# A Program Slicer for Java (Tool paper)*

Carlos Galindo[0000−0002−3569−6218], Sergio Perez[0000−0002−4384−7004], and Josep Silva[0000−0001−5096−0008]

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain
cargaji@vrain.upv.es, {serperu,jsilva}@dsic.upv.es

**Abstract.** Program slicing is a static analysis technique used in debugging, compiler optimization, program parallelization, and program specialization. However, current implementations for Java are proprietary software, pay-per-use, and closed source. Most public and open-source implementations for Java are not maintained anymore or they are obsolete because they do not cover novel Java features or they do not implement advanced techniques for the treatment of objects, exceptions, and unconditional jumps. This paper presents *JavaSlicer*, a public and open-source tool written in Java for slicing Java programs, which supports the aforementioned features. We present its usage, architecture, and performance.

**Keywords:** Program slicing · System Dependence Graph · System demo

## 1 Introduction

Program slicing is a static analysis technique used to automatically identify *what parts of a program may affect the value of a variable at a given position* (static backward slicing) or *what parts of a program may be affected by the value of a variable at a given position* (static forward slicing). The program point of interest (a set of variables in a line) is known as *slicing criterion*. The output, or *slice*, is the subset of the program that affects the slicing criterion.

Program slicing can be likened to automated scissors for code: given a pattern to target (a slicing criterion) it will remove all the code that is not relevant to that pattern. Consider Figure 1, in which a very simple program has been sliced. The criterion $\langle 10, sum \rangle$ indicates that we are interested in the elements that

* This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Sergio Pérez was partially supported by Universitat Politècnica de València under FPI grant PAID-01-18. Carlos Galindo was partially supported by the Spanish Ministerio de Universidades under grant FPU20/03861.

```
1  void f(int n, int m) {
2    int sum = 0;
3    int prod = 0;
4    int i = 0;
5    while (i < m) {
6      sum += n;
7      prod *= n;
8      i++;
9    }
10   log(sum);
11   log(prod);
12 }
```

**Fig. 1.** A simple Java program and its slice w.r.t. $\langle 10, sum \rangle$ (in black).

affect the value of the variable *sum* at line 10. The resulting slice has removed the lines used to compute *prod* because they have no influence on *sum*.

Program slicing is particularly useful for debugging (where the slicing criterion is a variable containing an incorrect value and, thus, the slice must contain the bug), but there are many other applications such as program specialization, and program parallelisation. Unfortunately, currently, there does not exist a public and open-source program slicer for modern Java since the existing ones are obsolete or proprietary. For instance, there does not exist a plug-in for IntelliJ IDEA or Eclipse, two of the most popular Java IDEs in the market.

*JavaSlicer* is a library and terminal client that creates slices for Java programs, using the System Dependence Graph (SDG). Its current version is JavaSlicer 1.3.1 (aka *scissorhands*). In this paper, we present its usage, structure, underlying architecture, and performance.

## 2 Background

The most common data structure used to slice a program is the System Dependence Graph (SDG) [3], a directed graph that represents program statements as nodes and the dependences between them as edges. Once built, a slice can be computed in linear time as a graph reachability problem by selecting the node that represents the slicing criterion and traversing the edges backwards/forwards (for a backward/forward slice, respectively).

The SDG itself is built from a sequence of graphs: each method is used to compute a Control-Flow Graph (CFG), then control and flow (aka data) dependences are computed and they are stored in a Program Dependence Graph (PDG). Finally, the calls in each PDG are connected to their corresponding declarations to form the SDG, making it the union of all the PDGs.

To compute a slice, the slicing criterion is located, and then a two-phase traversal process is used (so that the context of each call is preserved), which produces a set of nodes that can then be converted back to code or processed in other ways.

## 3 Producing slices with *JavaSlicer*

*JavaSlicer* is a very sophisticated tool that implements the SDG and its corresponding slicing algorithms with advanced treatment for object-oriented (OO)

features [5], exception handling [1], and unconditional jumps [4] (in Java, `break`, `continue`, `return` and `throw` are unconditional jumps). It includes novel techniques that improve the until now most advanced representation of OO programs, the *JSysDG* [2]. It is free/libre software and is publicly available at https://github.com/mistupv/JavaSlicer under the AGPL license. The sources can be built by using maven (following the instructions in the README), or a prebuilt jar can be downloaded from the releases page[1].

