Visual Breakpoint Debugging for Sum and Product Formulae

Oliver Moseler
Computer Science
University of Trier
Trier, Germany
moseler@uni-trier.de
0000-0003-3118-4968

Michael Wolz
Computer Science
University of Trier
Trier, Germany
kontakt@michaelwolz.de
0000-0002-9313-713

Stephan Diehl
Computer Science
University of Trier
Trier, Germany
diehl@uni-trier.de
0000-0002-4287-7447

***Abstract*—Software debugging is one of the most time consuming source code related tasks. Hence, we propose a novel approach to breakpoint debugging for formula code, i.e. source code implementing mathematical formulae. In this work, the focus is on source code which computes a numerical value via arithmetic operations as well as sum- and product formulae. We introduce and discuss breakpoints placed on an automatically inferred mathematical representation, i.e. in a common mathematical notation or by a mixed form of source code artifacts and maths symbols. Furthermore, we present visual debugging features aiming to facilitate the dynamic inspection of the formula code leveraging the mathematical representation. We briefly present a first prototype implementation of our formula debugging approach and indicate future directions of our work.**

***Index Terms*—debugging, breakpoint, formula code, maths, visualization, program comprehension**

I. Introduction

A software developer's daily work comprises not only the creation of new source code. There are other source code related tasks such as testing, refactoring, improving and debugging of the already existing source code. Specifically the task of debugging can become very time-consuming and difficult. In case a software failure occurs, a developer is confronted with three main tasks: localizing, understanding and correcting the fault(s) [1]. While the localization of faults can be supported by (semi-)automated debugging techniques ([2], [3]) the task of understanding the fault(s) remains challenging and is inevitable for their correction. Due to potential poor explanatory capabilities of (semi-)automated debugging tools applied in code comprehension tasks, developers demand traditional debugging with breakpoints [1]. In this paper, we present a novel approach for breakpoint debugging on implementations of mathematical formulae, in particular sums and products. Subsequently, we refer to source code implementing a mathematical formula as *formula code*. For instance, Figure 1 depicts the Leibniz formula for π (Equation 1) with a corresponding simple Java implementation of an approximation. This code's nature allows a straightforward alternative visual representation, i.e. expressing formula code in a common mathematical notation [4] or by a mixed form of source code artifacts and maths symbols. Our vision is that expressing the formula code in a mathematical representation

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... = \frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2*k+1} \tag{1}$$

```java
double quPi = 0;
for (int k = 0; k < n; k++) {
    quPi += Math.pow(-1, k) / (2 * k + 1); }
```

Fig. 1. The Leibniz formula for π (Equation 1) with a corresponding simple Java implementation.

$$\bullet\ quPi = quPi_0 + \overset{\bullet}{\underset{\bullet}{\sum}}{}_{k=0}^{n-1} \bullet\ \frac{(-1)^k}{2*k+1}\ \bullet$$

Fig. 2. Formula breakpoints (red dots) added to a mathematical representation of the formula code in Figure 1. The term $quPi$ stands for "quarter of pi" and $quPi_0$ refers to the value of variable $quPi$ prior to the `for`-loop.

results in a reduction of time a developer needs to comprehend formula code, which we further discuss in Section II. With the available mathematical representation of the formula code, novel opportunities to set breakpoints on this representation emerge. An example for an inferred mathematical representation of formula code with added formula breakpoints, depicted by the red dots, is presented in Figure 2. Note that the upper bound of the sum is replaced by a fixed number of iterations to execute $(n - 1)$.

In Section III, we present and discuss each breakpoint, state how we map them to available line breakpoint approaches in Java and outline resulting limitations and challenges. Furthermore, we introduce visual debugging features, which aim to assist the code comprehension task while inspecting dynamic behavior of formula code, e.g. on debugging or manually testing the formula code. Altogether, our contributions are:

- A basic approach on expressing source code in a common mathematical notation to support code comprehension tasks. In this work, we focus on source code which computes a numerical value via arithmetic operations as well as sum- and product formulae;

- A novel application of interactive breakpoint debugging where breakpoints can be placed on the mathematical representation of the particular source code; Visual debugging features to assist dynamic inspections of the formula code;
- A first prototype of our debugging approach implementing a subset of the breakpoints and features outlined in this work.

