Debugging and visualizing runtime information in Envision

Research in Computer Science Project

Lukas Vogel

Supervisors: Dimitar Asenov, Prof. Dr. Peter Müller
Chair of Programming Methodology
ETH Zürich

May 18, 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

As programs keep getting larger, tools to develop, test, and debug them should also provide more help. Existing tools often display runtime data in a textual way. Visualizations in some use cases could help to grasp the data faster.

We introduce the concept of probes. A probe is an examination of program data and data interactions at a specific statement in the program during runtime. Combined with visualizations, probes should provide a source of information about a program. This information could be useful in several development activities, such as debugging, testing, and optimizing programs.

We implement a prototype of this concept in Envision. Envision is an environment which uses visual and textual elements to represent code. We argue that such an environment can optimally support probes with visualized data. With three examples we show possible use cases and the utility of the introduced probes.

In the current implementation we can probe variables of primitive type and visualize their data over time with bar-plots, scatter-plots, and array visualizations. While the implementation is still limited, it provides a nice glimpse of how the concepts of probes work. The probe concept is more general and can naturally be extended to many additional cases.

Contents

1 Introduction

A debugger is an invaluable tool to find problems in program code, as well as to help testing and understanding a program. One specifically helpful feature is the possibility to show values of variables during runtime. This is essential to find a certain state which is invalid. Showing values can also help to understand an algorithm, where its functionality is not obvious from the code.

In existing tools, reading runtime values can be cumbersome for complex structures. Eclipse[1] for example provides an indented list to show array data. To search for a specific field value in an object array can thus quickly become a time consuming task, as each object entry has to be expanded first.

Debuggers can inform a user about values in a single state of a program. However debuggers often can not relate data between different states, or inside a single state. Therefore a user still has to do manual work to get data histories or relations in the data.

In this project we explore the concept of probes. A probe is an examination of runtime values of any amount of variables at a specific location in a program. The probe keeps a history of the runtime values of the observed variables. We think that suitable visualizations of this data can aid programmers in various activities.

We present a prototype implementation of the probe concept in Envision. Envision [2] is a research project which aims to provide a next generation IDE. Envision uses visual and textual elements to present code and can easily be extended with plug-ins. Instead of working on a text file, Envision uses a tree model to store and edit program code. This model simplifies the querying of information about the code. Considering this helpful model and the already visual environment of Envision, we think that probes fit very well into Envision.

[1] https://eclipse.org/

2 Probes

2.1 Concept

A probe is inspired by a measurement in electronics. An electrical engineer can connect a probe (a measurement device) anywhere in a circuit and measure various properties of the electrical signal. Both a multimeter or an oscilloscope can be used as a probe. While the multimeter can show a certain property of the signal at a certain time, an oscilloscope can show how this property changes over time. Thus an oscilloscope can diminish the manual post processing which might be needed when using only a multimeter.

We think in this analogy a traditional debugger is like a multimeter and a probe is more like an oscilloscope. A probe "measures" values of variables at runtime. It can be created at any statement of the program by issuing a probe command. The probe is then attached to this specific statement. The probe stores a history of the values of the variables it measures. Whenever the statement where the probe is attached to is executed, the history is updated. With arguments to the probe command the user can specify which variables should be captured by the probe and how they should be related.

Probes can capture data from various data structures existing in programs. A simple use case is to use a probe to get a history of a variable of primitive type. But a probe also works with arrays and index variables pointing into it. Even more interesting are complex data structures which are interconnected such as linked lists, trees, etc. With a specific argument syntax a probe could be created to track such structures. For example by specifying a start node and the next field, a probe could be declared to track and visualize data of a linked list. In complex structures there is often a lot of irrelevant data for a specific task, so a user should be able to specify which data should be tracked.

The collection of data from a probe can be visualized in many meaningful ways. For some cases, like the history of a variable, expressive visualizations can be chosen automatically by the program. For other cases, like a complex data structure with interconnections, the user needs to specify more information.

We think by using a probe a developer can focus on the data that is relevant to his task. In existing tools like Eclipse a developer would need to look through all values and pick out the relevant ones for his task. The developer would then still need to manually relate this data, e.g. by using external tools. With a probe the developer can specify all relevant variables and get a single view of this data and its relations. The visualization can help to read and understand this data.

The following section should demonstrate how the concept of probes can be translated into practice.

