Optimization Techniques for Algorithmic Debugging

David Insa and Josep Silva

Universidad Politécnica de Valencia,
Camino de vera s/n, 46022 Valencia, Spain
{dinsa,jsilva}@dsic.upv.es

Abstract

Nowadays, undetected programming bugs produce a waste of billions of dollars per year to private and public companies and institutions. In spite of this, no significant advances in the debugging area that help developers along the software development process have been achieved yet. In fact, the same debugging techniques that were used 20 years ago are still being used now. Although some alternatives have appeared, they are still a long way until they become useful enough to be part of the software development process. One of such alternatives is Algorithmic Debugging, which abstracts the information the user has to investigate to debug the program, allowing them to focus on what, rather than how, is happening. This abstraction comes at a price: the granularity level of the bugs that can be detected allows for isolating wrongly implemented functions, but which part of them contains the bug cannot be found out yet. This work is a short introduction of some published papers that focus on improving Algorithmic Debugging in many aspects. Concretely, the main aims of these papers are to reduce the time the user needs to detect a programming bug as well as to provide the user with more detailed information about where the bug is located.

Algorithmic Debugging [10] allows the user to debug their code without being aware of how the code works, but only what the code is supposed to do. However, despite being powerful, the technique also has some drawbacks. In this paper, we introduce some works that focus on mitigating or suppressing these drawbacks. Concretely, these works include some techniques to start the debugging sessions as soon as possible [3], to reduce the number of questions the user is going to be asked [4, 5, 8], and to augment the granularity level [6, 9] of those bugs that can be detected, allowing the debugger to keep looking for bugs even inside functions. In addition, due to the obsolescence of the original formulation [10], a new reformulation of the technique [7] has been defined. Besides these theoretical results, a

fully functional algorithmic debugger (DDJ) has been implemented [2] that contains and supports all these techniques and strategies. This debugger is written in Java, and it debugs Java code. To further increase its usability, the debugger has been later adapted [1] as an Eclipse plugin (HDJ), so that it could be used by a bigger number of users. These two debuggers are publicly available, so any interested person can access them and continue with the research if they wish so. The main contributions of these works are summarized below:

1. **Scalability:** Since Algorithmic Debugging was proposed, a debugging session has been traditionally divided into two independent and sequential phases: obtaining the execution tree, and debugging the program. During the development of the DDJ debugger, we developed the *Virtual Execution Trees* technique [3], which allows the debugger to overlap these two phases, permitting a reduction of the time that a user needs to start the debugging session from minutes to milliseconds. Moreover, the scalability problem has been encompassed incorporating a cache-based architecture in the debugger. This new architecture allows the debugger to be scalable in time and in memory. Moreover, in the HDJ debugger, three different debuggers have been combined to increase the scalability, the first of which is the own *Trace Debugger* of Eclipse. Its combination with DDJ improves the debugging sessions. By using the *Trace Debugger*, the user can place breakpoints into the source code to direct the search until a piece of code that contains the bug is found. Later, Algorithmic Debugging can be used to debug this piece of code in an abstract way, which helps to debug complex algorithms because it is only necessary to understand what they are supposed to do and not how they are implemented. This combination reduces the amount of information that Algorithmic Debugging has to store, because now, the debugger only stores the part of the code in which the user thinks the error is. Thus, the debugger only needs to store a suspicious subcomputation instead of the whole computation.

2. **Usability:** The data structure used to store the computation of the program to debug (the execution tree) is highly related with the usability of the technique. The more unbalanced it is, the more time is needed to explore it and thus to find the bug. When it is completely unbalanced, the user may be asked a linear number of questions with respect to the number of nodes of the tree, whereas when the tree is completely balanced, the debugger may only ask a logarithmic number of questions. Clearly, the structure of the tree is crucial for the technique. To improve the structure, we have developed techniques that are now integrated into DDJ and we have also developed some strategies that reduce the number of questions. In particular, DDJ

implements the *Tree Balancing* technique [8] that balances the tree by combining some nodes, and the *Tree Compression* technique [9] that removes some nodes when this change balances the tree. Moreover, *Optimal Divide & Query* [5] improves the best known search strategy (D&Q), and therefore it reduces the number of questions that the user is asked. Finally, the HDJ plugin for Eclipse has been developed allowing the user to combine DDJ with its *Trace Debugger*, and to use DDJ in a more familiar context.

