

École Polytechnique Fédérale de Lausanne

Immediate tracing
for smoother debugging and code exploration

Erwan Serandour

Master Thesis

Approved by the Examining Committee:

Prof. Clément Pit-Claudel

Thesis Advisor

Dr. Michel Schinz

External Expert

Shardul Chiplunkar

Thesis Supervisor

EPFL IC IINFCOM SYSTEMF

INN 316 (Bâtiment INN)

Station 14

CH-1015 Lausanne

June 30, 2024

Abstract

By recording each event that occurs in a program, its complete execution trace, it is possible to implement a graphical interface to interactively explore the program's execution. With the event log, it becomes easier to observe events spread over the entire execution of a program. We provide PrintWizard, an implementation of this idea that demonstrates the usability and feasibility of the concept.

In this project, we claim two contributions: a flexible event log that captures complete information about a program's execution, and a graphical interface to interactively explore the event trace.

GitHub repository: <https://github.com/thurgarion2/PrintWizard>

Contents

Abstract	2
1 Introduction	4
2 Demo of PrintWizard	8
3 Architecture	12
3.1 Instrumentation module	12
3.2 Logging module	13
3.3 Object data store	13
3.4 Source code formatting module	14
3.5 Display module	14
3.5.1 statement	15
3.5.2 sub-statement	15
3.5.3 flow control	16
4 Roadmap to deploy PrintWizard	17
4.1 Improving testing infrastructure	17
4.2 Fixing the issue with poor event formatting	17
4.3 Support external library in the object store	18
5 Related Work	19
5.1 Time traveling[12]	19
5.1.1 Record and replay	19
5.1.2 Omniscient debuggers	20
5.2 Snoop	20
5.3 Valentin and Baptiste work on event log based debugging	21
6 Conclusion	22
Bibliography	23

Chapter 1

Introduction

Observing the behavior of a program is a common practice when debugging or exploring a code base. In many cases, it is not enough to just read the code; it is also necessary to observe what happens during the actual execution of the program. This project is the result of frustration with the set of tools available when observing events spread over the entire execution of a program. In this context, an event is anything that happens during program execution. In our opinion, the solution to this frustration is a tool that records all events that occur during program execution and provides a graphical user interface to interactively explore the event trace. The rest of this report describes PrintWizard, a tool that aims to provide the above capabilities and to convince you that this is a reasonable opinion.

To observe the behavior of a program, there are two widely used approaches:

- Use an interactive debugger. A typical interactive debugger[24a] provides the following features: step through a program, stop at a specific event, pause a program, and explore its current state. An interactive debugger is excellent for observing a particular state of a program. However, the most difficult part is usually locating the error, and when the information about the error is distributed throughout the execution of the program, interactive debuggers face difficulties.
- Logging[Zel09] involves inserting logging statements into the source code to collect a record of the events that occurred during the execution of the program. Logging is excellent when we want to observe a few events. But to observe a large number of events, it quickly becomes impractical. This requires significant modifications to the source code, and exploring the logs quickly becomes complicated.

We illustrate our claims with two examples. In each example, logging and the interactive debugger have different shortcomings. In the first example, program A prints lines to a file used by program B.

Program B reports a problem at line 11 of the file. Identifying the problem should be straightforward. However, with the tools currently available, the following options are available:

- Find all the print calls in program A. There are 25, and a quick review of each one is not enough to identify the culprit.
- Using an interactive debugger, set 25 breakpoints and check them until line 11 is printed. In this case, we don't know which breakpoint is relevant, and even if we use conditional breakpoints, using the interactive debugger will be a pain.
- For each print statement, add a logging statement that specifies its line and the line it prints. We are in a situation where we want to observe a large number of events, and it would clearly be impractical to modify the source code with the 25 logging statements.

The second example is a simulation of a flock of boids[Rey87]. A flock of boids imitate the behavior of a flock of real birds using a set of simple rules. The state of the simulation is described by the velocity and position of each boid. The rules are forces that the boids apply to each other, such as an avoidance force to prevent collisions. At each step of the simulation, the position and velocity of the boids are updated using Newtonian mechanics. In our implementation, the boids are immutable. The state of the simulation is represented as a list of boids, and at each step the state is updated with a tick function. While running the simulation, we notice that under certain initial conditions the boids start disappearing from the screen a few seconds after the simulation starts.

