Immediate Tracing

Semester project, SYSTEMF laboratory, EPFL

Valentin Aebi, IN-Ma2

Supervisors:
Clément Pit-Claudel,
Shardul Chiplunkar

Spring semester 2023

Abstract

This semester project report describes a prototype tracing tool that records the behavior of a Java program during its execution. It explains the two methods that were attempted to implement the tool, one using the Java debug interface, which gave poor results, the other using JVM bytecode instrumentation, which gave much better results. The tracer is able to record which lines are visited during the execution, what values the variables and fields have before and after the execution of each line and what values are given as arguments to functions or returned by them. We also describe a GUI to visualize the data collected during the execution.

Project proposal: `https://gitlab.epfl.ch/systemf/systemf-lab/-/blob/main/projects/2023%20-%20Loop%20backtraces.md`

GitHub repository: `https://github.com/epfl-systemf/JumboTrace`

1 Introduction

When a computer program does not behave as intended, or when one wants to deeply understand how an unfamiliar piece of code works, the most reasonable thing to do is usually to execute it in a debugger. This makes it possible to see which control flow paths are taken by the program, and what value the variables have at some point during the execution.

Furthermore, when a program crashes or is stopped in a debugger, it is usually possible to see the position of the control flow, and sometimes also the values of the variables, by looking at its stacktrace.

However, both of these solutions only display one state of the program, or at least one at a time. Yet, this state is not independent of the others: most of the time, the value of a variable depends on its previous value, as well as on the value of other variables or fields.

This project is an attempt to solve this problem by generating a detailed trace of the execution of a program. More precisely, the execution of the program should produce a trace that shows the lines of code that were executed along with the value of the variables read or written at these lines.

To this end, we describe and implement a tool, named JumboTrace, which allows generating such traces for the execution of Java programs. Due to the sequential nature of traces and since the goal is not to solve concurrency issues, we focus on single-threaded Java programs.

2 Architecture of the tracer

Conceptually, the tracer consists of two parts: a backend, which handles the execution of the Java program, and a GUI frontend, which displays the trace generated by the backend. The original plan was that the backend would be an executor for Java programs. It would periodically stop the program to save its state, and generate a trace with this information.

However, due to technical issues that are explained in the Implementation and results section of this report, this approach failed. The backend therefore consists of an instrumenter that takes Java class files and produces copies of them, augmented with instructions that save the state of the program. To reduce the amount of code injected into the modified classes (because it is challenging to avoid introducing bugs when injecting code in the form of individual bytecode instructions), it adds additional code to the program in the form of other class files. The so modified programs generate traces as JSON files. The instrumenter is written in Scala, while the additional classes are compiled Java code. This architecture is described in Figure 1.

The GUI is a Scala program that parses the JSON files and the source files of the program, and generates a displayable trace as an HTML file. As a few features of the UI are dynamic, the HTML file also includes Javascript code.

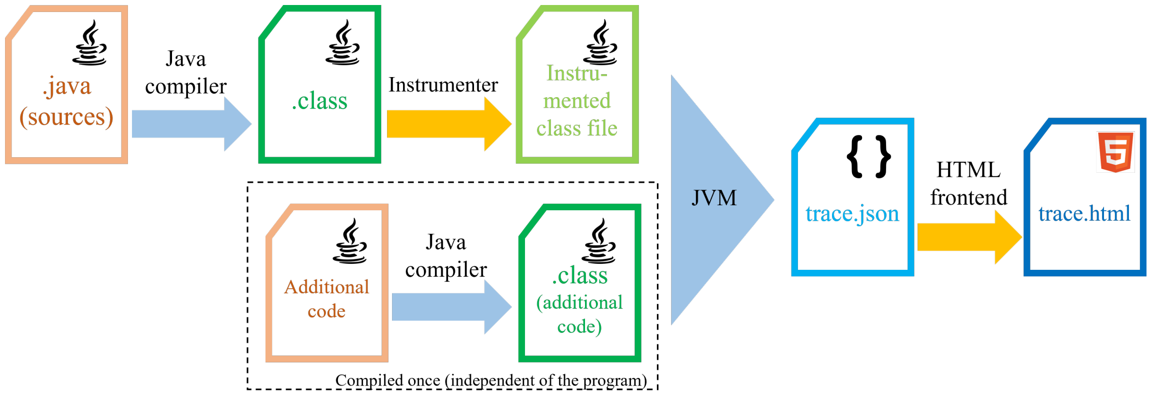


Figure 1: Architecture of the tracer. The orange arrows are the modules implemented in this project.

3 Timeline

Planned and actual timelines: figures 2 and 3.

The project was planned for one semester, i.e. 15 weeks including the Easter break. The first part of the semester (until week 5) was devoted to writing the project proposal, searching for

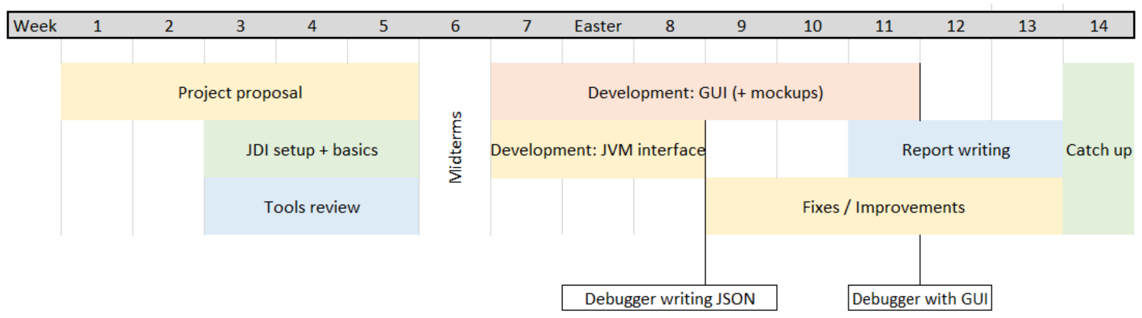


Figure 2: Planned timeline. The colors have no particular meaning. The two rectangles at the bottom are the programs that should be working at that point in time.

