A New Hybrid Debugging Architecture for Eclipse

Juan González, David Insa and Josep Silva

Universitat Politècnica de València, Valencia, Spain
{jgonza, dinsa, jsilva}@dsic.upv.es

Abstract. During many years, Print Debugging has been the most used method for debugging. Nowadays, however, industrial languages come with a trace debugger that allows programmers to trace computations step by step using breakpoints and state viewers. Almost all modern programming environments include a trace debugger that allows us to inspect the state of a computation in any given point. Nevertheless, this debugging method has been criticized for being completely manual and time-consuming. Other debugging techniques have appeared to solve some of the problems of Trace Debugging, but they suffer from other problems such as scalability. In this work we present a new hybrid debugging technique. It is based on a combination of Trace Debugging, Algorithmic Debugging and Omniscient Debugging to produce a synergy that exploits the best properties and strong points of each technique. We describe the architecture of our hybrid debugger and our implementation that has been integrated into Eclipse as a plugin.

1 Introduction

Debugging is one of the most time-consuming tasks in software engineering. However, the automatization of debugging is still far from being a reality. In fact, during many years, Print Debugging (also known as Echo Debugging) has been the most common method for debugging. Print Debugging allows us to easily know whether the computation traverses one specific point. Many bugs can be corrected with this information, and the programmer (maybe optimistically) prefers to use this method before loading a real debugger. Nevertheless, some bugs are almost impossible to detect with Print Debugging, specially in presence of random values, input, and concurrency.

Fortunately, all modern programming environments, e.g., Borland JBuilder [**?**], NetBeans [**?**], Eclipse [**?**], SICStus Prolog SPIDER IDE [**?**] or SWI-Prolog [**?**] include a trace debugger, which allows programmers to trace computations step by step. However, Trace Debugging is a completely manual task, and the programmer is in charge of inspecting the computations of the program at a low abstraction level. For this reason, other debugging techniques have been proposed to solve some of these problems, but they also suffer from other problems. For instance, Algorithmic Debugging [**?**,**?**] (also known as Declarative Debugging) is semi-automatic, i.e., the search for the bug is directed by the debugger instead of the programmer; and its abstraction level is so high that programs can be debugged without even seeing the code, but it suffers from scalability problems.

In this work we introduce a hybrid debugging technique that combines three different techniques, namely, Trace Debugging (TD), Omniscient Debugging (OD) and

Algorithmic Debugging (AD). The combination is done exploiting the strong points of each technique, and counteracting or removing the weak points with their composition. Our method is presented for the programming language Java—our implementation is an Eclipse plugin for Java—but the technique and the architecture of our debugger could be applicable to any other programming language. In summary, the main contributions of this work are the following:

- The design of a new hybrid debugging technique that combines TD, OD and AD.
- The integration of the technique on top of the JPDA architecture—which was conceived for tracing, but not for algorithmic or omniscient debugging—.
- The implementation of the technique as a Eclipse plugin.
- The empirical evaluation of the new architecture that demonstrates the practical scalability of the technique.

The rest of the paper is structured as follows: In Section 2 we describe TD, OD, and AD, analyzing their strong and weak points. Then, in Section 3 we present our new hybrid debugging technique and explain its architecture. In Section 4 we describe our implementation, which has been integrated into Eclipse. The related work is presented in Section 5. Finally, Section 6 concludes and outlines the future work.

2 Debugging Techniques

This section describes the three debugging techniques that we use in our hybrid method: TD, OD and AD. For each technique, we also analyze its strong and weak points and its applicability to Java.

2.1 Trace Debugging

The most used method for debugging is TD. It allows the programmer to traverse the trace of a computation step by step. The programmer places a *breakpoint* in a line of the source code and the debugger stops the computation when this line is reached. Then, the programmer proceeds line by line and, at each step, the programmer can inspect the state of the computation (i.e., variables' values, exceptions, etc.). During the traversal of the trace, when a call to a method is reached, the debugger can either enter the method (*step into*) or skip it (*step over*). Modern breakpoints are conditional, i.e., the breakpoint includes conditions over the values of some variables, or over the action performed where they are defined. For instance, it is possible to define a breakpoint that only stops the computation when an exception happens, or when a class is loaded. TD has one important advantage over other debugging techniques: scalability. The debugger only needs to take control over the interpreter to execute the program normally. Hence, its scalability is the same as the one of the interpreter. On the other hand, TD has four main drawbacks:

1. The whole debugging process is done at a very low abstraction level. The programmer just follows the steps of the interpreter, and she needs to understand how variables' values change to identify an error.

2. The debugger can generate an overwhelming amount of information.
3. The debugging process is completely manual. The programmer uses her intuition to place the breakpoints. If the breakpoint is after the bug, she has to place it again before, and restart the program. If the breakpoint is placed long before the bug, then she has to manually inspect a big part of the computation.
4. The inspection of the computation is made forwards, while the natural way of discovering the bug is backwards from the bug symptom.

