A Tool for Visualizing the Execution of Programs and Stack Traces
Especially Suited for Novice Programmers

Stanislav Litvinov, Marat Mingazov, Vladislav Myachikov, Vladimir Ivanov, Yuliya Palamarchuk, Pavel Sozonov and Giancarlo Succi
Innopolis University, Innopolis, Russia

Keywords: Visualizing Execution, Debugging, Teaching Programming.

Abstract: Software engineering education and training has obstacles caused by a lack of basic knowledge about a process of program execution. The article is devoted to the development of special tools that help to visualize the process. We analyze existing tools and propose a new approach to stack and heap visualization. The solution is able to overcome major drawbacks of existing tools and suites well for analysis of programs written in Java and C/C++.

1 INTRODUCTION

Software engineering education and training has many obstacles. Failures of students are quite frequent even at introductory level programming courses; failure rate is approximately 33% (Benstedsen and Caspersen, 2007; Pedrycz and Succi, 2005; Watson and Li, 2014). Apparently, students and novice programmers struggle with very basic concepts of programming, such as behavior of a program in run-time. Early failures in studying may dramatically decrease one’s motivation to become a programmer (Coman et al., 2008; Oroma et al., 2012; Succi et al., 2001; Dragoni et al., 2017). Hence, this is a serious issue for education institutions, if students do not have a clear idea of how a program is executed.

One possible way to resolve the issue involves special tools for visualization the process of program execution. This requires a clear and easy to use visualization of stack and heap memory. Despite many available solutions, very few existing tools are widespread (Sorva et al., 2013).

In this paper we propose a new approach to visualization of stack and heap memory of a running program. The approach is aimed at building a clear understanding of program’s memory organization. The visualization reflects a model of memory which suits to describe programs written on Java and C/C++.

Section 2 describes the state of the art. Section 3 presents major parts of the solution: architecture, user interface, and visualization model. Section 4 summarizes results and outlines the future work.

2 BACKGROUND

2.1 Modeling Memory to Support the Learning Process

Each program in Java and C/C++ usually contains three separate segments: a program area, a stack, and a heap. The program area is where the code is located and it is used to access the instructions to execute. The stack is used by processes or threads as a storage of arguments and local variables. This memory cannot be used if the data size is unknown (Doe, 2016; Yurichev, 2013; cs.umd.edu, 2003). The heap is more flexible, because size of data which will be stored in the heap can be determined at any moment, including run-time. However, usage of heap imposes significant overhead, since it needs additional processor instructions and memory space for storing memory allocation metadata (Doe, 2016; Yurichev, 2013).

Usually, novice programmers and students misunderstand memory organization of a running program (Safina et al., 2016). Typical learning challenges that novice programmers are facing with have been summarized in (Sorva et al., 2013): (i) treating a program as a run-time process, not only a piece of code; (ii) understanding of a computer working process; (iii) revealing implicit programming constructs (e.g., pointers, references); (iv) misunderstanding of program execution sequence and tracing.

Notably, that a standard debugger cannot help novice programmers due to its limited usability, not it
contains useful metrics for defect detection (Moser et al., 2008). A debugger requires a knowledge of memory organization of a program that a novice typically does not have yet. Moreover, most debuggers do not provide any explanations or hints (Sorva et al., 2013).

2.2 Overview of Visualization Tools

Most currently used programming languages for novice programmers are C/C++ and Java and existing visualization tools for them either emphasize their imperative features, or (for C++ and Java) the object oriented features (Scotto et al., 2004; Jermakovics et al., 2011; Bennedsen and Schulte, 2010; Corral et al., 2011; Pedrycz et al., 2015)). Considering Java and C/C++ is also particularly relevant, since these languages are in the top 5 most popular languages (Bissyandé et al., 2013). Needless to say that this implies that this study focuses on compiled languages rather than on scripting languages.

In Table 1 we summarize some of the most prominent tools for visualizing execution of programs. Most of the tools in Table 1 are research prototypes. Very few of them are still in active use. A limited number of tools have been used outside the place of its origin. However, in order to be effective in educational process a visualization tool should have high level of engagement (Juett, 2016; Moreno et al., 2004; Maurer et al., 1999). To overcome learning challenges a visualization of stack and heap memory should be easy to use.