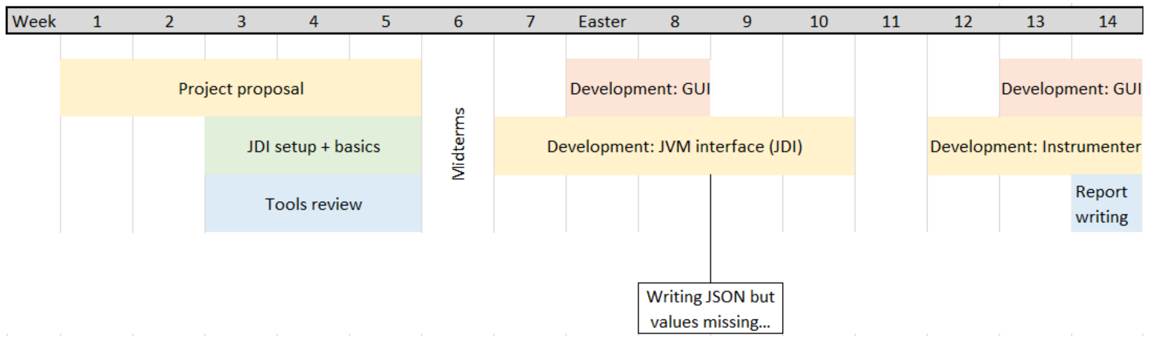


Figure 3: Actual timeline

existing projects that are similar to this one and, most importantly, understanding the basics of the Java Debug Interface (JDI), that we expected to be the basis of the project.

The second part, until week 8, consisted of implementing the first version of the tracer (both the backend and the corresponding GUI). At the end of week 8, the tool under development was able to generate and display traces. However, these traces were incomplete: some bugs were preventing the evaluation of some values (see the Implementation and results section for the details of these bugs). Until that point, the workflow was approximately following the planned timeline, except for these missing values, but the presence of minor bugs during the development was not surprising at all.

However, it turned out that these were not *minor* bugs. After spending some time trying to fix them between the Easter break and week 10, there were no improvements. Considering this, as well as the lack of documentation and the apparent fragility of the JDI, we decided to try to build a tracer with another approach, namely bytecode instrumentation.

The last part of the semester (weeks 12 to 14) was thus devoted to implementing the instrumentation approach. While it was expected that not everything would happen as initially planned, this new approach basically meant restarting the development process. As this of course took a lot of time, the writing of this report was postponed to the last week of the semester.

4 Implementation and results

4.1 JDI approach

(Corresponding code on the Github repository: `jdi` branch)

4.1.1 The Java debug interface

The Java Debug Interface (JDI) [7] is a high-level debugging framework that allows monitoring a running Java virtual machine, which we will refer to as the debuggee. A client application, usually a debugging tool, can request the JDI to send it notifications about what is happening in the debuggee. The JDI does so by observing the debuggee and sending events to the client when an action occurs that the client has requested to be notified of. For example, the client can send a breakpoint request for some line in the program, and every time the control flow of the debuggee reaches that line, the JDI will stop it and send a breakpoint event to the client. This event contains information like the identity of the thread on which the breakpoint was reached, as well as the value of the variables visible at that point. The execution of the debuggee is then suspended until the client instructs the JDI to restart it.

In order to include the information needed by the JDI, Java programs must be compiled with the debug flag enabled (`javac -g`). If the JDI encounters a class whose source file has been compiled without this flag, an `AbsentInformationException` is thrown and must be handled by the client.

4.1.2 JDI-based tracer

The first version of the tracing tool uses the JDI, and is implemented in Kotlin. At startup, it requests to be notified each time a class is loaded by the debuggee virtual machine. Then, when such a notification arrives and the class being loaded comes from a source file that was provided to the tracer, the tracer requests to be notified when a method of this class is entered or exited. Then, when a method is called (resp. returns), the tracer receives an event, and extracts the name of the method and its arguments (resp. return value). It then stores this information in a list of events.

We also want to record the lines that were visited. To do so, the first obvious solution is to use breakpoints. We could indeed set breakpoints at all lines and record the lines that are visited and the values of the variables. However, this only works when no request is made to monitor method calls and method returns. When events are requested simultaneously for breakpoints, method calls and method returns, some attempts to access the current stack frame (the JDI object containing visible variables and their values, among others) when the program is stopped on a breakpoint result in an `IncompatibleThreadStateException`. According to the code in the JDI that throws this exception, it is due to the fact that the thread on which the event happens has reached state `JDWP.Error.INVALID_THREAD`. The reason why this happens is unknown to us.

Another approach is to use steps instead of breakpoints. “Step” is a JDI command that continues the execution until the program flow moves to a new line. The principle here is to use this command to execute the program line by line, and at each line save the values of variables, like in the breakpoints solution. Unfortunately, this causes the same `IncompatibleThreadStateException` to be thrown. Several variants of this solution have been tried (e.g. by deleting the step request before processing the event and creating a new request afterward), without success.