II. Mathematical Representation of Formula Code

The general discipline of software engineering is studded with problem solving and therefore with mathematical reasoning [5]. Thus maths is omnipresent in the work of a software developer and represented in the software systems' code base by the utilization of one or more programming languages.

Presenting formula code in an alternative mathematical representation promises a reduction of time to comprehend the code. According to the cognitive load theory [6]–[8], a reduction of the extraneous cognitive load, i.e. the working memory load affected by the manner in which information is presented, can facilitate a learning or problem solving task. From our perspective, the mathematical representation of formula code achieves this in multiple ways: The overall number of symbols is reduced by omitting potential visual disturbances, e.g. type information, variable initializations and method calls. A method call to `Math.sqrt(x)`, for instance, is replaced by its mathematical counterpart symbol $\sqrt{x}$. In contrast to the one dimensional writing space of code editors, a maths notation avails itself of a two dimensional space. This can lower the distance between cohesive symbols, e.g. $x = \frac{a}{b}$ in contrast to `int x = a / b;`, resulting in a more compact depiction. Moreover, the understanding of arithmetic expressions can be facilitated since a two dimensional writing space allows a more comfortable visual grouping of symbols according to valid operator precedence, for instance `(a + b) / (9 * x)` versus $\frac{a+b}{9x}$. The semantics of for-loops are similar to the semantics of the maths symbols Σ and Π. This is due to the definition of the index variable with respective lower-bound, a step size of one (or other step sizes as well) and the definition of an upper-bound. In total, it makes the connection of formula code utilizing a `for`-loop to implement a sum or product formula to a mathematical representation straightforward. Thus, the usage of Σ or Π reduces the number of symbols used to describe the semantics of formula code for sum and product formulae and compacts its depiction into one annotated symbol. Future directions also include to embed the mathematical representation into the source code editor to first, add valuable context (the surrounding source code) to the visualization and second, avoid extraneous cognitive load caused by a potential split attention effect [9]–[11].

III. Formula Breakpoint Debugging Approach

The concept of breakpoints in the context of interactive debugging dates back at least to the sixties [12], [13]. Common interactive breakpoint debuggers allow programmers to define breakpoints at particular lines of code which makes it possible to suspend a program's execution, inspect and modify its state and then resume or stepwise continue the program. Modern IDEs, e.g. Jetbrains IntelliJ IDEA, provide traditional line breakpoints as well as exception- and method-breakpoints or field-watchpoints. Conditions can be attached to every type of breakpoint, e.g. pass counts, class filters or even complex expressions, on which a breakpoint is supposed to trigger. This enables a programmer to filter out interesting executions during the dynamic inspection of a program.

A. Breakpoints on Sum and Product Formulae

Subsequently, we consider the pseudocode for a simple `for`-loop (Figure 3, right) and its respective mathematical representation (Figure 3, left). Note that $expr_0$ and $expr_i$ are placeholders for arbitrary expressions and $expr_i$ might depend on the indexing variable i's value. Since breakpoints for the

