Automatic Dependency Identification for Isolation of Software Errors
(extended abstract of the MSc dissertation)

Alexandre Cardoso Pais de Almeida
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor João Garcia

Abstract—In a typical software development cycle, a large amount of resources is spent locating errors. To find the cause of an error, developers typically execute the programs step by step, checking many operations and variables that might not be relevant to the problem in question. It is difficult to detach the relevant parts that triggered the error from the source code. This paper presents a tool that provides the programmer with a view of the program execution, containing the statements and variables that might lead to an observed error. Thus, simplifying and facilitating the debugging process. This tool was developed in Java for Java programs. First, it instruments the application to save the information that we consider relevant: variable assignments, method invocations, and conditional statements. Then, the saved data from the execution that reproduces the problem is presented in a variable dependency graph and a graph of the statements that induced those dependencies. The tool introduces a performance penalty which is quite affordable considering the traditional process of software development and testing.

Keywords—Software Errors, Debugging, Diagnosis, Instrumentation

I. INTRODUCTION

Software errors are usually difficult to detect and diagnose. Despite the large investment on testing and debugging during the software development process, the software continues to be delivered with undiscovered flaws. Empirical observation suggests that the density of bugs in industrial-strength code has stayed relatively constant [1], yet the volume of code that goes into a general software product today has increased by many orders of magnitude. This means that the number of overall bugs is growing at an alarming rate. If this trend continues, software could significantly affect our society which relies on software every day [2]. Software errors impose several billions of Euros annually to Western economies in terms of maintenance costs [3], [4]. Therefore, there is an urgent need to explore more effective software testing and debugging tools and software engineering methods to minimize bugs that: i) are detected during the software development process; ii) are reported by the end-users during the production runs.

Fixing bugs is clearly the most complex and time-consuming activity in the software development process [5], [6], [7]. It is estimated that 80% of the total cost of a software system is spent on fixing bugs [8], [9], [5]. The complexity of error correction is related to the distance that often exists between the symptoms of the error and the error itself. This means that the place in the source code where the error manifests itself (e.g. an error or exception) might not be close to the location where the error was caused. Therefore, developers have to analyze large amounts of source code attempting to locate the error, where it is likely that a large portion of the code is not related to the problem.

Several diagnostic tools, particularly debuggers, can aid the process of finding bugs. However, these tools are not fully effective. When using a debugger, a considerable part of the programmer’s time is spent re-executing the program several times because there is no way to anticipate the optimal placement of breakpoints or how to choose the variables to observe to identify the causes of an error. Thus, the debugging process tends to be a repetitive cycle of execution/observation/speculation.

The main objective of this paper is to simplify the software error diagnosis by guiding developers towards the statements and variables that could be relevant for a particular error. This will relieve developers from having to fall back on the usual approach of re-executing the programs repeatedly with or without the debugger (reintroducing data, setting breakpoints, running step by step and observing the state of the program). This paper presents a tool that after the execution helps programmers locating the root cause of a program fault without any further executions. With this tool, when a programmer tests an application, the execution is monitored using program instrumentation to log the execution path. To accomplish this, we save every variable assignment, conditional statements and method invocations.

Imagine that the programmer detects a problem in the end of the execution (e.g. an error, an unusual output, among other symptoms). With our tool, the programmer is presented with an intuitive graphical representation of the execution path as well as the sequence of variable assignments that led to the current program state. Thus, the usual debugging process would be greatly simplified and accelerated because it would be done in reverse of the usual order of the program’s progress: the programmer would start debugging from the point where the program terminated. Additionally, we believe that presenting the dependency chain of every program variable would allow the developers to understand at any moment the reasons behind every variable value at any point. Therefore, programmers would save time gathering the necessary information without re-executing and exploring the application code. The tool
presented in this paper is implemented as an Eclipse IDE [10] Plug-in and it is available online.