In the absence of a good hypothesis about the origin of the problem, the best solution is to observe the evolution of the positions of the boids during the simulation. The following options are available:

- Using an interactive debugger, place a breakpoint at the end of the tick function and observe the position of each boid. Even if you can narrow the problem down to 10 boids and the bug only occurs after 300 steps of the simulation, you will have to click on the step functions 300 times. Using an interactive debugger will be a pain.
- Log the positions of the boids at the end of each simulation step. Assuming that we have narrowed down the problem to 10 boids and the problem occurs after 300 steps, it is reasonable to inspect the 300 lists of positions. However, the log only provides information about the positions of the boids. Now suppose we want to understand why a boid has a particular position; it will be necessary to create a new set of logging statements.

In the two examples above, we're interested in a set of events spread over the entire execution of the program. For each example, both Logging and the interactive debugger have different shortcomings. Logging works well when we are interested in a limited set of events, but scales difficulty. And it is difficult to find the location of a bug with an interactive debugger when information about it is spread throughout the execution of the program.



Figure 1.1: A flock of boids

A logger and an interactive debugger each make different trade-offs. In the context of debugging and exploring programs, we believe that the best way to observe a set of events spread over the execution of the program is to make different choices. We identify the following parameters that characterize different debugging tools: the tool's overhead on program execution, the information it records and if it provides interactive exploration.

An interactive debugger has low overhead in most cases, does not record information about the program, and provides interactive exploration. In our opinion, interactive debuggers are geared toward exploring the specific state of a program.

Loggers have low overhead, collect limited information about program execution, and do not provide interactive exploration. Because they are used in production, loggers are optimized for performance, which results in limited information available, event logs that are not always easy to explore, the inability to inspect object data, and the decision of what to include is made at a low granularity.

When observing a set of events spread over the execution of a program during debugging, we care about usability and to have access to the relevant events. The tool we developed in this project, PrintWizard, aims to satisfy the above constraints. It does so by making the following choices: high overhead, recording all information about the program's execution, recording information about object data, and providing a graphical interface to interactively explore the trace log. PrintWizard is divided into two parts: a backend that collects information about the program execution and a graphical interface that provides interactive exploration of the event log.

In this project, we claim the following contributions:

- A flexible event log that captures comprehensive information about program execution and information to aid in debugging. In the rest of the report, we also refer to the event log as an execution trace. The event log contains two categories of program events : Execution steps and event groups. Execution steps are events that occur during program execution. They capture information at the language level in our case java and not the assembly level. Event groups capture information about the organization of the program and allow better exploration of the trace. Event groups necessarily form a tree. The format is not limited to Java and should be easily adaptable to support other languages such as Python.
- A graphical interface for interactive exploration of the event record. The interface uses event group information to aid navigation. Each event group can be expanded and collapsed, allowing the user to hide irrelevant items and quickly navigate to the relevant part of the program execution. Ease of use depends on the right choice of event group. After some experimentation, we came up with a reasonable result given the use cases considered.

Chapter 2

Demo of PrintWizard

Our tool, PrintWizard, addresses the problems described in the previous section. In this part, we will use PrintWizard to find the location of the bug in the boid simulation. Notice that it is possible to understand what is happening without being familiar with the source code. As a reminder, the flock of boids tries to imitate the behavior of a flock of real birds using Newtonian mechanics. Under certain initial conditions, the boids begin to disappear from the screen a few seconds after the simulation starts, even though they should remain within the screen boundaries. To use PrintWizard, the necessary steps are: compile the program with our Java compiler plugin, run the program, and load the execution trace into the graphical user interface.