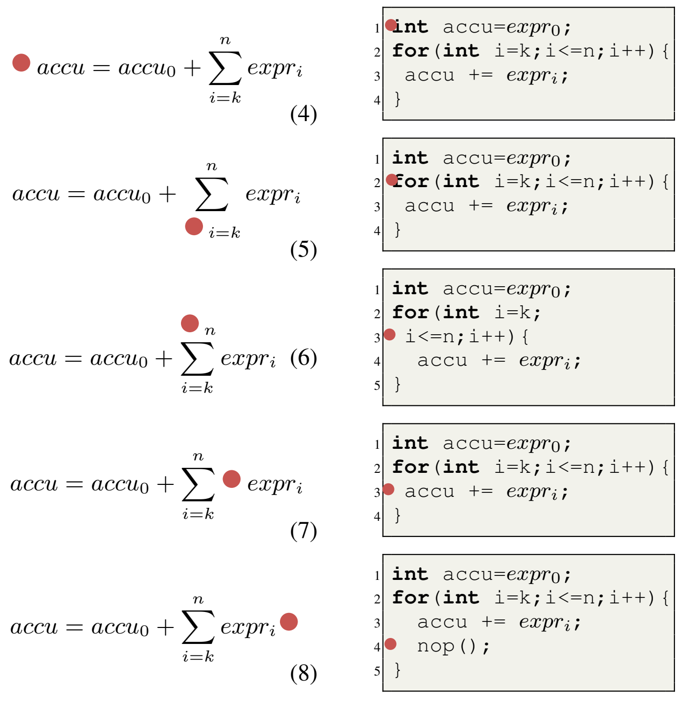
$$accu_0 = expr_0 \tag{2}$$

$$accu = accu_0 + \sum_{i=k}^{n} expr_i \tag{3}$$

```
int accu=expr₀;
for(int i=k;i<=n;i++){
  accu += exprᵢ;
}
```

Fig. 3. A simple for loop in pseudocode (left) and its respective mathematical representation (right).

formula presented in Equation 2 are trivial, we focus on the summation formula in Equation 3. Our intention is to define breakpoints which are most intuitive and self explanatory. In the following, we refer to equations listed in Figure 4 and respective lines of source code presented in the corresponding code listings next to the equations.



$$\bullet\ accu = accu_0 + \sum_{i=k}^{n} expr_i \tag{4}$$

```
●int accu=expr₀;
 for(int i=k;i<=n;i++){
  accu += exprᵢ;
 }
```

$$accu = accu_0 + \sum_{\bullet\, i=k}^{n} expr_i \tag{5}$$

```
 int accu=expr₀;
●for(int i=k;i<=n;i++){
  accu += exprᵢ;
 }
```

$$accu = accu_0 + \sum_{i=k}^{\bullet\, n} expr_i \tag{6}$$

```
 int accu=expr₀;
 for(int i=k;
●  i<=n;i++){
    accu += exprᵢ;
 }
```

$$accu = accu_0 + \sum_{i=k}^{n} \bullet\ expr_i \tag{7}$$

```
 int accu=expr₀;
 for(int i=k;i<=n;i++){
● accu += exprᵢ;
 }
```

$$accu = accu_0 + \sum_{i=k}^{n} expr_i\ \bullet \tag{8}$$

```
 int accu=expr₀;
 for(int i=k;i<=n;i++){
    accu += exprᵢ;
●   nop();
 }
```

Fig. 4. List of available formula breakpoints.

The first formula breakpoint presented in Equation 4 is in front of the whole formula. We map this breakpoint to a line breakpoint in line 1. When the execution suspends at this point, the value of $accu_0$ is not yet present. Assuming $expr_0$ does not take changes on k and n, the values for these variables are defined. Thus, we are able to update the mathematical representation of the formula code by replacing the occurrences of k and n in the formula with their actual values. By this, we add dynamic information to the mathematical representation. This leads to the term of a *dynamic formula* which, each time a variable is updated, represents a different state of the statically inferred formula. The breakpoint right before the summation-index definition (Equation 5) is understandable as a halt right before the definition of i. This maps to a line breakpoint in line 2. Then, the value of $accu_0$ is available and again, we can replace the variable $accu_0$ with its actual value in the mathematical representation.

