An Algorithmic Debugger for Java

David Insa and Josep Silva
Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Valencia, Spain
{dinsa,jsilva}@dsic.upv.es

***Abstract*—This work presents DDJ, an algorithmic debugger for Java. The main advantage of DDJ with respect to previous algorithmic debuggers is its scalability. DDJ has a new architecture based on the use of cache memories that allows it to scale both in time and memory. In addition, it includes new techniques that allow the debugger to start the debugging session even before the execution tree has been produced. We present the new architecture, and describe the main features of this debugger together with a usage scenario.**

I. Introduction

Debugging is one of the most important tasks in the software development process. It is necessary in all paradigms and programming languages both during the development and during the maintenance of software systems. In Shapiro's words [16]:

> "*It is evident that a computer can neither construct nor debug a program without being told (...) what problem the program is supposed to solve (...) No matter what language we use to convey this information, we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: we cannot know, at any finite point in time, all the consequences of our current assumptions.*"

Unfortunately, the efforts of the scientific community in producing usable and scalable debuggers has been historically low. One important example is the lack of an algorithmic debugger in Java. The debugging of Java programs has traditionally been done with the use of breakpoints that allow us to execute the program step by step, and inspect computations (manually) at a given point. However, the adaptation of semi-automatic debugging techniques such as algorithmic debugging has not been successfully done, and neither Sun Java Studio Creator, Borland JBuilder, NetBeans, JCreator, nor Eclipse implements an algorithmic debugger.

To the best of our knowledge there has been only one attempt (the algorithmic debugger JDD [10]) of implementing an algorithmic debugger for Java. Other debuggers exist that incorporate declarative aspects such as the Eclipse plugin JavaDD [8] or the Oracle JDeveloper's declarative debugger [7] however, they are not able to automatically produce questions and to control a search to automatically find the bug. This means that they lack of current strategies for algorithmic debugging implemented in standard algorithmic debuggers of declarative languages such as Haskell (Hat-Delta [6]) or Toy (DDT [4]).

The main drawback of JDD is that it suffers from important scalability problems: the internal data structure needed for algorithmic debugging—the so called *Execution Tree* (ET)—is huge (indeed gigabytes) and it does not usually fit in main memory. Even with the use of a database the construction of the ET is costly (e.g., minutes). This is the main cause of the lack of algorithmic debuggers for imperative and object-oriented languages such as Java.

For instance, we conducted some experiments to measure the time needed by JDD to start a debugging session[1] with a collection of medium/large benchmarks. Results are shown in column `JDD` of Table I. Note that, in order to generate the ET, JDD needs some minutes, thus the debugging session cannot start before this time.

Benchmark	JDD	DDJ
argparser	22 s.	407 ms.
cglib	230 s.	719 ms.
kxml2	1318 s.	1844 ms.
javassist	556 s.	844 ms.
jtstcase	1913 s.	1531 ms.
HTMLcleaner	4828 s.	609 ms.

TABLE I
Benchmark results

In this work we present the *Declarative Debugger for Java* (DDJ), a new implementation based on the old JDD but with a completely new architecture that incorporates many new features that make the debugging process scalable. In particular, column `DDJ` of Table I shows the time needed to start a debugging session with DDJ[2].

II. Tool Description

This section describes the main features of DDJ, and it explains why its new architecture allows the debugger to scale up in time and memory.

[1]These times correspond to the execution of the program, the production of the ET and its storage in a database.

[2]These times correspond to the execution of the program up to the generation of the first ET node and its storage in a database.

A. Architecture

The architecture of DDJ is new not only for Java, but for all paradigms. DDJ is the first algorithmic debugger that allows us to debug a program at the same time that the ET is being generated, and it is the first implementation that automatically balances the ETs to reduce the number of questions asked.

The architecture of DDJ is summarized in Figure 1. It uses the *Java Platform Debugger Architecture* (JPDA) [19] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface which helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Therefore, with JPDA the debugger can execute the source code and inspect the state of the memory at any time. In order to solve the memory scalability problem, DDJ uses a database to store the ET. Therefore, it can work with ETs of any size. Moreover, to avoid that the graphical memory is exhausted, only a part of the ET is shown in the GUI (e.g., the part associated to the current question); the user can configure the amount of ET nodes shown in the GUI. In order to solve the time scalability problem, DDJ implements an algorithm that allows strategies to work with uncompleted ET nodes. This means that the debugger can start the debugging session when a subcomputation has been already executed (i.e., when the ET has at least one node completed), but the program is still running, and thus, the ET is incomplete.

