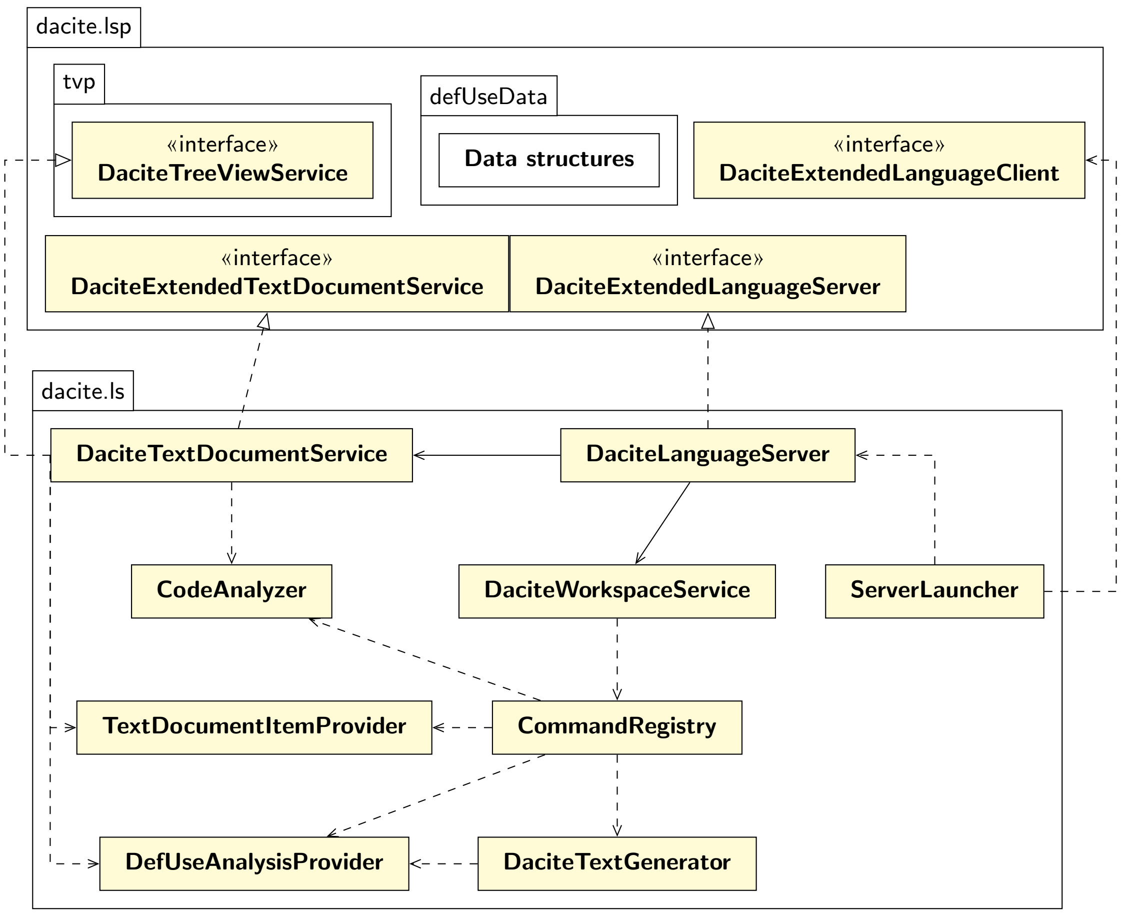


Figure 11: An overview of the language server implementation in the packages `ls` and `lsp`. Methods and attributes are omitted for the sake of clarity as well as dependencies and associations to the data structures of `lsp.defUseData` as the majority of the `ls` classes refer to these.

user wishes to analyze may be located in a different package. Hence, users can define a config file named `Dacite_config.yaml` within the Java project where they can specify a deviating package name which is automatically retrieved and replaces the default package name inferred from the JUnit test. Additionally, to be able to access the classes for the instrumentation and execution within the new Java process, the server has to make the classes of the project that should be analyzed available to the analysis process. Hence, the server collects directories with `*.class` files as well as `*.jar` files and adds these to the class path of the Java process [23]. After the analysis, the `CommandRegistry` invokes the `DefUseAnalysisProvider` to parse and transform the identified DUCs into the corresponding structure for the visualization. To enhance the overview of the list of DUCs and facilitate usability, the DUCs are sorted by their definition, variable, method, and class where the definition was made. Figure 12 shows an overview of this representation within Dacite. Each `DefUseClass` has a list of methods as `DefUseMethod` which has a list of variables as `DefUseVar` defined in this method. All of these classes inherit the common features from the abstract class `DefUseStructure`. Each `DefUseVar` has a list of definitions made for this variable represented by the class `Def` which again has a list of uses for this definition (class `Use`). Each definition and usage has a set of attributes such as the line number, whether it is currently highlighted within the editor, or the highlighting color, extracted by the abstract class `DefUseElement`. During this transformation, a distinct color is given (with the field `color`) to all definitions and usages of a variable to facilitate the visual differentiation between the variables. However, selecting a maximally distinct set of colors for larger numbers of variables is a non-trivial problem. To approach this problem a list of colors was generated[8] with respect to the CIE color distance so that subsequent colors have a maximal distance [1]. All DUCs for the same variable are then associated with one color taken from the list. The transformed result as a list of `DefUseClass` objects is stored within the provider class for further reference.

dacite.highlight: This command takes as an input parameter the tree view node properties for which the highlighting was invoked by the user. This can be a whole class, a method, a variable, or a single DUC. The `CommandRegistry` invokes the corresponding static method of the `DefUseAnalysisProvider` which sets for all subordinate DUCs the attribute *editorHighlight* (see Figure 12) to true or false given the input.

dacite.symbolicDriver: This command triggers the automatic generation of the symbolic driver. As explained in Subsection 5.2, a driver method for Mulib requires an Method Under Test (MUT) as the starting point for the symbolic execution (for more details see [24]). Which method of the to-be-analyzed program is the MUT is not immediately clear without further user interaction. To derive not yet covered DUCs, it is assumed that there already exists JUnit test cases for the to-be-analyzed program and its methods. These test cases give insights into which methods of the program are the starting point of execution and have to be tested. Hence, a static analysis using ASM was carried out on these test cases to derive the MUT, its input and output values, and their types. A method is considered a MUT when it is located in a class within the analyzed packages to ignore calls to internal Java functions. If more than one method is called within the test cases, several driver methods are generated, one for each MUT. This analysis with ASM is performed by the `DaciteTextGenerator`. Afterward, a corresponding String representing the class and a driver method for every identified MUT is automatically generated setting up the symbolic input values and invoking the MUT while adhering to the expected format of Mulib. The generated driver method (or methods) already contains a default setup sufficient for executing the method. Adaptations only need to be done manually if the domain of the symbolic inputs should be further restricted, e.g. by using domain knowledge of the MUT. Moreover, the generated driver also contains comments including an introduction to the driver method to enhance the comprehensibility and on how custom constraints can be set to prune the search space, e.g., limiting the symbolic length of an array. An exemplary driver method can be seen in Figure 15. The result is sent with the `workspaceEdit` request to the client (see Figure 10).