2.2 Omniscient Debugging

Omniscient debugging [**?**] solves the fourth drawback of TD with the cost of sacrificing scalability. Basically both techniques rely on the use of breakpoints and they both do exactly the same from a functional point of view. The difference is that OD allows the programmer to trace the computation forwards and backwards (chronologically). This is very useful, because it allows the programmer to perform steps backwards from the bug symptom. To do this, the debugger needs a mechanism to reconstruct every state of the computation. One of the most scalable schemas to do this is depicted in Figure 1. In this figure, we have an horizontal line representing an execution as a sequence of events. Some of these events are method invocations (represented with a white circle), and method exits (represented with a black circle). Each event is identified with a timestamp. From the execution, the omniscient debugger stores a variable history record that contains the values of all variables together with the exact timestamp where they updated each value. The omniscient debugger also stores information about the scope of variables that we omit here for clarity. With this information the debugger can reconstruct any state of the computation. For instance, in state 42, value `M.N.y` did not exist, and the last value of variables `O.x` and `O.v[3]` was 23 and 3 respectively.

Being able to reconstruct the complete trace also allows the programmer to start the execution at any point. Nevertheless, storing all values taken by all variables in an

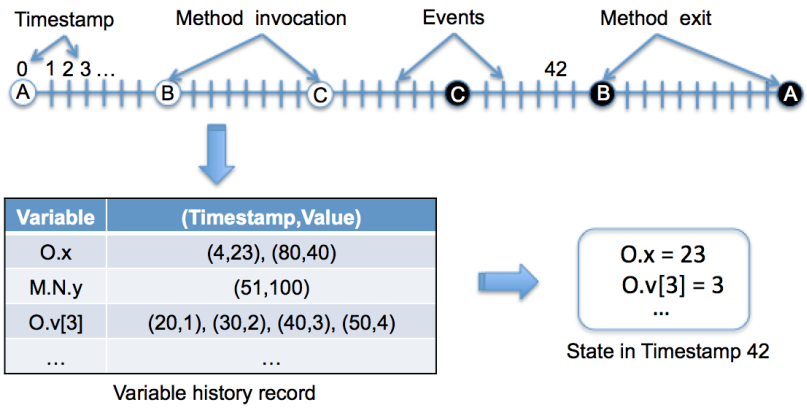

Fig. 1. Timestamps-based scheme to store traces in Omniscient Debugging

execution is usually impossible for realistic industrial (large) programs, and even for medium sized programs. Thus scalability is very limited in this technique.

2.3 Algorithmic Debugging

Algorithmic Debugging [**?**,**?**] is a semi-automatic debugging technique that is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. The questions are always whether a given result of a method invocation with given input values is actually correct. The answers provide the debugger with information about the correctness of some (sub)computations of a given program; and the debugger uses them to guide the search for the bug until a buggy portion of code is isolated.

Example 1. Consider the Java program in Figure 2 that simulates Tic-Tac-Toe games —we suggest the reader not to see the code now, and try to debug this program without seeing the code. This is possible with AD as it is shown in the following debugging session—. This program is buggy, and thus it does not produce the expected marks in the board. Class *Replay* reads from a file a new game and it reproduces the game using a *TicTacToe* object. The null character is represented in Java with `'\u0000'`.

An AD session for this program is shown below where boards are represented with a picture for clarity (e.g., {{X,",}{O,",}{",",}} is represented with ). For the time being ignore column `Node`:

```
Starting Debugging Session...
Node  Initial context          Method call           Final context       Answer
(2)  [turn='X',board= ]  game.mark('X',0,0)  [turn='O',board= ] ?  YES

(7)  [turn='O',board= ]  game.mark('O',0,1)  [turn='X',board= ] ?  NO

(8)  [turn='X',board= ]  game.win(0,1)=false [turn='X',board= ] ?  YES

Bug found in method: TicTacToe.mark(char, int, int)
Discovered with the call: game.mark('O',0,1)
```