The breakpoint before the upper-bound definition (Equation 6) refers to a halt of the program after the summation-index definition, but before the definition of the upper-bound. To map this, we alter the code such that the summation-index definition (`int i=k`) and the upper-bound definition (`i<=n`) are placed in subsequent lines of code by inserting a newline character before `i<=n`. That way, we can place a line breakpoint in the new introduced line, which than results in a halt right before the definition of the upper-bound value. This makes the actual value for i available which can be replaced in the mathematical representation. Note, we do not intend to alter the source code itself which would be confusing to a user. There are other ways to implement those alterations, e.g. by editing classes during the Java VM's class loading phase or by application of bytecode instrumentation. The breakpoint placed before $expr_i$ (Equation 7) is interpretable as a halt on the first line of the underlying loop's body (line 3) under the condition $i == k$. The condition ensures that the breakpoint only triggers in the first iteration of the loop. When this condition is omitted, the breakpoint triggers in every iteration of the loop. This also represents a valid interpretation of the breakpoint. The interpretations differ in the breakpoint's referring scope, either the whole summation or each summand. We find both meaningful. To visually distinguish between these two alternatives, we add two breakpoint markers in different sizes. On hovering with the mouse over a marker, the referring breakpoint's scope is indicated by a frame (Figure 5). On the first hit of this breakpoint, the value of the summation's upper-bound becomes available and its corresponding variable in the mathematical representation can also be replaced by its actual value. The interpretation of the breakpoint placed after $expr_i$ (Equation 8) is also ambiguous. It could either refer to a halt after each evaluation of $expr_i$, i.e. loop iteration, or at the end of the whole summation. To map this, we insert a NOP operation, e.g. a call to a method with an empty body, in the succeeding line of $expr_i$ and add a line breakpoint on this line (4). If only a halt on the end of the whole summation is intended, we simply add the condition $i == n$ to the breakpoint to suspend the execution on the last iteration.To visually distinguish between these two breakpoint scopes, we follow the same approach as before (Figure 5).

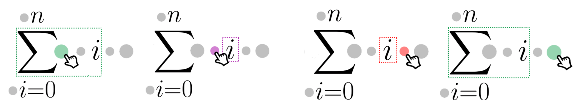


Fig. 5. Different size breakpoint markers. The referring breakpoint's scope is indicated by the frames.

$$accu = accu_0 + \bullet \sum_{i=k}^{n} expr_i \tag{9}$$

$$accu = accu_0 + \sum_{i=k \bullet}^{n} expr_i \tag{10}$$

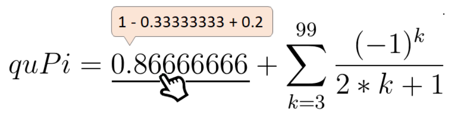
$$accu = accu_0 + \sum_{i=k}^{n \bullet} expr_i \tag{11}$$

Fig. 6. Discarded formula breakpoints.

In Figure 6, we list further possible formula breakpoints. We discarded them because their semantics are already implemented through the other formula breakpoints. The breakpoint right in front of the Σ symbol (Equation 9) maps to a line breakpoint in line 2. This is equivalent to the breakpoint in Equation 5. Setting a breakpoint after the summation-index and lower-bound definition (Equation 10) is similar with setting a breakpoint right before the upper-bound definition realized by breakpoint in Equation 6. The breakpoint after the upper-bound definition (Equation 11) is in line with the whole summation scope interpretation of the breakpoint in Equation 7, i.e. prior to the first iteration of the underlying `for`-loop.

B. Dynamic Formula with Summation History

Whenever a formula breakpoint triggers or a debug step is executed, the available values of the variables in the formula code's mathematical representation are replaced or updated, respectively. This feature enables a dynamic view and illustration of the formula's current progress. Furthermore, the value for the accumulation variable will update in every iteration of the underlying `for`-loop. We can store the current accumulation variable's value in the mathematical representation by utilizing the accumulation variable's initialization placeholder in the mathematical representation, e.g. $accu_0$ in Equation 3 or $quPi_0$ in Figure 2. We demonstrate this in Figure 7. We refer to the mathematical formula code representation presented in Figure 2, assume an upper-bound value of $n = 100$ and that the program suspended after the third iteration. Thus the current accumulation variable's value is $quPi_3 = 1 - \frac{1}{3} + \frac{1}{5} = 0.86666666$, k has a value of 3 (started at 0). On hovering with the mouse over the accumulation value, we show its summation history in a small popup above the value. In order to remain visually scalable, we only list the last three summands. If there are more than three summands, we indicate this by a leading "..." in the summation history. We suppose, that this will foster the summation's traceability and thus, supports program state inspections while a developer tests, debugs or refactors the source code.