dacite.analyzeSymbolic: This command triggers the symbolic execution to derive DUCs that were not covered by the given test set yet (see Figure 10). Analogously to the process of `dacite.analyze`, the `CommandRegistry` first derives the necessary arguments and class path and then invokes the corresponding process of Dacite Core invoking the symbolic execution of Mulib as a separate Java process. For this, next to the package name the user can customize the execution by also specifying different values replacing the default configuration values for the symbolic execution for Mulib, e.g. the maximal execution time, within the `Dacite_config.yaml`. Afterward, the `DefUseAnalysisProvider` parses the identified DUCs and `Solutions` (input-output-mappings for the test case generation, see Subsection 4.3) from the XML and compares it to the already identified DUCs from the `dacite.analyze` command. The difference, those DUCs that were not identified previously, are transformed for visualization (see Figure 12) and stored within the `DefUseAnalysisProvider`. During this transformation, the color red was assigned to all not-covered DUCs. This color is excluded for the previously defined DUCs during the `dacite.analyze` command and facilitates the distinction between covered and non-covered DUCs.

dacite.generateTestCases: This command triggers the automatic generation of test cases based on the not-covered DUCs. For this, the corresponding static method of the `DaciteTextGenerator` is invoked which first derives the distinct set of input-output mappings (`Solution` objects) from those not-covered DUCs. For each Solution object, the set of not-covered DUCs that were covered in the path of this solution or test case is derived. Thus, the coverage is represented as a `BitSet` where each set bit represents the unique identifier of one those not-covered DUC. Based on this representation, Mulib's test case generation is invoked which includes test case reduction. The resulting String representing a

[8] `https://mokole.com/palette.html`

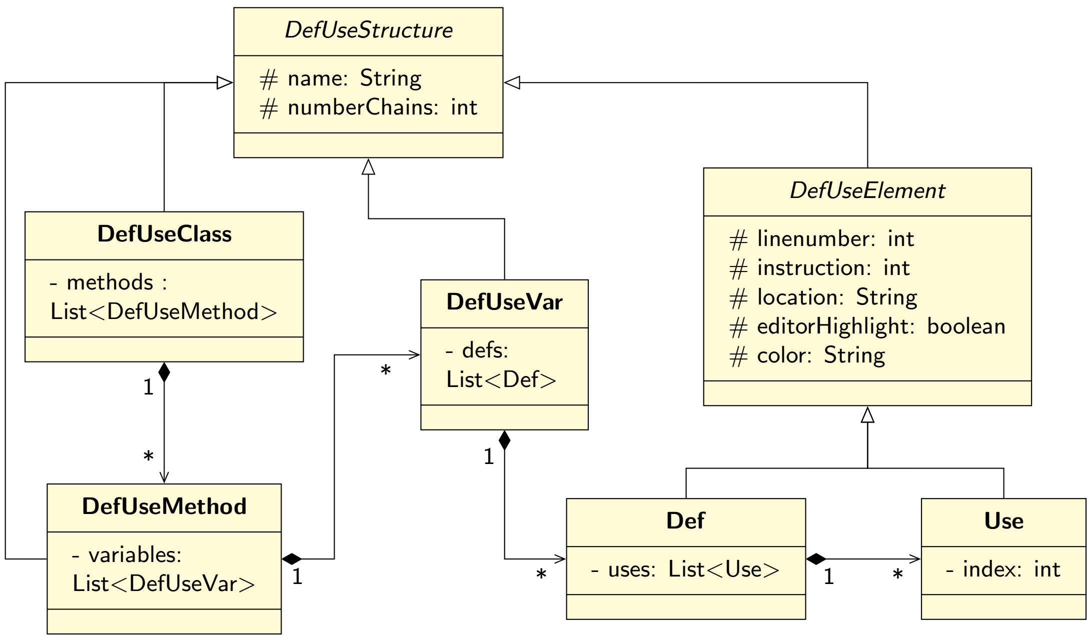


Figure 12: An overview of the transformed DUCs implementation for the visualization in the package `lsp.defUseData`. Methods are omitted for the sake of clarity [22].

JUnit test class with the reduced test cases is sent with the `workspaceEdit` request to the client (see Figure 10).

The `DaciteTextDocumentService` provides an implementation for the rest of the explained LSP message types used by Dacite (see Subsection 5.1). To account for the LSP extensions, two additional interfaces were implemented by the document service: the `DaciteExtendedTextDocumentService` adding the `inlayHintDecoration` as a message type and the `DaciteTreeViewService` adding the tree view protocol messages e.g. `treeViewChildren`. The package `dacite.lsp.tvp` entails all classes necessary for the tree view protocol including the `DaciteTreeViewService` as well as classes representing the exchanged tree view information e.g. `TreeViewNode`. For the implementation of the synchronization message (e.g. `didOpen` see Subsection 5.1), the `DaciteTextDocumentService` forwards the implementation to `TextDocumentItemProvider` which maps and stores the file's URI and its content. Depending on the synchronization type, information within the provider is added, changed, or removed. The LSP message type `codeLens` is implemented for Dacite by invoking the corresponding method of the `CodeAnalyzer`. This parses the source code of the given file retrieved from the `TextDocumentItemProvider` for the requirement necessary to display a code lens to the user for Dacite. Only if a file contains a JUnit test, the Dacite analysis code lens should be displayed (see Figure 14). So the source code is parsed to determine whether it contains a JUnit test case identified by the JUnit `@Test` annotation. In this case, the `CodeAnalyzer` returns the code lens for starting the dynamic analysis by providing the position where the lens should be displayed as well as the command (`dacite.analyze`) that should be triggered when the code lens is clicked. Analogously, if the given file adheres to the naming convention of the symbolic driver (class name starts with *DaciteSymbolicDriverFor*), the corresponding code lens for starting the symbolic execution is returned to the language client (see Figure 10).