2.2 Example use cases

In this section we show three selected use cases, which showcase how the current implementation can be used to help our imaginary developers, Alice, Bob, and Carol to better understand their programs.

2.2.1 Time measurements

Alice wonders about the complexity of her sorting algorithm. She already wrote a function which measures the runtime for different input sizes. The function is shown in figure 2.1. Alice planned to print the size versus the running time in a file and then create a plot. But recently she found out about probes in Envision and she decides to try them now. She creates a probe of the input size versus the elapsed time by typing the command: `probe size elapsed`. Once she executes the program the plot of the runtime is automatically generated for her. She notices that the time values are inconvenient to read as they are measured in nanoseconds. By adapting the probe to divide the elapsed variable (`probe size elapsed/1000000`) she can quickly generate a new plot which has the time value in milliseconds. The resulting plot is shown in figure 2.1. By looking at the plot Alice can see that her implementation shows a quadratic runtime.

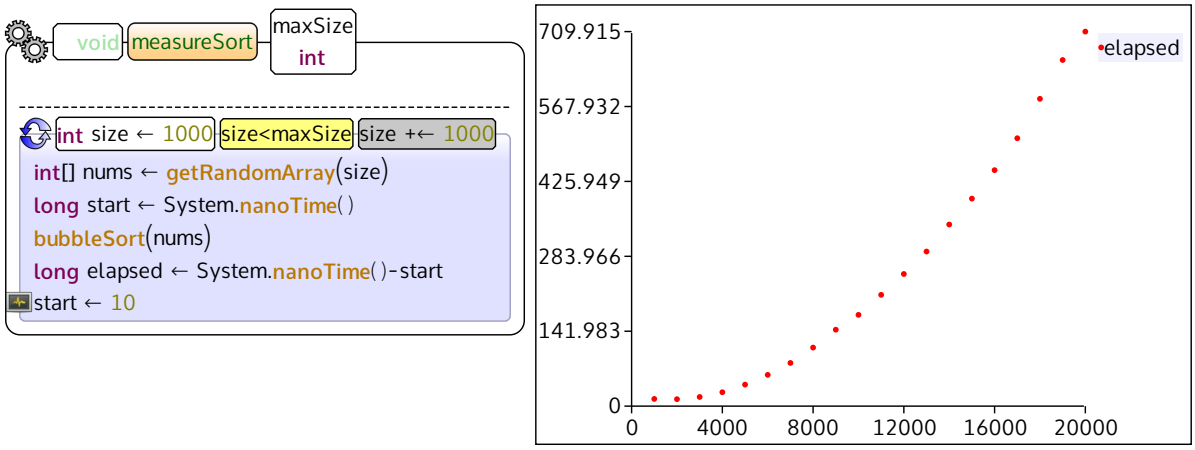

Figure 2.1: On the left we see a measurement function for the sorting function `bubbleSort()`. On the last statement of the method we introduced a probe with the following command: `probe size elapsed/1000000`, it is indicated by the small monitor icon. This probe will thus relate the size to the execution time in milliseconds. Note that we had to introduce a dummy statement on the last line, this is to make sure we have the correct value in the elapsed variable. On the right side we have the plot of the probe which was generated by running the program.

2.2.2 Error difference

Bob has to implement two different approximation functions for π. He received the following two formulas:

$$\texttt{pi4Approx} = 4 \cdot \frac{\pi}{4} = 4 \cdot (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots)$$

$$\texttt{pi2Approx} = 2 \cdot \frac{\pi}{2} = 2 \cdot (1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \cdots)$$

Once he has implemented both approximation functions he has to decide which of the two formulas converges faster. To test this, Bob wrote the function we show in figure 2.2. For different amounts of iterations this function calculates the error of each approximation function. Instead of manually comparing the values Bob creates a probe to visualize both errors using the following command: `probe iterations pi4Error pi2Error`. The resulting plot of the probe we show in figure 2.2. From this plot Bob can immediately see that the error of the `pi2Approx` function decreases faster than the error of the `pi4Approx` function.