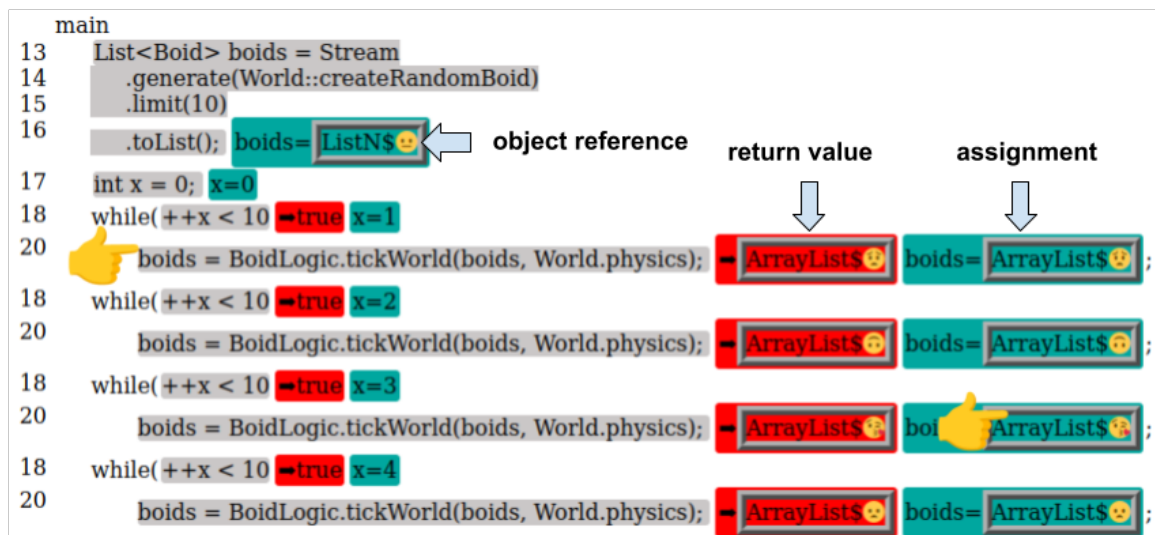


Figure 2.1: Initial GUI screen

The image above is the initial screen you see when you open the GUI. First, a brief description of the screen. The hand indicates that we can click on these elements. We can click on the assignment to expand it and see the events happening inside. We can click on the ArrayList to inspect its data. In red is the value returned by an expression, and in blue are the assignments that occur inside. An object reference is represented by its class name and an emoji identifying its pointer. Events within the body of a loop are indented to reflect a group of events.

The result is close to the source code. We don't need to be familiar with it to understand what is going on. The display clearly shows that ten boids are randomly initialized and that the tickWorld function is used at each iteration to update the state of the boids. If you remember, we're interested in the position of the boids. Let's click on the ArrayList element to see the state of the boids after the tenth iteration step.

