A Comprehensive Dynamic Data Flow Analysis of Object-Oriented Programs

Laura Troost and Herbert Kuchen
Institute of Practical Computer Science, University of Münster, Leonardo Campus, Münster, Germany

Keywords: Data Flow Analysis, Bytecode Analysis, Java Instrumentation.

Abstract: Studies have shown that in the area of testing data-flow coverage is often more effective in exposing errors compared to other approaches such as branch coverage. Thus, evaluating and generating test cases with respect to the data-flow coverage is desirable. Unfortunately, data-flow coverage is rarely considered in practice due to the lack of appropriate tools. Existing tools are typically based on static analysis and cannot distinguish between traversable and non-traversable data flows. They also have typically difficulties with properly handling aliasing and complex data structures. Thus, we propose a tool for *dynamically* analyzing the data-flow coverage which avoids all these drawbacks. In combination with our existing test-case generator, it enables the creation of an (almost) minimal set of test cases that guarantee all data flows to be covered. We have evaluated our tool based on a couple of benchmarks.

1 INTRODUCTION

Data-flow analysis has originally been developed for compiler optimizations and is concerned with identifying the data flow within a program (Allen and Cocke, 1976). The concept of data flow considers the definition and usage of data throughout the program. The idea is that for each value defined in a program it should be checked whether it causes problems at places where this value is used. Data-flow analysis has been continuously researched and applied to areas such as testing or test case generation (Su et al., 2017). For this, studies have shown that data-flow coverage is often more effective in exposing errors compared to other approaches such as branch coverage (Frankl and Iakounenko, 1998). Thus, evaluating and generating test cases with respect to the data-flow coverage is desirable.

However, only a few tools exist for tracking the data-flow coverage for a given program (Su et al., 2017). Even more importantly, the existing tools typically focus on a static analysis of the data flow. This causes severe limitations such as that a syntactically identified data flow may not be traversable due to semantic circumstances. Additionally, each existing tool does not consider every data flow aspect of the program in detail. For instance, some tools only focus on the data flow within one method (intra-procedural) (Vincenzi et al., 2003) while others regard objects and arrays as a whole instead of differentiating which field or element is defined/used (Bluemke and Rembiszewski, 2009; Misurda et al., 2005). Furthermore, although these existing tools have been researched, they are either not publicly available or not working reliably and outdated (see Section 5). As a consequence, due to the lack of appropriate tools, data-flow testing is largely ignored in practice.

This paper introduces an open-source prototype that dynamically identifies the reachable data flows of a given Java program. In combination with our existing test-case generator (Winkelmann et al., 2022), which produces one test-case for every path through the classes under test, it can be used to reduce this set of test-cases to an (almost) minimal set ensuring data-flow coverage.

Our prototype not only considers the data flow of a single method (intra-procedural) but also over methods and classes (inter-procedural) (Denaro et al., 2014). Moreover, typical challenges such as aliasing and the precise treatment of used array elements are taken into account. Hence, the prototype proposed in this paper contributes by providing the means to precisely and reliably analyze the executed data flow of a given program. This is especially relevant for evaluating and generating test cases. Our prototype is called Dacite (DAta-flow Coverage for Imperative TEsting).

This paper is organized as follows. Section 2 explains the fundamental concepts of data-flow analy-

Troost, L. and Kuchen, H.
A Comprehensive Dynamic Data Flow Analysis of Object-Oriented Programs.
DOI: 10.5220/0010984800003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 267-274
ISBN: 978-989-758-568-5; ISSN: 2184-4895

sis. Section 3 explains the implementation of the dynamic data-flow analysis. Therefore, it is first illustrated how the data flow is represented and then the two steps of the analysis, the bytecode transformation and the dynamic analysis, are explained. Afterwards, in Section 4, the results are evaluated and validated given benchmark examples. Section 5 summarizes the related work concerning dynamic analysis and data flow. Section 6 concludes the results of this paper.