II. RELATED WORK

This section presents and briefly describes several papers published in the debugging area. Debugging programs typically involves four stages, illustrated in Fig 1. Most of the existing papers in this area address one or more of the steps showed in this figure. As mentioned, this paper proposes a new technique for diagnosing software errors.

![Fig. 1. The four phases of the debugging process.](image)

A. Bug Detection vs. Bug Diagnosis

Bug detection and bug diagnosis are two concepts that are often confused. Briefly, detection involves finding out whether the program has the expected result or performance, whereas the diagnosis involves identifying and locating the causes of a faulty behavior.

A diagnosis is made by the developers, but this is not always the case for the detection: it can be done by programmers during the development cycle, testing and maintenance, but can also be carried out by the users. For this latter case we use error reporting tools [11], [12], [13].

1) Bug Detection: There are several strategies and tools that focus on bug detection. However, we think that those strategies can be used to help the diagnosis process since we believe that programmers “manually” employ some of them when diagnosing errors. Therefore, we briefly describe some strategies that we consider an inspiration to help the diagnosis process:

- **a) Model Checking:** is the algorithmic analysis of programs to prove properties of their executions, automatically checking whether the model of the program meets a given specification [14].

- **b) Program Slicing:** is the computation of a set of program statements (the program slice) that affect the values of variables at some point of interest [23]. Such a point of interest is called a slicing criterion and is typically specified by a location in the program in combination with the state of a subset of the program’s variables. This can be used in debugging to locate errors sources more easily [24].

- **c) Symbolic Execution:** is used to cycle through all possible executions of a program and determine which inputs cause each part of a program to execute [17], [18]. Opposed to normal execution that obtains actual inputs, symbolic execution assumes symbolic values for inputs. Thus, these input symbols can initially take any value.

- **d) Data Flow Analysis:** is a technique for gathering information about the possible set of values calculated at various points in a computer program [19], [20]. A program’s Control Flow Graph (CFG) [19] is used to determine those parts of a program to which a particular value assigned to a variable might propagate. It is a static representation of a program, representing all possible paths that might be traversed during the program execution. It is a directed graph where each node in the graph represents a code block.

2) Bug Diagnosis: There are several strategies and tools that focus on bug diagnosis. Here, we focus our attention in two main strategies: Statistical Debugging and Program Slicing, which we describe in the following paragraphs:

- **a) Statistical Debugging:** is a powerful technique for identifying bugs that do not violate programming rules or program invariants - the bugs that are most difficult to isolate using traditional debugging techniques. With this approach, the program dumps data during its execution, which is used by statistical methods to isolate software bugs by comparing the data collected from a large number of correct and faulty executions [21]. The challenge with this approach is identifying events that contributed to a failure and use this information to help support the debugging process, considering the large collection of report data dumped by a program, where each report is distinguished according to whether the run succeeded or failed. Statistical methods can handle uncertain and incomplete information while still providing best-effort clues about the root causes of software failure [22].

- **b) Program Slicing:** is the computation of a set of program statements (the program slice) that affect the values of variables at some point of interest [23]. Such a point of interest is called a slicing criterion and is typically specified by a location in the program in combination with the state of a subset of the program’s variables. This can be used in debugging to locate errors sources more easily [24]. However, there are other applications for it, including software maintenance, optimization, program analysis, and information flow control [25]. This is a technique for simplifying programs by focusing on selected aspects of their semantics [25]. The slicing process deletes those parts of the program that can be determined to have no effect upon the semantics of interest. This technique can be static where the slicing is applied on the source code with no other information than the source code or dynamic where the slicing is applied on a specific execution of the program (for a given execution trace) [26].

B. Debugging Practices

There are several studies that examine debugging practices, for instance [7], [27], [28], [29], [30], [31], [32]. These papers describe strategies commonly used by programmers (e.g. filtering, slicing, forward and backward reasoning). However,
these strategies are merely superficial since they describe methods and practices from a theoretical point of view. Indeed, using these strategies while debugging can help developers accelerate the debugging process, but they do not provide any implementation mechanism to diagnose software errors.