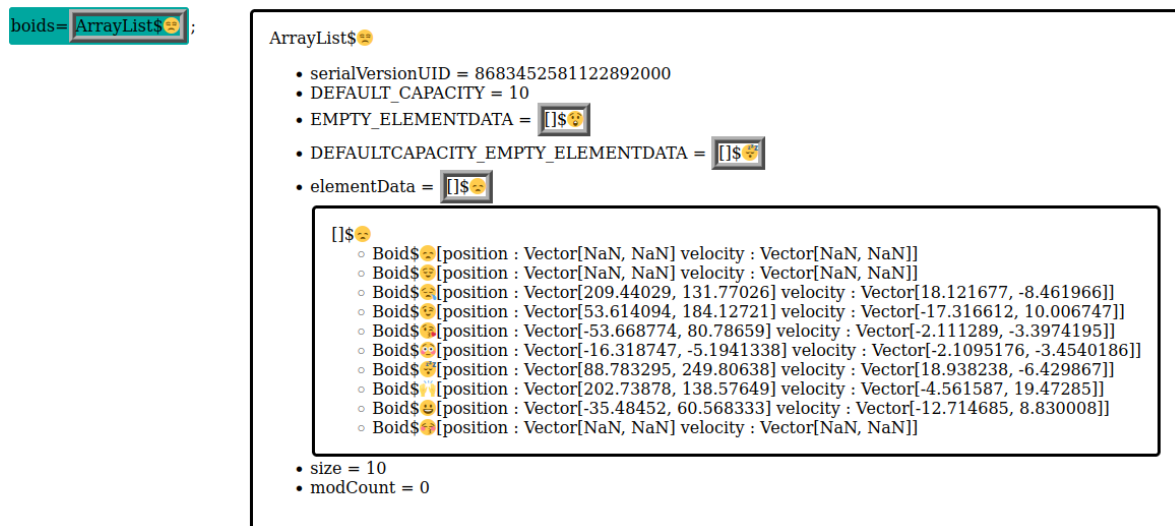


Figure 2.2: ArrayList data

The image above shows what can be seen when inspecting the data of an object. Several elements can be observed. A header with the class name of the object and an emoji identifying its pointer. A list of fields; if the value is also an object, we can expand it as in an interactive debugger. A quick look at the elements shows that the position of the first, second and last boid is equal to NaN. This explains why the boids disappear from the screen. Next question: what computation led to this value? Let's dig into the tickWorld function.

```

18  while(++x < 10 =>true x=9
20      BoidLogic.tickWorld(boids: ArrayList$, World.physics: Physics$); - ArrayList$
    tickWorld
163      List<Boid> newBoids = new ArrayList<>(10); newBoids= ArrayList$
164      for(int i=0; i<allBoids.size(); ++i){ i=0; i<allBoids.size(); ++i){
164      for(int i=0; i < allBoids.size() =>true
165      newBoids.add(tickBoid(allBoids.get(i), allBoids, physics): Boid$); =>true ;
164      for(int i=0; i<allBoids.size(); ++i i=1
164      for(int i=0; i < allBoids.size() =>true
165      newBoids.add(tickBoid(allBoids.get(i), allBoids, physics): Boid$); =>true ;
164      for(int i=0; i<allBoids.size(); ++i i=2
164      for(int i=0; i < allBoids.size() =>true
165      newBoids.add(tickBoid(allBoids.get(i), allBoids, physics): Boid$); =>true ;
164      for(int i=0; i<allBoids.size(); ++i i=3
164      for(int i=0; i < allBoids.size() =>true
165      newBoids.add(tickBoid(allBoids.get(i), allBoids, physics): Boid$); =>true ;

```

Figure 2.3: Event log of a tickWorld execution

After expanding tickWorld, we see that each boid is updated using the tickBoid function. The next step is to go inside tickBoid for one of the boids whose position is equal to NaN. Let's take a look at the first boid by clicking on *newBoids.add*.

```

149      tickBoid
      Geometry.Vector2 acceleration = totalForce(thisBoid, allBoids, physics); acceleration=Vector[NaN, NaN]
150      Geometry.Vector2 velocity = thisBoid.velocity().add(acceleration); velocity=Vector[NaN, NaN]
151      if (velocity.norm() > physics.maximumSpeed() =>false
155      if (velocity.norm() < physics.minimumSpeed() =>false
159      return new Boid(thisBoid.position().add(thisBoid.velocity()):Vector[NaN, NaN], velocity:Vector[NaN, NaN]); - Boid$ ;
      <init>
      ...
165      newBoids.add(tickBoid(allBoids.get(i), allBoids, physics): Boid$); =>true ;

```

Figure 2.4: Event log of a tickBoid execution

After examining the tickBoid function, we see that the new position is equal to the old position plus velocity. Velocity is the old velocity plus acceleration. The acceleration is equal to NaN. It would be interesting to know why. Let's move on to the totalForce function by clicking on it.

```

135      List<Boid> withinPerceptionRadius = boidsWithinRadius(thisBoid, allBoids.stream().physics.perceptionRadius().toList()); withinPerceptionRadius= ListNS
136      Geometry.Vector2 cohere = cohesionForce(thisBoid, withinPerceptionRadius); cohere=Vector[NaN, NaN]
137      Geometry.Vector2 align = alignmentForce(thisBoid, withinPerceptionRadius); align=Vector[0, 0]
138      Stream<Boid> withinAvoidanceRadius = boidsWithinRadius(thisBoid, withinPerceptionRadius.stream().physics.avoidanceRadius()); withinAvoidanceRadius=
139      Geometry.Vector2 avoid = avoidanceForce(thisBoid, withinAvoidanceRadius); avoid=Vector[0, 0]
140      Geometry.Vector2 contain = containmentForce(thisBoid, allBoids, World.Physics.WIDTH, World.Physics.HEIGHT); contain=Vector[0, 0]
142      return avoid.scale(physics.avoidanceWeight())
143      .add(cohere.scale(physics.containmentWeight()))
144      .add(align.scale(physics.alignmentWeight()))
145      .add(contain.scale(physics.containmentWeight())); Vector[0, 0]; Vector[NaN, NaN];

```

Figure 2.5: Event log of a totalForce execution

In the totalForce function, we see that the total force is the sum of four different forces, three of which are zero and one of which is equal to NaN. Next question: why is the cohesion force equal to NaN? Let's click on the cohesion force.

```

92      Geometry.Vector2 sum = Geometry.Vector2.zero(); sum=Vector[0, 0]
93      for(int i=0; i<boidsWithinPerceptionRadius.size(); ++i){ i=0; i<boidsWithinPerceptionRadius.size(); ++i){
93      for(int i=0; i<boidsWithinPerceptionRadius.size() false
96      boidsWithinPerceptionRadius.size(); 0
96      1 / (float)boidsWithinPerceptionRadius.size() NaN
96      sum.scale(1 / (float) boidsWithinPerceptionRadius.size(:NaN)); Vector[NaN, NaN]
96      scale
96      Geometry.Vector2 center = sum.scale(1 / (float) boidsWithinPerceptionRadius.size()); center=Vector[NaN, NaN]
97      return center.minus(thisBoid.position():Vector[106.17325, 87.61371]); Vector[NaN, NaN];