2 FUNDAMENTAL CONCEPTS

As described in the introduction, data flow considers the definition and usage of data throughout the program. To represent and describe this information, the definitions and corresponding usages can be connected to so-called definition-usage chains (DUCs), also referred to as definition-usage pairs or definition-usage relations. They can be defined as follows:

$$\begin{gathered}\text{let } X \text{ be a variable and } S, S', S'' \text{ instructions}\\ def(S) := \{\, X \mid S \text{ writes to } X \}\\ use(S) := \{\, X \mid S \text{ reads from } X \}\end{gathered}$$

If $X \in def(S) \cup use(S')$ and $X \notin def(S'')$ for each S'' on a path from S to S', (X,S,S') forms a *definition-usage chain* (Majchrzak and Kuchen, 2009).

The DUCs of a program can be identified statically by analyzing the program code or dynamically during the execution (Su et al., 2017). One main limitation of statically analyzing the program is that it is not possible to differentiate traversable and non-traversable DUCs. Consider the code snippet in Listing 1 and suppose that `p1(int)` and `p2(int)` are boolean methods defined elsewhere and that `p1(0)` implies `p2(0)`. In this case, the DUC `(x,x=0,y=100/x)` is *not traversable*, since `y = 100/x` is unreachable. On the other hand, the DUC `(x,x=0,y=x+1)` is *traversable*. When testing and executing programs, only traversable DUCs are of interest. Since it is in general not possible to decide statically whether a DUC is traversable, we will use a dynamic approach.

```
int x = 0; int y;
if (p1(x)) {
  if (p2(x)) { y = x+1;}
  else { y = 100/x;}
}
```

Listing 1: Example for traversable and non-traversable DUCs.

Moreover, when analyzing the data flow of a Java program, several aspects need to be considered. One important aspect is aliasing. When two variables point to the same object in Java, they are aliases. In

```
1  Point p1 = new Point(1,2);
2  Point p2 = p1;
3  p2.setX(3);
4  System.out.println(p1.getX());
```

Listing 2: Example for aliasing.

Listing 2, `p1` and `p2` are aliases referring to the same object (of some assumed class `Point` with getters and setters for its coordinates). Consequently, when the field `x` of `p2` is changed in line 3, `p1.x` is changed as well. This needs to be taken into account when generating DUCs, as although `p1.x` has not been accessed in line 3, its value has changed here. Hence, `(x, p2.setX(3), p1.getX())` is a DUC.

A dynamic analysis and hence also Dacite is able to precisely identify these aliases by directly gathering the relevant information during the execution while the static analysis typically has to rely on techniques such as overapproximation (Denaro et al., 2014). Additionally, as mentioned before, data-flow analysis can be distinguished into intra-procedural and inter-procedural analysis. While intra-procedural data flow focuses only on single methods, inter-procedural approaches enable the analysis to identify DUCs over the boundary of methods or classes (Harrold and Soffa, 1989). This inter-procedural approach is especially relevant for programs written in object-oriented languages such as Java which typically consist of several methods and classes. The dynamic analysis of Dacite takes inter-procedural DUCs into account. Here, parameter passing can also lead to aliasing. For instance, in Listing 3 two methods are defined. Within `method1` a variable *x* is defined (line 2) which is then passed to `method2` as a parameter (line 3). Here, the definition in line 2 and the usage in line 6 form a DUC although different variable names are used.

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

Listing 3: Example for inter-procedural DUCs.

Moreover, when analyzing objects and arrays, different levels of abstractions can be made for the identification of DUCs. Either objects and arrays can be regarded as a whole, i.e. changing one field or element results in the definition of the complete object or array, or each field or element can be regarded separately. Thus, separate DUCs are considered for the different fields and elements. While the first is easier to implement, the latter allows for a more detailed analysis of the data flow within the program and hence for identifying problems within the program. Dacite supports the detailed identification of DUCs for objects fields and array elements.

```
1  Point p = new Point(1,2);
2  int[] a = {1,2,3,4};
3  p.setX(3);
4  a[0] = 5;
5  System.out.println(p.getY(),a[1]);
```

Listing 4: Example for considering object fields and array elements separately.