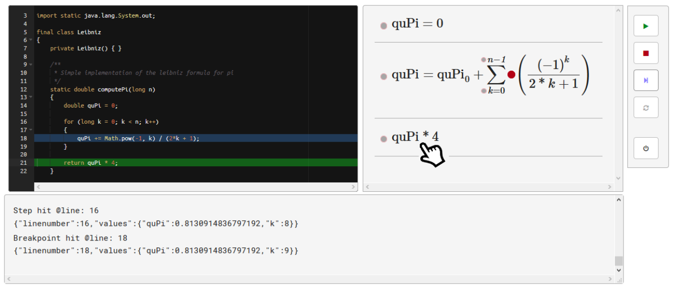


$$quPi = \underline{0.86666666} + \sum_{k=3}^{99} \frac{(-1)^k}{2*k+1}$$

Fig. 7. Illustration of the formula's progress and summation history of the accumulation variable.

We are convinced, that these features can support debugging, testing and refactoring tasks on formula code. In particular, for formula code in the shape of sum and product formulae such as the Leibniz formula for pi (Figure 1). With the use of the summation history, the correct computation of each summand can be ensured. With a clever usage of the formula breakpoints, variables can be inspected at ease in different stages of the formula computation. Furthermore, the correctness of the variable values, in particular the start, end and step values of `for`-loops, can be investigated in a lightweight visual manner.

IV. Prototype Implementation



```
 3  import static java.lang.System.out;
 4
 5  final class Leibniz
 6  {
 7      private Leibniz() { }
 8
 9      /**
10       * Simple implementation of the leibniz formula for pi
11       */
12      static double computePi(long n)
13      {
14          double quPi = 0;
15
16          for (long k = 0; k < n; k++)
17          {
18              quPi += Math.pow(-1, k) / (2*k + 1);
19          }
20
21          return quPi * 4;
22      }
```

● $\text{quPi} = 0$

● $\text{quPi} = \text{quPi}_0 + \sum_{k=0}^{n-1} ● \left(\frac{(-1)^k}{2*k+1}\right)$

● $\text{quPi} * 4$

```
Step hit @line: 16
{"linenumber":16,"values":{"quPi":0.8130914836797192,"k":8}}
Breakpoint hit @line: 18
{"linenumber":18,"values":{"quPi":0.8130914836797192,"k":9}}
```

Fig. 8. Screenshot of the formula debugger prototype.

We implemented a first prototype of our formula debugging approach. To this end, we developed a web application presenting the source code in an Ace code editor [14] and a list of the automatically inferred mathematical representations of formula code on the right next to the code in a separate view (Figure 8). We utilized the Java Debug Interface [15] to control and query the debuggee Java VM, i.e set breakpoints and retrieve variable values. To infer a mathematical representation from the source code, we utilize the Java parser *javalang* [16] written in Python and added routines to nodes of the abstract syntax tree to generate MathML (a markup language for mathematical notation). Note, this approach is not able to infer every possible formula code but is sufficient to deliver first working examples. When hovering over an inferred mathematical representation, the corresponding lines of source code in the editor are highlighted in green. Furthermore, gray dots are used as markers for the formula breakpoints, which are integrated into the inferred mathematical representations. On hovering over these markers, they turn red. With a click on those markers, the respective formula breakpoint will be activated or deactivated, respectively. We added controls to remotely run/resume, stop, rerun or step through the debuggee's execution as well as to quit the debugger. On a breakpoint hit, the corresponding line of source code on which the execution halts is highlighted in blue. The respective available variable values are displayed in a separate output view underneath the other two views. The selection between global summation and local summand scope of some formula breakpoints, the dynamic formula illustration and the summation history are not yet fully implemented but are planned to be completed in the near future.