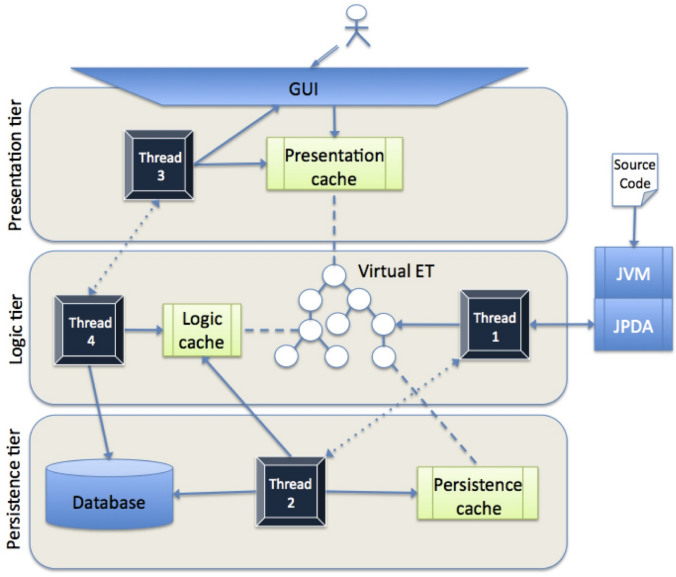

Fig. 1. Architecture of DDJ

The architecture is composed of three tiers that modularize the processes and limit the memory used in the graphical memory (presentation tier), in main memory (logic tier) and in secondary memory (persistence tier).

All the components of the architecture are described bellow:

GUI: The GUI is independent of the other components. It can be adapted to interact with other oracles apart from the user.

Database: The database stores the complete ET and it is independent of the architecture. The connection is done via JDBC, thus any database could be used. The current distribution uses MySQL by default.

JPDA / JVM: JPDA is used to control the execution of the source code in the JVM in order to produce the ET. The architecture only communicates with this component via thread 1; therefore, the debugger could be easily adapted for another language (e.g., C++) by only changing thread 1.

Virtual ET: A copy of the ET in the database is always in main memory. However, this copy called "virtual ET" only contains the structure of the ET, being the nodes empty. When it is required, the information of a cluster of nodes is loaded from the ET to the virtual ET. This allows us to control how much memory is used by the virtual ET.

Presentation Cache: It contains the subset of the ET shown in the GUI.

Logic Cache: It contains the subset of the ET stored in the virtual ET.

Persistence Cache: It contains a subset of nodes computed by JPDA that will be stored in the database with a single transaction. This clustering mechanism allows us to reduce the number of accesses to the database.

Thread 1: It constructs the virtual ET.

Thread 2: It stores the ET in the database.

Thread 3: It interacts with the user.

Thread 4: It controls what nodes of the ET are stored (also) in the virtual ET. It implements all the algorithmic debugging strategies.

B. Functionality

An enumeration of the main features and functionalities of DDJ follows:

1) *Algorithmic debugging strategies*. The strategy used strongly influences the performance of algorithmic debugging [18]. DDJ is the algorithmic debugger that currently implements more strategies, and moreover, it allows the user to change the strategy during a debugging session. Currently, 10 strategies are implemented: Top down [1], Single stepping [16], Heaviest first [2], More rules first [17], both Shapiro [16] and Hirunkitti's [9] Divide & query, Divide by rules & query [17] and the three Hat Delta heuristics [6].
2) *GUI*. Many algorithmic debuggers (e.g., Hat or Buddha) lack of a GUI because they are based on a semi-automatic search. However, the user can provide useful information to the debugger if she is allowed to freely (manually) explore the ET. This allows her, for instance, to select a subtree of the ET (e.g., the suspicious one or the one associated to the last changes, etc.) instead of selecting always the whole ET. It also allows her to change the state of some nodes and thus avoid the exploration of correct parts of the ET.
   Clearly, having a graphical (and interactively explorable) ET can speed up the debugging session. However, this comes with a cost: the graphical components are heavy,

Steps	Output	Comments
1.- Load a program	The ET is shown in the GUI	Select the .class file and dependencies (if any), and provide the arguments (if any)
2.- Select a strategy	The GUI shows a question	It is also possible to freely explore the ET and debug the program manually
3.- Answer questions	The ET is pruned in the GUI and a new question is shown	Possible answers: correct, wrong, I don't know, trusted
(...)	(...)	Step 3 is repeated until the bug is found
4.- Locate the bug	Bug highlighted in the source code	

TABLE II
USAGE SCENARIO

and the whole ET does not usually fit in the graphical memory. To solve this situation, DDJ provides a clustering mechanism (based on the use of the presentation cache) that permits to load the part of the ET that (i) is required by the user, or (ii) is required by the strategy being used. This mechanism automatically controls that the current node is shown in the GUI with a number of directly related nodes (ancestors, successors, siblings, etc.); ensuring that the amount of graphical memory used is limited.

