Exercises

Static and Dynamic Slicing

Points	Submission due	Status
0 / 30	Jul 22, 2025 17:00	No graded result

Tasks:

Static and Dynamic Slicing

In this third graded assignment, we will implement the dynamic slicing algorithm for Java byte code. Program slicing computes a set of program statements, called the *program slice*, that may affect the value of variables at some point in the program (that is, the *slicing criterion*). Slicing was originally introduced by Mark Weiser [Wei81, Wei84]. Since then, many different approaches have been introduced. A comprehensive overview over the various slicing techniques can be found in the survey of Silva [Sil12].

For this assignment, we will implement an *intra*-procedural dynamic slicer for Java. We will work at bytecode level, so you can build up on the experience gained while solving the previous assignment. The slicer takes as input a program P (an arbitrary Java class file in an arbitrary package available in the classpath), a slicing criterion S, and a test to execute (for dynamic slicing). The slicing criterion S of a program P is a tuple $\langle s, V \rangle$, where s is a program statement and V is a variable set of P. The output of the tool is the dynamic backward slice for the variables in V from the method start to the program statement s. *Note:* since we are considering an intra-procedural analysis, we limit the slice to the method containing the statement s.

In practice, various techniques exist to compute a slice. A popular—and straight forward—technique is to use a *Program Dependence Graph* (PDG). A PDG combines the control and data dependencies into one graph structure [FOW87, HAR02]. To compute a dynamic slice for a slicing criterion S, one can follow a two-step approach:

1. Simplify the PDG to obtain the *reduced PDG* by removing the *nodes* of the original PDG that do not belong to the history of the program execution—also known as the trace.
2. Compute the static slice on the *reduced* PDG, that is, traverse the *reduced* PDG backwards starting from the node that corresponds to the location s of the criterion S.

Implementation

Your task is to implement the dynamic slicing algorithm as presented in the lecture. This means, you need to compute a static slice on the *reduced* PDG starting at the location defined by the slicing criterion. Before you can compute the *reduced* PDG, you must have traced the program execution. Then, you need to remove the nodes from the original PDG that have not been covered by the execution. When doing so, remember to also remove the edges from the original PDG that are not valid anymore due to the removal of the nodes.

To implement this assignment you can refer to the topics covered in the lectures about control and data flow (for building the Program Dependence Graph), and the lectures on dynamic analysis and slicing (for computing the intra-procedural dynamic backward slice).

Static Slicing

We split the implementation task into two subtasks. This allows you to get some points on functionality, even if your implementation lacks some functionality. Please note that you cannot implement dynamic slicing without static slicing.

For the static backward slicing, please consider the following steps:

- Calculate the static backward slice, that is, a set of nodes from the (reduced) PDG beginning at a given node. This is done in the method `Set<Node> backwardSlice(Node)` of the `ProgramDependenceGraph` class. The parameter of the method is the node representing the slicing criterion and the variable to look at. It returns a set of nodes that are the static backward slice. The output routine does not expect a specific ordering of the elements in this set.
- Implement the computation of the program dependence graph in the class `ProgramDependenceGraph` by combining the control and data-dependence graph of a given method.
- Implement the computation of the control-dependence graph in the class `ControlDependenceGraph` as shown in the lecture. In the lecture slides, the final step of constructing the CDG is to mark all nodes that are not yet control dependent to be dependent on the entry node. For the sake of slicing, this step is not necessary and thus we do not include nodes being dependent on the entry to our CDGs.
- Implement the computation of the post-dominator tree in the class `PostDominatorTree` as described in the lecture.
- Implement the computation of the data-dependence graph in the class `DataDependenceGraph` based on the reaching-definitions algorithm as shown in the lecture. To compute the data-dependence graph and the reaching-definitions, you'll have to use the not yet implemented `definedBy()` and `usedBy()` methods of the `DataFlowAnalysis` class. In order to implement these two methods, you can make use of the asm-defuse library.

The `PostDominatorTree`, `ControlDependenceGraph`, `DataDependenceGraph`, and `ProgramDependenceGraph` classes extend the abstract class `Graph`, which sets up the *control-flow graph* (CFG) in its contructor and provides the abstract method `computeResult()`. Missing implementations are marked by comments (`// TODO`) and throw `UnsupportedOperationException`s.