This seems to indicate that the problem comes from the interaction between the requests for line visited (either breakpoints or steps) and the ones for method-related events (both method calls and returns). Therefore, another attempt was made that consisted of only saving the lines and variables, and using that information to infer calls and returns. In this setting, it is quite straightforward, at least in theory, to determine the values of the arguments that were given to a method: these are the values of the variables visible at the first line of a new method. However, this approach prevents the tracer from accessing the value returned by a method, which is usually provided by a method return event. Furthermore, the detection of a call or return should be based on information about the method contained in the stack frame accessed through the breakpoint event (or step event). Thus, the distinction between returns and calls should rely on the size of the stack. In the case where not all methods provide debug information (i.e. if their source file was compiled without the `-g` option), this can cause issues. And this happens quite often, as the methods of the Java standard library are not monitored by the tracer. For example, let a be a method compiled without debugging information, and f and g be methods compiled with debugging information. If a calls f and then g, and the stack size is larger at the time it calls g than when it calls f, the tracer may believe that g was called from f. Solving that problem would require analyzing the program itself (most probably the source files), and seems cumbersome, fragile and sensitive to edge cases.

The "best" version of the JDI-based tracer is the one that is on the `jdi` branch of the repository. It uses breakpoints and monitors method entries and exits. It evaluates variables, but replaces their value with an error message if this evaluation fails because of an exception thrown by the JDI. Catching the `IncompatibleThreadStateException` allows the program to continue its execution and thus to... throw other exceptions! The other kind of exception thrown by the tracer is `IllegalArgumentException` (with error message "frame method different than variable's method"), because the JDI detects that a frame comes from a method other than the current one. The exact mechanism by which this exception occurs is unknown to us, but it only occurs when trying to access the arguments of a method that has just been called, which seems to indicate that in some cases the first frame of the current method is not yet ready at the time the call event is processed. Interestingly, this is not always the case, as the arguments are sometimes evaluated correctly.

Also, the values missing in the trace are not always the same between two executions of the same debuggee. The fact that these bugs are non-deterministic seems to indicate that they are related to concurrency in the JVM. Unfortunately, I was not able to find documentation that would have allowed me to understand and solve the problem. A part of a trace obtained by running the tracer on the *Graphs* example from the repository is shown in Figure 4.

4.1.3 Comments on the issues with the JDI, and the decision not to use it

As explained in the previous paragraphs, the JDI caused a lot of problems in this project, some of which I did not manage to solve. While I do not claim that anyone else would have so many difficulties as I did trying to make it work, one of the reasons of this failure is the difficulty to find documentation on the interactions between the JDI and the JVM, and more precisely multithreading in the JVM. Relying on the JDI seems fragile in the sense that there are lots of ways of reaching edge cases that will cause it to throw exceptions. Also, it is possible that some of the problems encountered in this tracer are related to the heavy usage of the JDI: unlike a usual debugger where the user usually sets a few breakpoints, or does one step and then pauses the program for some time, the tracer repeatedly invokes the JDI and may cause several requests to be pending at the same time, which possibly leads the debuggee to become unstable. Finally, the JDI noticeably slows down the program being executed, which is one more reason to use another approach.

```
[52 (7)] CALL addEdge(from = 4,to = 3)
  [53 (52)] VISIT DirectedGraph.java:11 => { graph = [DirectedGraph
      -58: ( V = { 1, 2, 3, 4, 5 } ; E = { 1 -> 2, 2 -> 4, 2 -> 5, 4
      -> 3 } )] }
  [54 (52)] VISIT DirectedGraph.java:14 => { from = 3, to = 1 }
  [55 (52)] VISIT DirectedGraph.java:15 => <?? missing vars>
[56 (52)] EXIT addEdge --> return <void>
[57 (7)] VISIT Main.java:29 => { from = 3, to = 1 }
[58 (7)] CALL addEdge(from = <??>,to = <??>)
  [59 (58)] VISIT DirectedGraph.java:11 => { graph = [DirectedGraph
      -58: ( V = { 1, 2, 3, 4, 5 } ; E = { 1 -> 2, 2 -> 4, 2 -> 5, 3
      -> 1, 4 -> 3 } )] }
  [60 (58)] VISIT DirectedGraph.java:14 => { args = [], graph = [
      DirectedGraph-58: ( V = { 1, 2, 3, 4, 5 } ; E = { 1 -> 2, 2 ->
      4, 2 -> 5, 3 -> 1, 4 -> 3 } )] }
  [61 (58)] VISIT DirectedGraph.java:15 => <?? missing vars>
[62 (58)] EXIT addEdge --> return <void>
[63 (7)] VISIT Main.java:30 => <?? missing vars>
```

Figure 4: Part of the textual representation of the trace generated by the execution of the *Graphs* example, monitored by the tracer. The missing values are represented by the `<??>` and `<?? missing vars>` notations, the former being produced by an `IllegalArgumentException` on the evaluation of an argument and the latter by an `IncompatibleThreadStateException` when trying to access a stack frame. The numbers on the left of the events are the event identifier, followed by the identifier of its parent event (in parenthesis), i.e. the identifier of the call to the method executing it.

4.2 Code instrumentation approach

(Corresponding code on the GitHub repository: `main` branch)

The fact that the tool being designed is a tracer instead of a classical debugger makes it possible to use another approach than a debugging interface. Indeed, since no interaction with the user is needed at runtime, one can act on the program before it starts executing, instead of during execution. This could be done before compilation, by modifying the source files; during compilation, using a modified compiler; or after compilation, by modifying the bytecode. Assuming that the code is compiled with the `-g` flag to include debugging information in the bytecode, the simplest way to do that is probably to instrument the bytecode, as it requires little analysis.