3) *Portability*. The architecture of DDJ is similar to the architecture of a compiler. There is a front-end that depends on the source language, and a back-end that is independent of the source language and only depends on the intermediate representation that in this case is the ET. This means that the GUI, the database, the strategies, and all the components of the back-end are mostly independent of the language that is being debugged. Therefore, if we change the front-end, e.g., to work with C++, the debugger could debug C++ programs by only changing the implementation of thread 1.
4) *Uncompleted ETs*. Another feature that makes DDJ different from the other debuggers is that DDJ is able to work with uncompleted ETs. This feature does not only speed up the debugging session but it also allows us to make algorithmic debugging scalable. Traditional algorithmic debugging is based on two sequential phases: producing an ET and exploring the ET. In DDJ both phases can be overlapped so that the user is able (if desired) to start the debugging session long before the ET is completed, saving the user a waste of time (see Table I), since the generation of the ET is the bottle neck in algorithmic debugging. This feature has required the reimplementation and adaptation of the debugging strategies to work with uncompleted ETs [12].
5) *Balancing ETs*. DDJ is the first implementation of a novel technique [11] that allows to reduce the number of questions asked to the user during the debugging session. This technique is able to automatically balance ETs by a transformation that collapses some nodes and introduces new nodes. The result is an ET that is often bigger than the original, but it is more efficient when it is explored by the strategies.

III. USAGE SCENARIO

This section describes a typical usage scenario with DDJ. We describe the scenario in which the user decides to use the guided search for the bug. In this case, the debugger uses a strategy to search for the error, and the debugging session is semi-automatic, because the user only has to answer questions.

A summary of the steps described in the usage scenario is shown in Table II.

Step 1: The first step in a debugging session is to load the buggy program. This can be done with the window shown in Figure 2 where we see that the file *vector.class* is selected. This file communicates with other files that implements sorting algorithms such as quicksort. In addition to the *.class*, it is possible to select the dependencies that the program will use, and the arguments used for the execution of the program. In this example the arguments are 10 and 20.

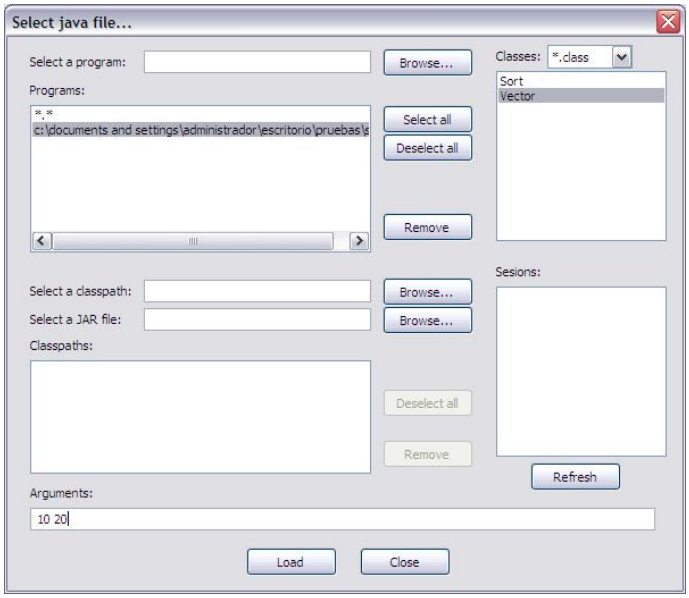

Fig. 2. Assistant to load a Java program

With this information, DDJ executes the program provided and gradually produces the associated ET. A portion of the ET associated to the program loaded is shown in the main pan of Figure 3. Even if the