C. Debuggers

Debuggers (e.g. [33], [34], [35], [36], [37]) are tools that give the user the possibility to manipulate the program through process control commands and to examine and modify the program’s state. However, knowing that they have a powerful debugger behind their code, developers lower their concentration as well as their concern about producing high quality code. Also, debuggers allow setting breakpoints to break the execution when it reaches a specific point in the flow, line, or when a variable is assigned a specific value. They allow stepping through code, line by line, and stepping through or over routines. They let program states be restored, stepping back to the point where a defect arose. Good debuggers grant full examination of data, including structured and dynamically allocated data. Debuggers enable programmers to analyze a chain of routine calls and promptly view the source code of any routine. Also, it is possible to change the program’s parameters within the debugger environment.

When debugging a program, the user must determine possible modes of failure and then stop the execution at key points to examine the program’s state. The user must be able to predict the state values to determine whether the state is correct or not [38]. The debugger is not a substitute for good thinking and thinking is not a substitute for a good debugger either. The most beneficial and effective combination is good thinking and a good debugger [1].

The main challenge with debuggers is where to set breakpoints and which variables to inspect, tasks that depend on the programmer’s intuition. We believe that when programmers are using debuggers, they usually follow a trial and error approach. This typically means that they have to re-execute the program several times until they set the right breakpoints and inspect the right variables, allowing them to diagnose the cause of the error.

III. Proposed Solution

A. Vision

Once a bug is detected in the application, the next step is to diagnose it. When software developers want to understand the reason for a program’s behavior, they must translate their questions about the behavior into a series of questions about code and speculate about the causes in the process [39]. The typical diagnosis process is based on several re-execution cycles [7], [27], [32] where we believe that programmers try to collect and inspect the execution flow information, the variables values, the variables dependencies, and the justification for every program variable. Usually, this process involves navigating through a large portion of the source code, trying to reproduce the same steps and branches that the execution took. This process might be done with the help of a debugger. However, as we have seen, the use of debuggers might not be effective and it is extremely time-consuming. When using debuggers we are confronted with problems like setting wrong breakpoints and inadvertently skipping statements that might be related to the bug cause.

Considering all these problems, we propose a tool that requires a single execution without needing breakpoints. Our tool offers the possibility to rapidly navigate through the executed statements and have immediate access to the justification of every program variable. We believe that these are crucial features to simplify the diagnosis process that the common bug diagnosis tools do not offer. This way, developers can rapidly inspect the value of a program variable, justify the execution flow by watching the variables values involved in the conditional branches at certain moments, analyze the variables dependencies at certain points and relate those dependencies with the executed statements.

Our tool consists in two major steps: i) instrument the application to record every variable assignment, every method invocation, and every conditional statement during the execution; ii) present graphically all the recorded information in a simple and intuitive manner.

Consider the program illustrated in Figure 2, which receives a number as input and performs several calculations based on this value. If we execute this program passing as argument the value 10 the program crashes presenting the stack trace with the following information: Exception in thread “main” java.lang.ArithmeticException: / by zero at example.DebugProcessExample.main(DebugProcessExample.java:14). Here, we lose all the information regarding the program execution and the stack trace showing the thrown exception is the only information that the programmer has about the execution. In this case, the programmer would probably want to understand why the variable den had the value 0. For the example in Figure 2 the proposed tool presents: i) a graphical representation of the execution flow until the crash; ii) the assigned values to the variables during the execution; iii) a graphical representation of the variables dependencies. In Figures 3 and 4 are illustrated the output of this tool for the example in Figure 2. Recording the variables assignments allow to obtain the variable dependencies chain, while the method invocations and conditional statements allow to present the execution flow.

We believe that presenting the executed statements graph and the variables dependencies graph give the developer a huge edge since he instantly has all the information regarding the executed statements, the variables values, the variables dependencies, and the justification for every program variable. Therefore, he has the possibility to rapidly inspect the program’s execution.