4.2.1 Java ASM

Java ASM [2] is a bytecode manipulation library for JVM bytecode. Following the visitor pattern, it allows its users to define vistors that traverse a class file to analyze and/or modify it. More precisely, to modify a method, one has to define a subclass of `ClassVisitor`, which will override the `visitMethod` method to let it return an instance of a user-defined `MethodVisitor`. This `MethodVisitor` in turn overrides methods corresponding to the changes that need to be performed. For example, if we want to replace all calls to a static method named "foo" by calls to another static method named "bar", we will override `visitMethodInsn` in the `MethodVisitor` as follows:

```
public void visitMethodInsn(int opcode, String owner, String name,
        String descriptor, boolean isInterface) {
    /* INVOKESTATIC is the opcode for calls to static methods */
    var newName =
        (name == "foo" && opcode == INVOKESTATIC)
        ? "bar" : name;
    /* By default, the methods in the superclass copy the bytecode
       without any modification. Here we simply replace the name if
       needed */
    super.visitMethodInsn(opcode, owner, newName, descriptor,
       isInterface);
}
```

4.2.2 Recorded events

The events recorded by the tracer, implemented as the `TraceElement` algebraic data type in the `traceElements` package, are the following:

- `LineVisited`: records the class name and line number of a line that has been executed by the program, as well as the other events taking place during this execution of the line (e.g. `VarSet`);
- `VarSet`: assignment to a variable, recording the identifier of the variable and the value assigned to it;
- `VarGet`: read access to a variable, also recording the identifier and value of the variable;
- `ArrayElemSet`: assignment of an array element, recording the array, the index of the updated element and the value;

- `ArrayElemGet`: symmetric to `ArrayElemSet`;
- `StaticFieldSet` and `StaticFieldGet`: assignment/access to a static field of a class;
- `InstanceFieldSet` and `InstanceFieldGet`: assignment/access to a field of an object;
- `Return`: method return with return value, records the name of the method and the return value;
- `ReturnVoid`: method return without return value, records the name of the method exiting;
- `MethodCalled`: beginning of the execution of a method. Records the name of the method, the class it belongs to, the arguments passed to it, a flag indicating whether the method is static or not, and all the events that happen during the execution of the method;
- `Initialization`: the very first element of the trace, containing the date and time at which the execution starts (more precisely at which the `___JumboTracer___` class (see below) is loaded);
- `Termination`: the very last element of the trace, containing a message explaining whether the program exited normally or with an exception.

4.2.3 The `___JumboTracer___` class

The `___JumboTracer___` class (in the `injected` directory) is added to the program before its execution starts and performs most of the work regarding the recording of the events. It maintains a list of events in a static field, and adds the events to this list when they happen in the program. The role of the instrumentation is therefore "only" to call the method of `___JumboTracer___` that corresponds to the event taking place in the program.

One technical issue is that most of these methods take values generated by the program as arguments. E.g. the `variableSet` method of `___JumboTracer___`, which records a write to a variable, takes as arguments the identifier of the variable and its value. The value can be both a primitive or an object. In a usual situation, Java would handle this case by boxing the primitive values automatically, and a method that takes an `Object` as an argument could fit any kind of value. However, this is not a usual situation, in the sense that the method will be called by code generated by the instrumentation, which means that the instrumenter would have to explicitly box the values. In order to reduce the amount of injected code and improve performance, it is preferable to provide specialized versions of each of these methods. However, Java has 8 primitive types. Along with the `Object` type that handles all references, this implies that each method should be replicated 9 times, changing only the type of one parameter. This is a lot of code and thus implies a high risk of introducing a bug. The safest solution I could think of to solve this problem is to use macros. Unfortunately, Java has no macros, so the file includes C-like macros that should be expanded using the preprocessor of C before running the Java compiler. Despite being a hack, I think that this is the best solution since it drastically reduces the amount of code that has to be written and guarantees that the code is exactly the same for each type, which helps prevent difficult-to-spot bugs that occur with only one type.

4.2.4 Bytecode instrumentation

As explained above, the methods of `___JumboTracer___` are called by code that is injected into the class files of the program. To achieve this result, the program is first traversed by the `ClassExplorer` and `MethodExplorer` classes of the `instrumenter` package. This "analysis pass" collects all the information needed to instrument the code. The instrumentation is then done by the `MethodTransformer` class, which seeks the instructions performing actions that need to be recorded (variable/field write/read, method return, etc.) and adds instructions to call the methods of `___JumboTracer___`, after having loaded their arguments on the stack. For example, an `ISTORE` instruction, that writes an integer to a variable, is replaced by the following sequence of instructions:

```
DUP                   // duplicate top stack element
LDC <variable name>   // load a constant
SWAP
INVOKE_STATIC ___JumboTracer___.variableSet(String,int)->void
ISTORE
```

The name of the variable is obtained from the data collected by the "analysis pass" performed before the beginning of the instrumentation. This code basically duplicates the value of the integer and passes one of the copies to `variableSet`, along with the name of the variable, which is known at the time of instrumentation and thus loaded as a constant (using `LDC`). Note that a slight improvement would be to swap the arguments of the `variableSet` method in order to get rid of the `SWAP` instruction, which has not been done here.

Other store instructions (*x*`STORE`, where *x* is a letter that depends on the type of the value it stores) are of course instrumented similarly. The same holds for accesses to variables (*x*`LOAD`) and reads and writes to arrays (*x*`ALOAD`, *x*`ASTORE`). Accesses to and modifications of field values (`PUTFIELD`, `GETFIELD`, `PUTSTATIC`, `GETSTATIC`) work similarly.

