CTutor

Tiago Alexandre Afonso Aguiar

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Nuno de Oliveira e Silva

Examination Committee

Chairperson:	Prof. João Emílio Segurado Pavão Martins
Supervisor:	Prof. João Nuno de Oliveira e Silva
Member of the Committee:	Prof. David Manuel Martins de Matos

November 2014

Resumo

CTutor é uma ferramenta de visualização de programas para a linguagem C.

O CTutor permite melhorar a compreensão dos vários elementos da linguagem de programação C, tais como tipos de variáveis, argumentos de funções e os seus outputs. Com isto, o CTutor pretende facilitar o processo de aprendizagem desta linguagem.

Uma ferramenta de visualização de programas também é útil como complemento do processo de aprendizagem de uma nova linguagem de programação. Através das visualizações disponibilizadas, consegue-se substituir os desenhos ou as imagens feitas em diapositivos pelos professores. No caso de C, estas imagens são normalmente usadas para explicar algoritmos de ordenação, ponteiros ou estruturas. Com o CTutor, as visualizações são construídas a partir do código-fonte, desenvolvido pelo estudante ou pelo professor. Este código-fonte é executado pelo *backend* do CTutor (ambiente de execução), por sua vez as visualizações são mostradas no *frontend* (visualizador).

Para além de afectar os estudantes e os professores ao longo do processo de aprendizagem, o CTutor foi desenvolvido de maneira diferente das demais ferramentas de visualização.

As ferramentas de visualização são compostas por um ambiente de execução e um visualizador, em que estes dois componentes estão fortemente ligados, dificultando assim o desenvolvimento de novas ferramentas que reutilizem um destes componentes. Usando esta mudança arquitectural demonstrada através do CTutor, é possível usar o ambiente de execução do CTutor e construir outro visualizador, resultando numa nova ferramenta, e adaptando-o às necessidades dos estudantes. Por outro lado, pode-se modificar o ambiente de execução para outra linguagem e usar o visualizador já disponibilizado, resultando também numa nova ferramenta.

Palavras-chave: Aprendizagem, Visualização, Estudantes, Linguagem C, Arquitectura

Abstract

CTutor is a program visualization tool for the programming language, C. As the name implies, CTutor serves as tutor for learning C. The common objective of program visualization is to enhance the comprehension of the several elements of a certain programming language, such as types of variables, function's arguments and outputs.

Program visualization tools are an useful complement to the learning process of a new programming language, replacing the typical images drew on blackboards, or the figures made in PowerPoint slides. In case of C, these images are often used to explain sorting algorithms, pointers and structures. With CTutor, the visualizations are produced from the source code, written by the student or the professor. This source code is executed by CTutor's backend (execution environment) and the visualizations are shown in the frontend (visualizer).

CTutor also presents a change in the the typical architecture of program visualization tools. Program visualization tools are composed by an execution environment and a visualizer, and usually are strongly coupled, thus hindering the creation of new tools that reuse one of these components in another tool. With this major change in the architecture, it is possible to use CTutor's execution environment and create a new visualizer and adapting it to student's needs. Or, it is possible to change the execution environment to execute another programming language and use it with the CTutor's visualizer.

Keywords: Learning, Visualization, Students, C language, Architecture

Contents

Resumo . . . iii
Abstract . . . v
List of Tables . . . ix
List of Figures . . . xii

1 Introduction **1**
1.1 Program Visualization . . . 2
1.1.1 Visualizer . . . 3
1.1.2 Execution Environment . . . 3
1.2 Target population . . . 3
1.3 Objectives . . . 4
1.4 Results . . . 4
1.5 Document Structure . . . 4

2 Related Work **7**
2.1 Visualization Tools . . . 7
2.1.1 Program Visualization Tools . . . 7
2.1.1.1 The Jeliot Family . . . 7
2.1.1.1.1 Eliot . . . 7
2.1.1.1.2 Jeliot I . . . 8
2.1.1.1.3 Jeilot 2000 . . . 10
2.1.1.1.4 Jeilot 3 . . . 12
2.1.1.2 CSmart . . . 13
2.1.1.3 Online Python Tutor . . . 14
2.1.1.3.1 Others Tutors . . . 16
2.1.2 Visual Debuggers . . . 17
2.1.2.1 DDD . . . 17
2.1.2.2 JIVE . . . 19
2.1.3 Comparisons . . . 21
2.2 Execution Environment . . . 21
2.2.1 Instrumentation . . . 21

2.2.2 Interpreter . . . 21
2.2.2.1 PicoC . . . 21
2.2.3 Debugger . . . 23
2.2.3.1 GDB . . . 23
2.3 Integration with Visualizer . . . 24
2.4 Complementary Tools . . . 26
2.4.1 Learning Environments . . . 26
2.4.1.1 Eclipse . . . 26
2.4.1.2 NetBeans . . . 26
2.4.2 Portability . . . 26
2.4.3 Conclusions . . . 27

3 CTutor **29**
3.1 Architecture . . . 29
3.2 Implementation . . . 30
3.2.1 Trace . . . 34

4 Extended Python Tutor **39**

5 Solution Evaluation **43**

6 Conclusions **47**
6.1 Future Work . . . 48

Bibliography **52**

List of Tables

2.1 Comparison between Visualization tools 22
2.2 Comparison between solutions to Execution Environment 25

5.1 CTutor's supported elements of language C 43

List of Figures

1.1 Forms of software visualization . . . 3

2.1 Eliot windows . . . 9
2.2 The phases of animation generation in Eliot . . . 9
2.3 Jeliot I user interface, with a representation of the Tower of Hanoi's algorithm . . . 10
2.4 The structure of Jeliot I . . . 11
2.5 Jeliot 3's user interface . . . 12
2.6 The functional structure of Jeliot 3 . . . 13
2.7 CSmart's example . . . 14
2.8 Online Python Tutor's user interface . . . 16
2.9 DDD user interface . . . 18
2.10 DDD Communication . . . 19
2.11 JIVE's user interface, object diagram and sequence diagram . . . 20
2.12 GDB structure . . . 23
2.13 The three different methods to build a program visualization tool . . . 25
2.14 Interactions among the components of Nacl . . . 28

3.1 The three different methods with CTutor . . . 30
3.2 The architecture of CTutor . . . 31
3.3 User Interface of CTutor . . . 32
3.4 Interactions between components of CTutor with Browser . . . 33
3.5 Interactions between components of CTutor with GUI . . . 34
3.6 Program to explain trace . . . 35
3.7 Visualization corresponding to trace 1 . . . 36
3.8 Visualization corresponding to trace 2 . . . 37
3.9 Visualization corresponding to trace 3 . . . 37
3.10 Visualization corresponding to trace 4 . . . 38
3.11 Visualization corresponding to trace 5 . . . 38

4.1 The architecture of Extended Python Tutor . . . 39
4.2 Interactions between components of Online Python Tutor . . . 40
4.3 Interactions between components of Extended Python Tutor . . . 41

5.1 Example of a multidimensional array 44
5.2 Example of a pointer to a char 44
5.3 Example of a pointer to an array of chars 45
5.4 Example of a struct and a pointer to it 45

Chapter 1

Introduction

This document introduces CTutor, a program visualization tool for the C programming language [1].

Nowadays, each person can learn programming in three different ways: self learning, e-learning or a programming course in a school. As a student, the learning process of a new programming language follows an approach bottom-up, starting with simple examples (”Hello World”) with the primitive data types and then the complexity increases gradually.

In terms of tools used in the learning process, the student starts with a simple editor to write the source code and when he is more comfortable with the new learned language, is common to use an integrated development environment (IDE), such as Eclipse [2] or NetBeans [3].

The described process is also complemented with some graphical illustrations for certain examples. These images are used with the objective of clarifying some doubts about the use of some complex types (e.g. Lists, Arrays, Dictionaries) or some paradigms and concepts (e.g. object-oriented with inheritance).

C is one of the most popular programming languages [4] and is one of the first languages taught to students. This fact also contributes to increase its popularity and it is largely used all over the world.

CTutor aims to be a widely used tool for program visualization. The common objective of this kind of tools is to help people better understand programs. These tools can be used by a beginner of programming or an expert and help users in the creation of mental models to easily associate the several elements of this language.

With this objective CTutor will be capable of fulfill a gap in the existing options to visualize programming languages in C, because the existing tools for this language are outdated.

Another goal of CTutor is to enhance the architecture of the program visualization tools.

The architecture of a typical program visualization tool uses an execution environment and a visualizer, and these two components are strongly coupled. Which means it is hard to separated them and reuse them to build other tools with same purpose. For example, if a professor needs a new visualizer for the program visualization tool, which he uses in his programming course, he will need to develop a new tool besides only a new visualizer.

A visualization tool that can reuse other visualizers, can present the elements of the language in different manners. This reuse of components makes possible adaption to the student’s or professor’s

needs. Also, changing the execution environment helps the student, which is familiarized with a specific visualizer, to use it with different programming languages.

CTutor will be based on an existing tool called Online Python Tutor [5]. This objective of CTutor is supported by a new architecture compared to the existing architectures in other tools.

The development of CTutor had two phases: first one was modifying Online Python Tutor's architecture, to split the visualizer and the execution environment; second phase was to develop a new programming tool to visualize C.

So, the first step of the implementation was splitting the visualizer and the execution environment into two separated components. This step started with the definition of the messages exchanged between those components, and the creation of a new component that will be the bridge of communication between them.

The second one was to analyze GDB/Machine Interface, to know how to interact with it and gather the information needed by the visualizer. This will help the development of the new execution environment that executes C programming language.

The following sections will introduce some concepts about a program visualization tool, such as the components, the target population, and the objectives of CTutor.

1.1 Program Visualization

The visualization of programs is increasing its use in the learning process, and with this fact tools are starting to be developed or upgraded the existing ones. In the process of learning, the tools that provide visualization are very useful to visualize algorithms (e.g. sorting algorithms) or programming examples (e.g. recursive functions). The visualization facilitates the comprehension of the several elements of a language.

According to Juha Sorva, in his PhD dissertation [6] from Aalto University, there are important concepts about program visualization. Figure 1.1 presents the various form of software visualization. First there is software visualization, which has the two subfields: program and algorithm visualization. Algorithm visualization systems visualize general algorithms, usually using a high level of abstraction (e.g. sorting algorithms, graph algorithms, binary tree operations), on the other hand program visualization systems visualize concrete, implemented programs and in a lower level.

Program visualization is divided in two areas, some visualizations represent program code (e.g. dependencies or code evolution), while others illustrate the runtime dynamics of programs. Also within program visualization, there are two concepts:

- Program animation refers to dynamic visualization of program dynamics – the common variety of program visualization in which the computer determines what happens during a program run, and visualizes this information to the student (e.g. as in a typical visual debugger).
- Visual programming attempts to find new ways of specifying programs using graphics rather than visualizing software that is otherwise in non-visual format.

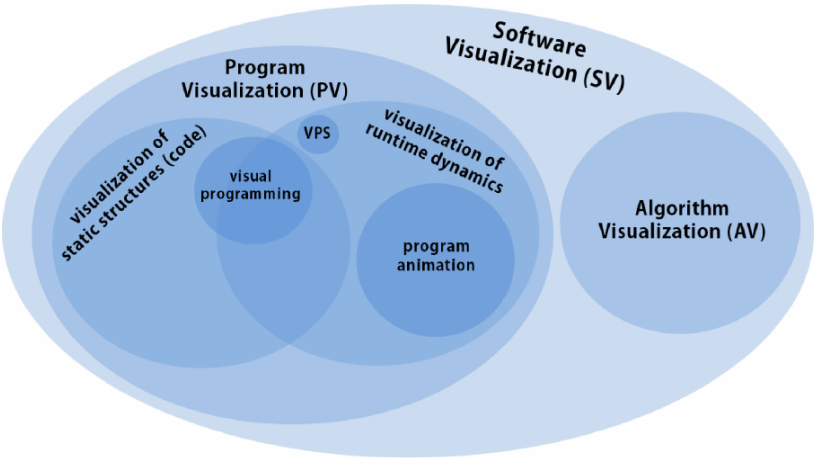

Figure 1.1: Forms of software visualization [6]

1.1.1 Visualizer

The visualizer is considered to be the part of the program visualization tool that shows to the student a graphic visualization of its own program. It is also responsible to know how to represent the different elements of a programming language.

1.1.2 Execution Environment

The execution environment is the part of the tool that executes the student's code. Additionally, it has to provide to the visualizer the several modifications that the elements suffer throughout the code. For example, if a variable change its value the execution environment has to expose this modification to the next component.

1.2 Target population

CTutor's users can be divided into three different entities, which have different goals when interacting with program visualization tools.

- The **student's** purpose is to use CTutor as a learning tool, using its visualizations to improve their knowledge about a certain programming language.
- The **professor** will use CTutor to teach a programming language, with the use of programming examples. With CTutor the teaching is more interactive than drawings on the blackboard or on slides.
- The **developer's** goal is extending or developing a new visualization tool. CTutor helps this actor with is new approach in terms of architecture of a program visualization tool.

1.3 Objectives

CTutor will be focused on program animation, which is the use of dynamic visualization techniques to enhance student understanding of the actual implementation of programs or data structures. This means that the student may examine the state of the program and related data, before and after execution of a particular line of code. To accomplish that CTutor defines two main objectives:

- As a program visualization tool, CTutor has the purpose of helping students of programming in learning or improving theirs programming concepts. A good analogy is a slow motion movie, because the source code of a program is transformed into several and sequential images of the operations or expressions represented in the code. The generated visualizations allows the student to analyze their code and understand precisely what the source code is doing in a certain line. Also, the student can assimilate the several components of a programming language and then create/improve his own mental models and reach the point where, in his mind, he can anticipate the next visualizations provided by the visualization tool.