B. Overview

Our architecture is shown in Figure 5. The remainder of this section describes the architecture’s main individual components and their interactions:

- **Instrumentation Component**: this component is responsible for instrumenting the application bytecode, injecting all the monitor invocations required to save
Fig. 2. A program example that abruptly terminates due to an arithmetic exception.

```java
public class DebugProcessExample {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Math.abs(a) - 8;
        int c = b - 2;
        int d = a - 8;
        int num = c + b;
        int den = methodAux(d, b);
        int result = num/den;
        System.out.println(result);
    }
}
```

Fig. 3. Output of the proposed tool for the example of Figure 2.

Fig. 4. Output of the proposed tool for the example of Figure 2 expanding methodAux.

Fig. 5. Proposed architecture.

all the execution information needed, particularly all the program assignments, method invocations, and conditional statements. Its input are the application’s .class files and it outputs the modified .class files.

- **Execution:** the application is executed with the modified .class files, which contain invocations to the *monitor* that is responsible for storing all the information that is logged during the execution.

- **Monitor:** this is responsible for maintaining and representing all the information that might be later relevant to find a bug cause (program assignments, method invocations, and conditional statements). The *instrumentation component* injects invocations to this *monitor* at key points in the application’s code, allowing the *monitor* to be populated with information during the execution.

- **Execution Analysis Component:** uses the *monitor* to inspect and organize the logged information, so that the statement, and the dependencies graphs can be built.

- **Presentation Component:** as the name says, this component is responsible for presenting the user with the execution information. This information is presented with two graphs: i) the executed statements graph, and ii) the variables dependencies graph.

Figure 6 presents our solution workflow. Is important to note that our solution not only applies to programs that crashed but programs that exhibit different symptoms, for instance, an incorrect output value. The solution operates as follows:

1) The program’s .class files are fed into the *instrumentation component*.
2) The *instrumentation component* injects statements in the developer code and outputs those modified .class files.
3) The program executes in the JVM with the modified .class files. During the execution all assignments, method invocations, and conditional statements are logged.
4) When the program execution terminates, the *execution analysis component* inspects and organizes the logged information.
5) After the information of the execution is structured; the statement and the dependencies graphs are built.
6) In the end, the *presentation component* is responsible for presenting those graphs in a pleasant manner.
C. Implementation

The implementation of our tool was developed using the Java Language and it is applicable only for Java programs. It resulted in an Eclipse Plug-in composed by one Action and one View, which is shown in Figure 7. The plug-in action triggers the instrumentation and the execution. Later, in the end of the execution it opens the view automatically, where the resulting graphs are drawn. Also, we used the Soot Framework [40], [41] to perform the application’s instrumentation.

D. Monitor

In order to be able to extract from the execution information that we consider relevant and to have a way of structuring and organizing that information, we choose to implement this monitor. As has been said, the monitor is responsible for maintaining and representing all the information that might be relevant to find a bug cause. It is a data structure, which is populated during the execution.

The first step for implementing the monitor was to build a representation for methods, local variables and all the relevant statements: if statements, invoke statements and assignment statements. Furthermore, we needed to convert the Soot Framework statement representation into the representation that we built. Additionally, to distinguish between different program contexts and multiple executions through the same point in the program, we developed our own trace and code location representations. The reason behind all these representations is the fact that during the tool workflow we need to have all the information well structured and consistent. Having the same representation along the full flow simplifies the process avoiding multiple mappings and conversions between representations in every step or component.

E. Instrumentation Component

This component was implemented using the Soot Framework. It allowed us to perform all the instrumentation, which was based on several requirements that have been established progressively. For every requirement, we improved our instrumentation process. Therefore, our approach to describe the implementation of this component will be based on those requirements. First, we give a brief overview of the implementation and then, we describe the implementation regarding each requirement.