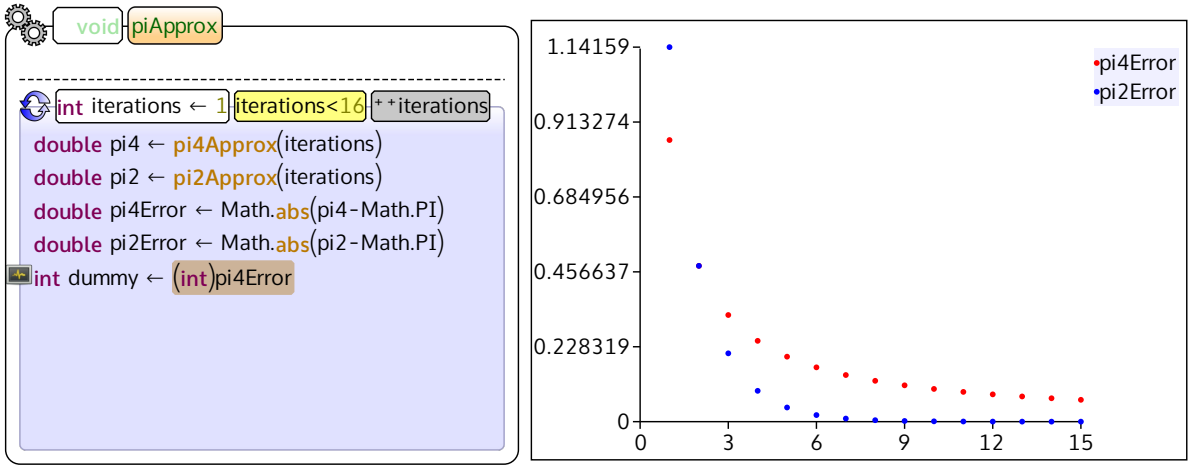

Figure 2.2: In this example we want to find out which approximation function converges in less iterations. The `piApprox` function calculates the absolute approximation error for each amount of iterations. We introduce a probe to plot both errors with the following command: `probe iterations pi4Error pi2Error`. During the execution the plot is filled with the error values. The legend indicates the colors of the values. Note that we had to introduce a dummy statement to assure that we fetch the correct value of the `pi2Error` variable from the debugger.

2.2.3 Algorithm understanding

Carol is training for her first technical interview. To start her practice session she writes a binary search function. The function is shown in figure 2.3. She then tests the function with several test cases and notices that in some cases the function does not return. By just inspecting the code she cannot determine the cause of this problem. She introduces a probe involving the array and the three indexes pointing in the array (command: `probe A imin imid imax`). From the probe plot shown in figure 2.4 and figure 2.5 she sees that after the third iteration of her

code the plot does not change. From this she can identify which index is wrongly updated and fix the bug in the code.

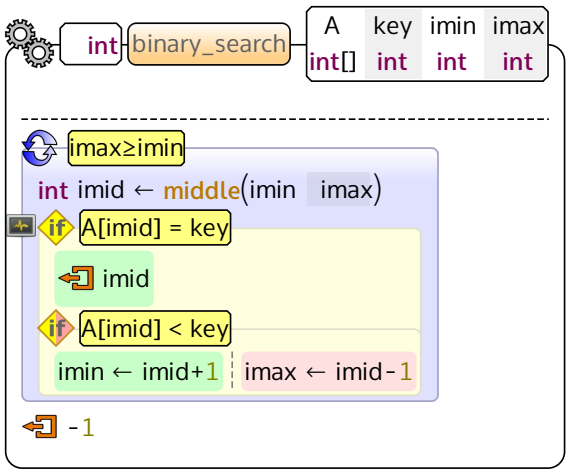

Figure 2.3: Here we show a binary search method with a slight bug related to `imin`. We introduce the probe: `probe A imin imax imid` at the first if statement. On the right side we see the plot of this probe in the second and third iteration.

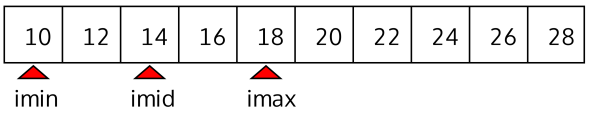

Figure 2.4: The plot of the `A` array with the corresponding indexes after the second loop iteration of the `binary_search` method.

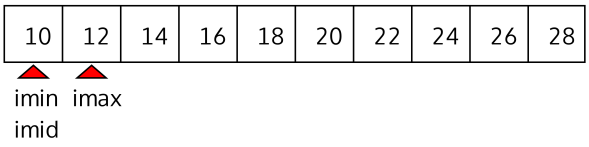

Figure 2.5: The plot of the `A` array with the corresponding indexes after the third loop iteration of the `binary_search` method. Any further iterations will result in the same plot, with `imin` at the same position. This leads us to the conclusion that `imin` is incorrectly updated.