- CTutor aims to facilitate the developing of new program visualization tools, which can be achieved with a different approach on the architecture of these tools. The developer can analyze the messages exchanged between the visualizer and the execution environment and then substitute one of them, or both. He can be concerned to develop only one component, and then the result will be a new tool, with different visualizations (in case of developing a new visualizer) or with a new language to be visualized (in case of developing a new execution environment). This fact will provide new program visualization tools and enable the customization of these tools with different visualizers to fulfill different requirements.

1.4 Results

This dissertation has two tools as outputs: a new program visualization tool for C programming language, CTutor; an improved version of Online Python Tutor, Extend Python Tutor (EPTutor).

1.5 Document Structure

This document is structured in the following chapters:

- **Chapter 2** exposes the related work, compares some program visualization tools and visual debuggers. Also, analyzes the three different methods that can be used as execution environment and explains what is necessary to be done to integrate with a visualizer. In the end, it is presented two learning tools that can be used with CTutor and a solution to the portability of CTutor.

- **Chapter 3** presents the architecture and explains some details about the implementation of CTutor.

- **Chapter 4** introduces the Extended Python Tutor, and compare it to the Online Python Tutor.

- **Chapter 5** performs the evaluation of the proposed solution. The evaluation is based on program examples taken from a book used as manual in programming course of C.
- **Chapter 6** concludes this document. It summarizes the developed work and introduces further work directions.

Chapter 2

Related Work

The intent of this section is to introduce the state of the art about program visualization tools. This subject is not a new research field, the first reference appears on the 1950's in an article about software visualization [7].

This section will be divided into three main topics: first one will present some program visualization tools and visual debuggers; then will introduce some tools that can be used as execution environments for language C. To support program visualization, the last topic exposes some complementary tools that could be useful to match with CTutor.

2.1 Visualization Tools

2.1.1 Program Visualization Tools

The tools presented in this section are program visualization tools, they are widely used with the purpose of help students to improve his knowledge about programming.

2.1.1.1 The Jeliot Family

The Jeliot family [8] consists of four program animation environments: Eliot [9], Jeliot I [10], Jeliot 2000 [11], and Jeliot3 [12] which have been developed in order to improve the teaching and learning of computer science, in particular, programming.

Study the evolution of this tool, was important for recognize some problems and understand the architectures used in each tool.

2.1.1.1.1 Eliot

Eliot was the first member of the family; it appears in 1996 as an interactive animation environment, which visualizes algorithms written in the C programming language, can be used in algorithm design, visual debugging and learning programming.

Eliot has its own programming language, called Eliot-C, and it is a dialect of the C programming language accepted by Eliot compiler. This dialect added some features to the standard data types of C, like animated array and tree data types, with that was needed some new operations to manipulated them. But, with that change of dialect it was imposed three limitations to Eliot-C:

- Programs cannot consist of more than one module;
- Animated variables cannot be manipulated with operations like ++ and – that increase or decrease the value of a variable with one;
- Pointer variables cannot be animated.

To animate an algorithm, the student writes it in Eliot-C, applying the visual data types of the Eliot library. Then, the student decides which data objects should be animated and selects the visual appearance of each object. After that, Eliot generates an executable program from the input code and produces the animation as an intentional side-effect.

Eliot supports multiple animation windows called stages, which can be displayed simultaneously. The selection of the animated variables and their characteristics on each stage is independent. Also, provides a feature to store the selection of variables and their visual parameters for later use.

This tool, with the objective of helping the viewer to focus on different aspects of the data in use, allows the student to define the attributes independently for each data object on each stage, see Figure 2.1. For example, it is possible to display the components of an array as numbers on one stage and as vertical bars on another.

Eliot has been implemented in the X11 environment and it runs on Linux workstations, and consists of three components:

- A library of visual data types that uses the services of the Polka package [13];
- A graphical user interface, has been implemented using the wxWindows library [14];
- A compiler, has been generated with lex and yacc [15]. Also contains parsing and code generating modules, and is called by the user interface whenever necessary.

The Eliot's process of constructing an animation, see Figure 2.2, is the following the parser constructs a parse tree and default appearance parameters from the algorithm. The student can modify the appearance parameters. The code generator produces a C++ source program, and then the resulting animation program is compiled and linked with the Eliot library.

2.1.1.1.2 Jeliot I

The main purpose of Jeliot I, was to allow a Web user to visualize his own algorithms, written in Java. Because porting Eliot would have been more difficult, the team decided to develop Jeliot I that is a similar system for World-Wide Web. This system is for animating Java programs and was finished, one year later than Eliot, in 1997.

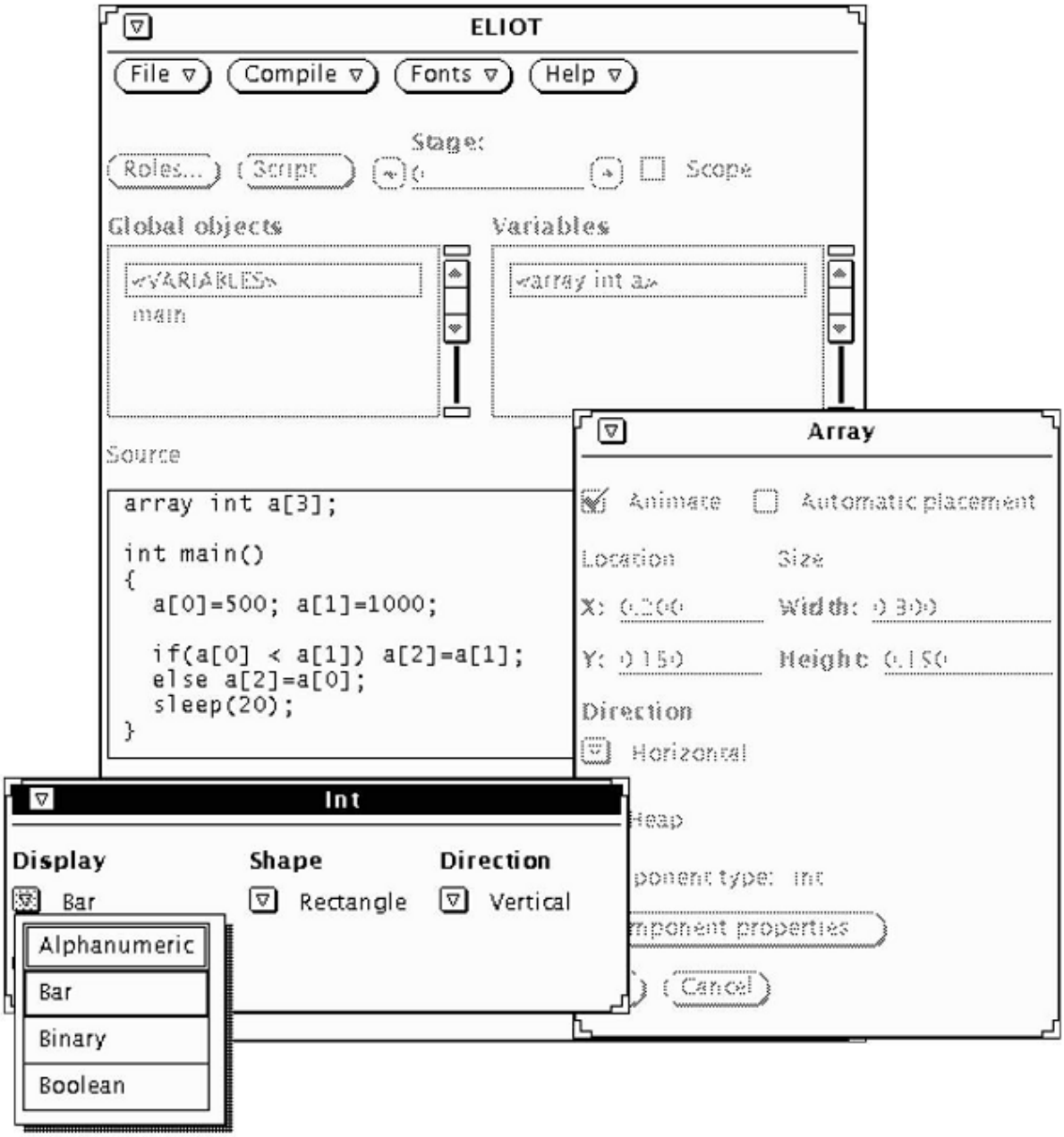

Figure 2.1: Eliot windows. The big main window let the student chose which variables to animate. The other two are used to set up the visual properties of the animated variables [9]

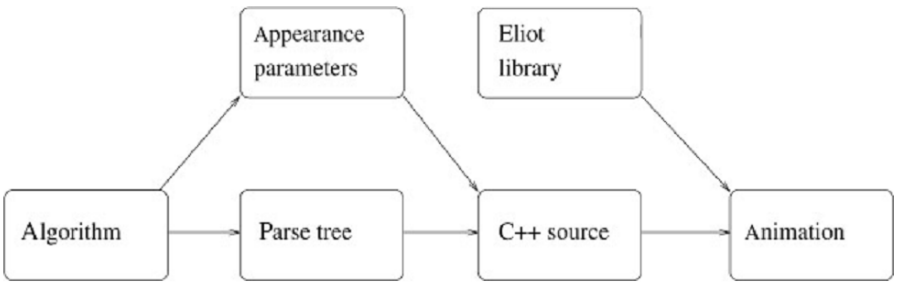

Figure 2.2: The phases of animation generation in Eliot [9]

The interaction with the students is over an internet page, he writes a Java program in a text field of a Web page, submits it, and gets back its animation. The animation, on a server, is generated automatically from the source code, and it is displayed on the student's screen. The model behind this interaction is a client-server, which is an example of a general framework where a Web server calls the compiler: the student inputs the source code or a specification to the server which returns a Java applet or other executable program to the student. The user interface of Jeliot I is shown in Figure 2.3.

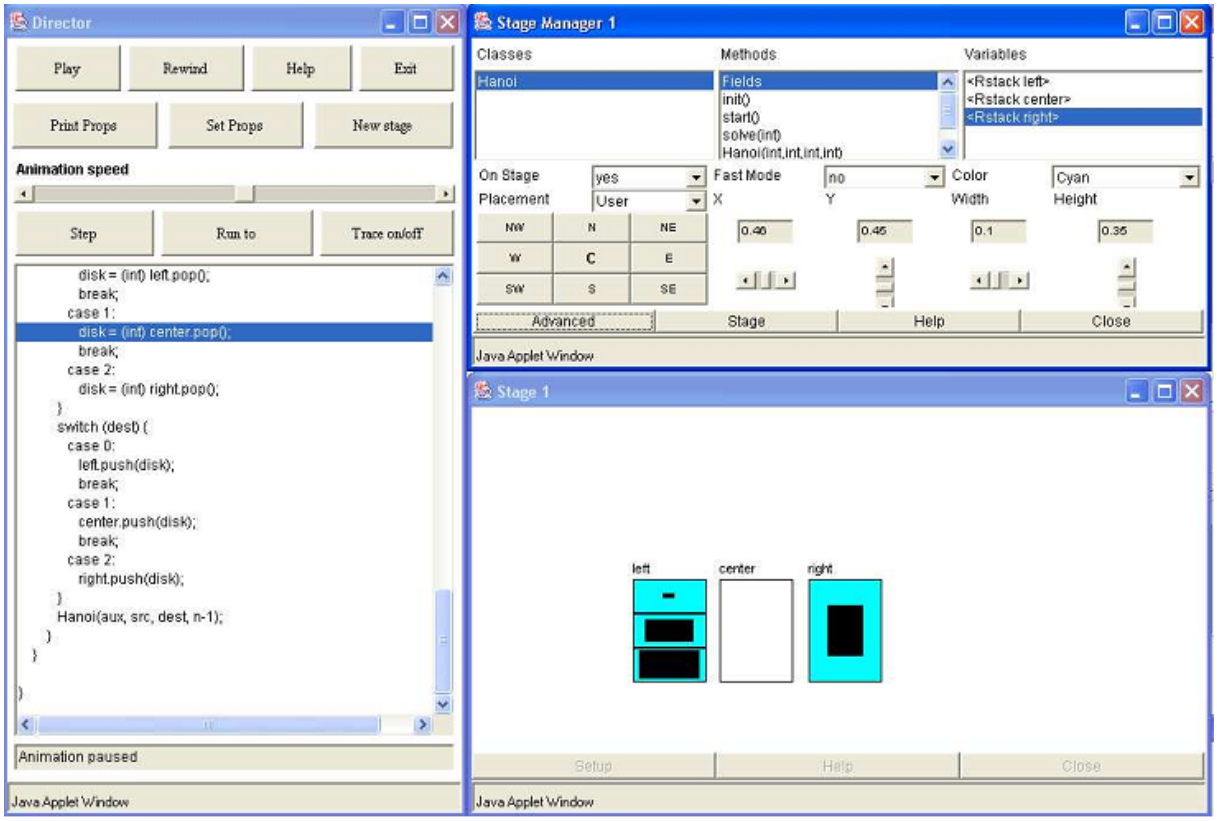

Figure 2.3: Jeliot I user interface, with a representation of the Tower of Hanoi's algorithm [10]

Besides the use, design and the operational framework of Jeliot is similar to those of Eliot, the technical design is completely different because the Web environment requires an architecture of another kind and also the different supported programming language, see Figure 2.4. As an example of these differences, the Polka library is not any more used instead; graphical primitives were implemented in using the standard Java libraries.

2.1.1.1.3 Jeilot 2000

Basically Jeliot 2000 is an upgrade of Jeliot I, in the terms that it changed the focus to helping total novices instead of assuming that the student has some familiarity with programming, and also the design has a simpler interface than his predecessor. The two principles that guided the design were completeness and continuity. The following examples explain these principles:

- Every feature of the program must be visualized; i.e. the use of a constant is animated by having the constant move from an icon for a constant store.