After the user has triggered the DUC highlighting which forwards the `dacite.highlight` command to the server, the client requests inlay hints for the opened file in the editor (see Figures 9 and 10). To return inlay hints for the language client's visualization, the exact positions of the DUCs whose attribute *editorHighlight* is set to true need to be identified. From the dynamic analysis, only line numbers are provided but not column numbers which are necessary to distinguish elements in one line. The language server overcomes this limitation by exploring the AST corresponding to the source code. During the source code parsing with the `JavaParser` library within the `CodeAnalyzer`, the AST is derived. Each element in the AST can be traced back to its exact position in the source code including line and column numbers. Hence, to find the position of an inlay hint corresponding to a variable definition or usage, the server identifies the AST node corresponding to the variable in the specific line with the variable name and instruction counter for when there is more than one occurrence of this variable name in one line (see Subsection 4.1). Then, the exact position including the column number can be obtained as a property of the AST node. This is done for both, the DUCs identified by the dynamic analysis of Dacite and those identified by the symbolic execution via Mulib. The inlay hint consisting of this position and the label that should be displayed (*Def* for definitions and *Use* for variable usages) is returned to the language client. Moreover, the result is additionally saved internally within the `DaciteTextDocumentService` mapping a file's URI to a map of positions and their corresponding `DefUseElement` (see Figure 12). This map is utilized for the implementation of `inlayHintDecoration`. The client requests for every position retrieved from the `inlayHint` request the decoration in the form of color by sending the file's URI and the position as parameters to the language server. The server uses this information to retrieve the corresponding `DefUseElement` from the map within `DaciteTextDocumentService`. The respective color is extracted and returned to the language client.

For the `treeViewChildren` request, the language client sends a node identifier to the language server for which all children nodes shall be returned and a String specifying whether the DUCs from the dynamic analysis are requested (*defUseChains*) or the derived DUCs from the symbolic execution (*notCoveredDUC*). If an empty identifier is given, i.e. the root node, the language server returns a list of `TreeViewNodes` containing the names and number of chains of all objects of `DefUseClass` of the corresponding DUCs structure stored within the `DefUseAnalyzerProvider`. Otherwise, the server iterates the structure to find a `DefUseStructure` object with a name matching the given node identifier. For this, the language server returns a list with all subordinate elements, e.g. if the requested node identifier matches the `DefUseVar` name, then a list of `TreeViewNodes` for the `Def` objects is returned. The `TreeViewNode`'s label is formatted to account for the necessary information, e.g. for instances of `DefUseElement` it contains information about the variable name and line number.

5.4 Language Client

As the language server extracts the IDE-independent features into a separate component, the language client is only responsible for visualizing the information exchanged via the LSP to the user and handling the user interactions (see Figures 9 and 10). This facilitates the integration into different IDEs. Moreover, by adhering to as many existing LSP message types as possible, the effort to implement the LSP features is minimized for IDEs that support the LSP. In this case, the functionality of sending and receiving these messages to and from the server and displaying the information to the user is already implemented and ready to use [23]. For Dacite, two IDE integrations are available. Subsection 5.4.1 describes the implementation of the language client for the IDE IntelliJ IDEA while Subsection 5.4.2 elaborates on the VS Code integration.

5.4.1 Intellij IDEA

As the approach of reducing the implementation cost for IDE integrations via LSP has emerged in recent years, many IDEs such as IntelliJ do not support LSP per default yet. However, there exists the client library `lsp4intellij`[9] to make the standard LSP functionality available to custom IntelliJ plugins [23]. While the functionality of communicating and synchronizing with the server is provided by the library, the visualization of standard LSP requests such as `inlayHint` or `codeLens` within the editor is not included and still has to be implemented for IntelliJ. Nevertheless, to provide the communication with

[9] https://github.com/ballerina-platform/lsp4intellij

the language server for the Dacite integration a custom IntelliJ plugin was developed with this library in Java.

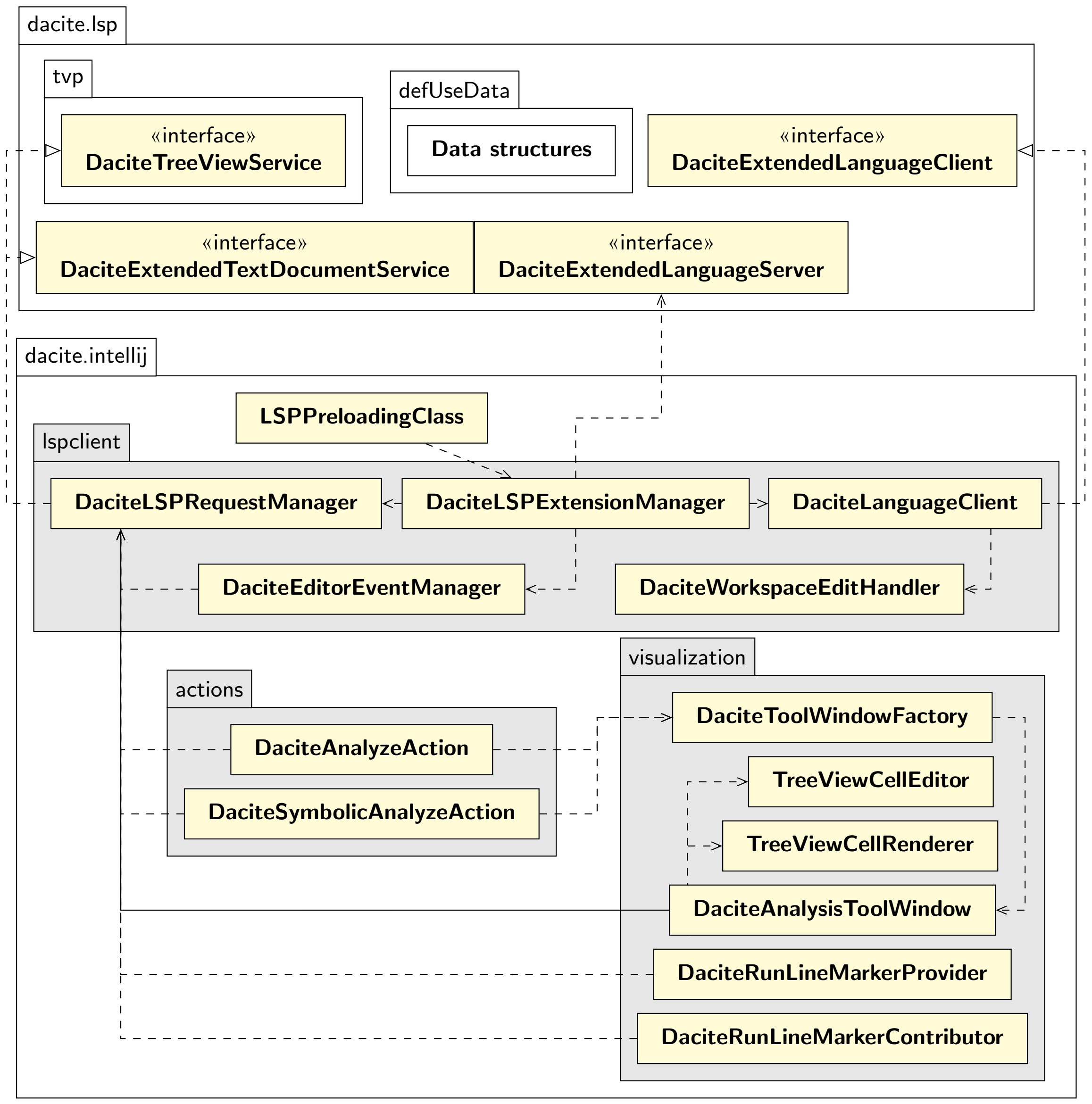
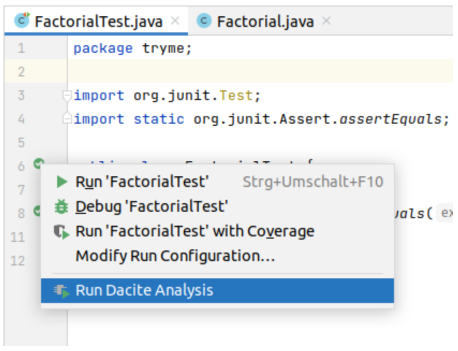