1) Overview: The implementation of this component is based on three classes: the main, the visitor and the auxiliaries. The main class is responsible for:

- Setting up all the Soot configurations: setting all the directories, setting the Soot classpath, setting the Soot parameters;
- Finding all the classes that need to be instrumented and loading them into Soot;
- Setting up the monitor location;
- Iterating over the classes and methods to perform the instrumentation using the visitor.

The visitor iterates through the statements and according to their types has an associated method with a specific behavior. Those methods are responsible for injecting in the application’s code the monitor calls. Also, those methods use the auxiliaries class, which has common behavior methods to avoid code repetition.

a) Chain of variable dependencies and values: The first requirement that we established was to be able to extract the dependencies between the local variables and their corresponding values during the program execution. In order to fulfill this requirement, we iterate over every statement that belongs to every method body, seeking assignment statements. Once we find an assignment, we handle it by performing all the verifications needed to know which information we must save. This is because an assignment statement might appear in several forms: i) \( a = 1 \) or \( a = b \), ii) \( a = b + 1 \) or \( a = b + c \), iii) \( a = method(b, c) \), and so on. Therefore, we need to know in which case we are, to inject the correct statement that will handle the corresponding case properly.

b) Conditional statements: Having control over any conditional statement is essential to trace the execution flow of any program. Thus, each time we find an if statement the typical procedure is to create a representation of the if statement considering the branch that was executed, the variables involved in the condition and their values.

c) Method invocations: When we are instrumenting the main method we consider it as a normal method invocation and we create a representation for it. Later, when we find an invoke statement we process it in the following way: (a) Before the method invocation, we notify the monitor that a method call will happen next. This notification is basically creating the representation for a new invoked method and adding it to the invoked methods stack; (b) After the method invocation, we notify the monitor that the method call reached its end. This notification just pops out the invoked method from the invoked methods stack.
d) Multiple executions through the same point in the code: One of the problems with the approaches that we described is that we have several instructions to be invoked repeatedly, both in the context of a method invocation or within a cycle. To resolve this problem we created our own representation of the stack trace. Thus, each time there is a call to a method we do the following: (a) before the call: we generate a new entry in our stack trace; (b) after the call: we remove the last entry from our stack trace. This way we can distinguish every statement in every context.

e) Incremental instrumentation: In order to ensure that when the developer makes small changes, our tool does not have to re-instrument all classes, we implemented the system so that the instrumentation is incremental. The first time that our tool is executed, it instruments all the project classes. The second time it only instruments the classes that were changed, therefore becoming a more efficient system. To find out which classes were modified, we loop through the class directory of the project and the class directory of the instrumentation. Along with this sweep we compare the last modified date of the project classes and the last instrumentation done.

F. Execution Analysis Component

The implementation of this component is extremely simple. As we said, our solution is embedded in an Eclipse Plug-in. We developed an action that starts the instrumentation, then the execution, and in the end launches our view. When the program execution reaches its end, all the stored data by the monitor is flushed to the hard-drive. Then, the execution analysis component loads the data from the hard-drive and organizes it in a way that the presentation component becomes simpler. This component works as a bridge between the monitor and the presentation component.

G. Presentation Component

The implementation of this component was based on the development of a Plug-in View for Eclipse IDE. This component is fed with the data provided by the execution analysis component and represents it graphically. The graphical representations that we used are based on simple rectangle shapes and arrows from the org.eclipse.draw2d package provided by the Eclipse IDE.

1) Eclipse Plug-In View: This view has two main components: i) the execution graph and ii) the variable dependencies graph, as shown in Figure 8. This view is triggered by our plug-in action after the execution reaches its end. The first thing to be handled is the execution flow graph and then the variable dependencies graph.