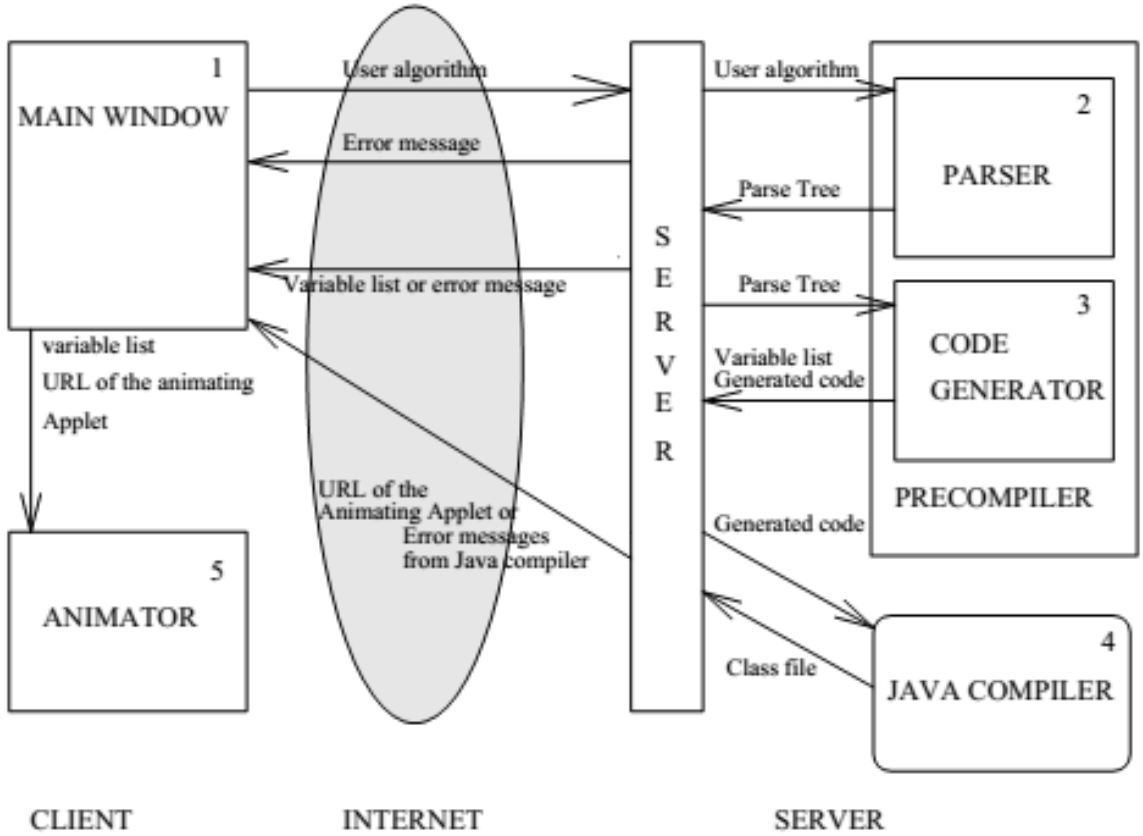

Figure 2.4: The structure of Jeliot I [10]

- The animation must make the relations between actions in the program explicit; i.e. the animation of the evaluation of an expression includes the animation of the evaluation of sub expressions.

The architectural driver of Jeliot 2000 was usability, with this concern a new life was given to the GUI. The final result was the inclusion of two panels: one for the source code, and another for the animation. The execution of the program is controlled through familiar VCR-like buttons (play, stop, rewind, etc). This tool has an explanatory feature about the evaluations made in the code, because they are essential for novice programmers.

To implement an animation, the source program must be interpreted; i.e. the animation display has to be a side effect of the interpretation. Animating JVM byte code would lose a layer of abstraction, so Jeliot 2000 directly interprets the program source code. The source code is parsed to produce a tree that has the declarations, statements, and expressions of the program as nodes. When the program is run, Jeliot 2000 traverses this tree, keeping track of the program's state and rendering the events of control flow and expression evaluation in the animation.

Jeliot 2000 is written in Java like is predecessor, but unlike the Web-based client-server architecture of Jeliot I, it is structured as a single application for simplicity and reliability.

2.1.1.1.4 Jeilot 3

Jeliot 3 is the last member of the family and it is the successor of Jeliot 2000 system, previously presented. This tool is currently being used in some Finnish and Israeli high schools and universities.

The Jeliot 3 user interface, identical to Jeliot 2000 and presented in Figure 2.5, is composed with two panels: on the left panel, the code and on the right, a larger panel where animation is shown. This larger panel, is also called theater (it is the visualizer of this tool), is composed with:

- method area, where method frames containing variables are stacked;
- constant area, where constants and static variables are displayed;
- object area, where objects and arrays are allocated and referred by the variables in the method area;
- expression evaluation area, where expressions are evaluated.

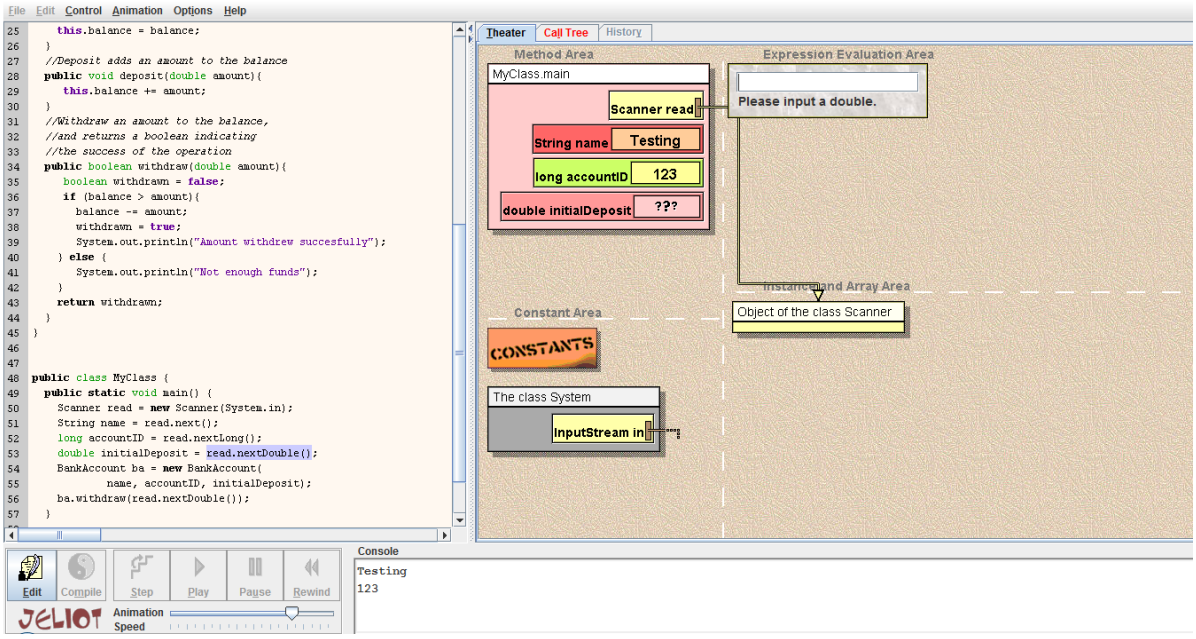

Figure 2.5: Jeliot 3's user interface

Instead of improving usability like in the transition of Jeliot I into Jeliot 2000, now the two main reasons are: to incorporate Object-Oriented support and to redesign the software architecture for better modularity and extensibility. To obtain that features, a complete redesign was needed. The interpretation and animation of Java programs are separated into two modules in Jeliot. For the former, DynamicJava [16], an open source Java interpreter, processes the student program. In the latter module, Jeliot's graphical engine creates an animation of the program interpretation. To connect these parts, an intermediate code, MCode [17], was designed.

MCode is a textual representation of the interpretation of a running program, or a program trace. It not only describes the changes in variables and stacks, as a normal Java debugger would do, but it also details the operations that produced those changes. All this information is required to animate

every step in the execution of a program. At present, there are two MCode interpreters in Jeliot 3: One that produces the complete animation of the program and another one that builds the method calls tree during the execution. Program animation designers can build their own animations and integrate them into Jeliot 3 by writing a new MCode interpreter. MCode aims to be language independent. For example, this version of Jeliot has integrated a Python interpreter to produce MCode for Python programs, which can be now visualized with Jeliot 3.

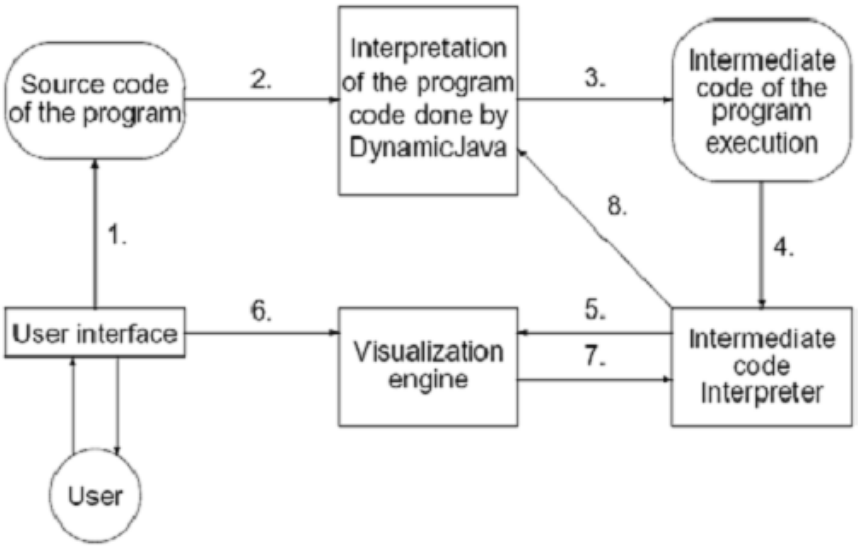

Figure 2.6: The functional structure of Jeliot 3 [18]

Figure 2.6 illustrates the functional structure of Jeliot 3. The meaning of the several interactions is:

- A student interacts with the user interface and creates the source code of the program (1).
- The source code is sent to the Java interpreter and the intermediate code is extracted (2 and 3).
- The intermediate code is interpreted and directions are given to the visualization engine (4 and 5).
- The student can control the animation by playing, pausing, rewinding or playing step-by-step the animation (6).
- The student can input data, for example, an integer or a string, to the program executed by the interpreter (6, 7 and 8).

2.1.1.2 CSmart

CSmart [19], developed by Roger Gajraj at 2011, is a learner's integrated development environment (L-IDE) proposed to increase the pedagogical value of the source code of sample programs by transforming them into self-explaining tutorials.

CSmart takes a different approach to visualization, instead of visualizing the execution of an existing program, the focus in CSmart is on already visualizing each statement of a C program to the student before the student types it in. It emphasizes teaching by example where the syntax and the semantics

of a language are taught by presenting samples of annotated program code prepared by experienced teachers. Tutorials are created dynamically by parsing the example source code files which are then presented in a rich format that aids comprehensibility.

Figure 2.7 shows how a line of code is parsed into an informal instruction within CSmart's editor. Rendered code in CSmart is annotated with the comments added to the example source code by the authors of the tutorial, who are expected to ensure that the explanations are appropriate.

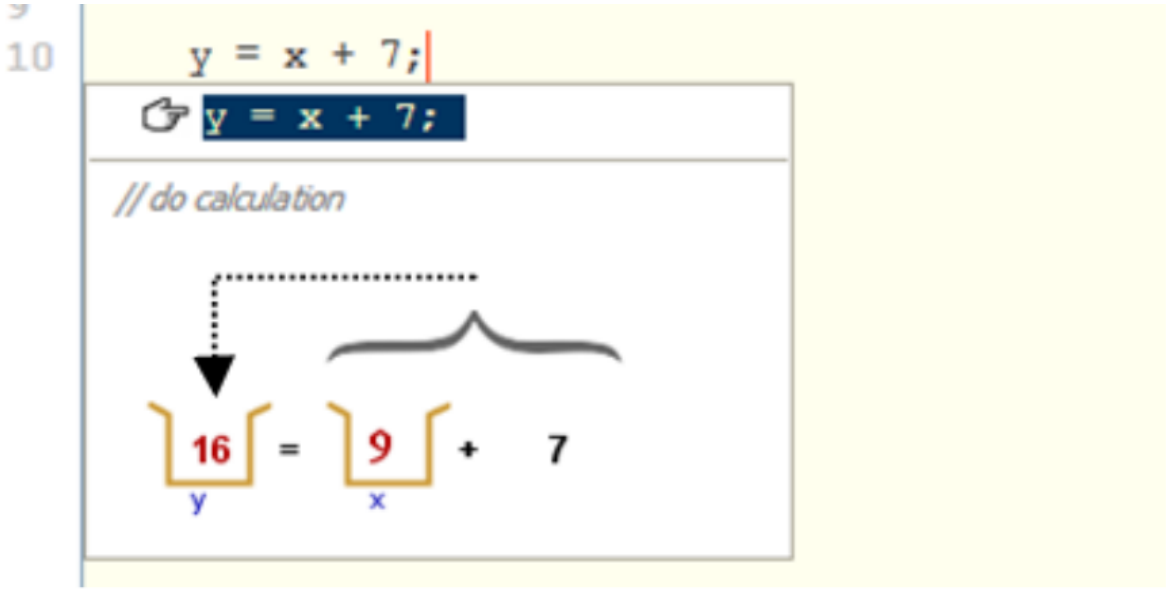

Figure 2.7: CSmart's example [19]

CSmart requires the examples to be manually annotated with syntactic and semantic information. This is required for every example created by the professor. The annotations added by the professor are confined to the example to which they are added. The IDE offers no provision for the professor to add these rules once and allow the learners to infer them across all examples when relevant. Students are limited to inferring the annotations provided by the professor with no options to instantiate and contextualize them.

2.1.1.3 Online Python Tutor

Online Python Tutor [5] is a web-based program visualization tool for the programming language Python, developed by Philip J.Guo. Using this tool, a teacher or student can writes a Python program directly in the web browser and visualize what the computer is doing step-by-step as it executes the program.