dacite.lsp
tvp
«interface» DaciteTreeViewService
defUseData
Data structures
«interface» DaciteExtendedLanguageClient
«interface» DaciteExtendedTextDocumentService
«interface» DaciteExtendedLanguageServer
dacite.intellij
LSPPreloadingClass
lspclient
DaciteLSPRequestManager
DaciteLSPExtensionManager
DaciteLanguageClient
DaciteEditorEventManager
DaciteWorkspaceEditHandler
actions
DaciteAnalyzeAction
DaciteSymbolicAnalyzeAction
visualization
DaciteToolWindowFactory
TreeViewCellEditor
TreeViewCellRenderer
DaciteAnalysisToolWindow
DaciteRunLineMarkerProvider
DaciteRunLineMarkerContributor

Figure 13: An overview of the language client implementation for IntelliJ in the packages `intellij` and `lsp`. Methods and attributes are omitted for the sake of clarity.

Figure 13 illustrates an overview of the language client implementation for IntelliJ in the package `dacite.intellij`. This package contains three subpackages and the class `LSPPreloadingClass`. This class is responsible for starting the language server as a separate process by invoking the main method of `ServerLauncher` (see Figure 11) and mapping it to the language client. The package `lspclient` comprises the classes responsible for the LSP implementation. The basic communication between the language server and client e.g. for synchronization is handled by the `lsp4intellij` library. In addition, the library allows an easy integration of protocol extensions by providing the interface `LSPExtensionManager` which allows overriding existing messages or adding new ones for the communication with the server. For Dacite, this interface is implemented by the `DaciteLSPExtensionManager`. This class maps the custom implementation of the language client `DaciteLanguageClient` implementing the DaciteExtendedLanguageClient interface of the dacite.lsp package with the corresponding interface for the language server. Moreover, it overwrites the standard request manager with the custom DaciteLSPRequestManager which implements all custom message types. Hence, this class implements the interfaces DaciteTreeViewService and DaciteExtendedTextDocumentService providing the message specification for the treeViewChildren and inlayHintDecoration requests, respectively. This request manager class is then utilized by all other classes whenever a request has to be sent to the language server. Furthermore, the extension manager also overwrites the standard editor event manager with the custom DaciteEditorEventManager. This class processes all editor events such as inlay hints and their decoration. Unfortunately, the lsp4intellij library only provides the functionality for the message exchange of inlay hints but not its visualization. Hence, this has to be implemented. Whenever an editor is opened e.g. by opening a new file within IntelliJ, the event manager requests the current inlay hints by sending the inlayHint request with the file's URI via the DaciteLSPRequestManager to the language server. The server returns a list of hints (see Subsection 5.3), which are iterated. For each inlay hint the inlayHintDecoration request is sent again to the language server to retrieve the DUC color. Afterward, the information about the inlay hint label, position, and color is utilized to add this as descriptive text into the editor model before the specified position (see Figure 1 on the left). Furthermore, the class DaciteWorkspaceEditHandler within the package lspclient is responsible for handling the workspace e.g. the creation of files. This is invoked when the DaciteLanguageClient receives the workspaceEdit request from the language server. In this case, the workspace handler creates a new file (either the symbolic driver or JUnit test case depending on the given parameters, see Subsection 5.2), fills it with the given content, and opens it within the editor so that it is directly visible to the user.

The packages actions and visualization handle the user interaction and the visualization for the user. To start the dynamic analysis of Dacite and also the symbolic analysis with Mulib, the respective code lenses have to be displayed to the user. To adhere to the visualization standard of IntelliJ, this is done via the action buttons on the left side of the editor denoted *run line marker* within IntelliJ (see Figures 14 and 15). For this, the class DaciteRunLineMarkerContributor in the package visualization inherits the corresponding interface of IntelliJ and adds a run line marker to the existing run options (e.g. run the JUnit test case) in the case of starting the dynamic analysis. Therefore, first, the DaciteLSPRequestManager of this project is retrieved and a codeLens request is sent to the language server which checks whether the file contains a JUnit test case. If this is the case, a run line marker is added for the returned position as shown in Figure 14. The given command dacite.analyze (see Subsection 5.3) is associated with this marker as an action that is invoked when the user presses the corresponding marker. In IntelliJ, actions are registered with an ID within a file plugin.xml. For the dacite.analyze command, the action DaciteAnalyzeAction is registered and consequently, associated with the visualized run line marker. Similarly, the class DaciteRunLineMarkerProvider provides a new run line marker to the editor if there are no markers present. While a JUnit test case contains methods that can be executed and therefore, already contains run line markers, the symbolic driver does not contain executable methods. Hence, for this, a new run line marker has to be added on the left side of the editor using the provided interface of IntelliJ. Analogously to the DaciteRunLineMarkerContributor, the DaciteLSPRequestManager is retrieved, the codeLens request is sent to the server and the run line marker is displayed at the given position with the corresponding action ID, in this case, dacite.analyzeSymbolic corresponding to the DaciteSymbolicAnalyzeAction (see Figure 15).

