Educational Program Visualizations Using Synthetized Execution Information

Rodrigo Mourato
Instituto Universitário de Lisboa (ISCTE-IUL), Portugal

André L. Santos
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL, Portugal

Abstract

Visualization is a powerful tool for explaining, understanding, and debugging computations. Over the years, several visualization tools have been developed for educational purposes. Most of these tools feed visualization engines using the raw program state data available provided by the debugger API. While this suffices in certain contexts, there are situations where additional relevant information could aid in building up more comprehensive visualizations. This paper presents two novel visualizations of Paddle, an educational programming environment based on synthesized program execution information. We generate execution traces and relevant program states through static and dynamic analysis of the execution data. The synthesized information captures program behaviors that facilitate the creation of comprehensive and rich visualizations involving arrays that depict position reads, writes, moves, and swaps.

2012 ACM Subject Classification Social and professional topics → Computer science education; Applied computing → Computer-assisted instruction

Keywords and phrases Introductory programming, visualization, comprehension

Digital Object Identifier 10.4230/OASIcs.ICPEC.2024.10

1 Introduction

Programming educators commonly use illustrations to explain algorithms, in different forms, namely in their slides (possibly with animations), whiteboard explanations in the classroom, or on paper when addressing learners individually. Hence, program visualization tools appeal to many programming educators. However, a study [4] has shown that only about 20% of programming courses regularly use visualization tools and that almost half do not use them at all. The survey included responses from over 250 programming teachers and their students, who were asked about their use of visualization. Visualization tools are more often used by teachers working with younger students. The topics in which visualizations are most often used are introductory programming and data structures and algorithms.

Visualization tools are often integrated with debuggers or execution animators (e.g., [5, 1, 12, 2, 9]), where the tool renders the program state at each step. Except for PandionJ [9], these tools do not perform code analysis for capturing semantic aspects of the program (e.g., variable roles [8]) towards richer visualizations. The visualizations are often a mere alternative graphical representation of the information available in the call stack frames. Furthermore, debuggers do not provide the execution data regarding what happened before the program suspension at a breakpoint, making it difficult to illustrate the current program state in context. This leads to illustrations of program states that are less expressive than those hand-drawn by programming instructors [10], and the overall picture is lost through the debugging process.

In this paper, we describe automated program visualizations based on execution information synthesized from execution data, capturing traces and intents that are conventionally unavailable, such as expression-solving steps, array moves, and array swaps. Our main goal is to provide learners with a richer means to understand some programming basics and principles, such as recursion and expression resolution, and facilitate detailed observation of algorithmic behavior on arrays, including when errors occur. When using our tool, users execute programs normally, and only if needed, may switch views to gain more execution insights without requiring specialized tool knowledge.

We developed a web-based platform that supports a subset of Java, covering all the fundamental primitives for writing algorithms. We present two views with novel characteristics: (a) invocation tree with expression evaluation tracing; and (b) heap view with array history of reads and writes (capturing moves and swaps). These views aim to automate the hand-drawn illustrations of programming instructors using the results of a previous study [10]. In particular, the visualizations of array manipulations are novel concerning the state of the art, as we are unaware of any educational tool that illustrates moves and swaps explicitly (beyond depicting the raw program state step by step).

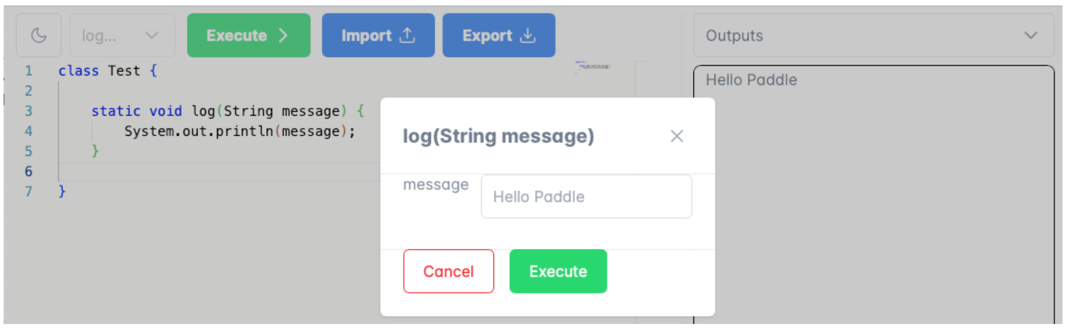
2 Related Work

Software visualization includes two broad areas, algorithm visualization, and program visualization, whereas the latter includes two further areas, visualization of static structures and visualization of runtime dynamics [11]. Algorithm visualization tools operate at a level of abstraction that is too high to be interesting for learning the basics of program execution. Our approach is focused on experimentation and debugging at an introductory level. Here we review tools that allow users to visualize the execution of their programs.