For instance in Listing 4, lines 3 and 5 for the object just as lines 4 and 5 for the array would *not* form DUCs when the elements are considered separately. Instead only lines 1 and 5 just as lines 2 and 5 form DUCs. However, if object and arrays are considered as a whole, this can no longer be differentiated leading to new definitions of the object and array in lines 3 and 4. Thus, the DUC of lines 1 and 5 just as lines 2 and 5 are no longer considered.

3 IMPLEMENTATION

After explaining the fundamental concepts of the dynamic data flow, we will discuss the implementation of Dacite. Our data-flow analysis is based on Java Virtual Machine (JVM) bytecode. Thus, it can analyze the data flow of Java programs but also of other programming languages which can be translated to JVM bytecode such as Kotlin and Scala. To be able to identify definitions and usages in bytecode, these occurrences have to be specified. In general, in JVM bytecode, a definition occurs when a value is stored from the operand stack into the variable table as is done, e.g., by means of the bytecode instruction `istore` for integer variables. Similarly, a usage is defined as loading a variable onto the operand stack as is done, e.g., using the instruction `iload`. Special cases like storing a value into an array or object or loading a value from an array or object to the operand stack are also considered. When dealing with inter-procedural analysis, variables passed on as parameters are not considered definitions at the entry of the called method or usages at the call site to account for the inter-procedural data flow. However, parameters at entry points, e.g. main methods, are treated as definitions when the code was called externally, i.e. not within the custom classes. Similarly, passed variables to library functions or internal Java functions are treated like usages at the call site (Santelices and Harrold, 2007). Before the implementation of the analysis can be explained, Section 3.1 first summarizes how the DUCs are represented.

In order to implement the dynamic identification of DUCs on the bytecode level, Java Instrumentation with the open-source framework ASM[1] is utilized. This framework enables the modification of existing classes in binary form before they are loaded into the JVM. To be able to access the necessary information during the program execution, methods tracking the definition and usages of variables can be automatically added by this instrumentation to the to be examined program code. These tracking methods then retrieve the necessary information during the execution and pass it to the class `DefUseAnalyzer` which dynamically collects and analyzes this information and derives passed DUCs. Hence, the dynamic data-flow analysis consists of two steps, the bytecode transformation which is described in Section 3.2 and the analysis during the execution which is explained in Section 3.3.

3.1 Representation of Data Flow

Before explaining how the relevant data-flow information is collected throughout the execution, first, it has to be defined which information is relevant. 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 index of the instruction and the method are used additionally to identify and associate definitions and usages of variables. To enable users to relate the definitions and usages to their program code, the corresponding line number and variable name within the program is stored for each definition or usage. Figure 1 depicts an overview of how the DUCs are represented within Dacite. A definition or usage as described is represented by the class `DefUseVariable`. A special form of this is the definition or usage of an object field

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

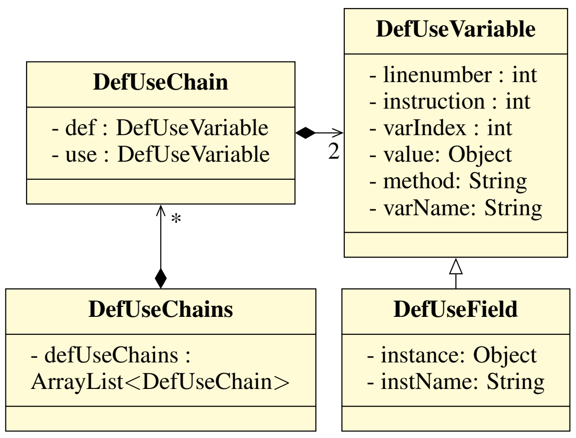

Figure 1: An overview of the DUCs implementation. Methods are omitted for the sake of clarity.

or an array element. For this purpose, a reference to the related object or array has to be saved next to the other information (see `DefUseField`). A DUC is represented by the class `DefUseChain` and consists of a `DefUseVariable` instance for the definition and one for the usage. `DefUseChains` is a wrapper class collecting all passed DUC and contains custom methods e.g. to find a variable usage with given arguments.