With the *sdg-cli.jar* file, an installation of Java 11 or later, and the Java sources that are to be sliced, producing a slice for is a simple task. E.g., for a file called *Example.java* and the slicing criterion $\langle 10, x \rangle$, the command would be:

```
$ java -jar sdg-cli.jar -c Example.java:10#x
```

The slice will be placed in a *slice* folder (which will be created if it does not already exist). The parameter `--output` or `-o` can set the output directory to any other location. The slicing criterion is given using the `--criterion` or `-c` parameter, with the following format: `FILE:LINE#VAR`. Alternatively, the criterion can be split into the `--file`, `--line`, and `--var` parameters.

### 3.1 Slicing more than one file

Most non-trivial programs are spread across multiple files, so it is also possible to produce slices w.r.t. a whole project. An additional parameter (`--include` or `-i`) must be passed so that a SDG is generated with all the files that make up the program. Assuming that the project is inside *src* and that the slicing criterion is $\langle 10, x \rangle$ in *src/Main.java*, the command would be:

```
$ java -jar sdg-cli.jar -i src -c src/Main.java:10#x
```

Any file from the project from which statements are included in the slice will appear in the *slice* folder. If the project is spread across multiple modules, they can be concatenated with commas (i.e., `-i x/src,y/src`).

### 3.2 Slicing with external libraries

A limitation of *JavaSlicer* is that the project must be compilable, so any external dependency must be included either in the SDG (as shown in the previous section) or added to Java's classpath. To do so, we can use the `-cp` parameter, concatenating multiple libraries with semicolons. For example, to slice a small program that depends on *JGraphT* with slicing criterion $\langle 25, res \rangle$, the command would be:

```
$ java -cp jgrapht-1.5.0.jar -jar sdg-cli.jar -c Graphing.java:25#res
```

[1] Available at https://github.com/mistupv/JavaSlicer/releases

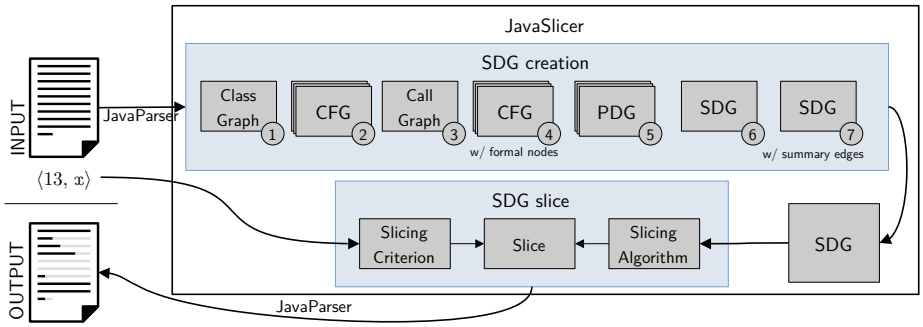


**Fig. 2.** Sequence of events that slice a program in *JavaSlicer*.

Of course, transitive dependencies must also be included (in our case, we would need to include *JGraphT*'s dependencies).

Each module, library, and dependency in a project must be included via `-i` or `-cp`. However, the SDG's behaviour changes in each. With the former, the files are included in the SDG (they are parsed, analysed, its dependences are computed, etc.), increasing precision but making the analysis take longer and more memory. The latter does not take into account the body of each function, speeding up the process at the cost of some precision. This gives the user the freedom of including/excluding specific libraries from the analysis.

## 4 Implementation

*JavaSlicer* is a Java project with 9.3K LOC[2] in two modules:

***sdg-core:*** The main program slicing library, which contains multiple variants of the SDG with their corresponding slicing algorithms.
***sdg-cli:*** A simple client that uses the core library to produce slices.

The main module contains all the data structures and algorithms required to run the slicer. Slicing a program with the library is as simple as creating a new SDG, building it with the parsed source code, and slicing it w.r.t. a slicing criterion. Internally, the construction of the SDG follows a 7-step process: (1) Compute the class graph (connecting classes, their parents and members) from the parsed sources. (2) Compute the control-flow arcs to create a CFG for each method in the program. (3) Compute the call graph, which represents methods as nodes and calls between them as edges. (4) Perform a data-flow analysis to locate the formal-in and formal-out variables, adding markers to the CFG such that formal nodes will be placed in the PDG. (5) Compute control and data dependence for each CFG, creating its corresponding PDG. (6) Transform the PDGs into the associated SDGs, connecting each call site to its corresponding

[2] Measured at release 1.3.1, excluding whitespace and comments, measured with `cloc`.