Jeliot [5] is a program animator supporting a subset of Java, where users play an animation of their programs. Visualizations are fine-grained, at the level of expression evaluation. Similarly, UUhistle [12] is a software tool to facilitate visual program simulation. It provides graphical elements that students can manipulate to indicate what happens during execution. The tool displays classes, functions, and operators that the program directly uses, enabling students to receive feedback on different types of errors, verify the accuracy of their answers, and obtain automated grading. These animation tools are useful to illustrate execution, but not practical when solving and debugging exercises because users have to go through the animation without traces of execution available. In our approach, we aim at an environment where programs are executed normally, i.e. in regular settings without any visualizations, and only if desired, behavior may be inspected in an aftermath manner through program traces that illustrate what happened.

Visualization tools are often integrated with debuggers within Integrated Development Environments (IDEs). JIVE is a declarative and visual debugging tool integrated with the Eclipse IDE [3]. jGRASP [2] is an IDE for visualizations to improve software comprehensibility through static and dynamic visualizations of programs. PandionJ [9] is an educational debugger for Java that combines static analysis and graphical visualization towards richer illustrations, such as depicting array iterator variables. While these tools are based on the standard Java debugger, in our present work we rely on a custom execution engine to collect more detailed information that is difficult to obtain otherwise (e.g., applying heavy program instrumentation).

Code Bubbles [7] is capable of displaying the debugging history as a UML sequence graph, the execution history of the current thread when it stops at a breakpoint, and information about a graphical user interface, including the widget hierarchy and the routines drawing at a selected pixel. Furthermore, it provides an interactive read-eval-print loop for the current context and a high-level view of the execution history in terms of threads, tasks, and



```
class Test {

    static void log(String message) {
        System.out.println(message);
    }

}
```

Figure 1 Paddle environment: executing methods.

transactions. This view is generated automatically based on data collected during previous debugging runs. Code Bubbles targets a non-beginner audience, while our tool aims at the first stages of programming learning.

The SRec Visualization System [13] employs graphical representations to illustrate recursion trees. Each node corresponds to a recursive call composed of two halves: the upper half contains the parameter values of the call, while the lower half contains the invocation's result. WinHIPE [6] is an integrated development environment (IDE) for functional programming based on rewriting and visualization. It also includes a powerful visualization and animation system that automatically generates visualizations and animations as a side effect of program execution.