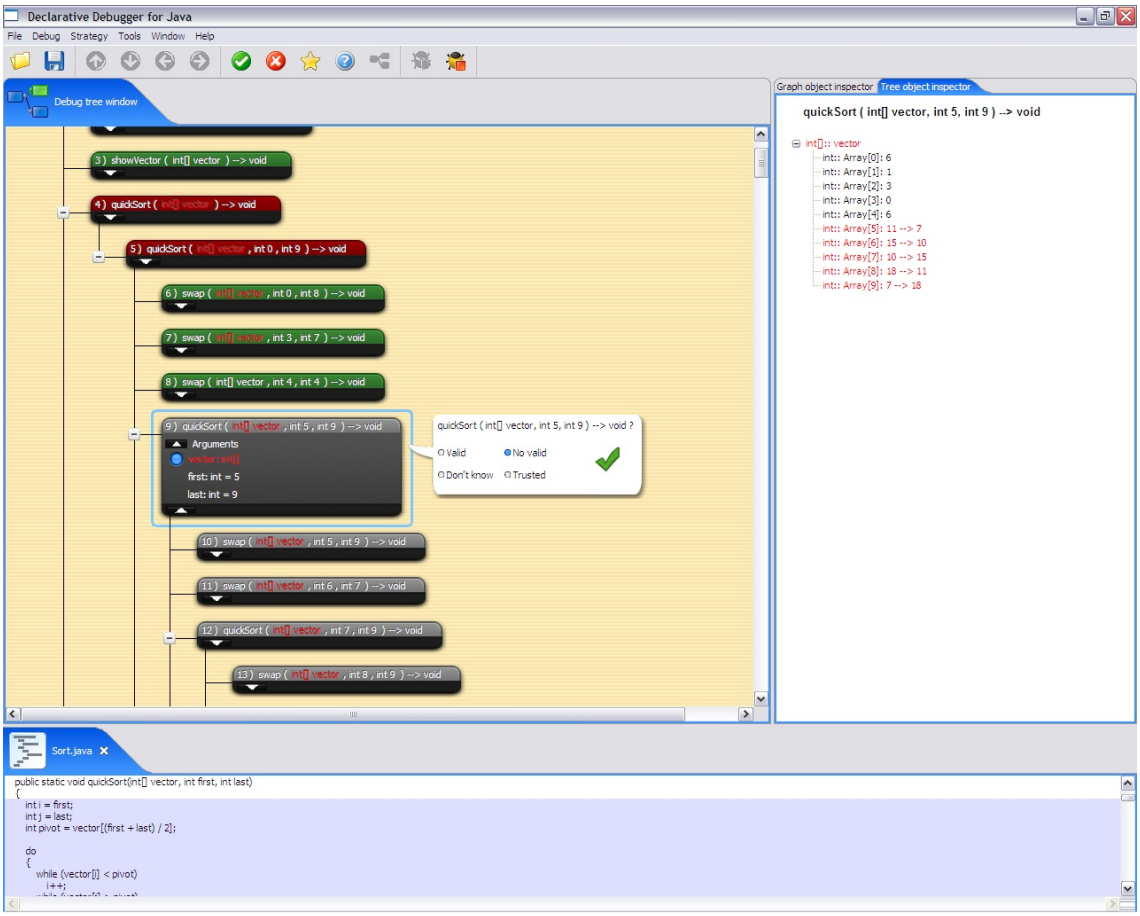

Fig. 3. Question produced by DDJ

ET is incomplete, the debugging session can start. In this case the debugger will only ask for those nodes that have been already completed, and, therefore, the user is able to determine whether the computation of the node is correct or not.

Step 2: Once a program has been loaded, the debugging session can start either manual or guided by the debugger. During a debugging session, at any moment, the user can change to manually or guided debugging several times. In manual debugging, the user can explore the ET and select wrong computations or discard computations that are correct. It is also possible to mark computations as trusted (e.g., because the code is reused, it is already tested, etc.). When a node is marked as trusted, the debugger automatically searches for all computations associated to the trusted code, and they are automatically discarded (i.e., marked as correct).

In guided debugging, the user only needs to choose the strategy to use (e.g., top-down) and start the debugging session with a single click.

Step 3: When the debugging session is guided, it is a dialogue between the debugger and the user. The debugger selects a node according to the strategy selected and ask to the user whether the computation of this node is correct. To answer the question, the user can inspect (it is interactively shown in the GUI) all the variables of the program that are in the scope of this computation. These variables are shown with their values before and after the execution so that the user can inspect the effects of the computation.

If the result of the computation is what the user expected, then the answer is *correct* and the node and its descendants are automatically marked as correct and thus discarded in the search for the bug. In the case that the user is sure that the method appearing in the question of the debugger is well implemented, she can answer *trusted*. As described before, this produces that all the nodes associated to this method are also marked as correct.