Note that the debugger generates questions, and the programmer only has to answer the questions with `YES` or `NO`. It is not even necessary to see the code. Each question is about the execution of a particular method invocation, and the programmer answers `YES` if the execution is correct (i.e., the output and the final context are correct with respect to the input and the initial context) and `NO` otherwise.
At the end, the debugger points out the specific call to a method in the code that revealed a bug in that method. In this case, method *TicTacToe.mark* is wrong. This method first checks whether the movement is correct (e.g, it is the player's turn, the mark is inside the board, etc.). If the movement is correct, then it places the mark in the corresponding position of `board`, it updates the next player to make a movement, and it finally checks whether this mark wins the game. Unfortunately, the programmer interchanged

```
public class Replay {
    public static void main(String[] args) throws IOException {
        TicTacToe game = new TicTacToe();
        FileReader file = new FileReader("./game.rec");
        play(game, file);
    }
    private static void play(TicTacToe game, FileReader file) throws IOException {
        BufferedReader br = new BufferedReader(file);
        String linea = br.readLine();
        while ((linea = br.readLine()) != null) {
            char player = linea.charAt(0);
            int row = Integer.parseInt(linea.charAt(2) + "");
            int col = Integer.parseInt(linea.charAt(4) + "");
            game.mark(player, row, col);
        }
    }
}

public class TicTacToe {
    private static boolean equals(char c1, char c2, char c3) {
        return c1 == c2 && c2 == c3;
    }

    private char turn = 'X';
    private char[][] board = new char[3][3];

    public void mark(char player, int row, int col) {
        if (turn == '\u0000' || turn != player
            || row < 0 || row > 2 || col < 0 || row > 2
            || board[row][col] != '\u0000')
            return;

        board[col][row] = player; // Bug!! Correct: board[row][col] = player;
        turn = turn == 'X' ? 'O' : 'X';
        if (win(row, col))
            turn = '\u0000';
    }
    private boolean win(int row, int col) {
        if (board[row][col] == '\u0000')
            return false;
        if (equals(board[row][0], board[row][1], board[row][2]))
            return true;
        if (equals(board[0][col], board[1][col], board[2][col]))
            return true;
        if (col == row && equals(board[0][0], board[1][1], board[2][2]))
            return true;
        if (col + row == 2 && equals(board[0][2], board[1][1], board[2][0]))
            return true;
        return false;
    }
}
```

Fig. 2. Example program

the row and the column producing a bug. This error can be easily corrected by replacing `board[col][row] = player` by `board[row][col] = player`.

Typically, algorithmic debuggers have a front-end that produces a data structure representing a program execution—the so-called *execution tree* (ET) [**?**]— and a back-end that uses the ET to ask questions and process the programmer's answers to locate the bug. Each node of the ET contains an equation that consists of a method execution with completely evaluated arguments and results. The node also contains additional

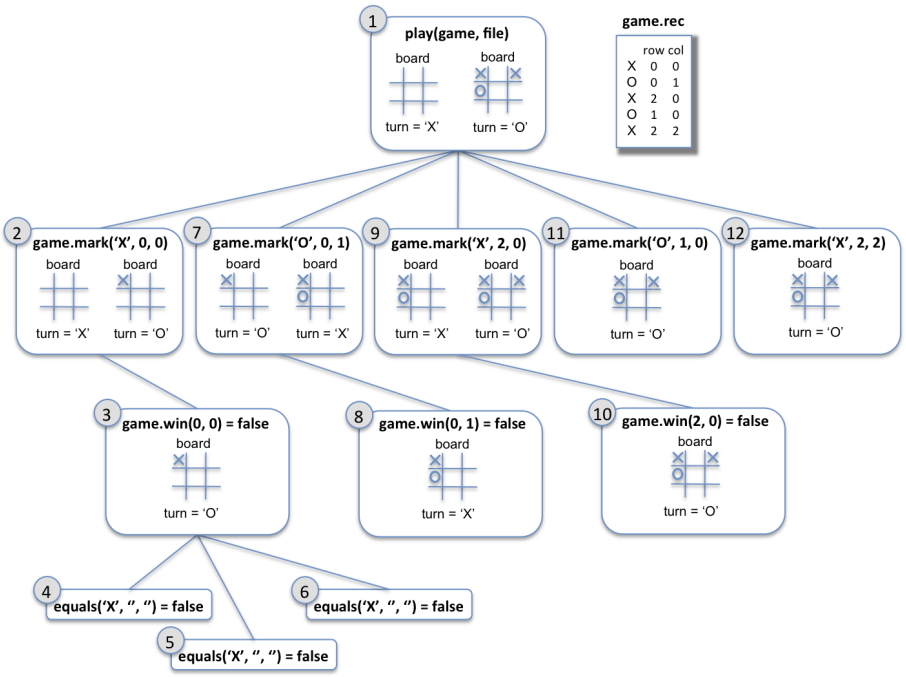

Fig. 3. ET associated with the call `play(game, file)` of the program in Figure 2

information about the context of the method before and after its execution (attributes values or global variables in the scope of the method).

Essentially, AD is a two-phase process: During the first phase, the ET is built, while in the second phase, the ET is explored. The ET is constructed as follows: The root node is (usually) the *main* function of the program; for each node n with associated method m, and for each method invocation done from the definition of m, a new node is recursively added to the ET as the child of n.

Example 2. Consider again the Java program in Figure 2. Figure 3 depicts the portion of the ET associated with the execution of the method `play(game, file)` using *game.rec* as the input file. Each node contains:

- A string representing the method call (including input and output) depicted at the top of each node.
- The variables (and their values) in the scope at the beginning and at the end of the method execution. When the value of a variable is modified during the execution of the method, the node contains both values on the left and on the right of the node respectively. When the variable is not modified, it is shown only once in the middle of the node.