3. **Granularity.** Algorithmic Debugger has the drawback of being only able to find the method that contains the bug, instead of the expression. DDJ now includes a novel technique called *Loop Expansion* [6, 9] that increases the granularity by converting the loops presented in the code into recursive calls. This transformation produces a completely different data structure in which the loops and their iterations are represented, helping the user to also discard that the loops have caused the bug. In addition, when DDJ has found the portion of code that contains the bug, the *Omniscient Debugger* included in HDJ can be used to further explore it. This debugger complements DDJ by permitting the user to find the expression that contains the bug.

4. **Uniformity:** The original formulation of Algorithmic Debugging was oriented to the logic paradigm, but it was quickly adapted to the functional and imperative paradigm respecting the basis of the formulation. However, this formulation is obsolete nowadays, and researchers frequently need to redefine specific parts of the formulation. In these works we have included a reformulation of the technique [7] that is paradigm-independent, and that can be used as a formal representation of all current existing techniques at the time of writing these lines. Moreover, it provides a classification of transformation techniques that affect the data structure of AD, as well as the properties that these transformations may fulfil. Researchers can use this classification to easily classify their Algorithmic Debugging techniques. Classifying a technique is useful because it provides other researchers with information about the properties that holds for this technique. In addition, an Algorithmic Debugging scheme is presented to provide a general view of how all the components of Algorithmic Debugging interact, helping the interested reader to better understand the technique.

As a result of these works, HDJ is the first algorithmic debugger that incorporates almost all search strategies from different paradigms, overlaps both Algorithmic Debugging phases, and incorporates new Algorithmic Debugging techniques such as *Virtual Execution Tree*, *Tree Balancing*, *Loop Expansion*, and *Tree Compression*, as well as combines different debuggers that share their information and collaborate in a single debugging session. In summary, HDJ can be considered as

the most advanced algorithmic debugger that exists either in the logic, functional or imperative paradigm.

References

[1] J. González, D. Insa, and J. Silva. A new Hybrid Debugging architecture for Eclipse. In *Proceedings of the 23rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2013)*, volume 8901 of *Lecture Notes in Computer Science (LNCS)*, pages 183–201. Springer, 2014.

[2] D. Insa and J. Silva. An algorithmic debugger for Java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–6, 2010.

[3] D. Insa and J. Silva. Scaling up Algorithmic Debugging with Virtual Execution Trees. In *Proceedings of the 20th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2010)*, volume 6564 of *Lecture Notes in Computer Science (LNCS)*, pages 149–163. Springer, 2010.

[4] D. Insa and J. Silva. Hacia una estrategia óptima para la Depuración Algorítmica. In *Proceedings of the 11th Spanish Workshop on Programming Languages (PROLE 11)*, sep 2011.

[5] D. Insa and J. Silva. An optimal strategy for Algorithmic Debugging. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *Proceedings of the 26th International Conference on Automated Software Engineering (ASE 2011)*, pages 203–212. IEEE Computer Society, November 2011.

[6] D. Insa and J. Silva. Automatic transformation of iterative loops into recursive methods. *Information and Software Technology*, 58:95–109, 2015.

[7] D. Insa and J. Silva. A generalized model for Algorithmic Debugging. In M. Falaschi, editor, *Proceedings of the 25th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2015)*, volume 9527 of *Lecture Notes in Computer Science (LNCS)*, pages 261–276. Springer, July 2015.

[8] D. Insa, J. Silva, and A. Riesco. Speeding up Algorithmic Debugging using Balanced Execution Trees. In M. Veanes and L. Viganò, editors, *Proceedings of the 7th International Conference Tests and Proofs (TAP 2013)*, volume 7942 of *Lecture Notes in Computer Science (LNCS)*, pages 133–151. Springer, jun 2013.

[9] D. Insa, J. Silva, and C. Tomás. Enhancing Declarative Debugging with Loop Expansion and Tree Compression. In E. Albert, editor, *Proceedings of the 22nd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2012)*, volume 7844 of *Lecture Notes in Computer Science (LNCS)*, pages 71–88. Springer, sep 2012.

[10] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.