V. Related Work

Work on automatic layout of formulae based on textual specifications as well as graphical formula editors [17]–[19] dates at least back to the seventies. More recent work includes the integration of a formula editor in a common IDE as a domain specific language extension using Jetbrains MPS [20]. The only work on reconstructing mathematical formulae from source code has been published by Moser et al. [21], [22]. Their RgB tool extracts formulae from annotated source code to produce a documentation of the source code. The tool was developed for Fortran and C++, requires manual annotations, uses static program analysis and covers only a small part of possible formulae. To some extent, implementations of mathematical formulae make use of numerical values such as integer or floating point numbers. Recently, studies on characteristics such as categories, symptoms, frequencies and possible fixes of numerical bugs have been conducted by Di Franco et al. [23]. While parts of numerical computations can be described by formulae, they are often more algorithmic in nature. A variety of visual debugging approaches have been developed. In [24], the dynamic program execution state is linked to an enhanced UML object diagram. A sequence diagram generated through traced executions paths is utilized to reveal fault inducing source code fragments in [25]. The tool *JIVE* [26] makes use of both, an object and sequence diagram to dynamically visualize the state and the history of a program's execution, respectively. The system *Lens* [27] integrates algorithm animation-style capabilities into a source-level debugger allowing a rapid creation of visualizations on defined breakpoints. The tool *VIDA* [28] recommends breakpoint candidates to a programmer based on the analysis of execution information and visualizes static dependency relations.

VI. Conclusion and Future Work

We presented a novel visual breakpoint debugging approach for formula code. To this end, we automatically infer a mathematical representation of the formula code which is meant to facilitate comprehension of the formula code. We leverage the mathematical representation to place formula breakpoints which we map to common line breakpoints on the underlying formula code. Furthermore, we developed visual debugging features in line with the mathematical representation aiming to facilitate the dynamic inspection of the formula code. In particular, this comprises the selection of breakpoints indicating their referring scope, either the whole summation or a single summand, the illustration of the dynamic formula's state including the summation history. Finally, we presented a first prototype implementation of our visual debugging approach.

Future work includes the investigation of other formula code in terms of syntactical structures used, e.g. foreach, while and nested loops, and mathematical types, e.g. vectors and matrices, as well as the qualitative and quantitative search for occurrences of formula code in software archives. Furthermore, by investigating bug databases to explore software defects related to formula code more requirements for our debugging approach could be identified. More important, user studies for both, the effect of the mathematical representation of formula code in program comprehension tasks and the usability and effectiveness of the visual breakpoint debugging approach itself, have to be conducted. A first qualitative user study where the participants need to explain the semantics of formula code, one group with and one group without an inferred mathematical representation at hand, would deliver first insights. A qualitative study where the participants in two groups, one with and the other without the help of our debugging tool, are confronted with fixing real-world formula code bugs appears promising. We would conduct such a study in an observational setting with encouraged thinking aloud and a subsequent participant interview.

Acknowledgment

We are grateful to Sebastian Baltes for his comments on an earlier version of the manuscript.

References

[1] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. ACM, 2011, pp. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[2] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. ACM, 2003, pp. 97–105. [Online]. Available: https://doi.org/10.1145/640128.604140

[3] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*, 2000. [Online]. Available: http://arxiv.org/abs/cs.SE/0012009

[4] F. Cajori, *A history of mathematical notations*. Chicago, IL: The Open Court Publishing Co., 1929.

[5] P. B. Henderson, "Mathematical reasoning in software engineering education," *Commun. ACM*, vol. 46, no. 9, pp. 45–50, 2003. [Online]. Available: https://doi.org/10.1145/903893.903919

[6] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988. [Online]. Available: https://doi.org/10.1016/0364-0213(88)90023-7

[7] J. Sweller, J. J. G. van Merrienboer, and F. G. W. C. Paas, "Cognitive architecture and instructional design," *Educational Psychology Review*, vol. 10, no. 3, pp. 251–296, Sep 1998. [Online]. Available: https://doi.org/10.1023/A:1022193728205