Once the ET is built, in the second phase, the debugger uses a strategy to traverse the ET asking an oracle to answer each question. For instance, each question in the

debugging session of Example 1 corresponds to a node (see column `Node`) of the ET in Figure 3. These nodes have been selected by the strategy Divide & Query [**?**]. After every answer, some nodes of the ET are marked as correct or wrong. When all the children (if any) of a wrong node are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated with this node [**?**].

Theorem 1 (Correctness of AD [?]). *Given an ET with a buggy node n, the method associated with n contains a bug.*

Theorem 2 (Completeness of AD [?]). *Given an ET with a bug symptom (i.e., the root is a method with a wrong final context), provided that all the questions generated by the debugger are answered, then, a bug will eventually be found.*

The most important advantage of AD is its high level of abstraction and its semi-automatic nature. The main drawbacks of this technique are:

1. Low scalability. Each ET node needs to record a part of the computation state (i.e., the context before and after the method execution). Storing the ET of the whole execution can be unpractical.
2. The strategy that traverses the ET can ask unnecessary questions until it reaches the part of the computation that contains the bug.
3. Low granularity of the error found. This technique reports a method as buggy, instead of an expression.

2.4 Comparison of the techniques and empirical analysis

Table 1 summarizes the strong and weak points of the techniques.

Feature	**Trace**	**Omniscient**	**Algorithmic**
Scalability	`Very Good`	`Very bad`	`Bad`
Error granularity	`Expression`	`Expression`	`Method`
Automatized process	`Manual`	`Manual`	`Semi-automatic`
Execution	`Forwards`	`Forwards and backwards`	`Forwards and backwards`
Abstraction level	`Low`	`Low`	`High`

Table 1. Comparison of debugging techniques

In our hybrid technique, we want to take advantage of the high abstraction level of AD. We also want to exploit the semi-automatic nature of this technique to speed up bug finding and to avoid errors introduced by the programmer when searching the bug. However, AD alone would explore all computations as if they all were suspicious. To avoid this, we want to take advantage of the breakpoints, which provide information to the debugger about what parts of the computation are suspicious for the programmer (e.g., the last changed code). Hence, we designed our technique to start using the breakpoints of the programmer, and then automatize the search using AD. Another problem

that must be faced is that AD is able to find a buggy method, but not a buggy expression. Therefore, once AD has found a buggy method, we can use OD to further investigate this method in order to find the exact expression that produced the error.

In order to analyze whether this scheme is feasible, we studied the scalability problem of both AD and OD. Operationally, AD and OD are similar. They both record events produced during an execution, and they associate with each event a timestamp. The main difference is that AD only needs to reconstruct the state of the events that correspond to method invocations and method exits (white and black circles in Figure 1). Moreover, AD does not need to store information about local variables—only about attributes and global variables—, which is an important difference regarding scalability.

We conducted some experiments to measure the amount of information stored by an algorithmic debugger to produce the ET of a collection of medium/large benchmarks (e.g., an interpreter, a parser, a debugger, etc.) accessible at:

`http://www.dsic.upv.es/~jsilva/DDJ/#Experiments`

Results are shown in Table 2.

Benchmark	var. num.	ET size	ET depth
argparser	8.812	2 MB	7
cglib	216.931	200 MB	18
kxml2	194.879	85 MB	9
javassist	650.314	459 MB	16
jtstcase	1.859.043	893 MB	57
HTMLcleaner	3.575.513	2909 MB	17

Table 2. Benchmark results

Column `var. num.` represents the total amount of variable changes stored. Column `ET size` represents the size of the information stored. Observe that the last benchmark needs almost 3 GB. Column `ET depth` is the maximum depth of the ET (e.g., in benchmark `jtstcase`, there was a stack of 57 activation records during its execution). If we consider that this information does not include local variables, then we can guess that the amount of information needed by an omniscient debugger can be huge. Clearly, these numbers show that neither AD nor OD are scalable enough as to be used with the whole program. They should be restricted to a part of the execution. For AD, we propose to restrict its use only to the part of the execution that corresponds to a breakpoint (i.e., the execution of the method where the breakpoint is located). For OD, we propose to restrict its use only to the part of the execution that corresponds to a single method (i.e., the method where AD identified a bug). This proposal is completely aligned with the previous ideas discussed: AD will only start in a suspicious area pointed out by a breakpoint, and OD will only be used when a buggy method has been found, and thus the programmer can trace backwards the incorrect values identified at the end of this method.

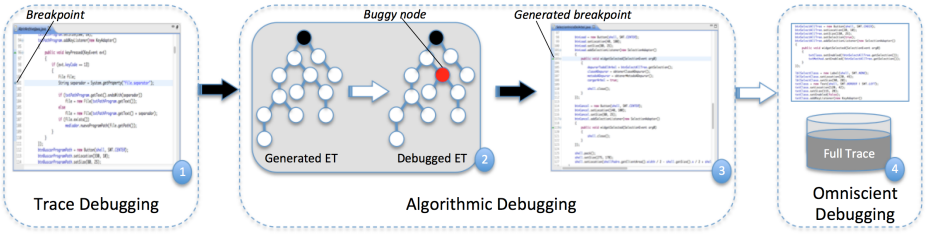

Fig. 4. Hybrid debugging with HDJ

3 Hybrid Debugging

In this section we present our hybrid debugger for Java (HDJ) based on the ideas discussed in the previous section. It combines TD, AD and OD to produce a synergy that exploits the best properties and strong points of each technique.