Array stores use a different approach: as it would be complicated to duplicate all 3 arguments to the store (the array, the index and the value), the instrumentation replaces the store instruction with a call to a method of `___JumboTracer___` that saves the event and preforms the array write itself. For array loads, the injected code only takes as arguments the array and the index, and loads the value itself before saving it.

The recording of method calls is a bit more complicated. Instructions are inserted at the beginning of each method to save the arguments using an *x*`LOAD` of the variable containing the argument. They are saved using the `saveArgument` method of `___JumboTracer___`. Finally, when all arguments have been loaded, `terminateMethodCall` is called by the instrumentation to write an event containing the values of the arguments and the name of the method.

Method returns work in the same way as loads and stores: the return value is duplicated and then a method is called to store it. The case of returning `void` is treated separately since here we do not even need to save any value from the program.

Finally, the code of the main method is wrapped in a `try-finally`, the `finally` containing instructions that call the method of `___JumboTracer___` that writes the saved events to a JSON file. That way, if the program crashes with an exception, these instructions will be executed and the JSON file will be written. Also, any return in the main method is preceded by similar instructions that trigger the writing of the JSON file, so that non-exceptional executions also produce a trace file.

4.2.5 Limitations and assumptions

Due to the way instrumentation is implemented, the tool has a few limitations, and therefore the following assumptions must hold on the instrumented program for the process to work correctly:

- The first limitation, and probably the most problematic one, is that the instrumented program should *never* exit through a call to `System.exit` (either directly or by calling methods that call `System.exit`). If it does so, the instructions that cause the JSON file to be written will not be executed and no trace will be produced. A straightforward solution would be to inject these instructions before any call to `System.exit`, but this would not work if `System.exit` is called from a non-instrumented method, itself called from the instrumented program. A better solution would probably be to trigger the writing of the JSON file from the finalizer of the main class, as suggested by my supervisors. But this has not been attempted yet.
- The second limitation is that the `toString` method of any object used in the program should not have side effects. The code that saves events in `___JumboTracer___` often calls `toString` to save only a textual description of the object, so having side-effects in this method could cause the tracer to modify the program behavior.
- The third limitation is rather straightforward: to avoid a name clash, the instrumented program should not define or use a class named `___JumboTracer___`.

Finally, only events happening in instrumented methods are recorded. In particular, calls to methods from the Java standard library are not recorded because the `MethodCalled` event is generated by the callee, which in this case is not an instrumented method.

4.2.6 ASM DSL

While Java ASM greatly facilitates the modification of Java bytecode, it still has a few drawbacks. Firstly, using it usually means writing long sequences of calls to methods that add new instructions to a visitor. These methods often take a lot of arguments, and the result can be difficult to read. Secondly, and more importantly, it does not completely handle the distinction between single-slot and double-slot values. Indeed, on the JVM, most values, when pushed to the stack, take only one stack slot. But doubles and longs need two slots. As a result, the same operation may have to be implemented with different instructions depending on which types of values are on the stack. E.g., `DUP`, that duplicates the topmost stack element, works fine with integers, floats, etc., but needs to be changed to `DUP2` when the element to duplicate is a double or a long.

To mitigate these issues, the transformation code uses a small "DSL", defined in the `AsmDsl` object. It consists of slightly higher-level "assembly" instructions that take as arguments the types of the stack values that they have to manipulate and use these values to decide which actual JVM bytecode instruction(s) to use. It is important to note here that using these instructions sometimes leads to unnecessarily long sequences of actual ASM instructions like `DUP`, `DUP_X1` (duplicate the top stack element and place the result below the second element), `SWAP`, etc., because they are generated at a higher level of abstraction.

5 GUI

The GUI is fairly simple. The `javaHtmlFrontend` package is a program that parses the Java source files and the JSON trace and produces an HTML page displaying the trace. HTML code is generated by the j2html library [4]. Java source code is parsed using the Javaparser library [6], and the JSON trace file is parsed by Play Json [9].

The GUI displays line visited events as lines of code. When the string representation of the value of a variable appearing in a line is not too long (i.e. no more than a few characters), it inserts this value into the line of code, next to the variable name. Clicking on such a line reveals all the variables and their values. These are obtained from the events like `VarSet`, `FieldGet`, etc. that occured at that line. E.g. if the event is a variable `x` set to `42`, it will display `x := 42`. If the variable is read instead of written, it will be displayed as `x:42`. Similarly, clicking on a method call reveals the events that happened during the execution of the method.

The generated HTML page has very few dynamic features. It is able to expand or collapse all method calls at the press of a button. When the mouse is on a line of code, it also highlights the parent calls of this line visited event (see Figure 5). This is implemented as a few JavaScript functions. As this script is short, it is simply hardcoded as a string in the Scala code generating the HTML page, for simplicity.

6 Further work

JumboTrace is currently a prototype. In particular, it has almost no automated test, which is a problem. Testing the instrumenter should probably be done by comparing the trace it generates with a hand-crafted trace. The `debugCmdlineFrontend` module, which essentially performs basic formatting on a trace would probably be helpful for such tests. The purpose of the `PrintStream` passed as an argument to the formatting methods of this module was exactly to be able to inject a mock stream for testing, even though this has not been done yet because of time constraints.