From 2010 to 2013, over 250,000 people in over 133 countries have used Online Python Tutor to understand and debug their programs, often as a supplement to textbooks, lecture notes, and online programming tutorials. This tool has reached some popularity among the program visualization tools used in a dozen universities such as UC Berkeley, MIT, the University of Washington, and the University of Waterloo. Also have been embedded within three web-based digital Python textbook projects [20, 21, 22].

Python is becoming a popular language for teaching introductory programming courses, to enforce

that idea MIT, UC Berkeley, Michigan State University switched their CS1 (First course of computer science) languages from Scheme or C++ to Python. Moreover, Online Python Tutor is one of the few tools for program visualization of this language.

Important features of this tool are: is the only Python program visualization tool that runs within a web browser without any required software or plugin installation, and also is the only that can be easily embedded within webpages.

The backend is written in Python and takes the source code of a Python program as input and produces an execution trace as output. The input is executed under supervision of the standard Python debugger module, bdb [23], which stops execution after every executed line and records the program's run time state. Then an ordered list of execution points is created, where each point contains the state right before a line of code is about to execute, including:

- The line number of the line that is about to execute;
- The instruction type (ordinary single step, exception, function call, or function return);
- A map of global variable names to their current values at this execution point;
- An ordered list of stack frames, where each frame contains a map of local variable names to current values;
- The current state of the heap;
- The program's output up to this execution point.

After execution ends, the backend trace is encoded in JSON format [24], serializing Python data types into native JSON types with extra metadata tags.

The Online Python Tutor frontend is a website hosted at `www.pythontutor.com`, an example of user's interface is presented in Figure 2.8. The student first insert the Python program into a text box. When the student clicks the button called *Visualize execution*, then interactions between frontend and backend start and are presented below:

1. The frontend sends the program inserted by the student to the backend;
2. The backend then sends back a JSON of all the execution trace;
3. For the last, the frontend renders in a GUI to the student.

Despite having a good visualizer , it is coupled to the execution environment. Also, Python Tutor is only web-based and it compiles all the source code before showing the visualization, so for each modification in the code Python Tutor will need to recompile and illustrate all the visualizations again. The visualizations are all made in the pre-execution, after that the visualizer has all the trace to do the visualizations.

With this pre-execution mode, an inconvenience arises. For example, when a program needs a user input at a specific line (**raw_input**). The visualization goes normally until that line, and then an input box

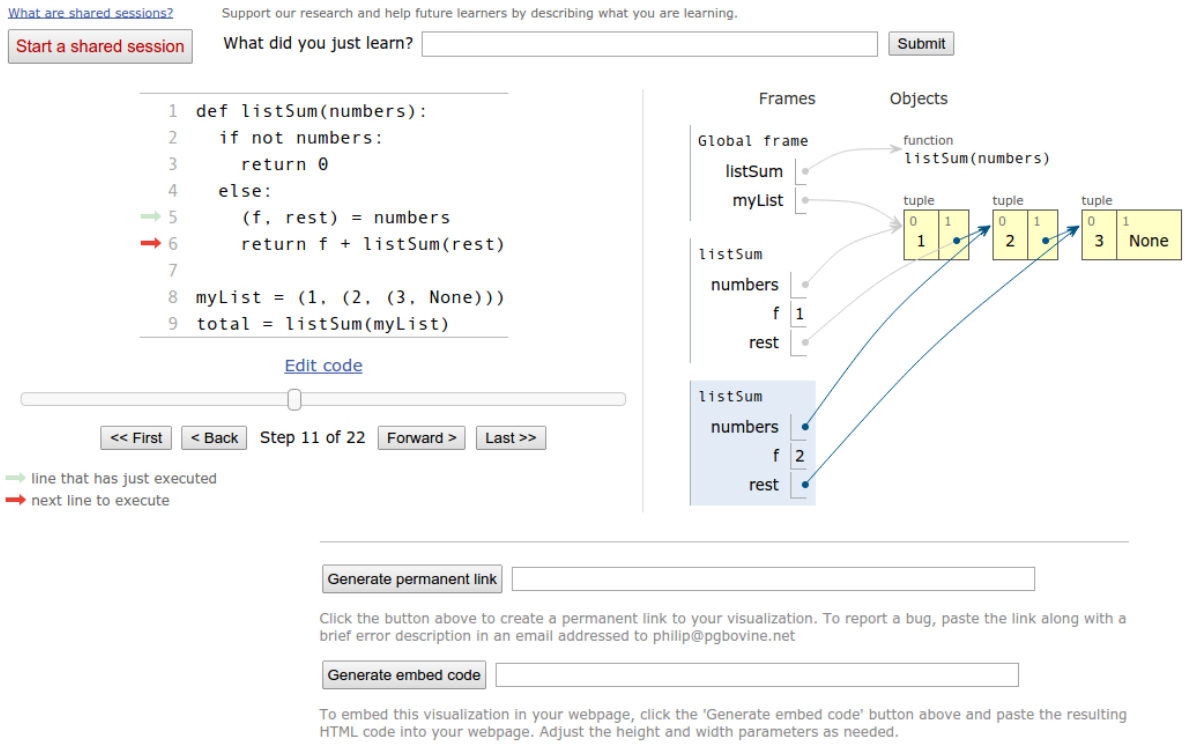

Figure 2.8: Online Python Tutor's user interface [5]

appears requesting for an input. After that, Online Python Tutor recompiles the program with this input, and starts over.

The explicit steps for this situation are:

1. The student inserts the source code in the frontend, and then is sent to the backend;
2. The backend executes the source code and sends to the frontend the program trace;
3. The student push forward in the visualization, and the frontend shows each visualization for each line;
4. When the next instruction needs the student input, the student inserts the input in a box;
5. The fronted sends all the source code again to the backend;
6. And the backend sends back a new program trace with the correct value for the raw_input;
7. The student begins the visualization from the start at this time there is no interruption in the raw_input.

2.1.1.3.1 Others Tutors

Based on Online Python Tutor, other tutors were developed, as Online Java Tutor [25], Online Ruby Tutor [26] and Online Javascript Tutor [27].

All Tutor's mentioned above are very similar to the Online Python Tutor in terms of visualization, but all of them are different in the backend and how they communicate with the visualizer.

The Online Java Tutor, uses *java_jail* [28] as backend, developed with the only purpose of creating the a Java version for Online Python Tutor. As a communication it is used a PHP script, with the intention of send the trace to the visualizer.

The Online Javascript Tutor, uses *Node.js* [29] as a backend and also uses it to send information to the visualizer.

The Online Ruby Tutor, is the one that has more differences in the visualizer, and as well in terms of backend. The backend is all in ruby.

These systems made major changes to be able to execute other languages. All these systems rebuild the backend in order of inspect each programming language, besides that they also changed the communication with the visualizer. The visualizer do not use the Google App Engine, to load it in the browser, instead the tutor's used other methods. Besides these modifications, these systems maintain the inconvenience existed in Online Python Tutor, all use a pre-execution state.

2.1.2 Visual Debuggers

A specialized debugging tool can significantly help to examine the dynamic behavior of a program. They provide support for inspecting the execution state in a symbolic way, and they allow for executing programs step-by-step or until a specific condition arises. Despite these functionalities, a graphical debugger also provides a display of the data structures, very useful for the user to follow and understand the execution flow of a program.

Unfortunately, the GDB [30] user interface is not suitable for casual and inexperienced users, since GDB recognizes more than one hundred of basic commands, each with its own set of options. Inexperienced users, as students, prefer a graphical user interface, emphasizing the most frequently used commands and providing direct manipulation facilities. To face that problem the debuggers turned to be more graphic, in other words the debuggers need to offer more visualization.

2.1.2.1 DDD

The Data Display Debugger (DDD) [31] is a graphical front-end for UNIX Debuggers. Despite the features of a debugger, like viewing source texts and breakpoints, DDD provides a graphical data display, where data structures are displayed as graphs.

DDD can run several debuggers, such as GDB, Perl Debugger (perl -d) [32], Bash Debugger (bashdb) [33], Python Debugger (pydb) [34], GNU Make Debugger (remake) [35], DBX [36], Ladebug [37].

DDD is presented to the user as a composition of three windows, as shown in the Figure 2.9, these are:

- The command window realizes the basic GDB command line interface, reflecting its editing and completion capabilities.
- The source window shows the source code as well as a breakpoint and the current execution position.

- The data window is the visualizer of DDD, is in this window where arbitrary symbols and expressions are displayed.

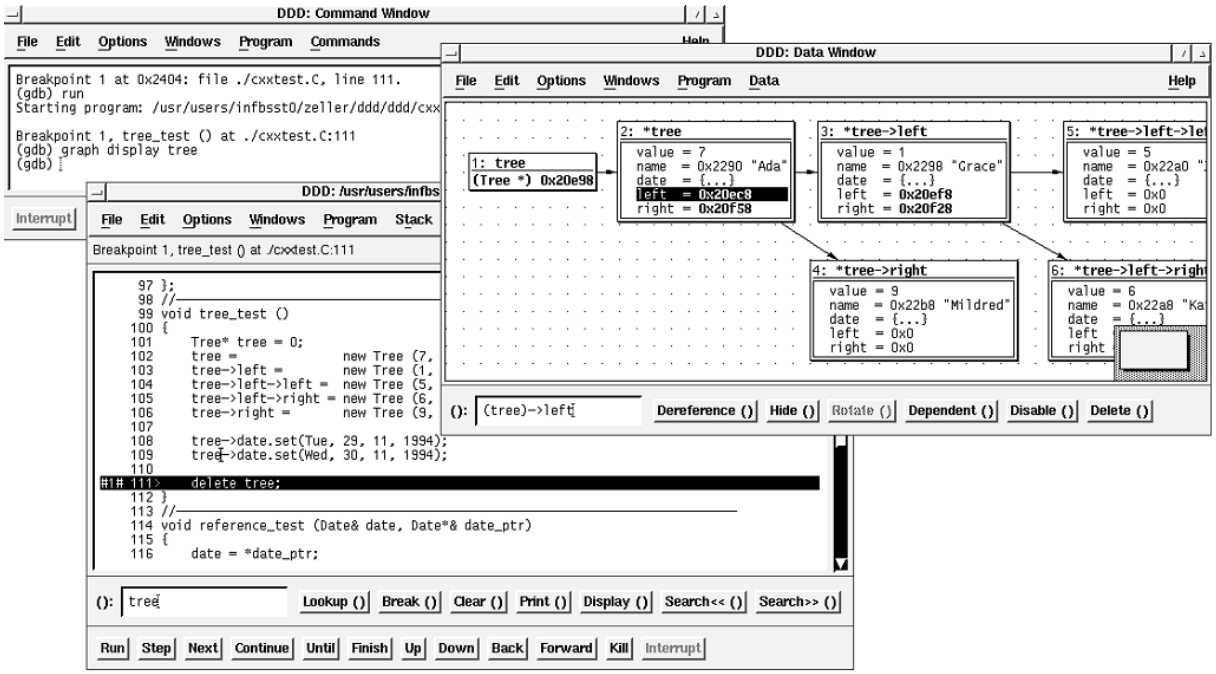

Figure 2.9: DDD user interface [31]

The Figure 2.10 illustrates the communication channels between the user, DDD, GDB, and the debugged process. The communication between DDD and GDB is all performed asynchronously, in order to minimize DDD response time. This feature is achieved with a command queue where it is placed the GDB command issues that are tagged with a callback routine and when the GDB output is available the callback routine processes it.

Another important aspect is the main event loop, which waits for user input, GDB output, and for GDB to become ready for input. When GDB is set to go, the next command is taken from the queue and sent to GDB. This asynchronous scheme has the advantage of avoiding that DDD is blocked while waiting for GDB output; incoming events can be processed any time.

The data window has several drawbacks for a beginner, the student has to explicitly open boxes to display information, for example the data type *date* inside of the structure tree represented in Figure 2.9, or he has to click directly on the code.

Another example is, if a student wants to visualize a certain behavior of the variable **x** located in *main* function, after he right-clicks it and chooses option **display**, he has another problem of visualization because this variable appear in data window without any link to the *main* function. Increasing the complexity of the problem, when a function *f* is call, for instance, from *main*, the environment (local variables) of the *main* function hides from the user.

In DDD function frames are not graphically represented, although they modify the way other data is represented.

This visualization features leads to a rejection from part of novice students and can also lead to flaws

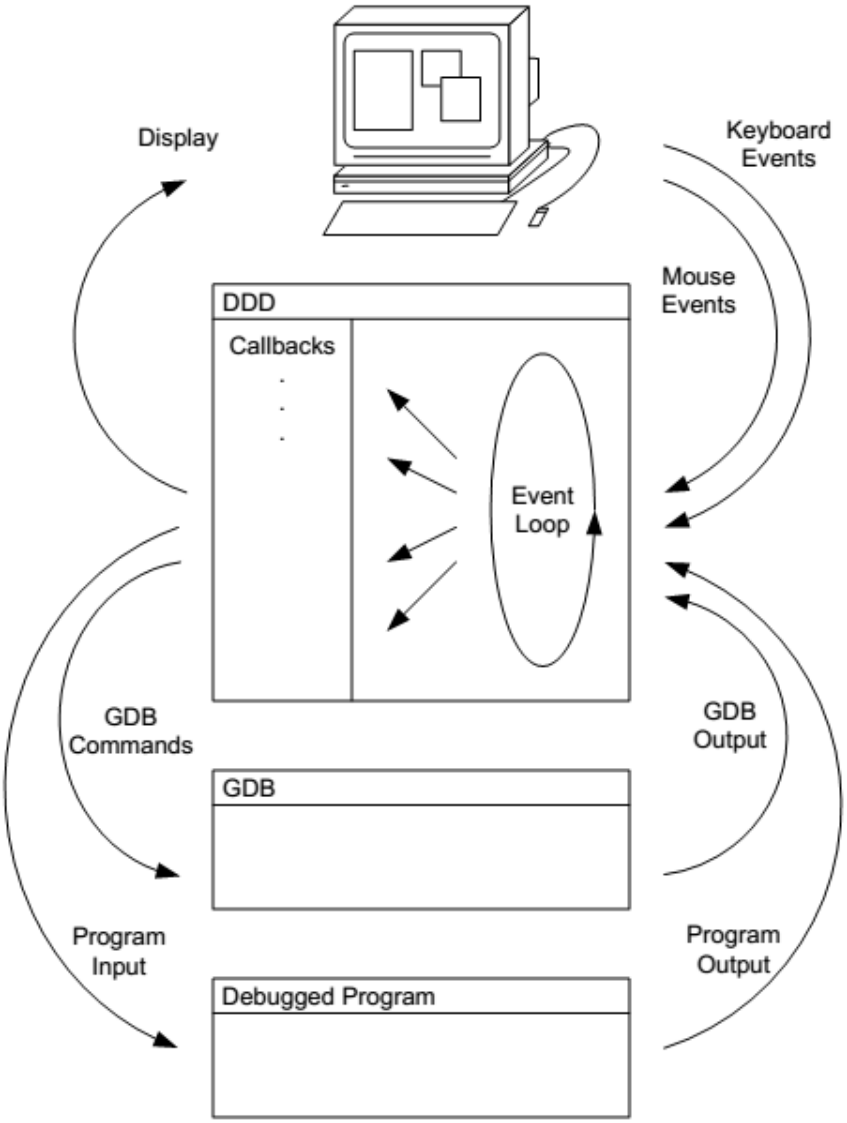

Figure 2.10: DDD Communication [31]

in the comprehension of simple things, such as definition of local scope of a function.