3.2 Bytecode Transformation

In order to automatically modify the classes of the given program, a Java agent is utilized. This agent is linked to the start of the program execution. This class or set of classes is specified by the user to determine the part of the program for which the data flow shall be analyzed. This can be a single JUnit Test or a batch of tests. The start of the execution is also the start of the bytecode transformation. When the JUnit tests are started, the Java agent invokes a custom subclass `DefUseTransformer` of the `ClassFileTransformer`. Here, method `transform` is invoked for every class starting from the defined JUnit test class. Within this method, the bytecode transformation takes place to facilitate the tracking of the relevant data-flow information. To avoid unnecessary overhead like analyzing internal Java classes, only the classes defined by the user are transformed.

Starting from the bytecode of a class which is given to the `transform` method, ASM is utilized to transform the bytecode into a traversable structure starting at a `ClassNode`. This `ClassNode` then references different `MethodNodes` representing the methods of this class. For each `MethodNode`, first, the given argument parameters are identified. As each method parameter definition may represent a potential definition of a variable, the relevant information has to be passed on to the `DefUseAnalyzer`. This information encompasses the current method name, the index where this value is stored in the local variable table, and the value of the parameter. To pass the information to the `DefUseAnalyzer`, new bytecode instructions are added which push the information onto the operand stack and afterwards, a method invocation to the corresponding method of the `DefUseAnalyzer` is added.

```
1 InsnList methodStart = new InsnList();
2 ...
3 boxing(types[typeindex],
    ↪ localVariable.index, methodStart,
    ↪ true);
4 methodStart.add(new
    ↪ IntInsnNode(BIPUSH,
    ↪ localVariable.index));
5 methodStart.add(new
    ↪ LdcInsnNode(mnode.name));
6 methodStart.add(new
    ↪ MethodInsnNode(INVOKESTATIC,
    ↪ "defuse/DefUseAnalyzer",
    ↪ "visitParameter",
    ↪ "(Ljava/lang/Object;I
    ↪ Ljava/lang/String;)V", false));
7 ...
8 insns.insertBefore(firstIns,
    ↪ methodStart);
```

Listing 5: A Java code excerpt inserting bytecode instructions for method parameters.

Listing 5 shows an excerpt which adds the new bytecode instructions. This is done by defining a list of instructions (line 1) for which different instruction nodes are added (lines 3-6), e.g. `MethodInsNode` for a method invocation, and afterwards, this list is added to the method instruction list (line 8). The method `boxing` (line 3) is used to convert the variable value based on its type to its corresponding Object wrapper classes to adhere to the type specifications of the called method in line 6 and consequently, to the implementation of the method within the class `DefUseAnalyzer`. As mentioned before, lines 3 to 5 are used to push the relevant information, i.e. value, index, and method name, to the operand stack. In line 6, the static method `visitParameter` of the `DefUseAnalyzer` is invoked which takes the three previously pushed values as method arguments.

After the definition of the method arguments, the bytecode instructions of the current method are iterated to find potential definitions and usages. Due to the many different types of instructions, e.g. `istore` or `iastore`, different approaches to retrieve the relevant information are required. For instance, while the `istore` instruction stores the index of the variable within the instruction, `iastore` gathers this information from the operand stack. Moreover, to not only account for objects and arrays as a whole but their individual parts, e.g. when only a specific field is defined and used, additional information including the reference to the whole object next to the identifier of the specific field needs to be forwarded to the `DefUseAnalyzer`. Hence, these different types are handled individually with corresponding method invocations to `DefUseAnalyzer` fitting their information.

In order to account for inter-procedural DUCs, at each new method invocation it is tracked which values are passed on from the current method to the invoked method. As the variables may be differently named and stored at different indexes in the variable tables of the two methods, this information is necessary to align the variables in the two methods. For instance, in Listing 3 in Section 2 at the point in line 3, it has to be stored that the variable `x` from `method1` is equal to the variable `z` to be able to later assign variable usages of `method2` to variable definitions in `method1` assuming no other definitions of `z` are made in `method2`.