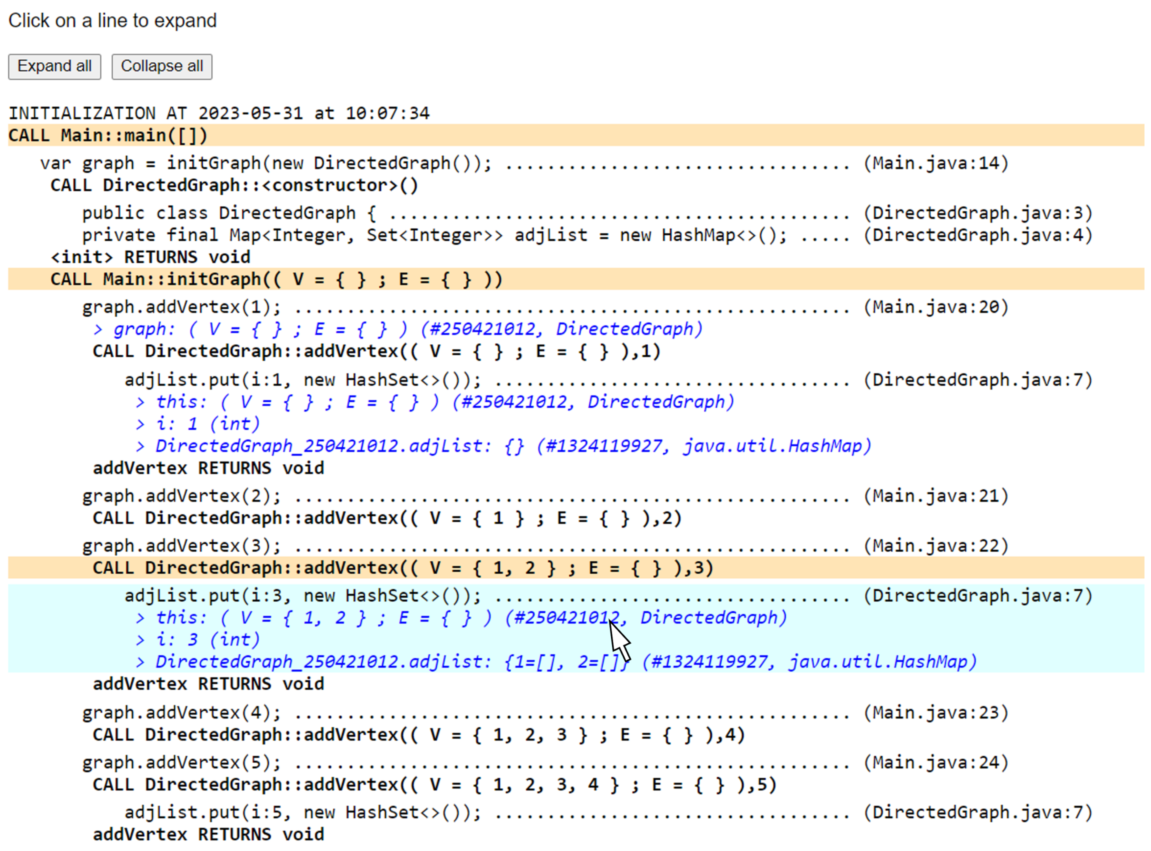
Apart from the correctness of the tracing, the usefulness of the concept should also be evaluated. It would be interesting to see how this tool can be used to debug or understand a program, or to teach programming to beginners. A good first step would likely be to test the tool on programs slightly larger than the ones we used until now.

The GUI could be improved in lots of ways. The values displayed in the HTML page could be made easier to read. A possibly useful feature to add would be a search system that would be able to answer queries like “go to a call to the method named ‘foo’ where its argument ‘x’ has value 1”.

Another useful feature would be to save not only the values of the fields and variables, but also those of the expressions evaluated during the execution. This would probably require modifying the program before or during compilation instead of instrumenting the bytecode, and the size of the traces generated by such a program would probably be truly gigantic.

Furthermore, the bytecode instrumenter should be able to handle not only Java programs, but also programs resulting from the compilation of other JVM languages. In particular, I made some quick tests with Scala (see the *Geom2D* example on the repository), and it seemed to work, even though Scala traces would probably contain a lot of closures passed as values, and thus be difficult to read if not handled properly. But it looks like implementing a module to display Scala traces in an HTML file, analogously to the `javaHtmlFrontend`, would be doable. Kotlin would possibly be worth a try too.

Last but not least, getting a better understanding of the JDI-related problems would be a good thing.


Click on a line to expand

Expand all Collapse all

```
INITIALIZATION AT 2023-05-31 at 10:07:34
CALL Main::main([])
   var graph = initGraph(new DirectedGraph()); ................................ (Main.java:14)
    CALL DirectedGraph::<constructor>()
       public class DirectedGraph { ........................................... (DirectedGraph.java:3)
       private final Map<Integer, Set<Integer>> adjList = new HashMap<>(); ..... (DirectedGraph.java:4)
    <init> RETURNS void
    CALL Main::initGraph(( V = { } ; E = { } ))
       graph.addVertex(1); .................................................... (Main.java:20)
        > graph: ( V = { } ; E = { } ) (#250421012, DirectedGraph)
        CALL DirectedGraph::addVertex(( V = { } ; E = { } ),1)
           adjList.put(i:1, new HashSet<>()); ................................. (DirectedGraph.java:7)
            > this: ( V = { } ; E = { } ) (#250421012, DirectedGraph)
            > i: 1 (int)
            > DirectedGraph_250421012.adjList: {} (#1324119927, java.util.HashMap)
        addVertex RETURNS void
       graph.addVertex(2); .................................................... (Main.java:21)
        CALL DirectedGraph::addVertex(( V = { 1 } ; E = { } ),2)
       graph.addVertex(3); .................................................... (Main.java:22)
        CALL DirectedGraph::addVertex(( V = { 1, 2 } ; E = { } ),3)
           adjList.put(i:3, new HashSet<>()); ................................. (DirectedGraph.java:7)
            > this: ( V = { 1, 2 } ; E = { } ) (#250421012, DirectedGraph)
            > i: 3 (int)
            > DirectedGraph_250421012.adjList: {1=[], 2=[]} (#1324119927, java.util.HashMap)
        addVertex RETURNS void
       graph.addVertex(4); .................................................... (Main.java:23)
        CALL DirectedGraph::addVertex(( V = { 1, 2, 3 } ; E = { } ),4)
       graph.addVertex(5); .................................................... (Main.java:24)
        CALL DirectedGraph::addVertex(( V = { 1, 2, 3, 4 } ; E = { } ),5)
           adjList.put(i:5, new HashSet<>()); ................................. (DirectedGraph.java:7)
        addVertex RETURNS void
```