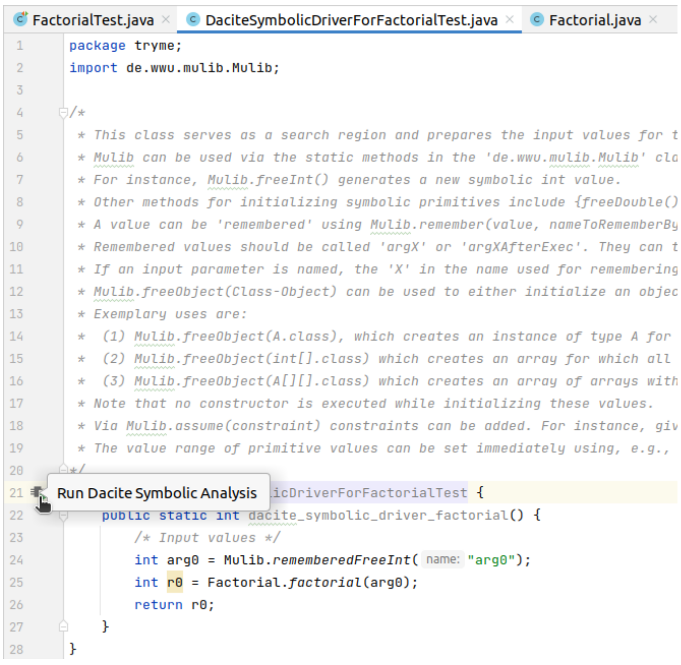
When the user triggers one of the Dacite run line markers, the corresponding action DaciteAnalyzeAction or DaciteSymbolicAnalyzeAction is invoked. Both actions send the respective command (dacite.analyze or dacite.analyzeSymbolic) as the executeCommand request via the DaciteLSPRequestManager to the language server. This triggers the internal process of the server executing the analysis (see Subsection 5.3). Afterward, to display the analysis results in the form of the identified DUCs the action creates a new instance of the class DaciteToolWindowFactory. This implements the interface of IntelliJ and is responsible for handling tool windows that display information next to the editor. This class registers the tool window for Dacite with the class DaciteAnalysisToolWindow, initializes its content, and adds it to the IntelliJ user interface on the right-hand side. To display the structured DUC information, the tool window first has to retrieve and render the results from the language server. Therefore, it sends the treeViewChildren request via the DaciteLSPRequestManager


FactorialTest.java
Factorial.java
Run 'FactorialTest' Strg+Umschalt+F10
Debug 'FactorialTest'
Run 'FactorialTest' with Coverage
Modify Run Configuration...
Run Dacite Analysis

```java
package tryme;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
```

Figure 14: Exemplary screenshot of the run line marker in IntelliJ to start the dynamic analysis for Dacite.


FactorialTest.java
DaciteSymbolicDriverForFactorialTest.java
Factorial.java
Run Dacite Symbolic Analysis

```java
package tryme;
import de.wwu.mulib.Mulib;

/*
 * This class serves as a search region and prepares the input values for t
 * Mulib can be used via the static methods in the 'de.wwu.mulib.Mulib' cla
 * For instance, Mulib.freeInt() generates a new symbolic int value.
 * Other methods for initializing symbolic primitives include {freeDouble()
 * A value can be 'remembered' using Mulib.remember(value, nameToRememberBy
 * Remembered values should be called 'argX' or 'argXAfterExec'. They can t
 * If an input parameter is named, the 'X' in the name used for remembering
 * Mulib.freeObject(Class-Object) can be used to either initialize an objec
 * Exemplary uses are:
 *  (1) Mulib.freeObject(A.class), which creates an instance of type A for
 *  (2) Mulib.freeObject(int[].class) which creates an array for which all
 *  (3) Mulib.freeObject(A[][].class) which creates an array of arrays with
 * Note that no constructor is executed while initializing these values.
 * Via Mulib.assume(constraint) constraints can be added. For instance, giv
 * The value range of primitive values can be set immediately using, e.g.,
 */
                icDriverForFactorialTest {
    public static int dacite_symbolic_driver_factorial() {
        /* Input values */
        int arg0 = Mulib.rememberedFreeInt( name: "arg0");
        int r0 = Factorial.factorial(arg0);
        return r0;
    }
}
```

Figure 15: Exemplary screenshot of the generated driver with run line marker in IntelliJ to start the symbolic analysis for Dacite.

to the language server requesting the children of the root and iteratively, for all children until the leaf nodes, i.e. uses (see Figure 12) are reached. The corresponding result is visualized using the GUI Java Swing elements such as JPanel or JTree (see Figure 1 on the right). To allow for a customized rendering of the tree elements, e.g. displaying the text elements in different colors to increase usability and readability, the class TreeViewCellRenderer was created which extends the default renderer provided by Java Swing. Moreover, to enable the interaction with the tree to highlight specific DUCs, checkboxes were added to each tree element. The TreeViewCellEditor implements the functionality of interacting with this checkbox by adding a listener and saving the result to the tree representation. When the editing is stopped and a checkbox has been selected, the listener invokes the corresponding command, in this case, dacite.highlight, and sends an executeCommand request via the request manager to the server. Afterward, the existing inlay hints are removed and the inlay hints are rendered again as described above to account for the requested highlighting. The process for visualizing the identified DUCs for the dynamic analysis via DaciteAnalyzeAction and the identified DUCs that were not covered yet by given test cases via DaciteSymbolicAnalyzeAction are similar. The only difference is the title of the visualized tree list and the placement (see Figure 2 on the right). For the dynamic analysis, only one list of DUCs is displayed whereas for the symbolic execution, both lists are displayed above another with the newest information, i.e. the DUCs that were not covered yet, displayed at the top.

The other two user interactions for generating the symbolic driver and generating the JUnit test cases (see Figure 9) are not displayed as run line markers as they are not triggered from a specific class but for the dynamic analysis in general. Hence, for both, interaction buttons with corresponding tooltip descriptions are added to the tool window (see Figure 2 in the right corner). A listener is added for both sending the executeCommand with the respective parameter (dacite.symbolicDriver or dacite.generateTestCases) via the request manager to the language server upon interaction. The server generates the corresponding content and sends a new workspaceEdit command to the language client (see Figure 9) which is received by the class DaciteLanguageClient. This forwards the information to the DaciteWorkspaceHandler which creates new files with the given names and fills their content accordingly with the results from the language server. The new class is opened within the editor for the user (see Figure 15).

5.4.2 VS Code

In contrast to IntelliJ, VS Code supports LSP per default. It provides a standard language client implementation including the communication with the server and the automatic visualization within the editor for all standardized message types, such as inlay hints and code lenses [13]. To integrate Dacite, only the two custom message types still need to be implemented. However, despite not being part of the standardized LSP, support for the tree view protocol has been implemented for VS Code and can be reused for the Dacite integration [12, 23].