When all `MethodNode`s of the class have been iterated and bytecode instructions accordingly added, a `ClassWriter` is utilized to convert the modified `ClassNode` into a byte array which is returned and loaded into the Java Virtual Machine. Thus, the program is executed with the method calls to the `DefUseAnalyzer` gathering the relevant data-flow information.

3.3 Dynamic Analysis

As described previously, the `DefUseAnalyzer` is responsible for collecting and analyzing the necessary data-flow information. The corresponding class is depicted in Figure 2. All fields and methods are static to enable the analysis throughout multiple classes as the bytecode transformation is performed for each class individually. It can be seen that the `DefUseAnalyzer` has three fields for collecting the information. The field `chains` comprises all DUCs which were identified until the current execution point. This is represented as `DefUseChains` (see Figure 1). The second field `defs` collects all variable definitions. As a DUC is defined so that on the path from the definition to the usage of a variable there is no other definition (see Section 2), `defs` is implemented in such a way that that the most recent definition comes first. This allows for a quick allocation of a variable usage to its most recent definition. The third field `interMethods` contains the allocation of variables passed on to methods as described in the previous section. A new allocation is added every time for every parameter when a method is invoked with the information from the calling method, i.e. the passed variable and invoked method name. Whenever a new method is entered, `interMethods` is checked against the given parameters and the information is complemented to allocate both sides, the information at the calling method and the invoked method.

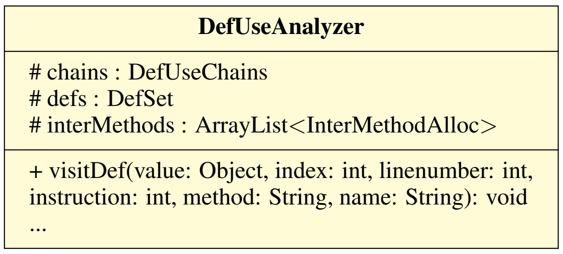
DefUseAnalyzer
# chains : DefUseChains
# defs : DefSet
# interMethods : ArrayList<InterMethodAlloc>
+ visitDef(value: Object, index: int, linenumber: int, instruction: int, method: String, name: String): void
...

Figure 2: An overview of the implementation of the class DefUseAnalyzer.

Invocations to the methods were added to the inspected program at the appropriate positions during the bytecode transformation. Each method then forwards its information for the analysis. For instance, during the execution when the method `visitDef` is invoked with the corresponding information, a new definition is added to `defs`. 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`.

4 EVALUATION

In order to evaluate and validate the results of this data-flow analysis tool, we executed it for a set of different examples. Eight examples are retrieved from the SV-COMP set of software verification (Beyer, 2021). SV-COMP is an annual competition for software verification that releases a publicly available repository as a benchmark suite for verification and validation software tools (Beyer, 2021).

We selected different types of algorithms, e.g., sorting, searching, highly conditional, and recursive algorithms, to showcase the wide applicability of this tool. Next to the example presented in this Section, we utilized eight different examples from the SV-COMP benchmark: The recursive Fibonacci algorithm, recursive algorithms to determine if a given number is even or odd and to check whether a number is a prime number. Moreover, a recursive algorithm that determines the greatest common divisor for

Table 1: Executed examples for the data-flow analysis. Run times in ms without and with instrumentation.

Example	**without instrum.**	**with instr.**
EvenOdd.evenOdd	4	151
Fibonacci.fibonacci	3	123
Hanoi.hanoi	3	145
InsertionSort.sort	2	177
SatPrime01.is_prime	3	142
RecursiveGcd.gcd	19	169
EuclidianGcd.egcd	10	148
TspSolver.search	12	260
BinaryTreeSearch.search	5	241

two given numbers, a recursive implementation of the Hanoi problem, and an insertion sort algorithm are regarded. These examples exhibit intra- and inter-procedural data flow as well as different data structures such as arrays. As more complex examples to account for aliasing and data flow between different classes, an implementation of the traveling salesperson and an implementation of the binary tree search are given.