On the other hand, if the result is not the expected, then the answer is *wrong*. This means that the execution of this node is wrong, and thus, the error has been generated by itself or by any of its descendant nodes. Finally, if the user does not know what is the expected result of the node, then she can answer *I don't know*, and the debugger will continue the search not taking into account this node and

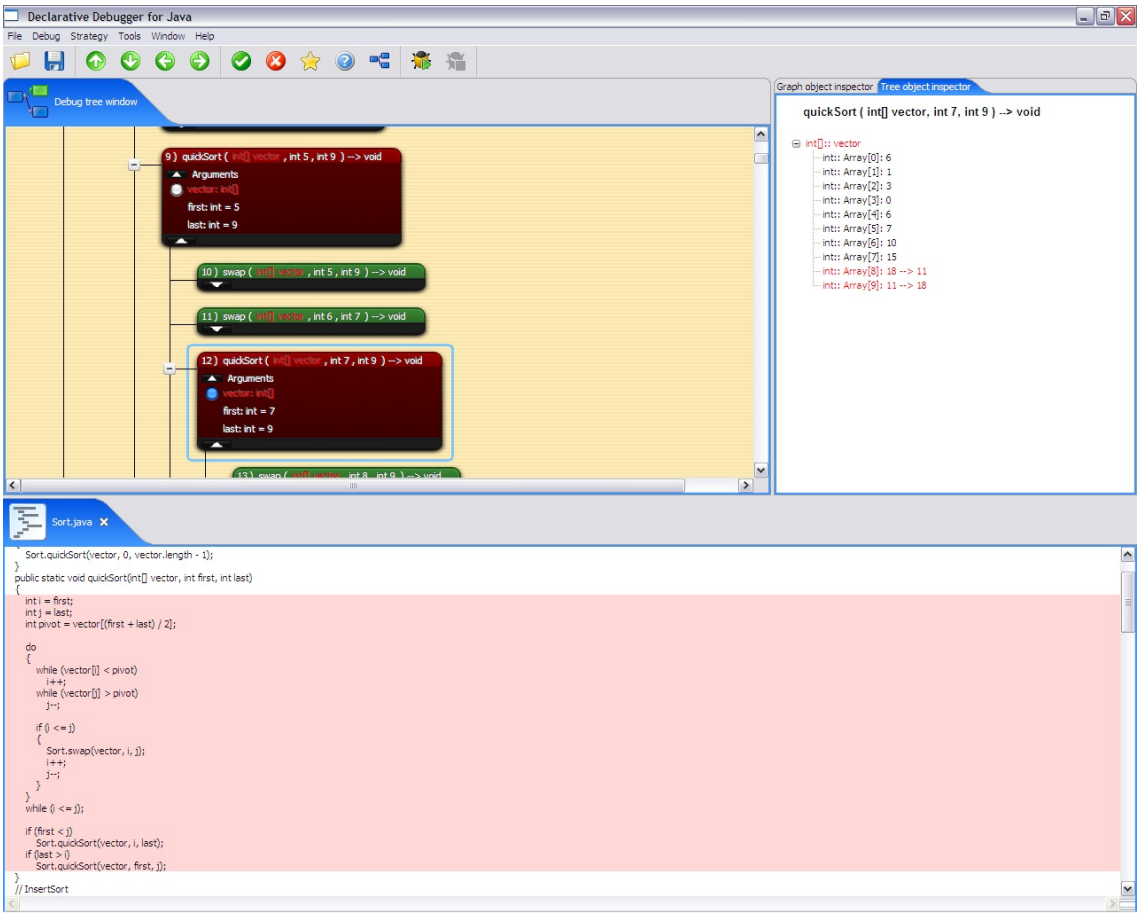

Fig. 4. Buggy source code highlighted by DDJ

producing a new question.

For instance, in Figure 3 we see DDJ with the ET of *vector.class* loaded. We see in the main pan the part of the ET associated with a quicksort computation. Some nodes are dark and others are light because the user already answered some of them stating their correctness. At this screenshot, DDJ is asking the question:

```
quickSort(int[] vector, int 5, int 9) --> void
```

where the values of the argument 'vector' are shown in the pan at the right. Here, those variables that changed during the call are in red and their initial and final values are shown. For instance, array[9] had initially value 7 and it was changed to 18.

The intended meaning of this call to quicksort is to order the values in the positions located between position 5 and 9 (both included). In the pan we see that the initial values were [11, 15, 10, 18, 7], and the final values are [7, 10, 15, 11, 18]. Therefore, the user should mark this computation as wrong.