2.3 Features of the prototype

The prototype implementation of probes in Envision is still quite limited in functionality. It only supports probing of multiple variables of primitive type and probing an array variable with multiple index variables. We describe how to interact with the current implementation of probes in section 2.3.1. The currently available visualizations are listed in section 2.3.2.

2.3.1 Interaction

To create a new probe one has to enter the `probe` command, with any number of arguments. The arguments can be variable names, numbers, or a combination thereof in a simple calculation. In figure 2.6 and figure 2.7 we show two examples of probe commands.

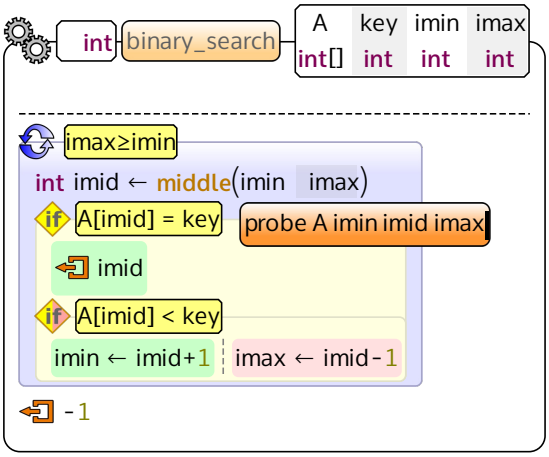

Figure 2.6: To visualize how a binary search method works we enter the `probe` command in the orange command prompt. As arguments we pass the array name (`A`) and the indexes (`imin, imax, imid`), which point in the array.

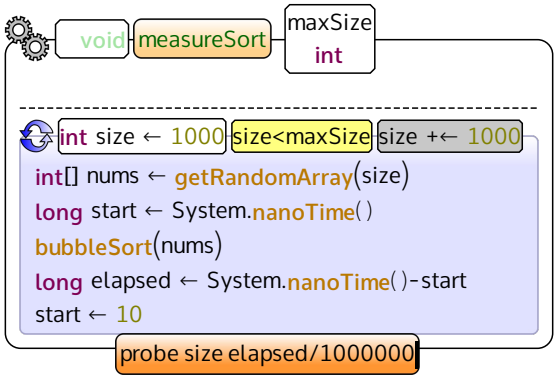

Figure 2.7: To test the execution time of an algorithm, often different sizes are measured and plotted. With a probe we can generate a plot of the array size versus the elapsed time for an algorithm (`bubbleSort`). With a simple calculation we can adapt the measured time to be plotted in milliseconds instead of nanoseconds.

The probe command can be confirmed by pressing Enter. Envision will check if all the specified variables are declared in this region. If this is the case a small monitor icon ( ) will indicate that there is a probe on this statement, additionally a plot overlay will appear. During the execution of the program the runtime values of the probed variables are inserted in the plot.

2.3.2 Visualizations

Currently there are three different visualizations which are chosen depending on the variable types:

1. For a single variable a bar plot shows its history.
2. For an array we show the array and its contents, additional variables are interpreted as indexes into the array.
3. For multiple scalar variables we show a scatter plot.

3 Implementation

This chapter presents important aspects of the implementation of the debug facilities we introduced in Envision. We first show a general overview in section 3.1. For the communication between Envision and the Java virtual machine (JVM), where the debuggee process is running, we chose to use the JDWP protocol[1]. We motivate this choice and explain the implementation of the protocol in section 3.2.

3.1 Overview

As a prerequisite for debugging we need the ability to compile and run a program. For this we introduce the corresponding classes as we show in figure 3.1. Figure 3.1 also shows the most relevant classes for developers of other plug-ins. In the following subsections we will present each of these classes separately. For users we introduced the commands `compile`, `run`, and `debug` so that they can directly take action from inside Envision.