```

Figure 2.6: Event log of a cohesionForce execution

The cohesive force appears to be equal to the center of the boids within the perception radius of the current boid minus its position. To calculate the center, we add the positions of all boids within the perception radius and divide by the number of boids. If there are no boids, we divide by zero. It seems we've found the bug, now we just have to fix it.

Chapter 3

Architecture

As a reminder, PrintWizard is divided into two parts: a backend implemented in Java and a graphical user interface implemented in Javascript. Since PrintWizard is aimed at Java programs, using Java for the backend was a natural choice. We chose Javascript because of our familiarity with the language. Performance shouldn't be an issue. As in the first instance, PrintWizard targets smaller programs to allow for faster exploration.

In the remainder of this section, we won't divide the project into a backend and a frontend, but into five different modules: the instrumentation module, the logging module, the object data store, the source formatting module and the display module. This decomposition emerged during the development of PrintWizard, in particular to facilitate certain modifications. We feel that the current organization works well. It allows many changes to be made to modules without affecting the rest of the program. The choice of the decomposition can be linked to the information hiding criteria proposed by David Parnas[Par72].

3.1 Instrumentation module

The instrumentation module is responsible for injecting logging functions into the program. It uses the logging module and hides the way the instrumentation is implemented. During this project, we proposed two different strategies for instrumenting a program: using a Java agent and modifying the bytecode of classes when they are loaded by the JVM, or implementing a Java compiler plugin and modifying the program's AST during compilation. When compiling a Java program into bytecode, we lose information, and this information is needed to record certain events. So, as with the Java compiler plugin, we would need to access the source code with the Java agent approach. Moreover, we would need a function to merge bytecode and source code information. This function is not impossible to implement, but the Java compiler approaches seemed much simpler.

3.2 Logging module

The logging module is responsible for recording the event tree, saving it to a log file and loading the event tree into the GUI. It provides a function for logging the different events. The logging module hides the format of the log file from the other modules. This allows us to easily change the way events are stored. We store the execution trace in a json file. Events are stored as json objects in an array in order. To represent a group of events, we use a start event and an end event.

```
[
{"eventId":21,"eventType":"statement","type":"GroupEvent","pos":"start"},
{"eventId":23,"eventType":"subStatement","type":"GroupEvent","pos":"start"},
{"eventId":23,"eventType":"subStatement","type":"GroupEvent","pos":"end"},
{"result":{"value":false,"dataType":"bool"},"type":"ExecutionStep","kind":"expression"},
{"eventId":21,"eventType":"statement","type":"GroupEvent","pos":"end"}
]
```

Figure 3.1: Example of a simple trace. Group event always have a start and end event. Execution steps are represented as a single event.

The logging module provides a simple API. The API is divided into two parts: simple logging functions for creating execution steps and more complex functions for creating event groups. Handling an event group is more complicated than the execution steps because it needs to be opened and closed. We implement the functionality as follows: first, we call a constructor to create an event group object, then the object exposes start and end functions. We also keep a stack of the currently open event group. This allows us to check at runtime that the event groups form a tree, and to close all event groups if the program stops before completion.

3.3 Object data store

The responsibility of the object data store is to support the inspection of values that contain other values, such as objects or arrays. The data store hides the strategy used to copy and store complex values. In PrintWizard, an object or array is identified by the hash code of its reference and the timestamp of its reading. Timestamps are increased whenever an object is read or updated. The data store supports the storage of the specific version of an object and loads the object version corresponding to a reference and timestamp.

The current strategy used by the data store is to save an object whenever one of its fields is updated or the object is created. When we retrieve an object, we return the most recent version stored for a given timestamp and hashcode. It's necessary to look for the most recent version because we don't

make a copy of an object every time the timestamp is increased. We serialize objects in json format.