After every answer of the user, the debugger uses the information provided to prune the ET and select a new question for the user. This process is repeated until the bug is found.

At any moment, it is possible to save the debugging session recording the answers of the user.

Step 4: After the dialogue with the user, the debugger identifies one node of the ET as buggy. This node is responsible of the wrong behavior of the program, and thus the source code associated to it contains the bug. When the bug is found, DDJ shows a message indicating the method that produced the wrong behavior and allowing the user to see the part of the source code that contains the bug (see Figure 5).

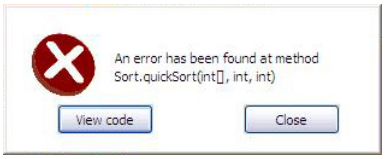

Fig. 5. Bug found

Figure 4 shows the source code pan (at the bottom) where the user can see the source code associated with any node of the ET. In this screenshot, the part of the code associated to the buggy node is highlighted. Observe that during the debugging session,

the user does not need to control the execution of the program or use breakpoints and she does not even need to see the source code.

IV. Tool Information

DDJ has been completely implemented in Java. It contains about 20400 LOC and it uses SWT for the graphical visualization, which is standard and hence, it can be used in different operative systems. Thanks to JDBC, DDJ can interact with different databases. The current distribution includes both a MySQL and Access databases. The last release of the debugger is distributed in English, Spanish and French.

All described functionalities in this paper are completely implemented in the last stable release. This version is open and publicly available at:

http://www.dsic.upv.es/ ˜jsilva/DDJ

In this website, the interested reader can find installation steps, examples, demonstration videos and other useful material.

V. Conclusion

Algorithmic debugging is a technique widely extended in the declarative paradigm. Since its introduction, many implementations have been distributed for functional and logic languages such as Haskell [14], [6], [15], Mercury [13], Toy [4], [5] or Curry [3]. Nevertheless, the technique has not been extended to the object-oriented paradigm, and, currently, there does not exist an implementation for C++ and only one implementation is distributed for Java.

The main problem of adapting algorithmic debugging to Java is scalability. In DDJ, we solve the memory scalability problem by introducing a database and a virtual data structure that represents the ET in main memory. And we solve the time scalability problem with a system of caches and with the ability of debugging incomplete ETs that allow us to start a debugging session before having generated the ET. The debugger also includes new features such as a balancing transformation that speeds up the debugging session.

DDJ is open and freely distributed. As it is completely implemented in Java with standard components we are currently producing a release that can be installed to Eclipse as a plugin.

References

[1] E. Av-Ron. *Top-down Diagnosis of Prolog Programs*. Ph.D. thesis, Weizmanm Institute. 1984.

[2] D. Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.

[3] B. Braßel. A Debugger for Functional Logic Languages. In *Proc. of the 25th Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, Technical Report 0811, pages 78–92, Christian-Albrechts-University of Kiel, Germany, 2008.

[4] R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, ACM Press, New York, USA, 2005.

[5] R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science, 2006.

[6] T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.

[7] D. Giammona. ORACLE ADF - Putting It Together. ADF Declarative Debugger Archives, Oracle's blog available at: http://blogs.oracle.com/DavidGiammona/adf_declarative_debugger, November 2009.

[8] H. Girgis and B. Jayaraman. JavaDD: a Declarative Debugger for Java. Technical Report 2006-07, University at Buffalo, March 2006.

[9] V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749, 1993.

[10] F. González, R. De Miguel and S. Serrano. Depurador Declarativo de Programas JAVA. Master's Thesis, Universidad Complutense de Madrid, 2006.

[11] D. Insa, J. Silva and A. Riesco. Balancing Execution Trees. To appear in *Proc. of the 10th Spanish Workshop on Programming Languages, (PROLE 2010)*, Valencia, Spain, 2010.

[12] D. Insa and J. Silva. Debugging with Incomplete and Dynamically Generated Execution Trees. *Proc. of The 20th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2010)*, Hagenberg, Austria, 2010.

[13] I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.

[14] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[15] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

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

[17] J. Silva. Three New Algorithmic Debugging Strategies. In *Proc. of VI Jornadas de Programación y Lenguajes (PROLE'06)*, pages 243–252, 2006.

[18] J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.

[19] Sun MicroSystems. Java Platform Debugger Architecture. Accessible from: http://java.sun.com/javase/technologies/core/toolsapis/jpda, 2010.