For all of the mentioned examples, JUnit test cases were added for the execution. When executing Dacite on these examples, overhead is introduced to the program in form of additional instructions in order to collect and analyze the data flow. Hence, it is reasonable that the JUnit execution with Dacite takes longer than the straightforward execution. In order to analyze this overhead, every example was executed with and without Dacite and consequently, with and without the instrumentation. The results can be seen in Table 1. While the runtime of the execution with instrumentation is higher than without, it is still considerably small ranging in milliseconds and thus, acceptable when analyzing Java programs.

```
1  public int egcd(int a, int b) {
2      if (a == 0)
3          return b;
4      while (b != 0) {
5          if (a > b)
6              a = a - b;
7          else
8              b = b - a;
9      }
10     return a;
11 }
```

Listing 6: An implementation of the Euclidian greatest common divisor.

As other data-flow analysis tools are either not available or not working correctly, a validation of the results was conducted manually. For each example, the possible DUCs were derived by hand and compared to the results of the dynamic analysis. For instance, consider the implementation of the Euclidian greatest common divisor in Listing 6.

This method has 18 possible DUCs as follows, nine DUCs for the variable a and analogously, nine for the variable b:

$$\begin{array}{lll}
(a, & egcd(\text{int } a, \text{int } b), & \text{if}(a == 0)) \\
(a, & egcd(\text{int } a, \text{int } b), & \text{if}(a > b)) \\
(a, & egcd(\text{int } a, \text{int } b), & a = a - b) \\
(a, & egcd(\text{int } a, \text{int } b), & b = b - a) \\
(a, & egcd(\text{int } a, \text{int } b), & \text{return } a) \\
(a, & a = a - b, & \text{if}(a > b)) \\
(a, & a = a - b, & a = a - b) \\
(a, & a = a - b, & b = b - a) \\
(a, & a = a - b, & \text{return } a) \\
(b, & egcd(\text{int } a, \text{int } b), & \text{return } b) \\
\ldots & &
\end{array}$$

As explained previously, these DUCs are derived manually. The only available and comparable static tool Jabuti (Vincenzi et al., 2003) was able to analyze this program as well. However, the result was not correct as it identified 23 DUCs. This highlights again the missing existence of a tool that is able to reliably identify the data flow for a given program and the added value this tool contributes.

Given a JUnit test case like in Listing 7, all these DUCs are identified by the dynamic analysis of Dacite. Moreover, when considering that the method is called by another method (inter-procedural data flow), this could result in DUCs no longer being traversable. For instance, if the calling method limits the range of values of the variable a to $a > 0$, then the condition in the first if-statement in line 2 will never be true. Consequently, the DUC $(b, egcd(\text{int } a, \text{int } b), \text{return } b)$ is no longer traversable in context of the program although the static analysis of the method still yields this DUC. However, through the dynamic analysis occurrences like this are not considered as only traversable DUC are taken into account.

5 RELATED WORK

As already stated in the introduction, in contrast to metrics such as branch coverage, there do not exist many tools to assess the data-flow coverage for a Java program (Su et al., 2017). JaBUTi (Java Bytecdoe Understanding and Testing), is one coverage tool that is able to identify data flow (Vincenzi et al., 2003). It statically analyzes the JVM bytecode to derive the DUCs. This is achieved by first con-

```
    @Test
    public void testGCD(){
        int i = egcd(94, 530);
        assertEquals(2, i);
        int i2 = egcd(940, 530);
        assertEquals(10, i2);
        int i3 = egcd(4, 4);
        assertEquals(4, i3);
        int i4= egcd(0,2);
        assertEquals(2, i4);
    }
```

Listing 7: An exemplary JUnit test case for the method for the Euclidian greatest common divisor in Listing 6.

structing a control-flow graph and then augmenting it with the data-flow information (Vincenzi et al., 2006). JaBUTi offers a graphical visualization of the identified data-flow graph and considers inter-procedural data flow. However, it is not able to identify the data flow of individual elements of objects and arrays, detect aliases, or distinguish between traversable and non-traversable DUC. As already mentioned in Section 4, this tool also does not identify the def-use relations reliably, often identifying more DUCs than the program actually has.