Figure 5: Screenshot of the GUI for the trace generated for the *Graphs* example (the mouse is artificial but its position corresponds to the highlighting). Not all events are expanded. At that point of the program, a graph is being initialized by the `initGraph` method. The part of the trace displayed here shows how the vertices set of the graph grows as the `addVertex` method is called repeatedly.

7 Related work

It seems that not much prior work exists on tracers similar to the one described in this report. However, debugging and program visualization have been the subject of numerous projects. One of those closest to this one is PythonTutor [3], especially its implementation for Java [5]. It is an online tool that displays the state of the stack and the values of the variables during the execution of a program. It allows visualizing the execution line-by-line and moving to the previous or next lines to see how the program state evolves. Interestingly, this tool seems to successfully use the JDI in a similar way as what I tried to do. However, the setup of the environment to execute this tool seems quite complicated, and the online tool is restricted to Java 8.

Another related tool is REMV (Repeatedly-Executed-Method Viewer) [8], a tool that also uses bytecode instrumentation to record what happens in a Java program and provides a GUI to visualize the execution. In particular, it focuses on comparing distinct executions of the same method and allows displaying in parallel several paths taken by the control flow at different times during the execution.

Finally, the visual tracing plugin for Eclipse described in [1] uses disabled JDI breakpoints to record what its authors call *silent hits*, and record information about the state of a program without stopping its execution. The tool also allows recording hits on watchpoints (a kind of breakpoints triggered when an object or variable is accessed and/or updated), and is able to display the history of modifications of a variable, which, among other benefits, makes it possible to visualize race conditions.

8 Conclusion

This project mostly consisted of searching for ways of recording what happens in a Java program during its execution, in order to be able to display its state and modifications as a readable trace. A first attempt using the Java debug interface did not work as expected at all. A second attempt using bytecode instrumentation, which at first seemed much more complicated than using a debugging interface, turned out to give better results. It is able to record the history of the execution of a Java program, which helps understand how a program works, or why it does not behave as intended. It is also able to display this history in a GUI that makes it reasonably easy to read. The tool has limitations, but it seems reasonable to expect that further work on this topic can get rid of them and produce easy-to-use tools that perform advanced analyses of the traces and facilitate the understanding of program behaviors.

A Appendix: Project proposal

A.1 Abstract

This project proposal describes a debugging tool that displays an interactive trace of the execution of a program. The trace is presented as an indented list of program statements, with the value of the variables when the statement was executed shown inline.

A.2 Problem

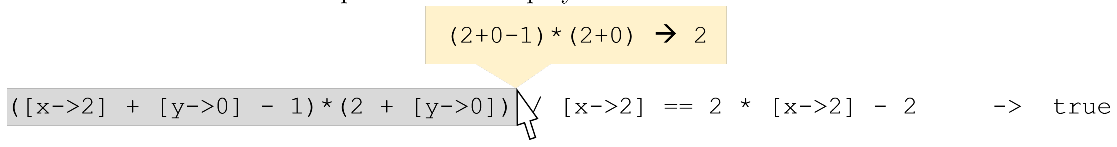
When a computer program does not behave as intended, or when one wants to deeply understand how an unfamiliar program works, the most reasonable thing to do is usually to run the program in a debugger. This makes it possible to see how data is propagated across the functions and data structures, and possibly to spot the place where the actual behavior starts deviating from the intended one.

Furthermore, when a program crashes because of a bug, it usually tries to help the programmer find out what went wrong by displaying stacktraces, showing the sequence of function calls that lead to the crash.

However, even using these techniques, debugging can be very difficult on some occasions. Stacktraces usually show nothing more (and often less) than the values of variables at the time of the crash. A debugger usually allows displaying the values of variables during the execution of the program, but it only focuses on a small part of the program at a time, instead of giving an overview of its execution.

A.3 Solution outline

The goal of this project would be to design a debugging tool that, given a program to execute, displays the execution of each line of the program. More precisely, for each execution of a line, it should display the line along with the value of each variable. These lines should be indented to make the resulting execution trace more readable (see Example section below). It would consist of a command-line program (alternatively: an IDE plugin) that takes as an argument the program to execute, executes it, and displays the resulting trace in a GUI. Ideally, the GUI should be interactive and allow the user to select an expression and display its value:



```
(2+0-1)*(2+0) → 2

([x->2] + [y->0] - 1)*(2 + [y->0]) \/ [x->2] == 2 * [x->2] - 2     ->  true
```

Ideally again, it should include a search function that finds points in the program where some event happened. For example, given a function ‘def foo(i: Int, j: Int, k: Int)‘ and assuming that we suspect an error to occur when ‘i=2‘ and ‘j=5‘, it should be possible to search for points in the program where ‘foo‘ has been called with these values of the arguments. Finally, it should be possible to highlight the parent stack frames of a selected line (i.e. the lines where the function calls and/or loop statements enclosing the selected line were executed).