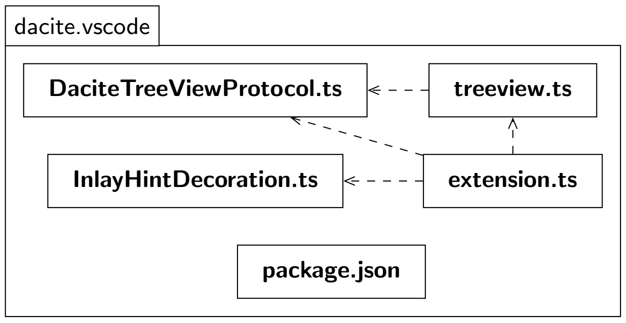
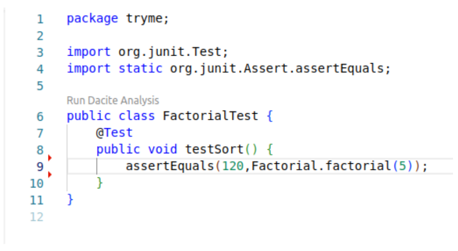


Figure 16: An overview of the language client implementation for VS Code in the package `vscode`.

Figure 16 illustrates the implementation of the Dacite language client in VS Code as a custom plugin. This plugin is implemented in the programming language TypeScript to adhere to the VS Code standard. The files `DaciteTreeViewProtocol.ts` and `treeview.ts` implement the functionality for the tree view protocol. While the `DaciteTreeViewProtocol.ts` defines the interfaces and namespaces for the communication, `treeview.ts` implements the functionality such as communicating the server and rendering the tree view children. This code was taken from [12] and minimally adapted for the Dacite integration. The file `InlayHintDecoration.ts` introduces the interfaces and namespace for the communication of the `inlayHintDecoration` request. The file `extension.ts` defines the Dacite integration as an extension of VS Code and is the starting point of the language client. Therefore, within this file, the language client is defined and mapped to the language server binary which starts the main method of the `ServerLauncher` (see Figure 11). Moreover, the existing implementation of `inlayHint` is extended so that for each identified inlay hint the decoration is retrieved via the request `inlayHintDecoration` and the existing inlay hint is displayed in the according color within the editor. The `extension.ts` also starts the tree view for the identified DUCs which is updated based on internal listeners within the `treeview.ts`. The file `package.json` is the extension manifest [27] and adds a new tool window for the Dacite tree view to the editor whose content is filled with the `extension.ts`.

The result of this integration into VS Code can be seen in Figures 17, 18, and 19. In VS Code, code lenses are automatically displayed as executable links within the editor, e.g. the trigger for starting the dynamic analysis of Dacite in Figure 17 above the class declaration. Figure 18 shows the result of the dynamic analysis of Dacite as a tree view on the left with the displayed inlay hints in the editor on the right. In contrast to IntelliJ, interactions with the tree view are conducted by clicking on a tree view element instead of interacting with checkboxes. Similarly, Figure 19 displays the complementary information about the not-covered DUCs identified during the symbolic execution and displayed in VS Code. All in all, integrating Dacite into this IDE required only a small effort due to the existing LSP functionalities. This underlines the added value of utilizing the LSP for the communication involved in analyzing and visualizing data flows [23].

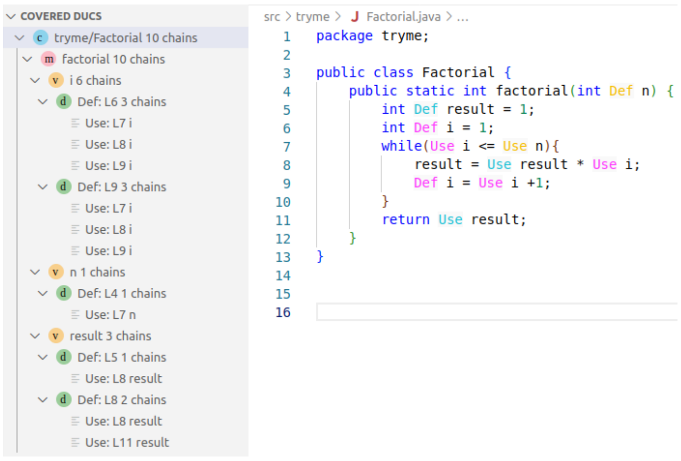


```
package tryme;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

Run Dacite Analysis
public class FactorialTest {
    @Test
    public void testSort() {
        assertEquals(120,Factorial.factorial(5));
    }
}
```

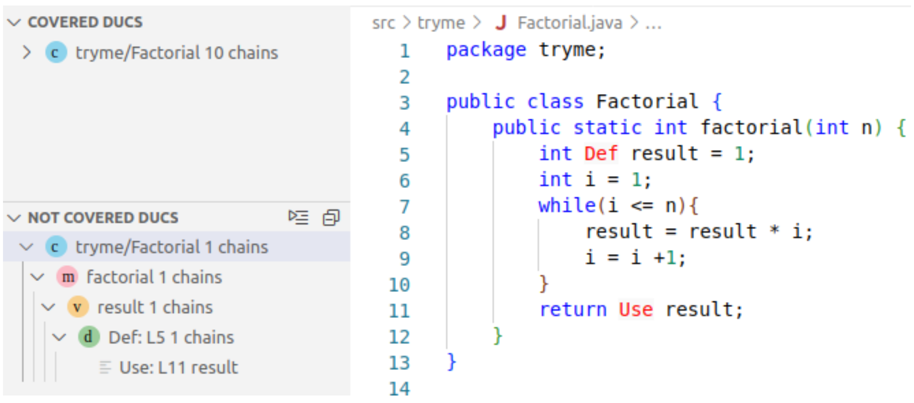
Figure 17: Exemplary screenshot of the code lens in VS Code to start the dynamic analysis for Dacite.



```
COVERED DUCS
  tryme/Factorial 10 chains
    factorial 10 chains
      i 6 chains
        Def: L6 3 chains
          Use: L7 i
          Use: L8 i
          Use: L9 i
        Def: L9 3 chains
          Use: L7 i
          Use: L8 i
          Use: L9 i
      n 1 chains
        Def: L4 1 chains
          Use: L7 n
      result 3 chains
        Def: L5 1 chains
          Use: L8 result
        Def: L8 2 chains
          Use: L8 result
          Use: L11 result
```

```
src > tryme > Factorial.java > ...
1   package tryme;
2
3   public class Factorial {
4       public static int factorial(int Def n) {
5           int Def result = 1;
6           int Def i = 1;
7           while(Use i <= Use n){
8               result = Use result * Use i;
9               Def i = Use i +1;
10          }
11          return Use result;
12      }
13  }
14
15
16
```

Figure 18: Exemplary Screenshot of the tree view and inlay hints of VS Code for the dynamic analysis.



```
COVERED DUCS
  tryme/Factorial 10 chains

NOT COVERED DUCS
  tryme/Factorial 1 chains
    factorial 1 chains
      result 1 chains
        Def: L5 1 chains
          Use: L11 result
```

```
src > tryme > Factorial.java > ...
1   package tryme;
2
3   public class Factorial {
4       public static int factorial(int n) {
5           int Def result = 1;
6           int i = 1;
7           while(i <= n){
8               result = result * i;
9               i = i +1;
10          }
11          return Use result;
12      }
13  }
14
```

Figure 19: Exemplary Screenshot of the tree view and inlay hints of VS Code for the symbolic analysis.