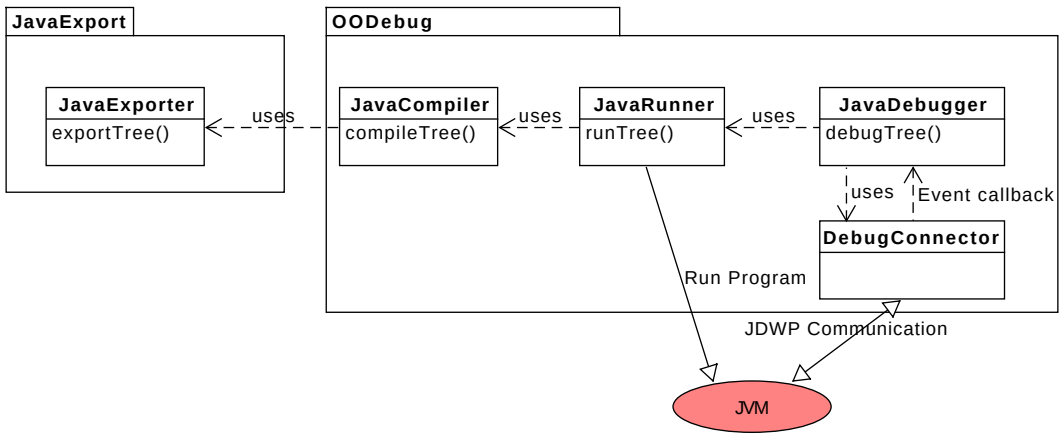

Figure 3.1: Overview of the relevant classes and their interactions in the debug plug-in. Note that the JVM and the interactions with it only exist during execution.

[1] http://docs.oracle.com/javase/7/docs/platform/jpda/jdwp/jdwp-protocol.html

3.1.1 JavaCompiler

The `JavaCompiler` class provides a `compileTree()` function, which will trigger a compilation of the current Envision tree. It uses the `JavaExporter` class to export the Envision model to Java code. The exported code is compiled by using the `CommandLineCompiler` class, which calls the `javac` compiler with the help of the `QProcess` class. The feedback from the compiler is parsed and displayed at the correct node's visualization. The `compileTree()` function returns a `CommandResult` object to notify the caller if everything went successfully.

3.1.2 JavaRunner

The `JavaRunner` class is responsible for running a compiled program. The `runTree()` method is the main interaction interface of this class. It assures that a compiled version of the program exists. The `MainMethodFinder` class helps to find the file which contains the main method. The name of the main file is then passed as an argument to the `java` command, which is executed in a `QProcess` wrapper. The output of the running process is redirected to a `ConsoleOverlay`.

3.1.3 JavaDebugger

The `JavaDebugger` class is a core part of this plug-in. It handles the interactions between the user and the debugger. The `JavaDebugger` class relies on the `DebugConnector` class which handles the communication to the JVM. To simplify the implementation most of this communication is synchronous, however there are also some events which are raised by the VM, like e.g. when a breakpoint is hit. We first explain the managing of probes and then the interaction between the `DebugConnector` and this class in separate subsections.

Probe management

When a user creates a new probe the `JavaDebugger` first checks if all the entered variables exist at this location. If that is the case, all relevant information for the probe is stored in a struct called `VariableObserver`. It contains the following details:

- For each variable the details which are needed to fetch the value from the debugger.
- For each variable a transformation function which transforms the variable values to the values expected by the probe. For example if the user enters `x+10` the transformation function for the variable with name `x` would be value + 10.
- A function which handles how the values are visualized.

To know which nodes have an attached probe, we introduced a map between nodes and `VariableObserver`s. At the probe location a breakpoint is added. Once a breakpoint is hit the handler function checks if there is a `VariableObserver` at this location. If there is an observer all the values of it are fetched and transformed. Then the stored handling function is called with the new values as an argument.

JavaDebugger and DebugConnector interaction

The interactions between the `JavaDebugger` class and the `DebugConnector` class is mostly synchronous. That means the debugger calls a method on the connector class which blocks until the response is back from the VM.

For communication from the VM to the debugger we introduced a different mechanism. As the `DebugConnector` is decoupled and independent of the `JavaDebugger`, the `JavaDebugger` has to register a callback for events. Once such an event happens the `DebugConnector` calls the registered callback with the event info as argument. The `JavaDebugger` then takes action for this event. With our probes mechanism we introduced an auto resume functionality, so that we only stop at a breakpoint for a short moment to fetch the values for the probe. If the user did not register a breakpoint at the same location we will just resume from the probe. It could however be that the user used a single step to reach the location in which case we do not want to resume automatically. In such a case 2 events happen at the same time and they have different opinions on whether they want to resume. To handle such cases properly we introduced an agreement protocol, which assures that the action expected by the user happens.