In general, these tools have the following advantages: (a) most of the applications have timeline or/and forward/reverse stepping, (b) one system has a flexible search mechanism which is able to work with many parameters (e.g., variable name, returned value, etc.) (Lessa et al., 2010), (c) some of the tools show multithreading and deadlocks, and (d) textual explanation is very useful (Smith and Webb, 1995).

On the other side, we can identify the following, quite generalized, drawbacks: (a) only a few tools show separate heap and stack, (b) not all of applications are convenient for beginners, (c) only two solutions work in Eclipse IDE, and (d) not all tools allow to use your own code.

3 A NOVEL APPROACH TO STACK AND HEAP VISUALIZATION

We have devised a new approach for visualizing program execution process. The main element of the visualization is the stack trace, which makes memory organization of C/C++ and Java programs explicit\(^1\).

We have implemented prototype to experiment our approach in three different environments: (1) Eclipse IDE for C++, (2) Eclipse IDE for Java, and (3) IntelliJ IDEA for Java.

3.1 System Architecture of the Prototypes

Figure 1 summarizes the general architecture of our prototypes. In the Java-based prototypes Eclipse and IntelliJ IDEA interact with Java Debugging Interface, and in the C/C++ prototypes Eclipse interacts with C/C++ Debugging Interface.

Java Debugging Interface (JDI) is a high-level Java-based interface, which is directly used in debugger applications. JDI provides access to Java threads, virtual machines state, Class, Array, Interface, and primitive types (Oracle, 2016).

C/C++ Debugging Interface (CDI) is a useful Java-based interface to custom debuggers in Eclipse environment. CDI can work with full-featured debugger provided by a development environment tooling (e.g., C/C++ Development Tooling (CDT)), or external debuggers (e.g., GDB). Eclipse plugins can interact with a debugger, and use all features of CDT environment, such as code-stepping, watchpoints, break-

\(^1\)The source code of the prototype is available at https://github.com/MaratMingazov/CMemvit
Table 1: A survey of tools for visualization of a program execution.

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYVISE (Reiss, 2009)</td>
<td>a standalone application for analysis of memory leaks, inefficient use of memory, unexpected changes in memory, etc.</td>
</tr>
<tr>
<td>JIVE (Lessa et al., 2010)</td>
<td>a plugin for Eclipse IDE; shows call history, method calls and object context; supports searching and stepping.</td>
</tr>
<tr>
<td>Trace (Alsallakh et al., 2012)</td>
<td>a plugin for Eclipse IDE; a timeline represented as line chart with breakpoints on the line.</td>
</tr>
<tr>
<td>EXTRA VIS (Cornelissen et al., 2008)</td>
<td>a prototype presents a program as an element in the circle with lines that represent relationships between classes and packages; and uses sequence diagram to overview events.</td>
</tr>
<tr>
<td>Memview (Gries et al., 2005)</td>
<td>an extension to the DrJava IDE; depicts call stack, static objects in heap, normal objects in heap in distinct boxes.</td>
</tr>
<tr>
<td>CoffeeDregs (Huizing et al., 2012)</td>
<td>supports multithreading, but it is mostly a teaching tool for object-oriented study.</td>
</tr>
<tr>
<td>JavaVis (Mehner, 2002)</td>
<td>a UML-based application; uses sequence diagram for time line and collaboration diagram for deadlock detection.</td>
</tr>
<tr>
<td>JAVAvis (Oechsle and Schmitt, 2002)</td>
<td>it supports multithreading; shows stack call; uses a sequence diagram for a time-line and parallel threads.</td>
</tr>
<tr>
<td>EVitor (Moons and De Backer, 2013)</td>
<td>is a plugin for Netbeans IDE. Advantage of this application is textual tips for the user with explanations.</td>
</tr>
<tr>
<td>JavaTool (Mota et al., 2008)</td>
<td>is a plugin for Moodle. It can be useful for small programs. Debugging occurs in the browser.</td>
</tr>
<tr>
<td>Labster (Juett, 2016)</td>
<td>a web-based system for visualization of memory representation and expressions evaluation.</td>
</tr>
<tr>
<td>Project S (Deitz and Buy, 2016)</td>
<td>a tool has graphical interface based on “Space Invaders”. The aliens are variables. Each variable has a text label.</td>
</tr>
<tr>
<td>Web-based tutor (Kumar, 2009)</td>
<td>a tool depicts memory representation of C++ code: global variables, stack and heap. In addition, this tool has a detailed explanation related to a current line.</td>
</tr>
<tr>
<td>VIP (Virtanen et al., 2015)</td>
<td>a tool explains how pointers work in C++ and demonstrates the process of expression evaluation.</td>
</tr>
<tr>
<td>Bradman (Smith and Webb, 1995)</td>
<td>an extended debugger which explains execution of each statement.</td>
</tr>
<tr>
<td>Teaching Machine (Bruce-Lockhart and Norvell, 2000)</td>
<td>a tool shows a stack, a heap and static memory. Special table includes a list of all variables: type, name and value.</td>
</tr>
<tr>
<td>jGRASP (Cross et al., 2004)</td>
<td>an IDE. It can show visualization of data structures, objects, instance variables.</td>
</tr>
</tbody>
</table>
return”), but also to be able to return to previous states at any moment.