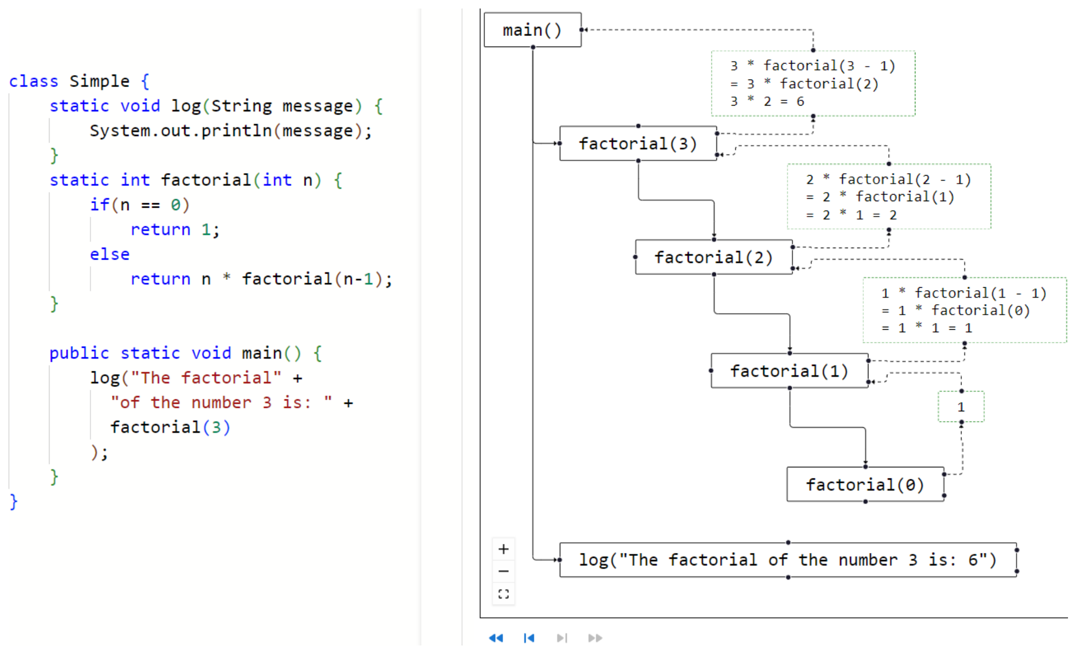
3 Paddle Environment

Paddle is an innovative educational programming environment providing visualizations that leverage synthesized program execution information. It generates representative execution traces and relevant program states through static and dynamic analysis of the execution data. The synthesized information captures diverse program behaviors to facilitate the creation of comprehensive and rich visualizations. The environment consists of a web application where the user can write code and obtain feedback about what happened during the execution as a trace illustration.

The user interface (UI) comprises two panels (see Figure 1): the left panel, where the user writes code and executes programs, and the right panel, where alternative visualization panels are presented. Figure 1 illustrates the elementary view for displaying console outputs. When clicking the "Execute" button, a dialog prompts the user to enter the values for each parameter, and the current code is sent to the server with the specified function and arguments. Afterward, the code result is returned to the web application, and the user may check the outputs and switch among the available visualization panels, which we detail next.

3.1 Invocation Tree View

Figure 2 presents a screenshot of the invocation tree view with the classic example of factorial calculation. Each node in the illustration represents one execution of a method, the solid edges represent invocations, and the dashed edges with the dashed nodes represent the return values of each invocation. If desired, the user may use the playback mode to go through each step, following the sequence of invocations. The related elements are selected in the code editor when clicking the view. When clicking an invocation node the function declaration is highlighted, whereas when clicking a value node the respective return expression is highlighted instead.



```
class Simple {
    static void log(String message) {
        System.out.println(message);
    }
    static int factorial(int n) {
        if(n == 0)
            return 1;
        else
            return n * factorial(n-1);
    }

    public static void main() {
        log("The factorial" +
          "of the number 3 is: " +
          factorial(3)
        );
    }
}
```

Figure 2 Invocation tree view illustrating recursive calls (factorial calculation).

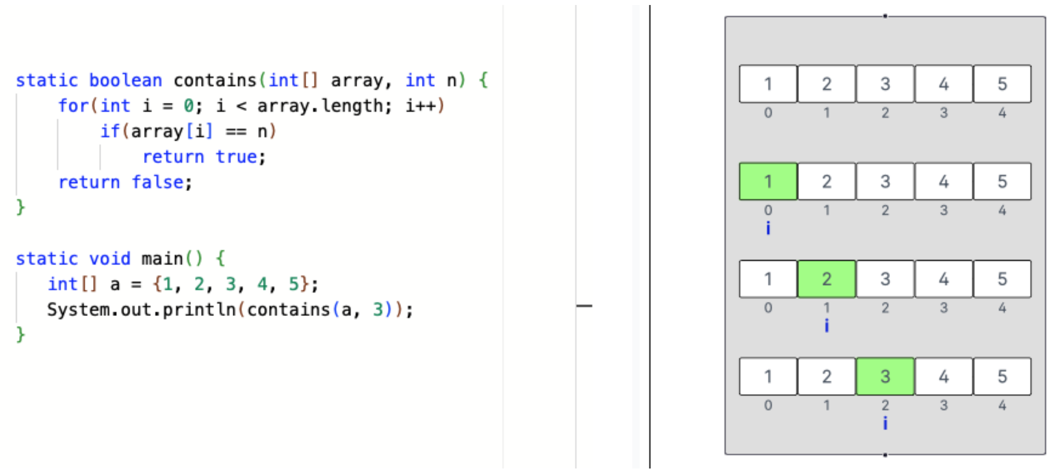
The main innovative feature of our view is the trace of expressions returned by the methods. In the example, the expression `3 * factorial (3 - 1)` is resolved to `3 * factorial(2)` and is finally resolved to `3 * 2`, which returns the final value of `6`. This enables the user to understand the return value of each invocation and how it was calculated. This information is synthesized from execution data, and is not available when using debuggers (both educational and professional). For performance reasons, the total number of resolutions has a limit. Programming instructors often use similar illustrations to explain the execution of recursive calls [10].