When the view is launched, the first step is to get the main method, so that we can represent its execution flow. We start by the main method, and thereafter the user controls what is shown. To represent the execution flow of any given method, the process is the following. For each statement from the execution flow we create a RectangleFigure with the statement representation as the figure text label and we set the figure tooltip according to the statement type:

- For an if statement we use the if outcome, the variables involved in the condition and their values. Consider the example in Figure 9 where can be seen the branch that the execution took and the variable values that were involved in the if condition.
- For an assignment statement we use the value that was assigned to the left variable. Consider the example in Figure 10 where can be seen that the value 10 were assigned to the variable a during the execution.
- For an invoke statement we simply use the statement string representation.

Fig. 8. Eclipse IDE after the execution of an application using our plug-in.

Fig. 9. Eclipse IDE after the execution of an application using our plug-in showing the information regarding an if statement.

Fig. 10. Eclipse IDE after the execution of an application using our plug-in showing the information regarding an assignment statement.
Then, we associate the statement with the resulting Figure to have a mapping between our statement representation and the graphical element that represents the statement. Finally, we add mouse listeners to every figure to achieve the following functionality:

- When pressing the mouse button over an execution flow figure, we synchronize the eclipse editor with this figure: we open the file where that statement belongs and we highlight the line where it is in the source code editor (see Figures 10 and 9).
- When double clicking over an execution flow figure that represents an `invoke statement`, we expand the graph by merging the `main` method flow with the execution flow of the `invoked method` (see Figure 9). If we double click again on the same figure, the execution flow of the `invoked method` is collapsed.

After handling the execution flow graph, we represent the dependencies graph. The procedure to represent this graph is similar to the execution flow graph: for each local variable from the `main` method, we create a `RectangleFigure` with the variable name as its text label and tooltip. Then, we associate the local variable's representation with the corresponding figure, as we did for the statement figures. Next, we add a mouse listener to every figure to highlight the execution graph statements that are related with the clicked local variable's figure (as shown in Figure 11, regarding the variable `c`). Finally, for every dependency that the `main` method has stored, we connect the corresponding figures. Unlike the execution flow graph where we defined the position of every statement figure, for the dependencies graph we used the Java Universal Network/Graph (JUNG) Framework [42] to get the positions of every figure, avoiding implementing a positioning algorithm for the variable dependencies. This way we achieve a simple and easy positioning for the variables figures.

It is important to point that every dependency graph is synchronized with the execution flow graph:

- When the execution flow graph represents the `main` method, the dependency graph is related to it. For example, in Figure 10 the dependency graph is according to the `main` method.
- When the execution flow graph is expanded to represent a method call, the dependency graph corresponds to that method dependencies. For example, in Figure 9 the dependency graph is according to the `methodAux` method.

IV. Evaluation

The tool that we described in this paper automates the diagnosis phase of the debugging process providing a set of valuable information to the developers. To gather this valuable information we needed to use program instrumentation, which imposes a natural performance penalty. No one will adopt this solution if it is slow, dull, and not useful for the developers. Based on this statement, we present some metric criteria that we used to conduct the tool evaluation:

- **Performance**: observe the implementation impact on the overall performance. Check the penalty that is introduced by the instrumentation and execution of the instrumented classes. For this, we performed some tests using different programs to compare the runtime with and without instrumentation.
- **Scalability**: observe the impact of our solution with respect to the program dimension. Obviously, this aspect is closely related to the performance criteria, since the greater the programs are the longer will take to instrument, and consequently, the longer will take to execute them. To observe this impact we performed some tests using different programs with different dimensions.

A. Performance and Scalability Tests

One of the objectives of this paper was to accelerate the debugging process regarding the diagnosis phase, freeing the developers from having to navigate the source code and reexecute the program multiple times. Therefore, we were not interested in having a solution that was time-consuming, since the objective was to accelerate the process. If our solution were time-consuming the users would stop using it because they would spend much time just to apply it.

Thus, in this section we present the evaluation that we conducted to ascertain if our solution is in fact, time-consuming, or if it has an acceptable performance. Of course, it will have a penalty in terms of performance due the use of program instrumentation. Every program will always take more time to instrument and execute since we are injecting more statements to be executed.