The framework contains an implementation of a `Node` type for nodes, the class `ProgramGraph`, which basically wraps a graph in the JGraphT library, and a `CFGExtractor` that creates the CFG from the Java byte code, using ASM.

Furthermore, the framework provides additional helpers related to the CFG and finding the right node for the slicing criterion. It is not permitted to change these classes, neither is it permitted to change the parameter passing nor the output routines.

Program Slicing and Dynamic Analysis

The dynamic slicing is built on top of the static slicing, which means that you need to implement the static slicing first. To implement the dynamic slicing, please consider the following steps:

- Given the PDG, you need to compute the reduced PDG by removing all the nodes (and corresponding edges) not covered by the given test execution in the `simplify()` method of the `SlicerUtil` class. Please note: the lecture also covered the creation of dynamic dependency graphs (DDGs). However, we do not consider DDGs in this assignment, and therefore you do not need to implement the DDG.
- We use a Java Agent for the on-the-fly instrumentation of the byte code, which allows us to manipulate the byte code during its loading. This shall be done with the ASM framework. The skeleton and the necessary settings for JAR generation are already predefined. Your task is to implement the byte-code manipulation in class `LineCoverageTransformer` and `InstrumentationAdapter`. Use the *core API* of ASM for this.
- The class `CoverageTracker` shall be used for the collection of execution data. It is partially implemented. Calling the static method `trackLineVisit` registers the execution of a line of code.
- In order to collect information about which lines of code were executed it is necessary to programmatically trigger the execution of a given test method (cf. the `-d` flag). Implement the execution of this test in `executeTest()` in class `SlicerUtil`. Refer to the JUnit 5 user guide for details.

Checks

We provide you a selection for graphs (CFGs, PDTs, CDGs, DDGs, PDGs) as dot and png files for download. The `ProgramGraph` class has a `toString` implementation that yields the respective graph in the dot format. Please note that GraphViz' dot layouting is not stable, i.e., while the graphs are isomorphic, they nodes and edges are not necessarily placed in the exact same positions every time you run the tool. In case of doubt, compare the dot files directly (they are plain-text strings, defining `<src> -> <tgt>` pairs for edges from source to target nodes).

Checks at the Unit Level

1. ◉ **Correct implementation of the Graph reversal** No results
2. ◉ **Correct implementation of the post-dominator tree** No results
3. ◉ **Correct implementation of the control-dependence graph** No results
4. ◉ **Correct implementation of the data flow analysis** No results
5. ◉ **Correct implementation of the data-dependence graph** No results
6. ◉ **Correct implementation of the program-dependence graph** No results
7. ◉ **Correct implementation of the backward slice** No results
8. ◉ **Correct implementation of CoverageTracker's trackLineVisit method** No results
9. ◉ **Correct implementation of the byte-code instrumentation** No results

Checks at the Functional Level

The following checks run the command line parameters provided in the respective `*.txt` file in the `expected-results` folder of the assignment. These files also provide the expected output. You can use these files, take the parameters, and compare your results against the expected results. For your convenience, you might want to use the `run.sh` script, which provides the same parameters as the tool, but combines the call to the JAR archive.

Static Slicing Configurations: these configurations only require static slicing.

1. ◉ **Correct output on the Calculator class** No results
2. ◉ **Correct output on the Complex class** No results
3. ◉ **Correct output on the GCD class** No results
4. ◉ **Correct output on the NestedLoop class** No results
5. ◉ **Correct output on the Rational class** No results
6. ◉ **Correct output on the SimpleInteger class** No results
7. ◉ **Correct output on the TestClass class** No results

Dynamic Slicing Configurations: these configuration require also dynamic slicing.

1. ◉ **Correct output on the Calculator class** No results
2. ◉ **Correct output on the GCD class** No results
3. ◉ **Correct output on the NestedLoop class** No results
4. ◉ **Correct output on the SimpleInteger class** No results
5. ◉ **Correct output on the TestClass class** No results

Coverage Values

1. ◉ **85% Branch Coverage** No results
2. ◉ **90% Line Coverage** No results
3. ◉ **70% Mutation Score** No results

Exercise details

Release date	Jul 2, 2025 10:00
Submission due	Jul 22, 2025 17:00
Complaint possible	No