**Table 1.** Time required to build and slice *re2j* (release 1.6).

| Slice size range (SDG nodes) | # SCs | Build time (s) | Slice time (ms) |
|---|---|---|---|
| $[0, 100)$ | 49 | $13.35 \pm 0.07$ | $0.927 \pm 0.018$ |
| $[100, 1000)$ | 95 | | $217.315 \pm 2.874$ |
| $[1000, 1400)$ | 122 | | $1164.423 \pm 13.093$ |
| $[1400, 1800)$ | 146 | | $1584.023 \pm 12.429$ |
| $[1800, \infty)$ | 31 | | $2943.965 \pm 15.702$ |
| $[0, \infty)$ - **Averages** | 443 | $13.35 \pm 0.07$ | $1095.440 \pm 12.039$ |

declaration (using the call graph). (7) Compute the summary arcs between each pair of actual-in and actual-out nodes that belong to the same call.

Finally, the graph can be stored for repeated use or a slice can be generated. Each child class of SDG contains a reference to the correct slicing algorithm required to slice it. The user only has to provide enough information to locate the node(s) that represent the slicing criterion (via an instance of *SlicingCriterion*). The resulting slice is a set of nodes from the SDG, that can be studied as-is or converted back to source code via *JavaParser*.

Figure 2 summarises the process through which the source code and slicing criterion are employed to build and slice the SDG.

## 5 Empirical evaluation

To evaluate the capabilities and performance of our tool, we chose *re2j*, a Java library written by Google to handle regular expressions. It contains 8.1K LOC across 19 Java files. We generated the SDG and then sliced it once per return statement (using the value being returned as the slicing criterion). In total we performed 443 slices. We repeated each action a hundred times to obtain the average execution time with error margins (99% confidence).

The results are summarised in Table 1. To show more relevant values, we grouped the slices by slice size, showing that the time required to slice scales linearly with the number of nodes traversed. Our tool produces slices between one and four orders of magnitude faster than it builds the SDG, which is expected and fits well into the typical usage of program slicers, in which the graph is built once and sliced multiple times. The amount of time dedicated to each phase in the creation of the graph can be seen in Figure 3.

## 6 Related work

The most similar tool in the state of the art is Codesonar, a proprietary tool for C, C++, and Java, that is being sold by *grammatech*©. On the public side, unfortunately, most Java slicers have been abandoned. For instance, Kaveri is an Eclipse plug-in that contains a program slicer, but it has not been updated since 2014 (8 years) and cannot work with maintained releases of Eclipse. The reason

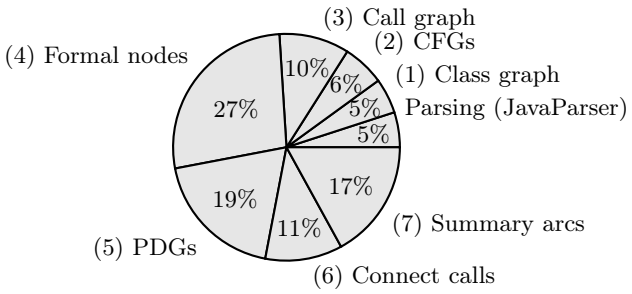


**Fig. 3.** Breakdown of the time dedicated to each step of the creation of the SDG.

is, probably, the difficulty of dealing with the new features of Java (functional interfaces, lambda expressions, record types, sealed classes, etc.). There is still, however, a public program slicer maintained for Java: the slicer contained in the WALA (T. J. Watson Libraries for Analysis, for Java and JavaScript) libraries. Unfortunately, this slicer does not implement the advanced extensions of the SDG for object-oriented (OO) features.

## 7 Conclusions

*JavaSlicer* is a novel free-software program slicing tool for Java. It efficiently implements the most advanced extensions of the SDG, including all the JSysDG extensions for object-oriented programs (inheritance, interfaces, polymorphism, etc.); specific exception handling treatment (`throw`, `try-catch`, etc.); and unconditional jumps (`return`, `break`, `continue`, etc.). It is both a library that can be used by other systems, and a standard program slicing tool.

## References

1. Allen, M., Horwitz, S.: Slicing java programs that throw and catch exceptions. SIGPLAN Not. **38**(10), 44–54 (June 2003)
2. Galindo, C., Pérez, S., Silva, J.: Data dependencies in object-oriented programs. In: 11th Workshop on Tools for Automatic Program Analysis (2020)
3. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. pp. 35–46. PLDI '88, ACM, New York, NY, USA (1988). https://doi.org/10.1145/53990.53994, http://doi.acm.org/10.1145/53990.53994
4. Kumar, S., Horwitz, S.: Better slicing of programs with jumps and switches. In: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002). Lecture Notes in Computer Science (LNCS), vol. 2306, pp. 96–112. Springer (2002)
5. Walkinshaw, N., Roper, M., Wood, M.: The java system dependence graph. In: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. pp. 55–64 (2003)