2.1.2.2 JIVE

JIVE (Java Interactive Visualization Environment) [38, 39] was created to improve the usual debugger offered by Eclipse. Originally it was a stand-alone system, but currently is implemented as an Eclipse plug-in. JIVES is a dynamic analysis tool incorporating traditional debugger features such as break-points, variable inspection, and stepping, as well as advanced features such as dynamic visualizations of program executions, forward and reverse stepping, and query-based debugging.

- Dynamic visualizations: JIVE's dynamic visualizations of a program's run-time state and method call sequence are extensions of UML's [40] object and sequence diagrams, respectively. Besides, these diagrams are normally used as design-time specifications, their use for depicting run-time behavior helps close the gap between design and execution.

The object diagram provides visual representations for objects, their internal states, associations and calls that have not returned yet. The sequence diagram captures interactions among objects at run-time. It consists of life lines placed horizontally across the top of the diagram and activation boxes arranged vertically along these life lines. Both diagrams are shown in the Figure 2.11.

- Reverse Stepping: JIVE records program executions in order to support stepping in both forward and backward directions, with this feature users may select and jump to any time point in the diagram and JIVE updates the execution state accordingly.

- Query-Based Debugging: JIVE supports eight different kinds of template queries, including: Variable Changed, Method Called, and Method Returned. In order to execute a template query, the user must provide one or more required parameters.

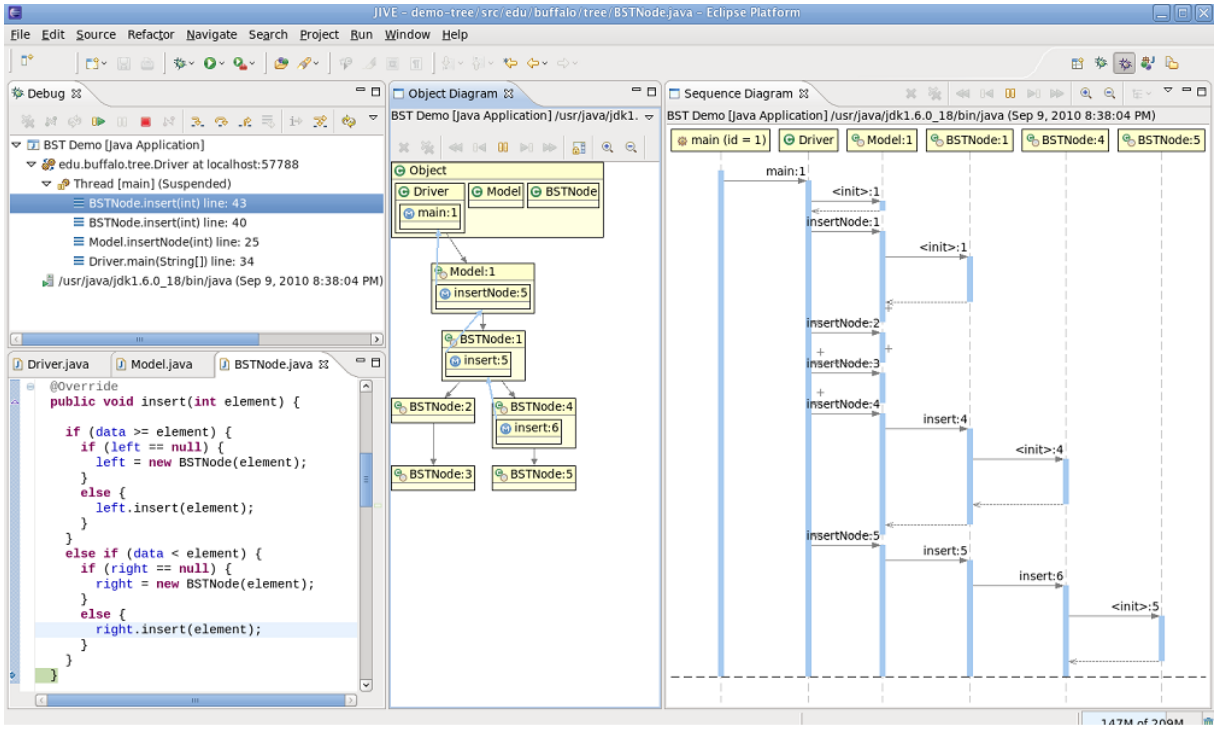

Figure 2.11: JIVE's user interface, object diagram and sequence diagram [38]

JIVE with these three features, show several views of the relationship between methods and objects, which are very important in an object oriented programming language. Also, it presents object-oriented semantics visually, e.g., inheritance, object members and associations, method invocations within object contexts, etc; aiding the programmer to better understand the execution state during the debugging phase.

As JIVE is coupled with Eclipse, makes it impossible to use JIVE separately. With that feature it is hard for a developer to reuse it as visualizer, in other tool.

JIVE lacks in visualization when using an approach without objects, e.g factorial example or a for loop. Using the classic example of a function that calculates the factorial of a number, JIVE will not explicit show the several values of the loop variable. The only way of showing the values is to make a

query on the variable and after this, the several values are showed. Although JIVE does not present it in the visualization.

2.1.3 Comparisons

The Table 2.1 summarizes the main characteristics of each visualization tool researched for this section. Some notes about the information represent in the table: all systems have the visualizer coupled with the execution environment and Jeliot 2000 is not represented because Jeliot 3 is an improvement in terms of architecture and they have a similar interface.

The main difference between visual debuggers and visualization tools is that, the visual debuggers provides weak visualization because there principal function is debugging the code. Instead of the Visualization tools, has the main objective of helping the student to better understand the execution flow of a program, so they are more concerned about visualization.

In Jeliot3 was presented the advantage of being useful for students , this feature is supported by some studies about the effects of program visualization on student's performance [41] and effectiveness of program visualization in learning Java [42]

2.2 Execution Environment

In order to show the internals of a program, it is necessary to execute it in a specific environment. The execution environment will provide information, so that other component of CTutor produces the visualization for the student. These tools have different goals and functionalities, although they all have in common an important feature which is executing the source code and then, with some modifications, provide information about the several changes of the language elements (e.g. when variable changes its value).

2.2.1 Instrumentation

Instrumentation is a technique used to monitor an application while it is being executed. With instrumentation, programmers can insert additional code to the application, this code triggers messages with the state of the application.

To conclude, instrumentation is a way to gather information from source code, but for each modification on the code is needed a recompilation of the application.

2.2.2 Interpreter

2.2.2.1 PicoC

PicoC [43] is a small C interpreter design for robotic, embedded and non-embedded applications, so it is multi-platform and portability. As an example of its functionality, it is used as the script language for a UAV's on board flight system. The core C source code is around 4500 lines of code. It is not a complete

System	Visualized elements	Disadvantages	Supported Language
Eliot	Algorithm visualization	Inactive and Outdated Execution environment coupled with Visualizer Transformation C in Eliot-C	Eliot-C
Jeliot I	Algorithm visualization	Execution environment coupled with Visualizer Needs a server	Java
Jeliot 3	Frames and variables	Execution environment coupled with Visualizer	Java
CSmart	Expressions evaluation	Requires annotations from the professor Being an IDE with limited visualization	C
Online Python Tutor	Frames and variables	Execution environment coupled with Visualizer Trace generated offline	Python
DDD	Frames and variables	Is a Debugger Difficult to interact with the visualization window	C
JIVE	Object Diagram and Sequence Diagram	No memory display	Java

Table 2.1: Comparison between Visualization tools

implementation of ISO C, but it has all the essentials. The code space and data space occupied is very small, an important feature to its purpose.

PicoC is now feature frozen. Since it is important that it remains small it is intended that no more major features will be added from now on. It is been tested on x86-32, x86-64, powerpc, arm, ultrasparc, HP-PA and blackfin processors and is easy to port to new targets.

PicoC has two ways of executing the source code, first one is interpreting the code and then presents the result and the second one is using a read–eval–print loop (REPL). In a REPL the user enters an expression of C, then it is evaluated and the result displayed, after that the REPL waits another expression to begin again the loop.

PicoC as an interpreter parses the source code and perform its behavior directly, without previously compiled into machine code. In internal terms PicoC is composed by its own parser, lexical analyzers (lex), table and heap. The parser uses the lex to create tokens from the sequence of input characters. Then the parser will fill the table of symbols, which stores the identifiers, its values and types. In the heap is stored the memory allocation values, for example the values of a variable created with a *malloc* expression.

2.2.3 Debugger

2.2.3.1 GDB

GDB, the GNU Debugger [30], came into existence around 1985, developed by Richard Stallman. In the UNIX world, it is one of the most popular debuggers and it provides a large range of functionality for all debugging purposes.

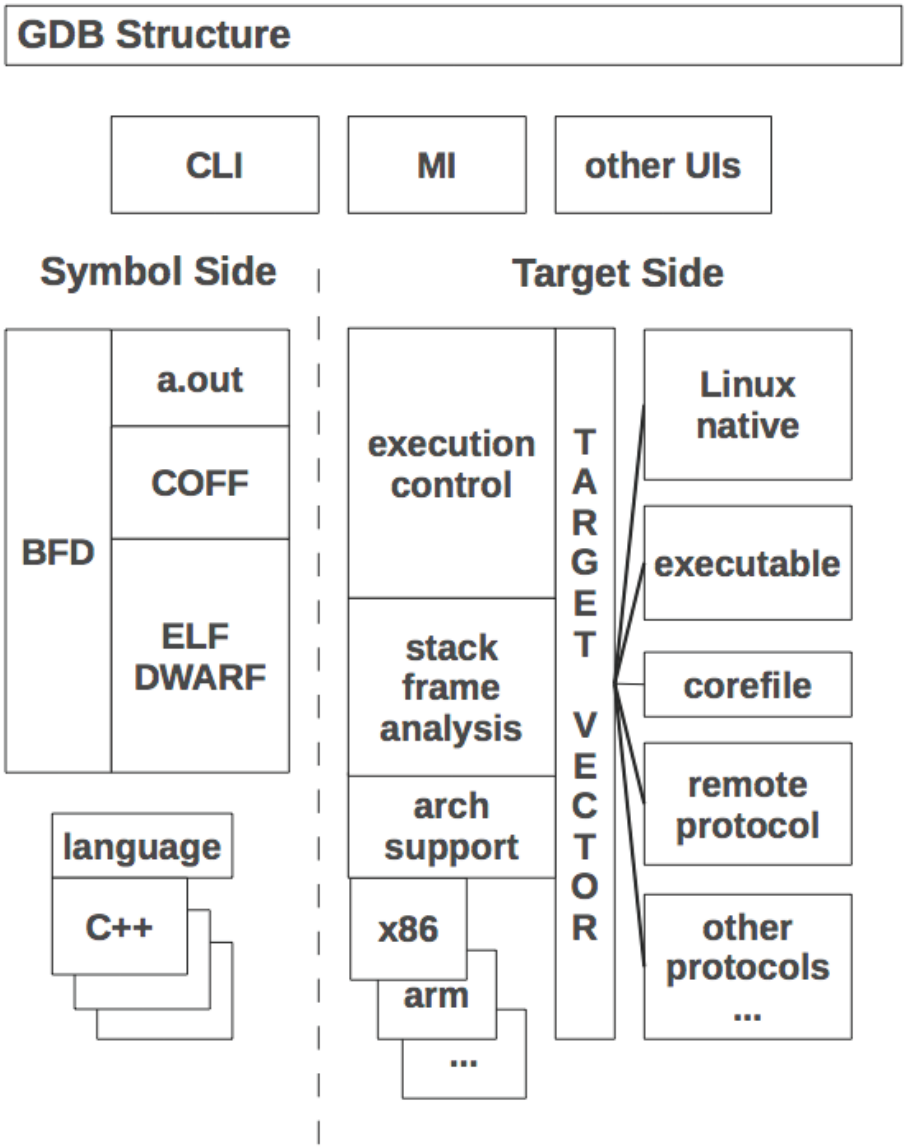

Figure 2.12: GDB structure [44]

GDB is divided in two sides, illustrated in Figure 2.12, each one having its own concerns:

- The "symbol side" is concerned with symbolic information about the program. Symbolic information includes function and variable names and types, line numbers, machine register usage, and so on. The symbol side extracts symbolic information from the program's executable file, parses expressions, finds the memory address of a given line number, lists source code, and in general works with the program as the programmer wrote it.