3.4 Source code formatting module

The responsibility of the source code formatting module is to collect information about the formatting of the various source code expressions, store this information in a file, and provide it to the display module. The purpose of the source code formatting store is to reduce data duplication in the execution trace. Now each event only stores the identifier of its source code information. For now, the identifier corresponds to the concatenation of the start and end index of an expression in the source code. It would be pretty simple to change the identifier.

With the source code formatter, each execution step only needs to store the identifier of its source code information.

The module's API supports retrieving the source code of an expression, the beginning of the line up to the expression, and the end of the line after the expression.

Currently, we have a problem with the Java compiler plugin because it doesn't store the start and end of all expressions. If we don't have this information, we use a fallback mechanism to display something, but it's far from perfect. In the fallback mechanism, we use the toString method of the AST node to represent an expression. One way of solving this problem would be to use another parser to extract the information from the source code.

3.5 Display module

The display module is responsible for displaying the event tree in the GUI. The module uses the source code formatter, the object data store, and the logging module to collect all the necessary information.

As a reminder, the frontend in the project is implemented in javascript. We translate the event tree into the dom tree, keeping the same structure. Each event is translated into a html element with the same parent and children. We play with the way we display each type of node to ensure a pleasant experience. In dom, events are divided into four different categories: statement, sub-statement, flow control, and execution steps. The characteristics of each category are described below.

3.5.1 statement

Since PrintWizard is aimed at reducing cognitive burden, we want the trace to be as close to the source code as possible. A statement should represent a line of code. It is displayed as a line of code, with the part corresponding to the expression grayed out. When collapsed, only the statement is shown, but when expanded, all subexpressions are shown as well.

```
main
7  int x = 0; x=0
8  while(x < 3 =>true
9    x = x + 1; =>1 x=1;
8  while(x < 3 =>true
9    x = x + 1; =>2 x=2;
8  while(x < 3 =>true
9    x = x + 1; =>3 x=3;
8  while(x < 3 =>false
```

Figure 3.2: In the image above, $x=x+1$ corresponds to a statement.

3.5.2 sub-statement

A sub-statement represents the various execution steps that make up a statement. A sub-statement appears only as the text of the expression being executed. It's useful to quickly see why a statement has a particular result. An action that is sometimes complicated in standard debuggers.

```
main
7  int x = 0; x=0
8  while(x < 3 =>true
9    x + 1 =>1
9    x = x + 1; =>1 x=1;
8  while(x < 3 =>true
9    x = x + 1; =>2 x=2;
8  while(x < 3 =>true
9    x = x + 1; =>3 x=3;
8  while(x < 3 =>false
```

Figure 3.3: In the image above, $x+1$ is a sub-statement.

3.5.3 flow control

The original motivation for the flow control category was to add indentation to the display. Now the goal is more general: a flow control node should indicate the change of flow control between contexts. For example, a function call or an if-else statement. Currently, there are two different types of flow control nodes: indentation and function definition. This is the part you can imagine being the easiest to customize.

Indentation nodes cannot be collapsed and add one level of indentation to all their children.

Function definition nodes, when collapsed, show the function name and ... to indicate that they can be expanded; when expanded, they show all the statements that happened during the function call.

Chapter 4

Roadmap to deploy PrintWizard

The goal is to use PrintWizard in the Introduction to Programming course[Jam] at EPFL given to first year computer science students at EPFL. In its first deployment, the tool would be limited to supporting single-file Java applications and single-threaded programs. Single-file support shouldn't be a problem, since all the exercises in the course can be solved with a single file, and this fits well with the simple program that the tool is supposed to target in the first place. Outcomes should include whether the student uses the tool and prefers it to interactive debuggers, how difficult it turns out to be to use, what sorts of bugs it is useful for and whether it helps him or her better understand what programs do. Here's a roadmap for achieving this goal.

4.1 Improving testing infrastructure

Currently we're checking that several Java programs compile without errors when using our plugin. There are two ways to improve the testing infrastructure. One is to test not only compilation, but also program execution and GUI output. The second is to add new examples to the currently limited test set and make sure all test passes.

4.2 Fixing the issue with poor event formatting

The Java compiler does not provide start and end index information in the source code for all expressions. This is a problem because we currently use the source code to display the execution steps, and the fallback mechanism is far from perfect. The goal of this section is to determine if we can use another program, such as the Java parser, to extract the information from the source code, or find another way to solve the problem. A second task would be to add syntax highlighting.