3.2 Heap View

Figure 3 presents the heap view illustrating a function to check if an element is contained in an array. This view collects any array allocations performed in user code and renders its evolution through snapshots, from top to bottom. In this case, the array content remains the same because there are no side effects. The green background depicts that the highlighted position was read, whereas red denotes that a write was performed. In the illustration, we can observe that the last accessed position was the third one. The iterator variables for accessing array positions (i in the example) are depicted below the respective index (as in [9]). Programming instructors often use similar illustrations to explain computations that involve array iterations [10].

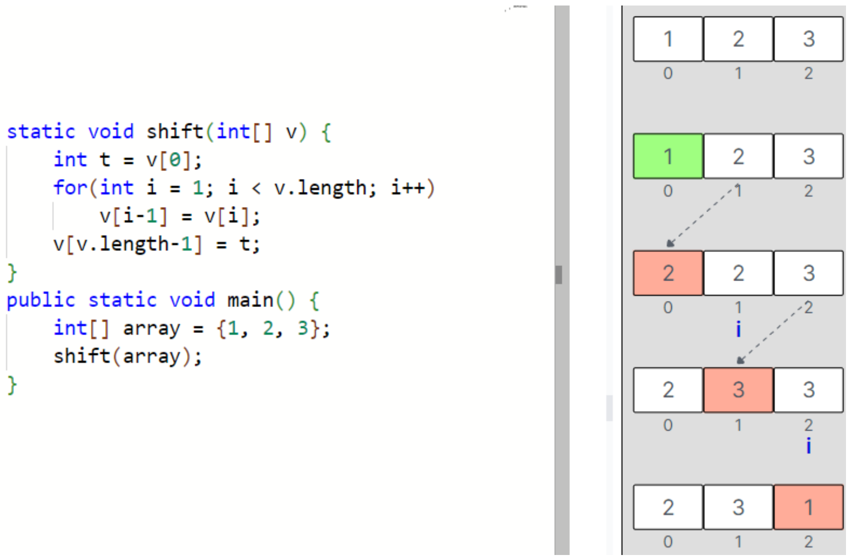
Figure 4 presents the heap view illustrating a procedure for left-shifting an array, exemplifying array writes. In the illustrations, a dashed arrow represents an array position move, that is, a value at one position is copied to another. This information is determined using a combination of static analysis and execution data.

Figure 5 presents the heap view illustrating a procedure to reverse an array. The array was initialized with five elements and the reverse function was invoked, which internally invokes the function to swap two array elements given their indices. Special attention is paid



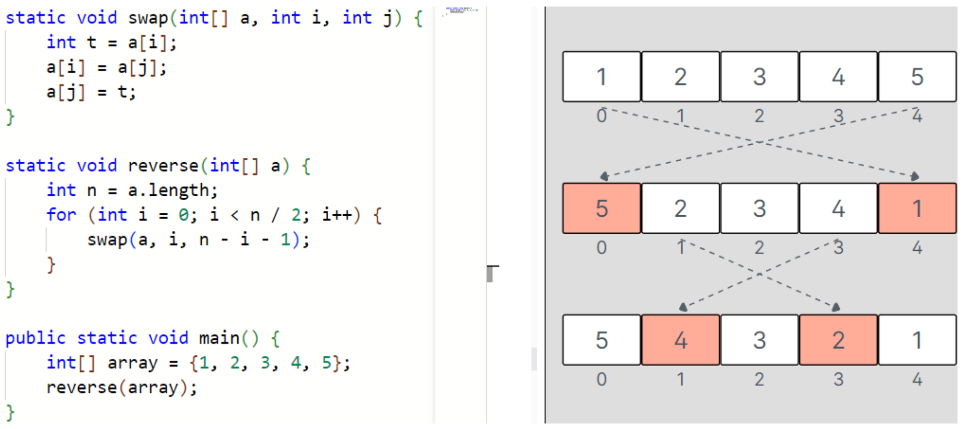
```java
static boolean contains(int[] array, int n) {
    for(int i = 0; i < array.length; i++)
        if(array[i] == n)
            return true;
    return false;
}

static void main() {
   int[] a = {1, 2, 3, 4, 5};
   System.out.println(contains(a, 3));
}
```

Figure 3 Heap view illustrating array reads (check if element exists).