- The ”target side” is concerned with the manipulation of the target system. It has facilities to start and stop the program, to read memory and registers, to modify them, to catch signals, and so on. Most Unix-type systems have a system call named *ptrace* that gives one process the ability to read and write the state of another process.

GDB provides two interfaces, command-line (CLI) and machine interface (GDB/MI [45]). Command-line interface basically uses the standard GNU library *readline* [46] to handle the interaction with the user. It is also able to do things like use cursor keys to go back in a line and fix a character.

One way to provide a debugging GUI is to use GDB as a backend to a graphical interface program, translating mouse clicks into commands and formatting print results into windows. But, it is not the ideal approach because sometimes results are formatted for human readability, omitting details and relying on human ability to supply context.

To handle this problem exists the second interface provided, machine interface. It has the following features: Commands and results have additional syntax that makes everything explicit - each argument is bounded by quotes, and complex output has delimiters for subgroups and parameter names for component pieces. Also, commands can be prefixed with sequence identifiers that are echoed back in results, ensuring reported results are matched up with the right commands.

This GDB/MI is more verbose than the command-line, but it is more useful for software to parse the information and use it.

2.3 Integration with Visualizer

After presenting the three different tools that can be used as execution environment, and as support for the visualizer. How to accomplish the integration with the visualizer will be explained next.

Figure 2.13 represents three architectures used for the developing a program visualization tool. These three architectures have as a starting point the source code, produced by the student, and the goal is achieved when the visualizer creates the visualization.

1. The use of instrumentation, requires the modification of the source code. After that the instrumented application executes the modified source code. The visualizer 1 is responsible for create the trace and to control the execution. This trace is send to the visualizer 2.

2. Alteration of the execution environment enables the possibility of the environment to execute the source code and gather the information to send for the visualizer 1+2. For example, examining the parse tree to know when its created a new variable.

3. After the compilation of the source code, it is possible to use an execution environment to inspect the application. This execution environment will automatically create triggers that can be exploit with an interface. This interface is used by the visualizer 1 to gather the information and send it to the visualizer 2.

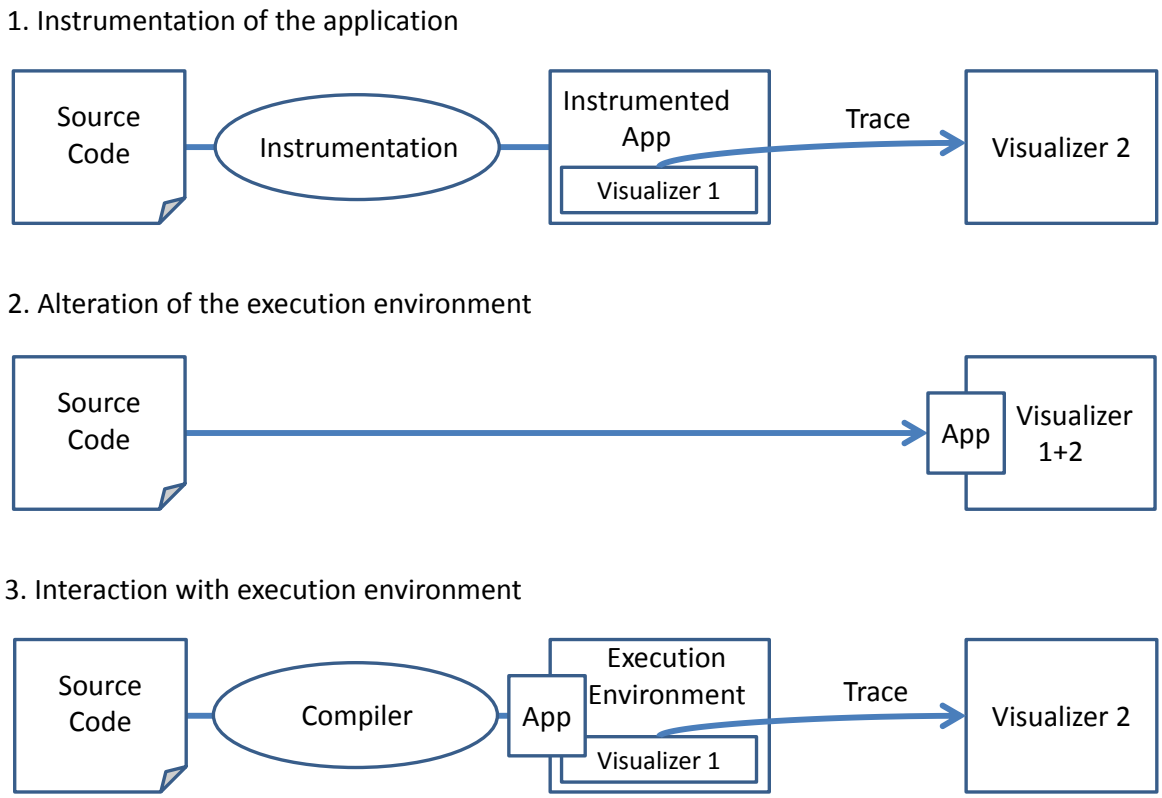

Figure 2.13: The three different methods to build a program visualization tool

Table 2.2 presents those methods. Independently on the architecture, there are several ways to collect information.

Execution environments	Advantage	Disadvantage
Instrumentation	Performance	Too complex Needs recompilation
Interpretation	Adaptable to the browser Used to develop an interactive tool	Only supports a subset of the language Change language implies changing interpreter
Debugging	Support several languages Easy access to application state Automatic code modification (gcc -g)	

Table 2.2: Comparison between solutions to Execution Environment

2.4 Complementary Tools

2.4.1 Learning Environments

The tools presented in this section are used in the learning process of a student, but they are not learning environments because they do not offer visualization or explanations to the student. The use of these tools, sometimes earlier then desire, can create some gaps in the learning process.

These tools are Eclipse and NetBeans, characterized as integrated development environments. They have the purpose of enhancing the efficiency and facilitate the programming.

2.4.1.1 Eclipse

Eclipse [2] is an open-source integrated development environment (IDE), used mostly to develop applications in Java. By means of various plug-ins, Eclipse may also be used to build software in other programming languages. With this use of plug-ins Eclipse provides all the functionality within and on top of the runtime system. Eclispe offers several development environments, for example: the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

Elcipse provides integrated debugging tools, but despite advances in development tools, these debuggers are still intrinsically based upon the classic DBX. The interfaces are based on interactions of text rather than interactions among graphical components, and so important relationships among methods and objects are lost.

2.4.1.2 NetBeans

The NetBeans [3] project consists of a full-featured open source IDE written in the Java programming language and a rich client application platform, which can be used as a generic framework to build any kind of application.

NetBeans IDE offers support for Java, C/C++ and PHP developers, providing comprehensive editors and tools for their related frameworks and technologies. In addition, the IDE has editors and tools for XML, HTML, PHP, Groovy, Javadoc, JavaScript, and JSP.

2.4.2 Portability

The student is the principal actor of program visualization tools, so it is important to consider portability to reach the maximum population.

Each operative system compiles a programming language differently and with that a program visualization tool can works in Windows and can't work in Unix systems or vice versa. Nowadays, a common method for achieve portability is to export the tool to the web, so that it works into the browser and in every systems.

Native Client and Portable Native Client can be the solution to the portability's problem, in C language, for that reason its architecture and purpose are presented below.

Native Client (NaCl) [47] is an open-source project being developed by Google, and is specifically designed to run native code securely inside web browsers. It may also be used for securing browser plugins, and in the future parts of other applications or full applications.

Since its launch in 2011, NaCl has provided developers with the ability to build web applications that harness the computational power of client machines to a much fuller extent than traditional web technologies, by running compiled C and C++ code securely and at near-native speeds, and taking advantage of multiple cores with shared memory.

Native Client consists of toolchains and runtime components. The first ones are collections of development tools, such as compilers or linkers, that transforms C/C++ code to Native Client modules. The second ones are components embedded in the browser or other host platforms that allow execution of Native Client modules securely and efficiently.

While Native Client provides operating system independence, it requires developers to generate architecture-specific executable modules (*nexe* modules) for each hardware platform. This is not only inconvenient for developers, but architecture-specific machine code is not portable and thus not well-suited for the open web. The traditional method of application distribution on the web is through a self-contained bundle of HTML, CSS, JavaScript, and other resources (images, etc.) that can be hosted on a server and run inside a web browser.

To solve the non-portability of NaCl, Google developed a new technology called Portable Native Client (PNaCl). With PNaCl, a developer generates a single *pexe* (portable executable modules) from source code, using the Low-Level Virtual Machine (LLVM) [48] *bitcode* format, rather than multiple platform-specific nexes. The pexe provides both architecture and OS-independence. Since the pexe uses an abstract, architecture-independent format, it does not suffer from the portability problem described above. Future versions of hosting environments should have no problem executing the pexe, even on new architectures. Moreover, if an existing architecture is subsequently enhanced, the pexe doesn't even have to be recompiled—in some cases the client-side translation will automatically be able to take advantage of the new capabilities.

The following diagram, Figure 2.14, shows how to use PNaCl (left side) and NaCl (right side) and how components of NaCl interact.

2.4.3 Conclusions

The two solutions that were presented in this section, will invite future developers to improve CTutor, and also demonstrate that it is possible to integrate CTutor with other tools widely used in software development.

CTutor could be used entirely through the browser, using Native Client, which it is possible to run C and see its visualizations only in a web page. Another important aspect, using Native Client, is the possibility of executing a previously compiled application in several architectures or operative systems. The use of previously compiled applications enables the creation of learning packages with pre-compiled examples.

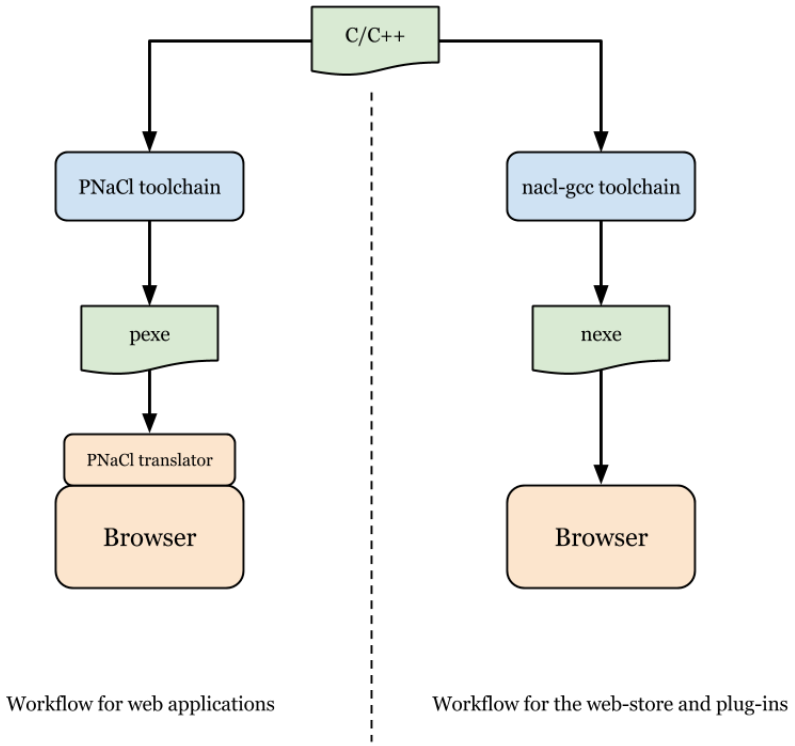

Figure 2.14: Interactions among the components of Nacl

CTutor, instead of a stand alone solution, could be an integrated module in existing tools, as Eclipse and NetBeans. Besides, all the advantages of using an IDE in terms of debugging a program in C, the Eclipse and NetBeans offers as much functionality as GDB. As manner of fact, both IDE's implement the Machine Interface provided by GDB on its environment.

Chapter 3

CTutor

This chapter will explain the choices made in the architecture of CTutor, i.e which execution environment will be used, and which visualizer. The decisions are made with the purpose of accomplish the objectives of CTutor, being a program visualization tool for C, and facilitate the creation of new tools with existing visualizers or existing execution environments.

The second section will present the components of CTutor and their implementation, as well the interactions between them. Also will demonstrate the communication protocol between the execution environment and the visualizer, called program trace.

3.1 Architecture

The Figure 3.1 is a modified version of the Figure 2.13, and it explains the future role of CTutor in each of the three architectures, and also where it will be necessary CTutor to be implemented.

1. Using instrumentation, will require changes the source code and as illustrated CTutor will have direct contact to it, then it will communicate to the visualizer the different state changes;

2. The interpreter will be the responsible for reading the code. Then, CTutor will change the interpreter to capture that information and send it to the visualizer;

3. When using a debugger, such as GDB, it will analyze the code and then it will provide to CTutor information in its own format, in case of GDB it is an GDB/MI. CTutor will need to know this format and how to interpret it, having then the information and transmit it to the visualizer.

To use instrumentation as backend to CTutor, a modification in GNU Compiler Collection (GCC) [49], would be a complicated task.

The interpretation, for instance with PicoC, needs to transverse the parse tree, and for that it will be needed to learn the code of this interpreter. Also it will be needed its modification, not only for intercept the cases for the visualization, as well for communicate with the visualizer.