4.3 Support external library in the object store

Currently, we only make a copy of an object when it is created or when one of its fields is written. The code of external libraries is not instrumented. Thus, if we use an external library such as the java standard library, it's impossible to know if a field was updated when we call a function defined in the library. A quick solution is to store a hash of the object data and then each time the object is read, visit the object's graph and see if anything has changed. This is not efficient at all, but it should work.

Chapter 5

Related Work

PrintWizard is related to a number of different projects. Debugging backward in time[Lew03], which aims to improve the typical interactive debugger by adding the ability to go back in time. **Snoop** and the work of previous students in the lab, which uses event logs to help debug programs.

5.1 Time traveling[12]

Typical backward debuggers retain the same interface as a standard interactive debugger. In our opinion, they are still focused on exploring a particular state of the program, it's just that it's easier to find the relevant point in the execution. They can't be compared to PrintWizard in this respect. They remain relevant because they face similar problems to our tool. Below we describe different techniques for implementing time traveling debuggers.

5.1.1 Record and replay

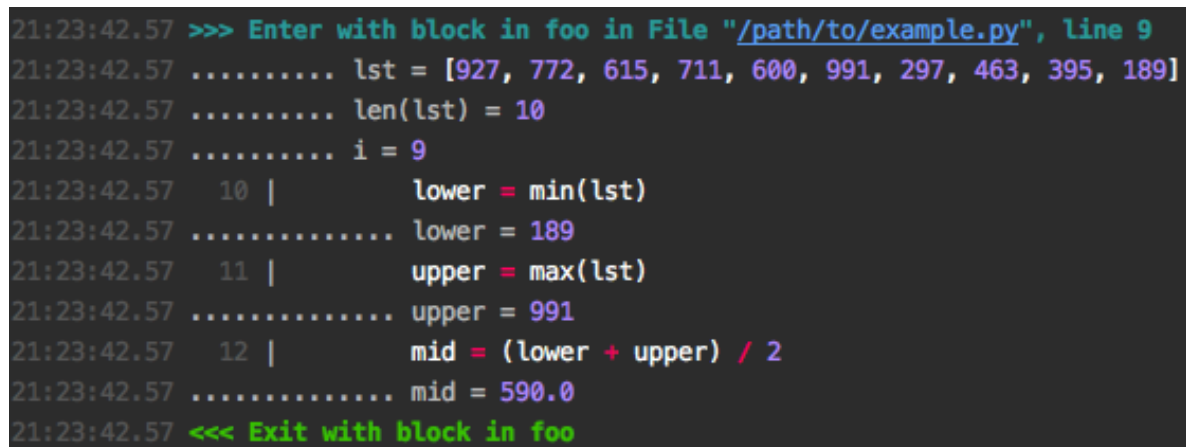
Time travel can be achieved by replaying the program to the desired point. The main challenge of the replay technique is the deterministic re-execution of a program, which can be more or less complex depending on the context. Some replay-based debuggers record the state of the program at regular intervals in order to speed up the backward navigation in time, since now we only need to re-execute the program from the last checkpoint. The advantage of record and replay debuggers is low overhead, but at the expense of having to wait for the program to re-execute. An example of a record and replay debugger currently available is **Mozilla RR**[24c] with an overhead of only about 1.2 times. However, the main motivation behind **Mozilla RR** is to improve the debugging of non-deterministic bugs and not to support backward-in-time debugging.

5.1.2 Omniscient debuggers

Time travel can be achieved by recording all state changes of a program. Debuggers that use this technique are now called omniscient debuggers. Compared to record and replay-based techniques, it provides near-instantaneous navigation and the ability to query the history. What queries are possible depends on the project, but a typical feature is to find out why a variable has a particular value. However, the overhead is much higher. For typical omniscient debuggers and typical programs, it can easily be 100 times[PT09]. Omniscient debuggers don't seem to be really used. The only one we know of that is currently available is **Pernosco**[24b]. **Pernosco** is built on top of **Mozilla RR** and interestingly solves the performance problem by using a remote server.