3.2 JDWP protocol

3.2.1 Debug protocol choices

To connect to a java debugger programmatically there are several options. The documentation from Oracle[2] shows an overview of those possibilities. We outline an evaluation of each possibility in the following subsections.

Java Debugger (JDB)

JDB is a console application to debug Java programs. To interface with it from a program one would need to parse the console output. However the format of this output is not defined and it is not clear how to distinguish the output from the debugger and from the program which is debugged. Additionally it limits the possible features to the features offered by JDB, which might not be as complete as the other possibilities. One advantage of this approach is that it should be easy to quickly have some running code, however the maintainability might lack. Overall we decided that the disadvantages outweigh the advantages for this approach.

Java Debug Interface (JDI)

The JDI is a set of Java APIs, which provide useful information about the JVM. Those APIs would simplify the implementation of a debugger written in Java. Internally JDI uses the JDWP protocol to communicate with the JVM.

Envision is written in C++ thus we would need to either wrap the Java program into a native interface or write some communication protocol to communicate between Envision and the Java program. However both approaches are offered already: the JVMTI offers a native

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html

interface to the VM and JDWP is a protocol to communicate with the VM. Thus we decided to not use JDI.

JVM Tool Interface (JVMTI)

The tool interface is a set of native, i.e. C/C++, APIs. A C++ API would certainly be helpful for our project. Yet some implementation details render this approach uninteresting for us. One has to provide an agent which will be called once the JVM is started. This means the code will execute in the VM process. Thus we cannot easily have the code running in the Envision process. Having separate processes would not work for our purposes, as Envision needs the information from the debugger. Additionally this would mean that the debugger competes with the VM and thus the debuggee for the resources. We did not find those problems with the JDWP approach and therefore decided to not further investigate the JVMTI.

Java Debug Wire Protocol (JDWP)

The JDWP protocol defines the interface between a debugger front end and the VM back end. It offers clear separation between the debugger and the debuggee. The protocol has a clear documentation. As it is a network protocol we can rely on known C++ and Qt APIs and do not have to add other APIs, like for the JVMTI. The decoupling even facilitates to debug a remote application. On the downside implementing a network protocol brings its own issues, e.g. incomplete data handling, etc. We believe that the advantages outweigh the network issues.

For the reasons stated above we think that JDWP outperforms the other choices so we decided to use this protocol to implement our debugger.

3.2.2 Implementation of the protocol

The JDWP protocol[3] specifies many message types in different formats. We want to parse them with the goal that it should be as easy as possible to support additional messages from the protocol. We achieved this by implementing a helpful framework which makes it possible to partially generate the code for the messages from the documentation.

We introduced a `MessagePart` class which represents a part of a message or even a whole message as defined in the protocol. The class serves as a base class for all messages, however as each command and reply have a fixed set of fields at the beginning we introduced the corresponding `Command` and `Reply` classes as bases for clients. The `MessagePart` class defines the reading and writing operators for the `QByteArray` data from the network. Each message then consists of a set of `MessageFields`, which register a reader and a writer function in the containing `MessagePart`. When the read or write operator on a `MessagePart` implementation is called, all registered reader or writer functions are executed. With this abstraction we achieve that a specific message implementation only has to specify the fields the message contains and nothing else.

[3] http://docs.oracle.com/javase/7/docs/platform/jpda/jdwp/jdwp-protocol.html

To implement the version info reply from the VM as shown in the JDWP documentation[4] we can copy the reply data table to a text editor which shows tabs between the columns. We then search with a regular expression for the following: `([^\t]+)\t([^\t]+)\t([^\n]+)\n`. The first group matches the type the second group the name and the third the description of the field. If we do not need the documentation we can replace the matches with the following expression: `MessageField<\1> \2{&VersionInfo::\2, this};\n`. After this replacement we only have to adapt the types to actual types. It should be fairly easy to make this process even more automated with a script, which automatically replaces the types by the correct C++ types. The code for the reply in listing 3.1 should demonstrate the ease of implementation of a message. Note that we only need the destructor for a correct vtable placement.

```
1 struct OODEBUG_API VersionInfo : public Reply {
2       virtual ~VersionInfo() override;
3
4       MessageField<QString> description{&VersionInfo::description, this};
5       MessageField<qint32> jdwpMajor{&VersionInfo::jdwpMajor, this};
6       MessageField<qint32> jdwpMinor{&VersionInfo::jdwpMinor, this};
7       MessageField<QString> vmVersion{&VersionInfo::vmVersion, this};
8       MessageField<QString> vmName{&VersionInfo::vmName, this};
9 };
```