We start by describing the steps followed in a hybrid debugging session. Consider the diagram in Figure 4 that summarizes our hybrid debugging method. We see three main blocks that correspond to TD, AD and OD. These blocks contain four items that have been numbered; and these items are connected by arrows. Black arrows represent an automatic process (performed by the debugger), whereas white arrows represent a manual process (performed by the programmer):

Trace Debugging. First, after a bug symptom is identified, the user explores the code as usual with the trace debugger and she places a breakpoint b_1 in a suspicious line (probably, inside one of the last modified parts of the code).

Algorithmic Debugging. Second, the debugger identifies the method m_1 that contains breakpoint b_1, and it generates an ET whose root method is m_1. This is completely automatic. Then, the user explores the ET using AD until a buggy node n is found. Note that, according to Theorem 2, if method m_1 is wrong, then it is guaranteed that AD will find a buggy node (and thus a buggy method). From n, the AD automatically generates a new breakpoint b_2. b_2 is placed in the definition of the method m_2 associated with n. And, moreover, b_2 is a *conditional* breakpoint that forces the debugger to stop at this definition, only when the bug is guaranteed to happen. The condition ensures that all values of the parameters of m_2 are exactly the same as their values in the call to m_2 associated with n.

Example 3. Consider a buggy node $\{x = 0\}\ m(42)\ \{x = 1\}$, where the definition of method m, *void* $m(int\ a)$, is located between lines 176 and 285. Then, the conditional breakpoint generated for it is $(176, \{x = 0, a = 42\})$. Alternatively, another conditional breakpoint can also be generated at the end of the method.

According to Theorem 1, because node n is buggy, then method m_2 contains a bug.

Omniscient Debugging. Third, the debugger acts as an omniscient debugger that explores method m_2 by reproducing the concrete execution where the bug showed up during AD. The user can explore the method backwards from the final incorrect result of the method. Observe that the OD phase is scalable because it only needs

to record the trace of a single method. Note that all method executions performed from this method are known to be correct thanks to the AD phase.

The three phases described produce a debugging technique that takes advantage of all the best properties of each technique. However, one of the most important objectives in our debugger is to avoid a rigid methodology. We want to give the programmer the freedom to change from one technique to another at any point. For instance, if the programmer is using TD and decides to use OD in a method, she must be able to do it. Similarly, new breakpoints can be inserted at any moment, and AD can be activated when required. The architecture of our tool provides this flexibility that significantly increases the usability of the tool, and we think that it is the most realistic approach for debugging.

3.1 Architecture

This section explains the internal architecture of HDJ, and it describes its main features. HDJ is an Eclipse plugin that takes advantage of the debugging capabilities already implemented in Eclipse (i.e., HDJ uses the Eclipse's trace debugger), and it adapts the already existent Declarative Debugger for Java (DDJ) [**?**] to the Eclipse workbench. The integration of HDJ into Eclipse is described in Figure 5.

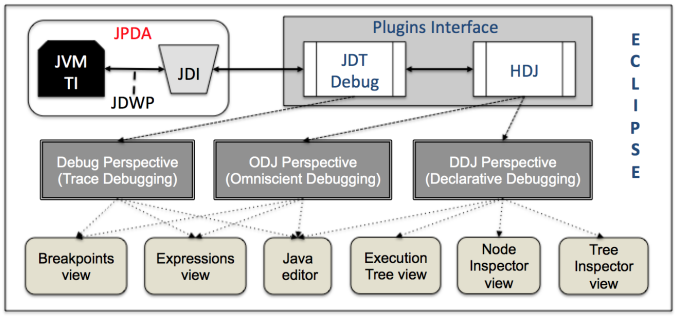

Fig. 5. Integration of HDJ into Eclipse

One of the debuggers, the trace debugger, was already implemented by an Eclipse plugin called JDT Debug. The other two debuggers have been implemented in the HDJ plugin. The tool allows the programmer to switch between three perspectives:

Debug: This perspective allows us to perform TD. It is the standard perspective of Eclipse for debugging. It is composed of several views and editors and it offers a wide functionality that includes conditional breakpoints, exception breakpoints, watch points, etc.