5.2 Snoop

Snoop[Hal24] is a Python debugging library. It provides annotation to instrument blocks of code and print the record of events to the terminal. The typical use is to instrument one function at a time. By default, **Snoop** collects incomplete information. For example, it doesn't collect data of objects or events that happen in inner function calls. The main weakness of **Snoop** is that it lacks a structured event log format, making it difficult to develop a GUI to consume the event record. The event log quickly becomes difficult to navigate, and we often don't have all the information we need out of the box. It is therefore necessary to customize the output to extract the relevant information. Unlike PrintWizard, **Snoop** doesn't have a front end and doesn't support inspecting data of objects or arrays. However, it does support runtime instrumentation and flexible customization of the event log.



```
21:23:42.57 >>> Enter with block in foo in File "/path/to/example.py", line 9
21:23:42.57 ..... lst = [927, 772, 615, 711, 600, 991, 297, 463, 395, 189]
21:23:42.57 ..... len(lst) = 10
21:23:42.57 ..... i = 9
21:23:42.57 10 |         lower = min(lst)
21:23:42.57 ..... lower = 189
21:23:42.57 11 |         upper = max(lst)
21:23:42.57 ..... upper = 991
21:23:42.57 12 |         mid = (lower + upper) / 2
21:23:42.57 ..... mid = 590.0
21:23:42.57 <<< Exit with block in foo
```

Figure 5.1: Example of an output from **Snoop**

5.3 Valentin and Baptiste work on event log based debugging

The focus of Valentin's[Aeb23] and Baptiste's[Lam23] work is how to capture all the language-level events that happen during program execution. This is already a challenge, as we are interested in much more precise information than in previous work. The work of Valentin is more relevant to PrintWizard since he focused on Java and Baptiste on Python. Valentin considered using the Java debugging interface, instrumenting the Java bytecode, and implementing a Java compiler plugin. The first two approaches proved impossible or too difficult to implement. The result was JumboTrace, a Java compiler plugin that instruments Java classes and records every event that occurs during their execution and stores the event log in a binary format. Unlike PrintWizard, JumboTrace doesn't have a GUI and doesn't record information about the object data.

Chapter 6

Conclusion

We have introduced PrintWizard, a new observation tool designed for debugging and code exploration. It has already been used successfully to locate complex bugs. PrintWizard is based on a flexible event log that captures comprehensive information about a program's execution, and a graphical interface that allows interactive exploration of the event trace. The next step is to complete PrintWizard and make it available to students learning Java. Although PrintWizard can already be used to debug programs, it does not yet support all valid Java programs, and sometimes we display events incorrectly due to a lack of information. We are optimistic and believe that this should not cause critical problems.

Promising directions for further development would be to create a lightweight version of PrintWizard with reduced overhead, where the programmer interactively selects what he wants to observe, or to extend PrintWizard to new languages such as Python.

Bibliography

- [12] *Reverse History*. Accessed: (26 June 2024). 2012. URL: <https://jakob.engbloms.se/archives/1547>.
- [24a] *Debugger*. 2024. URL: <https://en.wikipedia.org/wiki/Debugger>.
- [24b] *Pernosco*. 2024. URL: <https://pernos.co/>.
- [24c] *rr project*. 2024. URL: <https://rr-project.org/>.
- [Aeb23] Valentin Aebi. *Immediate tracing for Java programs*. EPFL SYSTEMF laboratory, 2023.
- [Hal24] Alex Hall. *Snoop*. 2024. URL: <https://github.com/alexmojaki/snoop>.
- [Jam] Sam Jamila. *CS-107, Introduction à la programmation*. URL: https://isa.epfl.ch/imoniteur_ISAP/!itffichcours.htm?ww_i_matiere=784637359&ww_x_anneeacad=2301874614&ww_i_section=946228&ww_i_niveau=6683111&ww_c_langue=fr.
- [Lam23] Baptiste Lambert. *State-recording traceback for smoother debugging in python*. EPFL SYSTEMF laboratory, 2023.
- [Lew03] Bil Lewis. “Debugging Backwards in Time”. In: *Computing Research Repository* cs.SE/0310016 (Oct. 2003).
- [Par72] David Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Communications of the ACM* 15 (Dec. 1972), pp. 1053–. DOI: 10.1145/361598.361623.
- [PT09] Guillaume Pothier and Éric Tanter. “Back to the Future: Omniscient Debugging”. In: *IEEE Software* 26 (Nov. 2009), pp. 78–85. DOI: 10.1109/MS.2009.169.
- [Rey87] Craig W. Reynolds. “Flocks, herds and schools: A distributed behavioral model”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406. URL: <https://doi.org/10.1145/37401.37406>.
- [Zel09] Andreas Zeller. “How failures come to be”. In: *Why Programs Fail*. Elsevier, 2009, pp. 1–23.