A.4 Example

An implementation of bubble sort in Scala is used as an example, but the exact language for which the debugger will be designed remains to be determined.

A.4.1 Source code

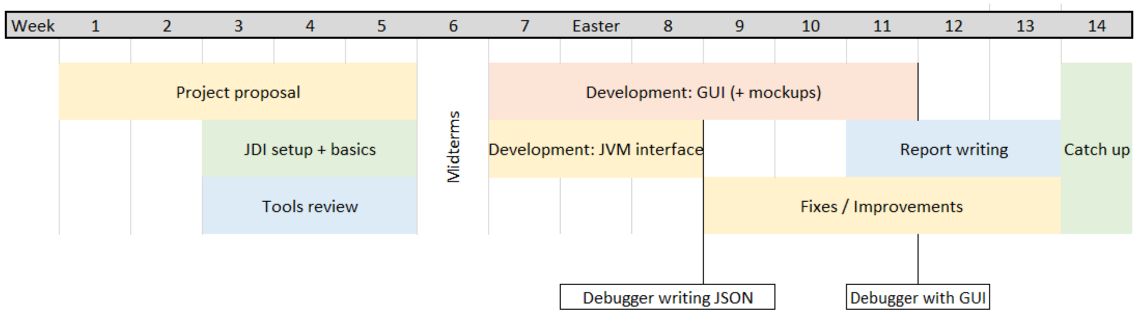
```
def swap(a: Array[Int], i: Int, j: Int): Unit = {
  val tmp = a(i)
  a(i) = a(j)
  a(j) = tmp
}

def bubbleSort(a: Array[Int]): Unit = {
  var continue = true
  while (continue) {
    continue = false
    for i <- a.indices.dropRight(1) do {
      if (a(i) > a(i+1)) {
        swap(a, i, i+1)
        continue = true
      }
    }
  }
}

def main(args: Array[String]): Unit = {
  val array = Array(10, 15, 12, 2, 7, 19, 4, 3, 12, 21, 0, 3)
  bubbleSort(array)
  println(array.mkString("[", ", ", "]"))
}
```

A.4.2 Trace

```
CALL main(args=[]) {
  val array = Array(10, 15, 12, 2, 7, 19, 4, 3, 12, 21, 0, 3)
  CALL bubbleSort(a-->[10,15,12,2,7,19,4,3,12,21,0,3]) {
    var continue = true
    // start while loop: while (continue)
    while continue-->true {
      while (continue-->true) {
        continue = false
        // start for loop: for i <- a.indices.dropRight(1)-->Range 0 to 10
        for i-->0 <- a.indices.dropRight(1) {
          if [a(i-->0)-->10] > [a(i+1-->1)-->15]-->false
        }
```

```
for i-->1 <- a.indices.dropRight(1) {
  if [a(i-->1)-->15] > [a(i+1-->2)-->12]-->true {
    CALL swap(a-->[10,15,12,2,7,19,4,3,12,21,0,3], i-->1, j-->2) {
      val tmp = a(i-->1)-->15
      a(i) = a(j-->2)-->12
      a(j) = tmp-->15
      RETURN () from swap
    }
    continue = true
  }
}
for i-->2 <- a.indices.dropRight(1) {
  if [a(i-->2)-->15] > [a(i+1-->3)-->2]-->true {
    CALL swap(a-->[10,12,15,2,7,19,4,3,12,21,0,3], i-->2, j-->3) {
      val tmp = a(i-->2)-->15
      a(i) = a(j-->3)-->2
      a(j) = tmp-->15
      RETURN () from swap
    }
    continue = true
  }
}
for i-->3 <- a.indices.dropRight(1) {
  if [a(i-->3)-->15] > [a(i+1-->4)-->7]-->true {
    CALL swap(a-->[10,12,2,15,7,19,4,3,12,21,0,3], i-->3, j-->4) {
      val tmp = a(i-->3)-->15
      a(i) = a(j-->4)-->7
      a(j) = tmp-->15
      RETURN () from swap
    }
    continue = true
  }
}
for i-->4 <- a.indices.dropRight(1) {
  if [a(i-->4)-->15] > [a(i+1-->5)-->19]-->false
}
for i-->5 <- a.indices.dropRight(1) {
  if [a(i-->5)-->19] > [a(i+1-->6)-->4]-->true {
    CALL swap(a-->[10,12,2,7,15,19,4,3,12,21,0,3], i-->5, j-->6) {
      ...
```

A.5 Timeline


Week
1
2
3
4
5
6
7
Easter
8
9
10
11
12
13
14
Project proposal
JDI setup + basics
Tools review
Midterms
Development: GUI (+ mockups)
Development: JVM interface
Report writing
Fixes / Improvements
Catch up
Debugger writing JSON
Debugger with GUI

References

[1] Bilal Alsallakh, Peter Bodesinsky, Alexander Gruber, and Silvia Miksch. Visual Tracing for the Eclipse Java Debugger. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 545–548, 2012.

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. 01 2002.

[3] Philip J. Guo. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery.

[4] J2html JSON parsing library. https://j2html.com/.

[5] Java jail backend for the Java version of the Python tutor. https://github.com/daveagp/java_jail.

[6] Javaparser Java parsing library. https://javaparser.org/.

[7] Documentation of the Java Debug Interface. https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/.

[8] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-Executed-Method Viewer for Efficient Visualization of Execution Paths and States in Java. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 253–257, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Play JSON library. https://github.com/playframework/play-json.