Listing 3.1: Code of the version info reply

[4] http://docs.oracle.com/javase/7/docs/platform/jpda/jdwp/jdwp-protocol.html#JDWP_VirtualMachine_Version

4 Related Work

4.1 Visualizing data structures

Visualizing data structures and objects is an interesting field. For some objects it is clear how the visualization should look like, for others we have more flexibility, and for some we might even ask the user how to represent them. For example for a color object you can show the color. Other objects like arrays may have multiple visualizations which make sense, like a histogram, or just a plain visualization of the array. On the other hand for user defined objects it can be harder to find a predefined scheme, thus it might be necessary that the user specifies the visualization.

B. Alsallakh et al. [1] designed a plug-in to show array contents and collections data in a visual way. With this plug-in the user can choose to show the data in a tabular view, a histogram, or in a line chart. Compared to the existing data presentation in Eclipse, the visualization of the data in this plug-in is easier to overview and interact with. Similar to this plug-in, our prototype can visualize an array in a plot. Additionally our probe prototype can show relations of indexes to the array.

Zeller et al. [10] created the GNU data display debugger (DDD). DDD can visualize runtime data in various ways. It can display multidimensional arrays in plots and visualize dynamic data structures such as trees. While DDD offers a wide variety of visualizations the tool is stand alone and not integrated in a development environment. We think that an integrated tool could improve the ease of use. It might even encourage users to use the debugger more often as one does not have to switch between different environments.

Rozenberg and Beschastnikh [9] implemented a templated approach to visualize arbitrary objects. The user can specify the visualization of an object by using HTML code with CSS templates. This makes it possible to visualize a locale with a flag, a linked list with arrows between boxes and more. While this provides great possibilities it is not that effortless as users have to write their own HTML code.

Larch [7] is an IDE for Python which combines the code and the execution result in a visual canvas. The idea is that each class can implement a `__present__` method which is a visual equivalent to the to string method `__str__`. The `__present__` method is used to visualize the object during the execution. Due to python's dynamic nature the visualization adapts simultaneously with a code change.

4.2 Visualizing temporal changes

While visualizing data in a certain state of the execution can be helpful, it might also be interesting to see how the data evolves over time. For example if we know a variable should be monotonically increasing and we can just plot its value during the execution, we can quickly see if this is really the case.

Beck et al. [3] made an Eclipse plug-in which can show a history graph of global variables and fields. The plot is attached in a small form to the variable declaration but can be extended to a more detailed view. It shows an overview of the variable history, the minimum and maximum values, and the number of read accesses for each value. The probe implementation we have in Envision can do a very similar thing, except that our prototype does not store the number of accesses to each value. Probes additionally enable a user to relate and plot different variables against each other. With simple calculations the plot of the probed variables can even be tweaked.

4.3 Visualizing relations

Instead of just viewing a certain state or how it evolves over time, it is often even more important to see how values relate to each other. Consider an algorithm to sort an array which uses several index variables. Here we are not only interested in how the array values evolve but also how the indexes move on the array.

The jGRASP IDE [8] has a viewer canvas, which can show visualizations of data. For the array and collections viewer a user can add index expressions. jGRASP can animate the changes to those index expressions by automatically stepping through the program. This can help novice programmers which are trying to understand how a certain algorithm works. Additionally it could aid finding a bug for an experienced programmer.

The concept of viewers in jGRASP has some similarities to probes. With probes one can visualize arrays and indexes pointing to it as well. While the viewers provide selectable visualizations, they seem to only support this very specific use case of data relation. With probes we want to have the possibility to relate any data.

5 Future work

5.1 Call stack visualization

Currently when a breakpoint is hit we get an indication of this event. However it is not possible to see from where the execution reached this point. In existing tools a call stack is often a clickable list of previous statements. In Envision we could re-arrange the methods to visually present the call stack. This could behave similar to the call stacks in Debugger Canvas [6].