ODJ: This perspective allows us to perform OD. It contains the same views and editors that form the standard debug perspective. Therefore, although the programmer is using a different debugger with a totally different debugging mechanism, their GUI

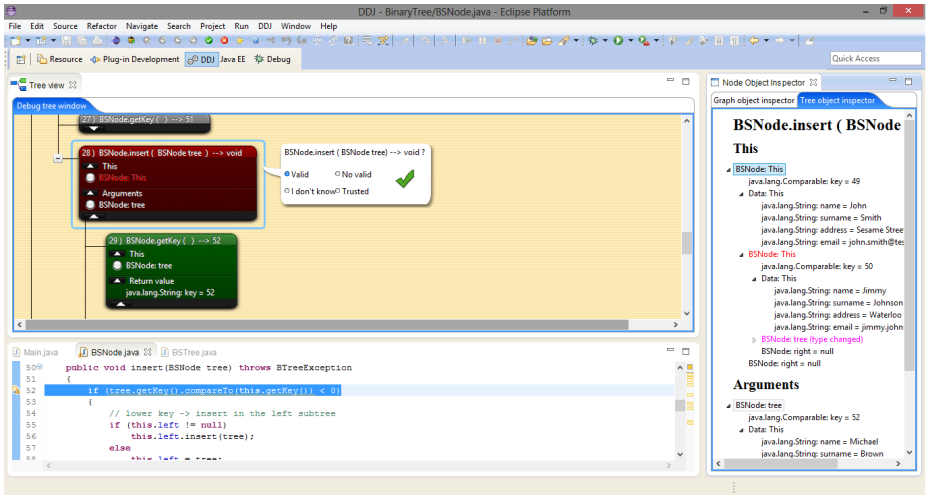

Fig. 6. Snapshot of HDJ (DDJ perspective)

is exactly the same; and thus, the internal differences are transparent for her. The only difference is that ODJ allows us to explore the execution backwards. Internally, it uses a trace of the execution (as the one described in Section 2.2) that is stored in a database.

DDJ: This perspective allows us to perform AD. An usage example of this perspective interface is presented in Figure 6. In the figure we can see two of its three views and one editor. First, on the left we see the `ET view`, which contains the ET and the questions generated by the debugger. Second, on the right we see the `Node inspector`, which shows all the information associated with the selected ET node. This includes the initial context, the method invocation and the final context, where changes are highlighted with colors. Third, at the bottom we see the `Java editor`, which contains the source code and the breakpoints. This editor is shared between the three debuggers, and thus, all of them manipulate the same source code, and handle the same breakpoints of the programmer.

One of the important challenges when integrating two new debuggers into Eclipse was to allow all of them to debug the same program together (i.e., giving the programmer the freedom to change from one debugger to the other in the same debugging session). For this, all of them must have access to the same target source code (e.g., a breakpoint in the target source code should be shared by the debuggers), and use the same target Java Virtual Machine (JVM) and the same execution control over this target JVM. In the figure, this common target JVM is represented with the black box. The Java Virtual Machine Tools Interface (JVM TI) provides both a way to inspect the state and to control the execution running in the target JVM. The debuggers access it through the Java Debug Interface (JDI) whose communication is ruled by the Java Debug Wire Protocol (JDWP). This small architecture to control the JVM is called Java Platform Debugger Architecture (JPDA) [**?**].

The integration of HDJ into Eclipse implies having three different debuggers accessing and controlling the same JVM where the debuggee is being executed. Therefore, our architecture uses two different JVMs that run in parallel and communicate via JPDA. The first JVM is where the debuggee is executed. The second JVM is where the debuggers are executed. It is important to remark that the information of one JVM cannot be directly accessed by the other JVM. Controlling one JVM from the other must be done through JPDA.

A first idea could be to execute the program in the target JVM and stop it when the statement that the programmer wants to inspect is reached. However, this would imply to re-execute the program once and again every time the programmer wants to perform a step backwards (i.e., to inspect the previous statement). Obviously, this is a bad strategy, because every time the program is re-executed, the state could change due to, e.g., concurrency, nondeterminism, input, etc. Therefore, even if we reached the same statement, it could vary between executions, and the information shown to the programmer would not be confident. Hence, we need to use some memorization mechanism to store all relevant states of a single execution.

Prior to our current implementation, our first design was conceived in such a way that the JVM of the debugger directly controlled the JVM of the debuggee using communication through JPDA. This implementation had to establish communication between both JVMs after every relevant event. This produced a heavy interaction with a massive message passing that was not scalable even for small programs. Therefore, we designed a second strategy whose key idea is to let the JVM of the debuggee to control itself. More precisely, before executing the debuggee in the JVM, we load a thread in this JVM so that, this thread directs the debugging of the program, thus, avoiding unneeded communication thorough JPDA. Figure 7 summarizes the internal architecture of the debugger to control the execution of the debuggee.