Another tool is DFC (Data Flow Coverage), which is an Eclipse plug-in for analyzing the data-flow coverage. Instead of completely deriving the data flow automatically, this is done by interacting with the user and inquiring first which methods modify object states and which only use them. This information is then utilized to statically constructs a def-use graph (DUG) based on the source code. While this tool also offers a simple graphical visualization of the DUG, it does only focus on intra-procedural DUCs, regards objects and arrays as a whole for the data-flow analysis (Bluemke and Rembiszewski, 2009). This tool, however, is not publicly available.

DaTeC (Data flow Testing of Classes) focuses on deriving data-flow information for integration testing and thus, inter-procedural information across methods and classes. Therefore, the authors enhance the definition-usage information by *contextual* information containing a chain of method invocations that led to the respective definition or usage. This is achieved by statically analyzing the JVM bytecode. While this tool regards objects and their fields in detail for the data-flow analysis, arrays are treated as a whole (Denaro et al., 2008). Since the tool is based on a static analysis, it cannot distinguish between traversable and non-traversable DUCs. While the source code of this tool is open-source, its dependencies and code are outdated leading to a not executable tool.

Another tool is DuaF (Def-Use Association Forensics) (Santelices and Harrold, 2007). It statically analyzes the intra- and inter-procedural data flow. Afterwards, the program is instrumented to be able to monitor the identified data flow and then executed. Thereby, it offers the possibility to utilize branch coverage to infer the covered DUCs to reduce the overhead. For this purpose, the DUCs are categorized during the dynamic monitoring into three categories, i.e. definitely covered, possibly covered, and not covered. If the users want a more precise result, they can also choose to directly monitor the identified DUCs. As the analysis is performed statically, the tool is restricted to the limitations of a static analysis as mentioned before (Santelices and Harrold, 2007). Although it is able to identify some form of aliases, these are derived using approximations which leads to a reduced precision in comparison with a dynamic analysis (Pande et al., 1994; Denaro et al., 2014). This tool, however, is also not publicly available.

These four tools are the more advanced and detailed tools for statically analyzing the data flow. There also exist some tools such as Jazz (Misurda et al., 2005) or BA-DUA (de Araujo and Chaim, 2014) which derive the data flow more superficially. However, all of them have some limitations due to their static nature. Execution information like the aliasing information cannot be accessed during static analysis. Moreover, it is in general not possible to distinguish between traversable and non-traversable DUCs for static tools (Su et al., 2017). For instance, in the field of test-case generation, this could lead to time spent finding a test case for a DUC which is not traversable during execution. On top of that, as could be already noticed, all tools are either not available or not working reliably.

To the best of our knowledge, there exists only one tool which analyzes the data flow of a program dynamically on the level of abstraction of static techniques, DReaDs (Dynamic Reaching Definitions). As the name suggests, this tool focuses on the variable definitions and their reachability. Instead of identifying variable usages and associating definition-usage pairs, for each definition it is tracked to which execution the definition is propagated until it is eventually overridden. This impedes the comparability to other data-flow test tools as the majority focuses on the definition-usage data flow representation. Moreover, this tool only considers the data flow for class state variables, i.e. class fields, and neglects all other defined variables such as local variables defined in methods (Denaro et al., 2014; Denaro et al., 2015). Like most of the other tools, this data-flow tool is also not publicly available.

This demonstrates that to the best of our knowledge our tool Dacite is the only tool that analyzes the data flow of an object-oriented program considering the fields of objects and the elements of arrays in detail, respecting aliasing, focussing on traversable DUCs and hence providing a comprehensive data-flow analysis.