The use of GDB, allows the use of a generic backend without modifications on it, CTutor needs to interact with it. This interaction is done through its GDB/Machine interface, presented in the section

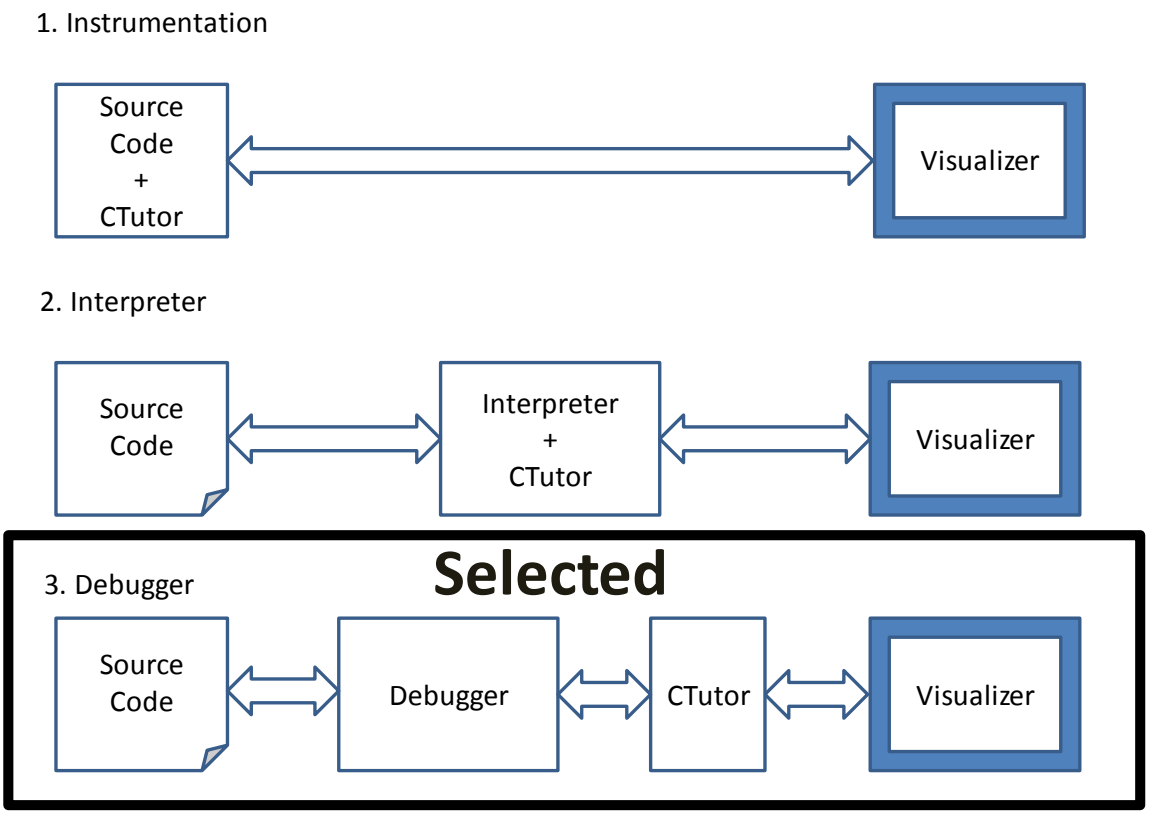

Figure 3.1: The three different methods with CTutor

2.2. Another interest of using GDB is the possibility, with slightly changes, of using other programming languages without the inconvenience of construct of a new backend.

The Visualizer used in CTutor is the same as the Online Python Tutor. This visualizer has been used in some programming courses, in universities as well as in e-learning. It has also some features, that can be handy to the learning process, the possibility of embed in a webpage the visualization, and sharing live visualization sessions.

The resulting architecture of CTutor is represetend in the Figure 3.2.

3.2 Implementation

The implementation of CTutor starts in the separation of the execution environment and the visualizer of Online Python Tutor (OPT). This separation between the visualizer and the execution environment, enables the reuse of the visualizer of OPT.

Compared to OPT, CTutor has a different execution environment code and the TutorStub to link both parts. As shown in Figure 3.2.

The execution environment is made by the GDB and a component called, CProxy. The CProxy handles the interaction with GDB, through an existing library, and the communication with TutorStub,

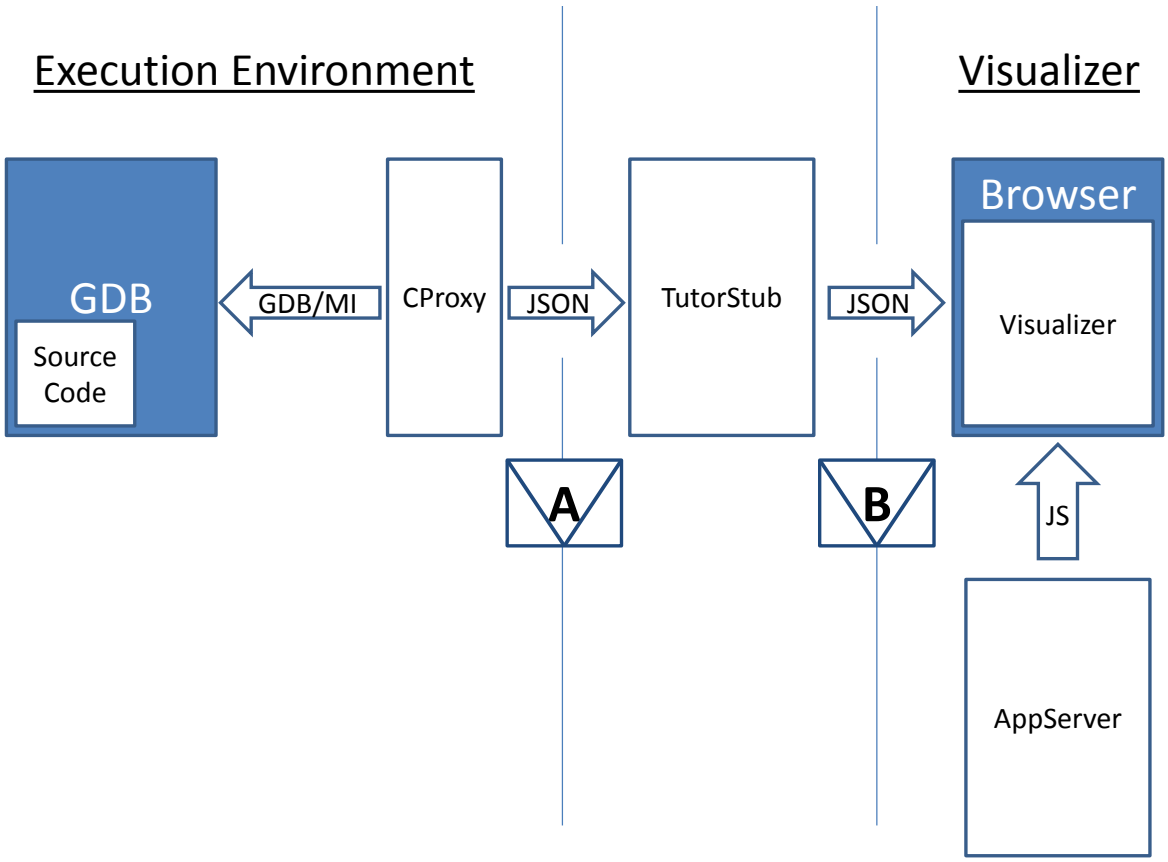

Figure 3.2: The architecture of CTutor

throughout JSON objects,.

The CProxy is implemented in the file **cproxy.c**. Since the communication, with TutorStub, is done by JSON objects. CProxy uses a library that maps C **structs** in that format. This library is cJSON [50], portable, single-file, ANSI-C compliant JSON parser.

The GDB/Machine Interface library (libmigdb) [51] implements the GDB/MI protocol, allowing CProxy to controll the application and retrieve its state.

It was necessary to add two methods to libmigdb: **gmi_break_insert_main** and **gmi_stack_lis_variables**. The first one is used to create a breakpoint into main function, as is the only entry point of C programming language, so that all visualizations starts always in main function. The **gmi_stack_list_variables** is used to gather the local variables and function arguments for the selected frame.

For each line, after the breakpoint, CProxy verifies the stack of frames (**gmi_stack_list_frames**) and the stack of variables (**gmi_stack_list_variables**). The outputs of these two methods are processed with the intention of getting information to fill the cJSON struct, for example name and value of a certain variable.

The TutorStub is a HTTP-Server, based on the python web framework Bottle [52]. This server provides two services *getSourceCode* and *getNextInstr*.

Comparing to OPT, CTutor changes the way students inserts source code to run. OPT, which uses a code editor on the browser to get student's code, while the CTutor gets the script from a file on the hard

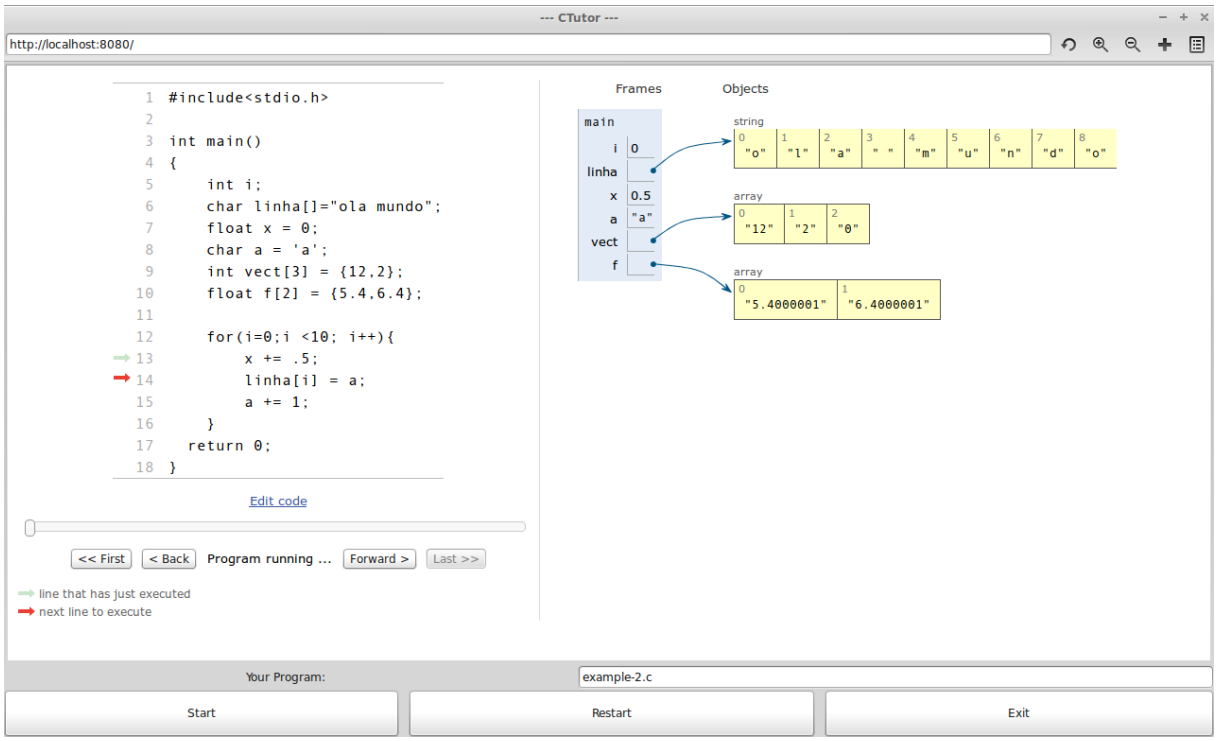

Figure 3.3: User Interface of CTutor

drive. The *getSourceCode* is invoqued from the javascript code on the visualizer.

The *getNextInstr* is used by the browser to request the next instruction trace, thus it can show the visualization. This request is sent when the student press the button **forward** on the visualizer.

The methods corresponding to the construction of the program trace are in **trace.c**, that is invoked by CProxy.

The program trace is a list of JSON objects, where each object correspond to a single instruction trace and it represents a message with the changes that the visualized environment suffers over the time. This changes are declaration of variables, modification of its values and execution of functions.

The TutorStub reduces the existing coupling between backend and frontend, working as a proxy. The TutorStub receives an instruction trace (A) from the CProxy and send to the browser the program trace (B), that are represented as envelopes in Figure 3.2.

The appserver is a Google App Engine that allows the execution of Python code, using a pre-loaded Python interpreter in a safe ”sandboxed” environment, used in OPT to execute the python debugger in the browser and to send the trace to the visualizer. In CTutor, as the execution environment uses the GDB and is not executed in the browser, also the TutorStub takes the responsibility of sending the trace to the visualizer, this server only loads all the contents of the visualizer (e.g. html, js and css files) to the browser.

There are two ways of launch CTutor, one is using a python script **main.py** and the other is using a graphical user interface (GUI) **gui.py**. As CTutor is a tool for students the gui is more suitable, but for developers the main script can be more useful to understand how to launch the components.

The main script is used to launch the two servers, the browser and for the user specify which file he

wants to analyze in the visualizer (browser).

This GUI, presented in Figure 3.3, was developed using the *pywebkitgtk* [53], this package enables the embedding the a web-page into the GUI. Another feature is about with the selection of the program, it as an input text box which is checked in the start of the server to get the file name of the file that the student wants to visualize.

The Figure 3.4 and 3.5 are sequence diagram to explain the interactions between the components of CTutor, also they demonstrate the difference between the CTutor using the main script and using the GUI.