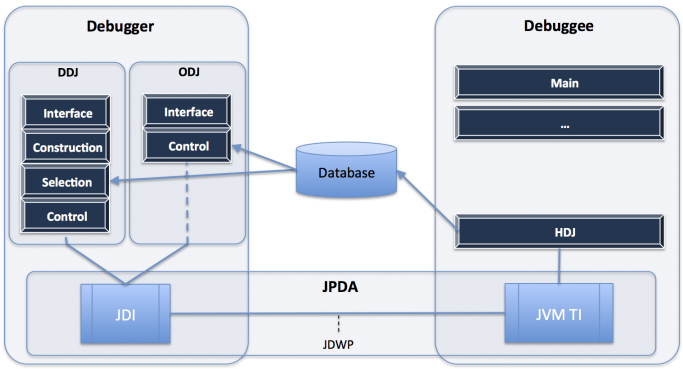

Fig. 7. Architecture of HDJ

The big boxes represent two JVMs. One for the debugger, and one for the debuggee. The debugger has two independent modules that can be executed in parallel: The algorithmic debugger DDJ, and the omniscient debugger ODJ. Each dark box represents a

thread. DDJ has four threads: *interface* to control the GUI, *construction* to build the ET, *control* to control and communicate with the debuggee JVM, and *selection* to select the next question. ODJ has two threads: *interface* and *control* that perform similar tasks as in DDJ. In the debuggee, a new thread is executed in parallel with the program. This thread, called *HDJ*, is in charge of collecting all debugging information and storing it in a database. This information is later retrieved by threads *control*. Thread *HDJ* makes this approach scalable, because it allows to retrieve all the necessary information with a very reduced set of JPDA connections. In the case of OD, the information stored in the database by thread *HDJ* contains all changes of variable values occurred during the execution of the method being debugged.

Example 4. Consider again the debugging session in Example 1. In this debugging session AD determined that method `mark` is buggy, and that the bug shows up with the specific call `game.mark('O',0,1)`. With this information, HDJ automatically generates a conditional breakpoint to debug this call. The information stored in the database by thread *HDJ* for this call is shown in Figure 8. Observe that only the variables that changed their value during the execution are stored.

Variable	(Timestamp, Value)
player	(0, 'O')
row	(0, 0)
col	(0, 1)
turn	(0, 'O'), (5, 'X')
board	(0, [['X',,],[,,],[,,]]), (4, [['X',,],['O',,],[,,]])

Fig. 8. Information stored in the database by the omniscient debugger

4 Implementation

HDJ has been completely implemented in Java. It contains about 29000 LOC: 19000 LOC correspond to the implementation of the algorithmic debugger (the internal functionality of the algorithmic debugger has been adapted from the debugger DDJ with some extensions that include the communication with JPDA trough JDT Debug, and the perspective GUI), 8300 LOC correspond to the implementation of the omniscient debugger that has been implemented from scratch, and 1700 LOC correspond to the implementation of the own plugin and its integration and communication with Eclipse. The debugger can make use of a database to store the information of the ET and the trace used in OD (if the database is not activated, the ET and the trace are stored in main memory). Thanks to JDBC, HDJ 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/HDJ/`

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

4.1 Empirical evaluation

In order to measure the scalability of our technique, we conducted a number of experiments to achieve the time needed by the debugger to start the debugging session. The scalability of TD is ensured by the own nature of the technique that reexecutes the program up to a breakpoint, and then shows the current state. In fact, we use the Eclipse's standard trace debugger that is scalable no matter where the breakpoint is placed. In the case of AD, scalability could be compromised if the debugger is forzed to generate the ET of the whole execution. Even in this case, we are able to ensure scalability: (i) The memory problem is solved with a database. Our debugger never stores the whole ET in main memory. It uses a clustering mechanism to store and load from the database the subtrees of the ET that are dynamically needed by the GUI. (ii) The time problem is solved by allowing the debugger to start the debugging session even if the ET is not completely generated (i.e, our debugger is able to debug incomplete ETs while they are being generated) [**?**]. In the case of OD, we cannot ensure scalability if it is applied to the whole program. For this reason, we limit the application of OD to a single method. This is scalable as demonstrated by our empirical evaluation whose results are shown in Table 3.

Benchmark		Execution	Omniscient
Statements	Objects	Time (ms)	Time (ms)
0 - 9 (294)	0 - 1 (96)	5	2060
	2 - 5 (97)	273	3265
	6 - 18 (101)	27	4099
10 - 19 (29)	7 - 18 (10)	51	6527
	19 - 24 (10)	739	7348
	25 - 32 (9)	2062	12379
20 - 57 (10)	25 - 39 (3)	83	3999
	40 - 54 (4)	117	6347
	55 - 100 (3)	176	3757

Table 3. Benchmarks results for OD

This table summarizes the results obtained for 333 benchmarks. Each benchmark measures the time needed to generate all the information used in OD (the information stored in the database by thread *HDJ* in Figure 7). After this time, the debugger contains the state at any point in the method, and thus, the programmer can make backwards steps, jump to any point in the method and show the values of the variables at any

point. These benchmarks correspond to all methods executed (333 different methods) by the loops2recursion Java library [**?**] applied over a collection of 25 Java projects. This library automatically transforms all loops in the Java projects to equivalent recursive methods.