The proposed architecture makes the development process scalable. Uniform representation of an intermediate JSON file precisely defines data to be extracted from debugger. A shared format optimizes development of the visualization subsystem. Indeed, instead of developing three different visualization modules we need to develop only one. In the future plugins for another IDE (or even other languages) can be easily developed and integrated in our architecture.

3.2 User Interface

Technically, our application is an extension of an IDE (Java or Eclipse), which interacts to a built-in debugger. In a basic scenario user puts a breakpoint somewhere in the source code and then steps forward and backward, observing execution states. User interface of the tool is presented in Figure 2.

The user interface consists of: (a) IDE standard window; (b) source code editor, which also highlights current execution line, and its breakpoints managing functional; (c) standard debug control buttons (e.g., "step into", "step over", "step return") of IDE; (d) view tab with all visualization tables along with additional buttons for back-stepping, and visualization preferences. The visualization model of the application is illustrated with an example. To this end we use the following simple Java program.

Figure 3 shows a heap and a stack state in the breakpoint. Inside the stack one can see arguments and local variables. Description of variables includes the following fields: type, name and value. For simplicity, names of standard classes are shown without prefix java.lang (e.g., we show String instead of java.lang.String). In addition, we show only user’s objects in the heap (only those objects, which have reference to them in the stack). Otherwise, the heap visualization might be littered with numerous system objects. C++ visualization has a very similar structure, but it also contains global/static variables block, memory addresses of local variables, and memory addresses of heap objects instead of identifiers.

Listing 1: Example of a Java program visualized in Fig. 3

```java
public class Sample {
    public static void main(String[] args) {
        Demo obj = new Demo();
        obj.i = 70;
        obj.c = 'Z';
        int a = 5;

        int b = obj.i;
        String s = "Hello";
    }
}
```

```java
class Demo {
    int i;
    char c;
}
```

During the execution process each new stack frame appears on the top of the stack, and old frames move down. When a function finishes its execution, the stack frame of the function is removed from the stack. New heap objects or static variables appear on the top of the heap and static/global memory areas. Thus, even if user will work with a large program, visual representation will grow only vertically. This means that our approach allows users to effectively observe all the information about the program state using scrolling, and to have recent data always on the top. In addition, a user can customize the application. There is a possibility to automatically minimize all stack frames excluding the upper one and manually minimize or maximize any block (i.e., heap, global/static memory, stack or distinct stack frames). A variable or an object which was changed/created during the last step is highlighted. The field “name” is added into a table which represents the heap. This facilitates understanding the relationship between pointers/references in the stack and objects in the heap.

4 CONCLUSION AND FUTURE WORK

This article presents a new solution for visualization of program execution. Right now we have available only prototypes, but soon we are going to develop a working version and to test it.

The prototypes are plugins that allow us to monitor memory content of programs during execution step by step and that will be released with an Open Source license (Kovács et al., 2004). It would be fruitful to pursue further research about including a timeline and textual explanations. If a timeline is shown as a sequence diagram, then we will be able to depict multithreading in our application. We have considered several advantages and disadvantages of existing visualization systems. Thus, we are going to gather some major advantages in one solution and eliminate flaws. So that novice programmers will obtain a powerful tool for understanding how programs execute and how memory is typically organized.
ACKNOWLEDGEMENTS

We thank Innopolis University for supporting this research.

REFERENCES


Doe, P. (2016). Java heap space vs stack – memory allo-