6 Conclusion

This section concludes this report. Therefore, first in Subsection 6.1 the features and implementation of Dacite is summarized. Afterwards perspectives for future research is provided in Subsection 6.2.

6.1 Summary

This report outlines the implementation of the open-source data-flow analysis tool Dacite. Using bytecode instrumentation with ASM, this tool is able to dynamically identify reachable DUCs for a given Java program and its JUnit test set. Dacite can be distinguished from other tools by taking variable aliases, inter-procedural data flow, and detailed consideration of the data flow of objects and arrays into account. Moreover, by deriving the DUCs while executing the program with instrumented methods tracking the data flow, only those DUCs are identified that are reachable during the execution. To give feedback about the data-flow coverage of a program, symbolic execution is utilized with the symbolic execution engine Mulib. By systematically traversing the program during symbolic execution, the data flow can be derived which was not covered yet by the existing JUnit test cases. On top of that, test cases are automatically generated and suggested to the user covering these not-yet-covered DUCs. Furthermore, Dacite provides feedback and visualizations directly to the user with integrations into standard IDEs, IntelliJ and VS Code. This is accomplished using the LSP to reduce the implementation effort of developing different IDE integrations by extracting IDE-independent features into a separate component, the language server.

6.2 Future Work

Given the described Dacite prototype, IDE integrations for IntelliJ IDEA and VS Code are provided to the user. However, because of market segmentation there exists a variety of development environments that are utilized by different users (e.g., Eclipse vs. IntelliJ). To make Dacite greatly available for Java developers, future work should integrate the language server via the Language Server Protocol (LSP) to additional popular IDEs such as Eclipse or NetBeans. While Eclipse provides a plugin with rudimentary LSP functionalities like the communication with the server excluding the visualization for inlay hints and code lenses similar to IntelliJ[10], NetBeans offers built-in support for different features including code lenses[11]. However, as many of these features have been developed recently with the emerging popularity of LSP, it can be assumed that these sets of supported features for the different IDEs will be further increased in the future.

Additionally, the customization of the Dacite analysis could also be further integrated into the LSP visualization. For now, the user has the opportunity to specify custom values e.g. the to-be-analyzed package name or configuration values for the symbolic execution within a config file named `Dacite_config.yaml`. To increase the usability, it would be preferable to optionally inquire this information within the editor user interface. For instance, when the user triggers the dynamic analysis via the code lens, an additional window would be visualized enabling the user to change the execution configurations before continuing the dynamic analysis.

Moreover, in the future, one could further optimize the symbolic execution for deriving the DUCs not covered by the initial test cases. While current results are acceptable for unit testing and early integration testing, scaling to larger problems leads to the well-known problem of path explosion [3]. One way to mitigate this problem would be to prune the search region based on the existing JUnit test cases and consequently the already covered DUCs for those paths. If, for instance, a region of the program cannot contain any more DUCs, these paths do not need to be regarded any longer during

[10] https://github.com/eclipse/lsp4e
[11] http://bits.netbeans.org/dev/javadoc/org-netbeans-api-lsp/overview-summary.html

symbolic execution [24]. Alternatively, other approaches such as search-based techniques could be applied to derive not-covered DUCs for larger programs. This has already been applied to the area of static data-flow analysis for test case generation and evaluated with larger programs [28]. For Dacite, the search-based execution could be utilized to explore for each iteration whether new DUCs can be covered given the corresponding input-output mappings.

References

[1] Simone Bianco and Andrea G Citrolo. "High contrast color sets under multiple illuminants". In: *Computational Color Imaging: 4th International Workshop, CCIW 2013, Chiba, Japan, March 3-5, 2013. Proceedings*. Springer. 2013, pp. 133–142.

[2] Ilona Bluemke and Artur Rembiszewski. "Dataflow testing of java programs with dfc". In: *IFIP Central and East European Conference on Software Engineering Techniques*. Springer. 2009, pp. 215–228. doi: 10.1007/978-3-642-28038-2_17.

[3] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Commun. ACM* 56.2 (Feb. 2013), 82â€“90. issn: 0001-0782. doi: 10.1145/2408776.2408795.

[4] Oracle Corporation. *The Java Virtual Machine Specification*. https://docs.oracle.com/javase/specs/jvms/se11/html/index.html. Last accessed Dezember 11, 2023. 2018.

[5] Giovanni Denaro, Mauro Pezze, and Mattia Vivanti. "On the right objectives of data flow testing". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 71–80. doi: 10.1109/ICST.2014.18.

[6] Giovanni Denaro et al. "Dynamic data flow testing of object oriented systems". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 947–958.

[7] EclEmma. *JaCoCo*. https://www.jacoco.org/jacoco. Last accessed November 14, 2023. 2017.

[8] Phyllis G. Frankl and Stewart N. Weiss. "An experimental comparison of the effectiveness of branch testing and data flow testing". In: *IEEE Transactions on Software Engineering* 19.8 (1993), pp. 774–787. doi: 10.1109/32.238581.

[9] Phyllis G. Frankl and Elaine J. Weyuker. "An applicable family of data flow testing criteria". In: *IEEE Transactions on Software Engineering* 14.10 (1988), pp. 1483–1498.

[10] Hadi Hemmati. "How effective are code coverage criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 151–156. doi: 10.1109/QRS.2015.30.

[11] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. https://www.jsonrpc.org/specification. Last accessed November 14, 2023. 2013.

[12] Metals. *Core Components and Native Components*. https://scalameta.org/metals/docs/integrations/tree-view-protocol. Last accessed November 14, 2023. 2022.

[13] Microsoft Corporation. *Language Server Extension Guide*. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. Last accessed September 01, 2023. 2023.

[14] Microsoft Corporation. *Language Server Protocol Specification - 3.17*. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification. Last accessed September 01, 2023. 2023.

[15] Jonathan Misurda et al. "Jazz: A tool for demand-driven structural testing". In: *International Conference on Compiler Construction*. Springer. 2005, pp. 242–245.