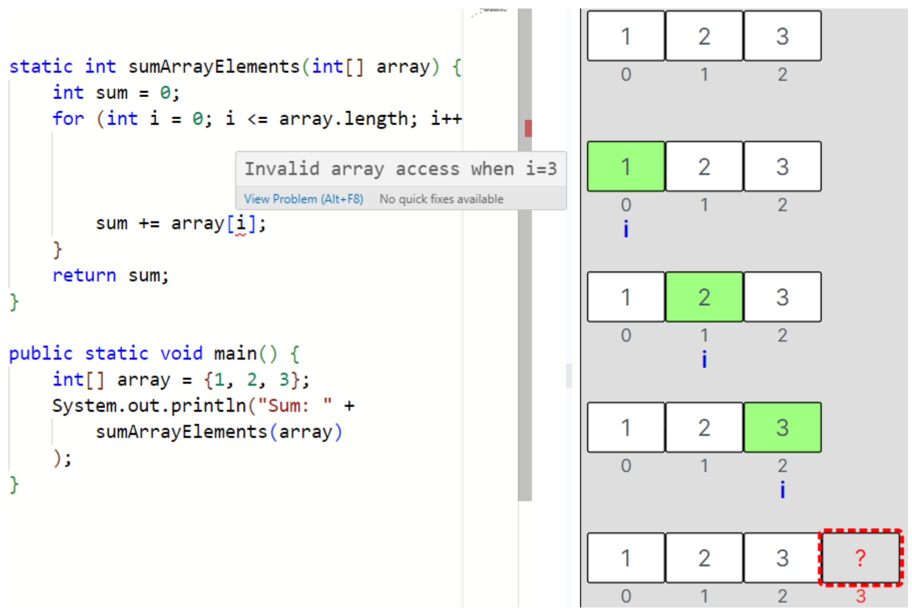
```java
static void shift(int[] v) {
    int t = v[0];
    for(int i = 1; i < v.length; i++)
        v[i-1] = v[i];
    v[v.length-1] = t;
}
public static void main() {
    int[] array = {1, 2, 3};
    shift(array);
}
```

Figure 4 Heap view illustrating array moves (left shift of array elements).



```java
static void swap(int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}

static void reverse(int[] a) {
    int n = a.length;
    for (int i = 0; i < n / 2; i++) {
        swap(a, i, n - i - 1);
    }
}

public static void main() {
    int[] array = {1, 2, 3, 4, 5};
    reverse(array);
}
```

Figure 5 Heap view illustrating array swaps (reverse the array).



```
static int sumArrayElements(int[] array) {
    int sum = 0;
    for (int i = 0; i <= array.length; i++

        sum += array[i];
    }
    return sum;
}

public static void main() {
    int[] array = {1, 2, 3};
    System.out.println("Sum: " +
        sumArrayElements(array)
    );
}
```

```
Invalid array access when i=3
View Problem (Alt+F8)    No quick fixes available
```

Figure 6 Heap view illustrating an illegal access to an array position.

to array swaps – information synthesized from execution data. As in array moves, a dashed arrow represents a move. Since a swap consists of two moves that exchange the values of the positions, the corresponding arrows are depicted simultaneously.

If an array index out-of-bounds error occurs during execution, we illustrate the error in the view, as depicted in Figure 6. The expression that led to the invalid index is also marked with precision in the code. Recall that conventional support for this type of error typically consists of an error message that only includes the line number and invalid index (if multiple array accesses are in that line, the user must figure out which is causing the problem).

4 Implementation

The implementation of our prototype is based on a REST API, where program executions are performed, and a web-based frontend to display the results and visualizations. Ideally, the whole application could run on the browser, but we needed unavailable JavaScript libraries to execute the Java programs and synthesize the required information for the visualizations.

The backend was constructed using Spring Boot[1], a JVM-based framework that simplifies the development of standalone application servers. The API calls respond JSON messages holding the execution results, outputs, traces, etc, that are necessary for building the visualizations.

Program execution and analysis are performed using Strudel[2], a programming library comprising classes that model structured programming, providing a virtual machine capable of interpreting those models, simulating the call stack-based execution. This enables clients to observe every aspect of execution in detail, including errors, tracking variable values, loop iterations, call stack, and memory allocation. We developed execution listeners to gather the

[1] `https://spring.io/projects/spring-boot`
[2] `https://github.com/andre-santos-pt/strudel`

necessary information to render the views. Regarding the resolution of expressions, EvalEx[3] was employed. EvalEx is a convenient expression evaluator for Java that enables the parsing and evaluation of expression strings.