6 CONCLUSIONS

In this paper, we have presented Dacite, a tool for a comprehensive data-flow analysis for object-oriented programs. This tool is able to dynamically analyze the data flow of a given program and its JUnit tests. By collecting the necessary information during the execution, more detailed and precise information such as the occurrences of aliases can be derived. Moreover, we have argued that by dynamically analyzing the program, only traversable data-flow relations are identified. It is noteworthy, that all other existing tools for analyzing the data flow of object-oriented programs only provide limited identification of the data flow, being restricted either by static analysis or by a specific type of variables (see Section 5). Furthermore, these tools are either not publicly available or not working reliably. Consequently, to the best of our knowledge, Dacite is the only available tool to provide a comprehensive and detailed data-flow analysis. This can be utilized to evaluate and assess test suites for object-oriented systems.

In the future, we plan to extend Dacite by graphical visualization of the DUCs to further facilitate the comprehensibility of the passed data flow. Moreover, by deriving the data-flow information during the execution, only those paths and their data flows are considered that have been passed. When comparing JUnit tests, this is sufficient. However, when generating new Junit tests, it would be beneficial to also have information on which data flows have not been passed yet. In order to ensure that all data flows are covered, Dacite can be combined with our existing test-case generator (Winkelmann et al., 2022) which is based on a symbolic execution of the classes under test. Dacite can then be used to restrict the number of generated test cases to those which are required to ensure data-flow coverage.

REFERENCES

Allen, F. E. and Cocke, J. (1976). A program data flow analysis procedure. *Communications of the ACM*, 19(3):137.

Beyer, D. (2021). Software verification: 10th comparative evaluation (sv-comp 2021). *Proc. TACAS (2). LNCS*, 12652.

Bluemke, I. and Rembiszewski, A. (2009). Dataflow testing of java programs with dfc. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 215–228. Springer.

de Araujo, R. P. A. and Chaim, M. L. (2014). Data-flow testing in the large. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 81–90. IEEE.

Denaro, G., Gorla, A., and Pezzè, M. (2008). Contextual integration testing of classes. In *International Conference on Fundamental Approaches to Software Engineering*, pages 246–260. Springer.

Denaro, G., Margara, A., Pezze, M., and Vivanti, M. (2015). Dynamic data flow testing of object oriented systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 947–958. IEEE.

Denaro, G., Pezze, M., and Vivanti, M. (2014). On the right objectives of data flow testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 71–80. IEEE.

Frankl, P. G. and Iakounenko, O. (1998). Further empirical studies of test effectiveness. *SIGSOFT Softw. Eng. Notes*, 23(6):153–162.

Harrold, M. J. and Soffa, M. L. (1989). Interprocedual data flow testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):158–167.

Majchrzak, T. A. and Kuchen, H. (2009). Automated test case generation based on coverage analysis. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 259–266.

Misurda, J., Clause, J., Reed, J., Childers, B. R., and Soffa, M. L. (2005). Jazz: A tool for demand-driven structural testing. In *International Conference on Compiler Construction*, pages 242–245. Springer.

Pande, H. D., Landi, W. A., and Ryder, B. G. (1994). Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403.

Santelices, R. and Harrold, M. J. (2007). Efficiently monitoring data-flow test coverage. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 343–352.

Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., and Su, Z. (2017). A survey on data-flow testing. *ACM Computing Surveys (CSUR)*, 50(1):1–35.

Vincenzi, A., Wong, W., Delamaro, M., and Maldonado, J. (2003). Jabuti: A coverage analysis tool for java programs. *XVII SBES–Simpósio Brasileiro de Engenharia de Software*, pages 79–84.

Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., and Wong, W. E. (2006). Establishing structural testing criteria for java bytecode. *Software: practice and experience*, 36(14):1513–1541.

Winkelmann, H., Troost, L., and Kuchen, H. (2022). Constraint-logic object-oriented programming for test case generation. In *Proceedings of the 37th ACM/SIGAPP Symposium On Applied Computing*, pages 1490–1499.