[16] Oracle Corporation. *Chapter 6. The Java Virtual Machine Instruction Set*. https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-6.html. Last accessed November 01, 2023. 2018.

[17] Jonas Kjær Rask et al. "The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions". In: *Electronic Proceedings in Theoretical Computer Science* 338 (Aug. 2021), pp. 3–18. doi: 10.4204/eptcs.338.3.

[18] Henrique L Ribeiro et al. "Evaluating data-flow coverage in spectrum-based fault localization". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2019, pp. 1–11. doi: 10.1109/ESEM.2019.8870182.

[19] Raul Santelices and Mary Jean Harrold. "Efficiently monitoring data-flow test coverage". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 343–352.

[20] Ting Su et al. "A survey on data-flow testing". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–35.

[21] Ed. T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7158. 2013. doi: 10.17487/RFC7158.

[22] Laura Troost and Herbert Kuchen. "A Comprehensive Dynamic Data Flow Analysis of Object-Oriented Programs". In: *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*. INSTICC. SciTePress, 2022, pp. 267–274. isbn: 978-989-758-568-5. doi: 10.5220/0010984800003176.

[23] Laura Troost, Jonathan Neugebauer, and Herbert Kuchen. "Visualizing Dynamic Data-Flow Analysis of Object-Oriented Programs Based on the Language Server Protocol". In: *Proceedings of the 18^{th} International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. Ed. by SciTePress. Prague, Czech Republic: SciTePress, 2023, pp. 77–88. isbn: 978-989-758-647-7. doi: 10.5220/0011743500003464.

[24] Laura Troost, Hendrik Winkelmann, and Herbert Kuchen. "An Integrated Visualization Approach Combining Dynamic Data-Flow Analysis with Symbolic Execution". In: *Proceedings of the 19^{th} International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. Ed. by SciTePress. (accepted). Prague, Czech Republic: SciTePress, 2024.

[25] AMR Vincenzi et al. "JaBUTi: A coverage analysis tool for Java programs". In: *XVII SBES–Simpósio Brasileiro de Engenharia de Software* (2003), pp. 79–84.

[26] Auri Marcelo Rizzo Vincenzi et al. "Establishing structural testing criteria for java bytecode". In: *Software: practice and experience* 36.14 (2006), pp. 1513–1541.

[27] Visual Studio Code. *Extension Anatomy*. https://code.visualstudio.com/api/get-started/extension-anatomy. Last accessed Dezember 14, 2023. 2023.

[28] Mattia Vivanti et al. "Search-based data-flow test generation". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2013, pp. 370–379.

[29] Hendrik Winkelmann and Herbert Kuchen. "Constraint-Logic Object-Oriented Programming on the Java Virtual Machine". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, 1258â€"1267. isbn: 9781450387132. doi: 10.1145/3477314.3507058. url: https://doi.org/10.1145/3477314.3507058.

[30] Hendrik Winkelmann and Herbert Kuchen. "Constraint-Logic Object-Oriented Programming with Free Arrays of Reference-Typed Elements via Symbolic Aliasing". In: *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC. SciTePress, 2023, pp. 412–419. isbn: 978-989-758-647-7. doi: 10.5220/0011849200003464.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004.

Nr. 2 Teubner, A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. 2005.

Nr. 3 Teubner, A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. 2007.

Nr. 6 Teubner, A.; Mocker, M.: A Literature Overview on Strategic Information Management. 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.

Nr. 9 Majchrzak, T.; Jakubiec, A.; Lablans, M.; Ükert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration. 2010.

Nr. 10 Majchrzak, T.; Kuchen, H: Muggl: The Muenster Generator of Glass-box Test Cases. 2011.

Nr. 11 Becker, J.; Beverungen, D.; Delfmann, P.; Räckers, M.: Network e-Volution. 2011.

Nr. 12 Teubner, A.; Pellengahr, A.; Mocker, M.: The IT Strategy Divide: Professional Practice and Academic Debate. 2012.

Nr. 13 Niehaves, B.; Köffer, S.; Ortbach, K.; Katschewitz, S.: Towards an IT consumerization theory: A theory and practice review. 2012

Nr. 14 Stahl, F., Schomm, F., Vossen, G.: Marketplaces for Data: An initial Survey. 2012.

Nr. 15 Becker, J.; Matzner, M. (Eds.).: Promoting Business Process Management Excellence in Russia. 2012.

Nr. 16 Teubner, A.; Pellengahr, A.: State of and Perspectives for IS Strategy Research. 2013.

Nr. 17 Teubner, A.; Klein, S.: The MÃ¼nster Information Management Framework (MIMF). 2014.

Nr. 18 Stahl, F.; Schomm, F.; Vossen, G.: The Data Marketplace Survey Revisited. 2014.

Nr. 19 Dillon, S.; Vossen, G.: SaaS Cloud Computing in Small and Medium Enterprises: A Comparison between Germany and New Zealand. 2015.

Nr. 20 Stahl, F.; Godde, A.; Hagedorn, B.; Köpcke, B.; Rehberger, M.; Vossen, G.: Implementing the WiPo Architecture. 2014.

Nr. 21 Pflanzl, N.; Bergener, K.; Stein, A.; Vossen, G.: Information Systems Freshmen Teaching: Case Experience from Day One (Pre-Version of the publication in the International Journal of Information and Operations Management Education (IJIOME)). 2014.

Nr. 22 Teubner, A.; Diederich, S.: Managerial Challenges in IT Programmes: Evidence from Multiple Case Study Research. 2015.

Nr. 23 Vomfell, L.; Stahl, F.; Schomm, F.; Vossen, G.: A Classification Framework for Data Marketplaces. 2015.

Nr. 24 Stahl, F.; Schomm, F.; Vomfell, L.; Vossen, G.: Marketplaces for Digital Data: Quo Vadis?. 2015.

Nr. 25 Caballero, R.; von Hof, V.; Montenegro, M.; Kuchen, H.: A Program Transformation for Converting Java Assertions into Controlflow Statements. 2016.

Nr. 26 Foegen, K.; von Hof, V.; Kuchen, H.: Attributed Grammars for Detecting Spring Configuration Errors. 2015.

Nr. 27 Lehmann, D.; Fekete, D.; Vossen, G.: Technology Selection for Big Data and Analytical Applications. 2016.

Nr. 28 Trautmann, H.; Vossen, G.; Homann, L.; Carnein, M.; Kraume, K.: Challenges of Data Management and Analytics in Omni-Channel CRM. 2017.

Nr. 29 Rieger, C.: A Data Model Inference Algorithm for Schemaless Process Modeling. 2016.

Nr. 30 Bünder, H: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. 2019.