The user interface was implemented using React[4], a popular JavaScript library for user interface development. The Redux Toolkit[5] was used for store management, providing utilities and abstractions to streamline common Redux tasks, such as creating actions, reducers, and store configuration. The code editor is provided by Microsoft Monaco[6], a lightweight, browser-based, highly versatile code editor providing features such as syntax highlighting, code completion, and IntelliSense. Monaco is the engine behind the Visual Studio Code editing experience and can be embedded in Web applications to edit code directly in the browser. Finally, the visualizations were implemented using React Flow library[7], a JavaScript library for developing interactive and visual flowcharts, diagrams, and graphs within React applications. It offers a flexible and customizable API to develop complex data visualization components, thereby enabling developers to incorporate drag-and-drop functionality, node-based layouts, and connection handling with relative ease. This library enabled the creation of custom nodes and edges, as illustrated in this paper's figures.

5 Conclusions and Future Work

Our prototype demonstrates that rich program visualizations can be obtained in a post-execution manner by making use of synthetized execution information. Our visualizations are inspired by illustrations often made by programming instructors (e.g., in slides, animations, or hand-drawn). In particular, the array manipulation illustrations are unavailable in other visualization tools supporting arbitrary user code, and without having to execute the program step-by-step (as when using a debugger). We argue that our views are a quick means to illustrate the execution of simple programs involving invocations and arrays, with minimal need to learn any particular tool features.

As future work, we plan to evaluate how programming instructors perceive the usefulness of our visualizations. Evaluating the tool from the perspective of programming beginners could also inform how easily and accurately they interpret the visualizations. Even if the visualizations have no expressive effect on novices working autonomously, they may serve as an aid to instructors when assisting learners in lab classes or remotely, sparing time that otherwise would be spent on figuring out what went wrong with the program execution and manually drawing illustrations for further explanations.

Regarding tool improvements, we plan to support objects in the heap view, which are important to illustrate elementary data structures such as linked lists and trees and to elaborate on the illustrations of errors (e.g., stack overflows). Furthermore, we believe that more interactivity between the views and the source code could improve the user experience, and we acknowledge that strategies to cope with large drawings are necessary for good usability.

[3] `https://github.com/ezylang/EvalEx`
[4] `https://react.dev`
[5] `https://redux-toolkit.js.org`
[6] `https://microsoft.github.io/monaco-editor`
[7] `https://reactflow.dev`

References

1 G. Cattaneo, P. Faruolo, U.F. Petrillo, and G.F. Italiano. Jive: Java interactive software visualization environment. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 41–43, 2004. `doi:10.1109/VLHCC.2004.34`.

2 James Cross, Dean Hendrix, Larry Barowski, and David Umphress. Dynamic program visualizations: An experience report. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 609–614, New York, NY, USA, 2014. ACM. `doi:10.1145/2538862.2538958`.

3 Jeffrey K. Czyz and Bharat Jayaraman. Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM. `doi:10.1145/1328279.1328286`.

4 Essi Isohanni and Hannu-Matti Järvinen. Are visualization tools used in programming education?: By whom, how, why, and why not? In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 35–40, New York, NY, USA, 2014. ACM. `doi:10.1145/2674683.2674688`.

5 Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers and Education*, 40(1):1–15, 2003.

6 Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. WinHIPE: an ide for functional programming based on rewriting and visualization. *SIGPLAN Not.*, 42(3):14–23, March 2007. `doi:10.1145/1273039.1273042`.

7 Steven P. Reiss. The challenge of helping the programmer during debugging. In *2014 Second IEEE Working Conference on Software Visualization*, pages 112–116, 2014. `doi:10.1109/VISSOFT.2014.27`.

8 Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, HCC '02, pages 37–, Washington, DC, USA, 2002. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=795687.797809`.

9 André L. Santos. Enhancing visualizations in pedagogical debuggers by leveraging on code analysis. In Mike Joy and Petri Ihantola, editors, *Proceedings of the 18th Koli Calling International Conference on Computing Education Research, Koli, Finland, November 22-25, 2018*, pages 11:1–11:9. ACM, 2018. `doi:10.1145/3279720.3279732`.

10 André L. Santos and Hugo Sousa. An exploratory study of how programming instructors illustrate variables and control flow. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research, Koli, Finland, November 16-19, 2017*, pages 173–177, 2017. `doi:10.1145/3141880.3141892`.

11 Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):15.1–15.64, 2013. `doi:10.1145/2490822`.

12 Juha Sorva and Teemu Sirkiä. Uuhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 49–54, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1930464.1930471`.

13 J. Ángel Velázquez-Iturbide and Antonio Pérez-Carrasco. How to use the SRec visualization system in programming and algorithm courses. *ACM Inroads*, 7(3):42–49, August 2016. `doi:10.1145/2948070`.