First, we characterize the programs that we used. Then, we present the evaluation results.

B. Characterization of the Test Cases

It is important to characterize our test cases in a way that the reader might comprehend their context, scope, and representativeness. We intended to use real applications to perform this evaluation. However, our tool does not support threads yet, which is one major aspect of most real applications. Therefore, we developed every test case that we used, most of them composed by a single Java class using two little auxiliary classes just to instantiate them and invoke some methods during the execution.
1) **Multiple classes vs. Single class:** Regarding our implementation, instrumenting multiple classes consisting of a certain number of lines of code does not have much more impact than instrumenting only a single class that consists of the same number of lines of code. In order to make this statement we conducted four experiments with different program dimensions and they were performed comparing the instrumentation times for 22 class files against 3 class files:

- **Experiment A:** 22 class files with approximately 10355 LOC vs. 3 class files with approximately 10355 LOC.
- **Experiment B:** 22 class files with approximately 5310 LOC vs. 3 class files with approximately 5310 LOC.
- **Experiment C:** 22 class files with approximately 1250 LOC vs. 3 class files with approximately 11250 LOC.
- **Experiment D:** 22 class files with approximately 670 LOC vs. 3 class files with approximately 670 LOC.

The results are shown in section IV-C1. Based on those results, we choose to perform subsequent tests using just a single class as a matter of simplicity.

2) **Test Cases:** In order to evaluate our solution regarding its performance and scalability, we used four test cases to extract the instrumentation and execution times. Every test consists of three class files: 1 main class and 2 auxiliary classes. In each test case, the main class has different dimensions regarding its number of lines of code. The behavior associated with this main class is similar in every test case, and the behavior associated with the auxiliary classes is always the same.

   a) **Test case A:** The main class totals a number of 520 LOC. This class is composed of a main method and seven methods with similar behavior:
   - **Method Main Composition:** 117 statements, which 85 are method calls for the own class, 10 are invocations to the `System.out.println` method, 11 are variable assignments, 3 `if-else` statements, 2 invocations to the `substring` method, 3 cycles (2 `for` cycles and 1 `do-while` cycle, all nested), 1 new statement instantiating the Auxiliary Class A and 2 method invocations for the instantiated class.
   - **Remaining Methods Composition:** 7 methods each of which with around 26 statements which 9 are variable assignments, 3 `if-else` statements, 1 `for` cycle, 2 invocations to the `substring` method, 8 are invocations to the `System.out.println` method, 1 new statement instantiating the Auxiliary Class A and 2 method invocations for the instantiated class.

   b) **Remaining test cases B, C, and D:** The main class totals a number of 1070, 5240, and 10050 LOC, respectively. Every class follows the same method and statement pattern of the classes from test case A. They are composed by a method main and several methods (17, 98 and 191, respectively) with similar behavior:
   - **Method Main Composition:** 187, 117, and 202 statements, respectively. They are composed by several assignments, `if-else` statements, cycles and method invocations.
   - **Remaining Methods Composition:** 29, 29, and 33 statements, respectively. They have a similar composition and behavior to the methods that compose the classes from the test case A.

c) **Common to all test cases:** Two classes that are instantiated during every test case, both performing a total of 70 LOC:

- **Auxiliary Class A:** composed of two constructors, two methods, and one field. This class totals a number of 44 LOC and 16 statements: 7 assignments, 1 method invocation to the `System.out.println` method, 2 `if-else` statements, 2 `return` statement and 1 method call for the Auxiliary Class B.
- **Auxiliary Class B:** composed of two methods totaling a number of 30 LOC. This class has 16 statements: 4 assignments, 6 method invocations to the `System.out.println` method, 2 `if-else` statements, 2 new statements, and 2 invocations to the `substring` method.

C. Results

The experimental results are derived from the execution of the test cases that we described in the previous section and correspond to the average of five executions for every experimental test case. All the experiments were conducted in an Intel Core i5 machine at 2.4 Ghz, with 8 GB of RAM and running Mac OS X with the Eclipse IDE Version 4.3.2.