5.2 Support for multi-threaded programs

Once we have a suitable call stack presentation we should extend it to support programs with multiple threads. Especially when threads have the same execution path a suitable visualization might help to detect possible race conditions. Probes could be enhanced to render data from multiple threads.

5.3 Improve probe implementation

As we have explained in section 2.3 the current probe implementation is still quite limited. It only supports primitive types and arrays and automatically uses default visualizations.

The prototype implementation should be extended to handle and visualize heap structures. The command syntax should allow a user to declare probes for those complex structures. For example a new command could look like this: `probe diagram head:*.next show:*.value,*.name`. This command specifies that the user wants a diagram of a linked list starting from the node `head` and following through the `next` fields. The `show` argument specifies that only the `value` and `name` field of the node should be contained in the diagram.

Instead of always using a default visualization an improved implementation should enable a user to specify a different visualization.

5.4 Working set view for debugging

Code Bubbles [4, 5] and Debugger Canvas [6] present the working set view in a visual context. We can imagine a working set view with attached probes. Such a working set should also store additional information. For example if we have an approximation function we could store

certain maximum error values for each iteration in the working set. If a user tries to optimize the runtime of the approximation function the user can use the working set to check if the errors are still below the bound. With such a working set this could conveniently be done using the plot of the probe where the maximum error would be shown as a fixed line and the actual values as dots.

5.5 Comparing data between program versions

With probes we can relate data from a single execution of a program. It could be interesting to compare the data from one version of the program to another version of the same program. This could help to experiment with parameters to an algorithm. One could compare how those parameters influence the output in a unified plot. Additionally it might help to pinpoint the version which introduced an undesired behavior with the help of a version control system.

5.6 Support for multiple processes

For now we just support running code from a single main method. For a project with multiple main methods, for example a client server project, there is no way to select which main method to run. To support such a project we would need a map from methods to the running processes. Once this is in place it would make sense to have a separate terminal overlay for each process. This implementation might also allow to start a process multiple times concurrently. Regarding probes and concurrently running processes there should be a way to specify which probe belongs to which process. A possible solution for this could be to have a working set per process, which contains all probes.

5.7 Plotting library

In the current state we implemented the plotting by using primitive drawing operations directly on an overlay. By using a plotting library we possibly get greater flexibility in how our plots look at less implementation cost. One license compatible Qt plotting library is Qwt[1].

[1] http://qwt.sourceforge.net/

6 Conclusion

In this project we aimed to explore and experiment with a visual debugging interface. We started by introducing basic debugging capabilities in Envision. Once this was in place we experimented with the concept of probes. We implemented a probe prototype which can be used in Envision.

We presented a well thought-out concept for probes. We have explained how probes for programs are similar to oscilloscopes for electronic circuits. With example use cases we have shown how our prototype implementation can be valuable to developers to better understand a program.

We outlined many ideas on how to extend our current work. By implementing those ideas we think that Envision can be a very helpful tool for developers for various activities in a development cycle. Especially probes for heap structures are a very interesting topic to explore further.

Appendix A

Debugger Manual

Here we list the available debugger interactions.

Toggle breakpoint `F8`

Continuing from a breakpoint `F6`

Single stepping `CTRL + F6`

Variable tracking `F12` (add a value to the plot every time the variable is referenced)

Add a probe Select a statement hit `ESC` enter `probe` and some variable name(s)

Remove a probe Select the statement where the probe is and hit `ESC` enter `probe -`

Start debug session `ESC` and enter `debug`

References

[1] B. Alsallakh, P. Bodesinsky, S. Miksch, and D. Nasseri. Visualizing arrays in the eclipse java ide. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 541–544, March 2012.

[2] D. Asenov and P. Müller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–12, 2014.

[3] Fabian Beck, Fabrice Hollerich, Stephan Diehl, and Daniel Weiskopf. Visual monitoring of numeric variables embedded in source code. In Telea et al. [3], pages 1–4.

[4] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[5] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[6] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.

[7] G.W. French, J.R. Kennaway, and A.M. Day. Programs as visual, interactive documents. *Software: Practice and Experience*, 44(8):911–930, 2014.

[8] T. Dean Hendrix, James H. Cross, II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. *SIGCSE Bull.*, 36(1):387–391, March 2004.

[9] D. Rozenberg and I. Beschastnikh. Templated visualization of object state with vebugger. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 107–111, Sept 2014.

[10] Andreas Zeller and Dorothea Lütkehaus. Ddd - free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.