[8] R. E. Mayer and R. Moreno, "Nine ways to reduce cognitive load in multimedia learning," *Educational Psychologist*, vol. 38, no. 1, pp. 43–52, 2003. [Online]. Available: https://doi.org/10.1207/S15326985EP3801_6

[9] P. Ginns, "Integrating information: A meta-analysis of the spatial contiguity and temporal contiguity effects," *Learning and Instruction*, vol. 16, no. 6, pp. 511 – 525, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0959475206000806

[10] P. Ayres and J. Sweller, *"The Split-Attention Principle in Multimedia Learning," in The Cambridge handbook of multimedia learning*. Cambridge university press, 2005, pp. 135–146, r. E. Mayer.

[11] J. Sweller, *"Implications of cognitive load theory for multimedia learning," in The Cambridge handbook of multimedia learning*. Cambridge university press, 2005, pp. 19–30, r. E. Mayer.

[12] T. G. Evans and D. L. Darley, "On-line debugging techniques: a survey," in *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, ser. AFIPS Conference Proceedings, vol. 29. AFIPS / ACM / Spartan Books, Washington D.C., 1966, pp. 37–50. [Online]. Available: https://doi.org/10.1145/1464291.1464295

[13] ——, "DEBUG - an extension to current online debugging techniques," *Commun. ACM*, vol. 8, no. 5, pp. 321–326, 1965. [Online]. Available: https://doi.org/10.1145/364914.364952

[14] Ace, "Ace - The High Performance Code Editor for the Web," https://ace.c9.io/, October 14, 2019, October 2019.

[15] Oracle, "Overview (Java Debug Interface)," https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/, Accessed 10/14/2019, October 2019.

[16] GitHub, "GitHub - c2nes/javalang: Pure Python Java parser and tools," https://github.com/c2nes/javalang, Accessed 10/14/2019, October 2019.

[17] B. W. Kernighan and L. L. Cherry, "A system for typesetting mathematics," *Commun. ACM*, vol. 18, no. 3, pp. 151–157, Mar. 1975. [Online]. Available: http://doi.acm.org/10.1145/360680.360684

[18] D. E. Knuth, "Mathematical typography," *Bull. Amer. Math. Soc.*, vol. 1, no. 2, pp. 337–372, 03 1979. [Online]. Available: http://projecteuclid.org/euclid.bams/1183544082

[19] M. Levison, "Editing mathematical formulae," *Softw., Pract. Exper.*, vol. 13, no. 2, pp. 189–195, 1983. [Online]. Available: https://doi.org/10.1002/spe.4380130208

[20] Jetbrains, "Mps: Domain-specific language creator by jetbrains," https://www.jetbrains.com/mps/, Accessed 10/13/2019, October 2019.

[21] M. Moser and J. Pichler, "Documentation generation from annotated source code of scientific software: position paper," in *Proceedings of the International Workshop on Software Engineering for Science, SE4Science@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 12–15. [Online]. Available: https://doi.org/10.1145/2897676.2897679

[22] M. Moser, J. Pichler, G. Fleck, and M. Witlatschil, "Rbg: A documentation generator for scientific and engineering software," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 464–468. [Online]. Available: https://doi.org/10.1109/SANER.2015.7081857

[23] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 509–519. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115662

[24] T. Jacobs and B. Musial, "Interactive visual debugging with uml," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis '03. New York, NY, USA: ACM, 2003, pp. 115–122. [Online]. Available: http://doi.acm.org/10.1145/774833.774850

[25] O. Pilskalns, S. Wallace, and F. Ilas, "Runtime debugging using reverse-engineered uml," in *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 605–619. [Online]. Available: http://dl.acm.org/citation.cfm?id=2394101.2394156

[26] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 31–35. [Online]. Available: http://doi.acm.org/10.1145/1328279.1328286

[27] S. Mukherjea and J. T. Stasko, "Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, pp. 215–244, Sep. 1994. [Online]. Available: http://doi.acm.org/10.1145/196699.196702

[28] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "Vida: Visual interactive debugging," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 583–586.