1) **Single Class vs. Multiple Classes:** As can be seen by the chart in the Figure 12 the difference between the instrumentation time of the 22 class files and the 3 class files is not significant. On average this difference is around 319 ms which is a quite low value. That said, we can state that this being such a small value would not impact on our tests. Therefore, and as we can extract from the above chart, the impact of instrumenting several classes against a single class with the same number of lines of code is not significant.

![Multiple Classes vs. Single Class](image)

Fig. 12. Multiple Classes vs. Single Class: Instrumentation time.

2) **Instrumentation Time vs. LOC:** As expected, we can observe an increase in the instrumentation time as the program size increases. The results are shown in the chart in Figure 13. The instrumentation time is almost directly proportional to the size of the application in terms of lines of code. Given the fact that the instrumentation is incremental - only instrumenting the necessary recompiled Java classes - the instrumentation cost in each development and debugging cycle is, for most programmers, very low, since we believe that the
number of classes that a developer changes before re-test his program is typically low. Also, developers already expect that the compilation takes longer when further changes are made.

3) Execution Time vs. LOC: As we can see by the chart in Figure 14, there is an increase in the execution time of the instrumented program regarding the execution of the program without instrumentation. This was expected since we are injecting many statements in the application’s code to log and extracting the information that we consider useful. The instrumented code suffers from a small penalty constant of about 280 ms. Beyond that small penalty, the execution times of the code follow approximately the ones of the instrumented source code as the program size increases. Furthermore, we consider that the additional time added by the instrumentation during the execution will be reimbursed by the reduction of the number of iterations during the debugging cycle.

D. Impact in the Development Cycle

Given the results that we obtained, we can be satisfied with the impact that our tool has on the development and debugging cycle. It would not be acceptable for a programmer to have to wait too long before beginning the debugging process. We consider that the additional time added by the instrumentation during execution is compensated by the reduction in the number of iterations in the development and debugging cycle. Thus, we believe that the impact that the tool has in terms of performance will ultimately be rewarded in terms of its usability and usefulness.

V. Conclusions

Nowadays all developers face a huge challenge when developing applications: software continues to be deployed with undiscovered flaws even after large investments on testing and debugging. Due to the time to market becoming shorter and shorter, debugging tasks cannot take up a higher percentage of the development time and cost. Therefore, the goal for every developer is to save time and costs during debugging. This paper presented a solution that makes an important contribution to reducing costs of debugging by changing the usual approach to debugging. Typically, debugging is cyclical because each time programmers want to explore a theory that explains a program error, they have to re-execute it, with or without a debugger. Using the proposed tool, after one execution, developers have access to all the information about the execution (executed path, explanation for that path, variables state) and they can rapidly navigate through the executed statements and have immediate access to the justification of every program variable.

We explained the architecture of our proposal and how it applies to the applications. We could see that our tool suffers from a small performance penalty. However, we believe that the impact that the tool has in terms of performance will ultimately be rewarded in terms of its usability and usefulness.

VI. Future Work

Given that the achieved results with our evaluation are quite acceptable; it is essential to perform a user testing evaluation to determine the usability and the usefulness of our tool. We believe that the impact that the tool has in terms of performance will ultimately be rewarded in terms of its usability and usefulness. Therefore, in the future we intend to perform the user testing using several buggy applications with an associated logic to contextualize the users.

Considering the limitations that our implementation has, it would be interesting to extend it to become more robust and wider. First, concurrency is one of the aspects that we intend to address. Being able to use and experiment our tool with real applications would be a great achievement. Another aspect that we must take into account in the future is the presentation of program cycles. Although our tool supports program cycles, their presentation is not done in an intuitive and simple manner. Therefore, we intend to improve the presentation of the program cycles. Also, we can extend our implementation so that it supports more types and more instructions, being able to handle a wider percentage of the Java Language.

REFERENCES