All benchmarks have been grouped into three categories according to the number of statements executed in the method (0-9, 10-19, and 20-57). Inside each category, we indicate the number of benchmarks that fall on this category between parentheses. Categories have been divided in subcategories that indicate the number of objects that have changed during the execution of the method (i.e., the number of objects that must be inspected and stored in the database). We have a total of 9 subcategories. Each of them indicates the average time needed to execute the methods in that subcategory (`Execution Time`), and the time needed to generate the information for OD (`Omniscient Time`). All the information is generated between 2 and 12 seconds. The variability between the rows is dependent on the size of the objects changed. Clearly, row 6 has less objects to store than rows 7, 8 and 9, but these objects are bigger, and thus both the execution and omniscient times are higher.

5 Related Work

While a trace debugger is always present in modern development environments, algorithmic debuggers and omniscient debuggers are very unusual due to their scalability problems already discussed. There exist, however, a few attempts to implement algorithmic debuggers for Java such as the algorithmic debugger JDD [**?**] and its more recently reimplemented version DDJ [**?**]. Other debuggers exist that incorporate declarative aspects such as the Eclipse plugin JavaDD [**?**] or the Oracle JDeveloper's declarative debugger [**?**] however, they are not able to automatically produce questions and to control the search to automatically find the bug. This means that they lack the common strategies for AD implemented in standard algorithmic debuggers of declarative languages such as Haskell (Hat-Delta [**?**]) or Toy (DDT [**?**]). None of this debuggers can work with breakpoints as our debugger does.

The situation is similar in the case of omniscient debuggers. To the best of our knowledge, OmniCore CodeGuide [**?**] is the only development environment for Java that includes by default an omniscient debugger. Nevertheless, for the sake of scalability, this debugger uses a trace limited to the last few thousands events. Some ad-hoc implementations exist that can work stand-alone or be integrated in commercial environments [**?**,**?**,**?**,**?**,**?**]. Almost all these works focus on how to make OD more scalable [**?**,**?**]. For instance, by reducing the overhead of trace capture as well as the amount of information to store using partial traces that exclude certain trusted classes from the instrumentation process [**?**]. Other works try to enhance OD, e.g., with causality links [**?**] that provide the ability to jump from the point a value is observed in a given variable to the point in the past when the value was assigned to that variable. This can certainly be very valuable to resolve the chain of causes and effects that lead to a bug.

There have been several attempts to produce hybrid debuggers that combine different techniques. The debugger ODB [**?**] combines TD with OD. It allows the user to debug the program using TD and start recording the execution for OD when the user

prefers. The debugger by Kouh et al. [**?**] combines AD with TD. Once the algorithmic debugger has found a buggy method, they continue the search with a trace debugger to explore this method (forwards) step-by-step. This idea is also present in our debugger, but we use OD instead of TD, and thus we also permit backwards steps. The debugger JIVE [**?**] combines TD, OD and dynamic slicing. It does not use AD, but allows the programmer to perform queries to the trace.

To the best of our knowledge, JHyde [**?**] is the only previous technique that combines TD, OD and AD. Unfortunately, we have not been able to empirically evaluate this tool (it is not publicly accessible); but considering its architecture, it is highly probable that it suffers from the same scalability problems as any other omniscient debugger. Unlike our solution, their architecture is based on program transformations that instrument the code to store the execution trace in a file as a side effect. First, this instrumentation and the execution of the trace usually takes a lot of time with an industrial program, so that the programmer has to wait for the instrumentation before starting to debug; and second, they store the trace of the whole program, while our scheme only needs the trace of a single method. The common point is that both techniques are implemented as an Eclipse plugin, and they both use the same data structure for OD and AD. This is important to reuse the trace information collected by the debugger. Another important feature implemented by both techniques is the use of a color vocabulary used in the views. This is very useful to allow the programmer to quickly see the changes in the state.

6 Conclusions and Future Work

Trace Debugging, Algorithmic Debugging and Omniscient Debugging are three of the most important debugging techniques. Some of them are more suitable for one specific kind of program, while for other programs the other techniques can be better. Furthermore, it is possible that one technique is desirable to debug one part of a program, while other technique is preferable for other part of the same program. For these reasons, in any development environment the three techniques should be available.

In this work, we introduce a new debugger called HDJ that implements and integrates the three techniques. The implementation uses a new debugging architecture that allows the three techniques to share the same target virtual machine, and the same target source code. This allows the programmer to change from one technique to the other in the same debugging session. Moreover, we present a new model for debugging that combines the three techniques. Our new debugging architecture is particularly interesting because it exploits the best properties of each technique (e.g., high precision, high abstraction level, etc.) and it minimizes the problems such as scalability. HDJ is open and freely distributed as an Eclipse plugin.

As future work, we plan to incorporate in our debugger the causality links functionality [**?**], which allows the programmer to click on a variable and jump to the statement that produced the value of this expression. We are also further improving the integration between AD and OD. In particular, we want to allow the programmer to select a node in the ET and automatically start an omniscient debugging session with the information of this node.

7 Acknowledgements

This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. David Insa was partially supported by the Spanish Ministerio de Eduación under FPU grant AP2010-4415.