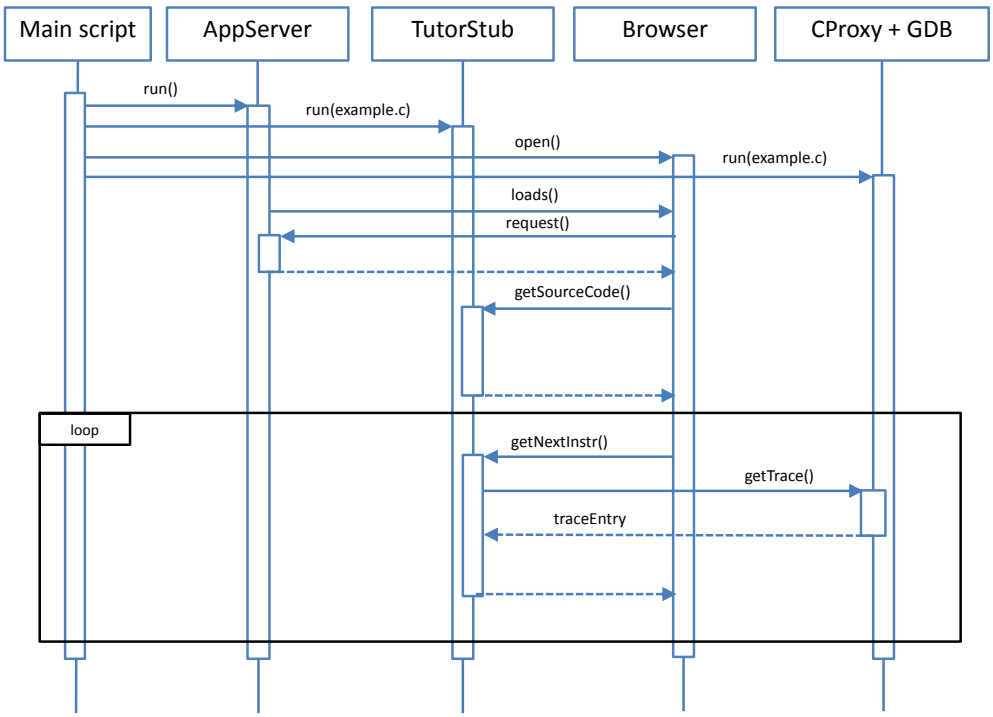

Figure 3.4: Interactions between components of CTutor with Browser

Observing the Figure 3.2, that shows the architecture of CTutor, it is possible to identify in the sequence diagrams the division between visualizer and execution environment. There are no direct interactions between the visualizer (Browser) and the execution environment (GDB). The TutorStub and the CProxy between them, reduce the coupling.

The loop represents the visualization process, as the student is visualizing each instruction it will send a *getNextInstr* request to the browser (whit button forward on the visualizer), passing through the TutorStub and transformed into a *getTrace* request to the CProxy. After that the reverse path, start with the execution of the instruction in GDB, the CProxy interacts with it to know which elements of the language were change, builds the trace and then send it to the TutorStub as a traceEntry (piece of trace, called instruction trace).

The biggest difference between the use of a Main script or a GUI, is the presentation to the student. As the GUI embeds the browser and occults the use of a console to do the start of the other components

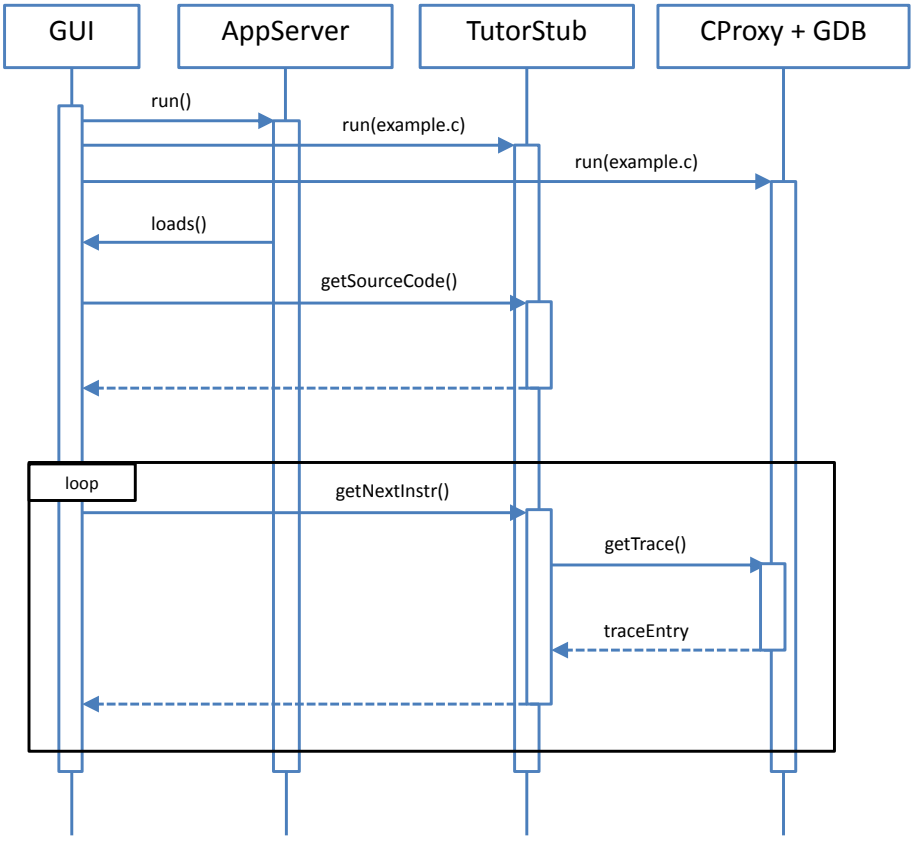

Figure 3.5: Interactions between components of CTutor with GUI

it is more suitable for students. The Main script can be more useful for developers to analyze the traces exchanged between the components.

3.2.1 Trace

The trace is the message passed by the backend to frontend, and is built in the execution environment. In OPT, the execution environment has the complete trace from the start until the end of the execution, with the appearing of TutorStub this situation has changed.

Now, the execution environment only builds the trace of the executed line (instruction trace), the TutorStub has now the responsibility of join all trace lines and send the complete trace to the visualizer (program trace).

This modification helps the developer of future tools, to analyze and comprehend the instruction trace and the information needed to build the program trace. Also, facilitates the developing of a new execution environment because the developer only needs to build an instruction trace for each new instruction, instead of a program trace with the previous instructions traces.

In order of explain the basics of trace, it is used as example the program illustrated in the Figure 3.6. It is also represented above the instruction traces and the visualizations corresponding to the execution of the program.

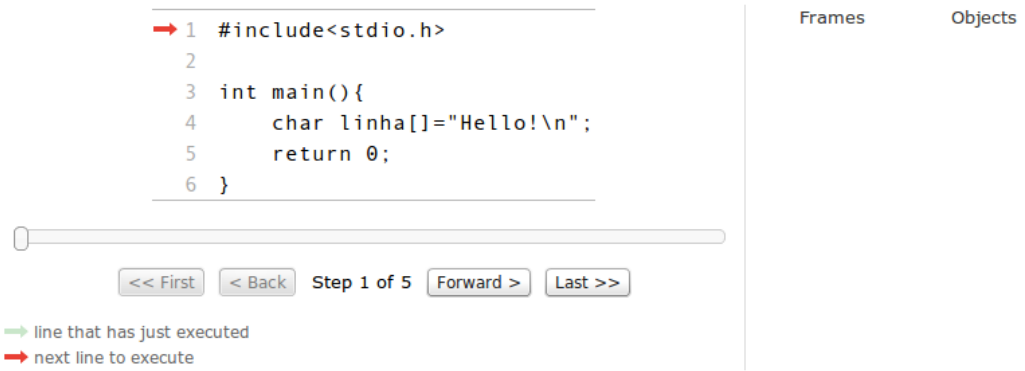

Figure 3.6: Program to explain trace

An instruction trace is a dictionary constituted by eight objects, each of one representing different information needed by the visualizer. The objects will be present below, also the order, of the objects in the dictionary, is not relevant.

- **ordered_globals -** Is a list of the global variables it is order of appearance, in case of CTutor it is the variables created in main.
- **globals -** It is a dictionary, in the keys has the name of the variables and in the objects has the value or a reference to the heap. As an object whose key are unsorted, the visualizer needs the list of *ordered_globals* to present the variables in the correct order.
- **stdout -** In Online Python Tutor is used as output string to show in the visualizer, but as CTutor uses the console to print the output this object is never used.
- **func_name -** Presents the name of the actual function, in case of CTutor ¡module¿ is the main function.
- **stack_to_render -** This object is used to represent a new frame, it is a dictionary and has several objects. The unique_hash is a unique name for the function that is called.

  The booleans *is_parent*, *is_highlighted*, *is_zombie* are used to classify the function. The only used in CTutor is the *is_highlighted*, that tells to the visualizer if the function is active and then it appears with a blue background. The *parent_frame_id_list* is a list of parents of the frame, but is not used in CTutor.

  The *ordered_varnames* and the *encoded_locals* made the same job as the *ordered_globals* and the *globals*, but for variables created in this new frame.

```
'stack_to_render': [{
'unique_hash': 'add_f2',
'is_parent': False,
'is_highlighted': True,
'is_zombie': False,
```

```
'ordered_varnames': ['n'],
'frame_id': 2,
'encoded_locals': {'n': 1},
'func_name': 'add',
'parent_frame_id_list': []}],
```

- **heap -** The heap contains the complex types of the language, as well as strings and arrays. The reference of a complex type is in the *globals* or *encoded_locals*, and the heap maps its value, see trace shown below.
- **line -** Represents the line that will be executed in the next step.
- **event -** The event can be a *step_line*, *call*, or a *return*. The call means the change to another function, i.e it will occur a function call. The return means the event is the return of a function. The step_line indicates that an ordinary single-line step event is about to occur.

1.
```
{'ordered_globals': ['linha'],
'stdout': '',
'func_name': '<module>',
'stack_to_render': [],
'globals': {'linha': ['REF', '1']},
'heap': {'1': ['STRING', '\\0', '\\0', '\\0', '\\0', '213', '204', '004', 'b']},
'line': 3,
'event': 'step_line'}
```

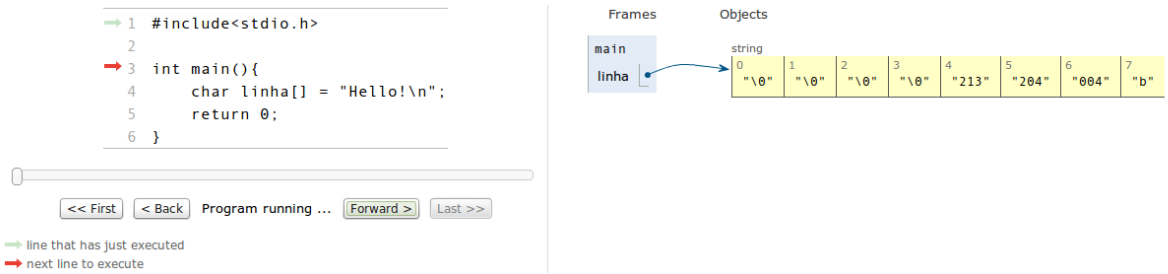

Figure 3.7: Visualization corresponding to trace 1

2.
```
{'ordered_globals': [],
'stdout': '',
'func_name': '<module>',
'stack_to_render': [],
'globals': {},
'heap': {},
'line': 4,
'event': 'step_line'}
```

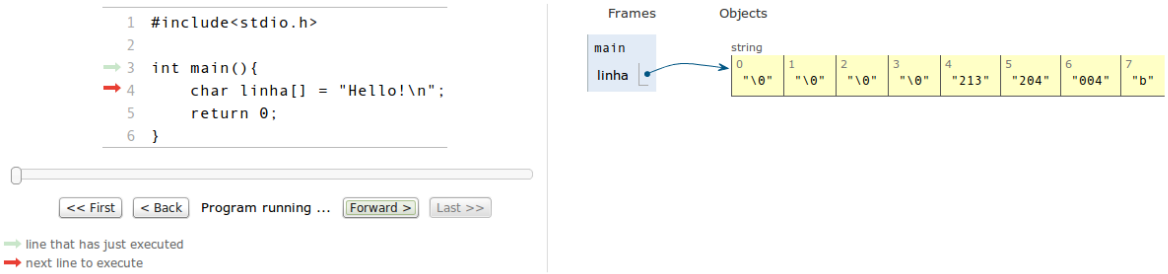

Figure 3.8: Visualization corresponding to trace 2

3. {'ordered_globals': ['linha'],

   'stdout': '',

   'func_name': '<module>',

   'stack_to_render': [],

   'globals': {'linha': ['REF', '1']},

   'heap': {'1': ['STRING', 'H', 'e', 'l', 'l', 'o', '!', '\\n']},

   'line': 5,

   'event': 'step_line'}

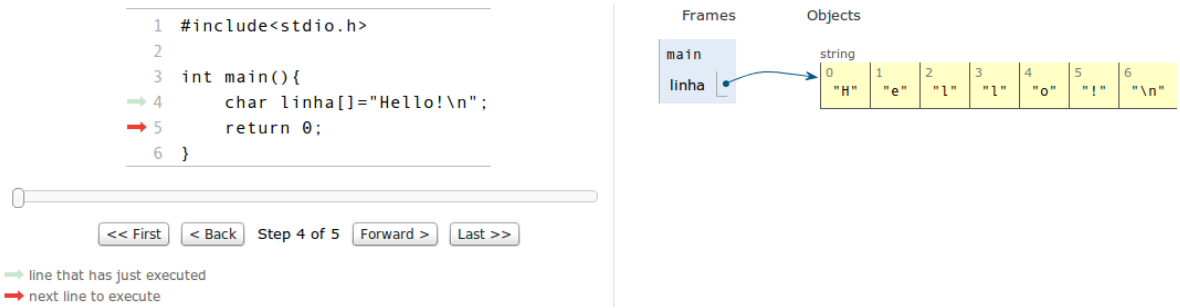

Figure 3.9: Visualization corresponding to trace 3

4. {'ordered_globals': [],

   'stdout': '',

   'func_name': '<module>',

   'stack_to_render': [],

   'globals': {},

   'heap': {},

   'line': 6,

   'event': 'step_line'}

5. {'ordered_globals': [],

   'stdout': '',

   'func_name': '<module>',

   'stack_to_render': [],

   'globals': {},

```
1  #include<stdio.h>
2
3  int main(){
4      char linha[]="Hello!\n";
5      return 0;
6  }
```

<< First | < Back | Step 5 of 5 | Forward > | Last >>

line that has just executed
next line to execute

Frames | Objects

main
linha

string

0	1	2	3	4	5	6
"H"	"e"	"l"	"l"	"o"	"!"	"\n"

Figure 3.10: Visualization corresponding to trace 4

```
'heap': {},
'line': 6,
'event': 'return'}
```

```
1  #include<stdio.h>
2
3  int main(){
4      char linha[]="Hello!\n";
5      return 0;
6  }
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute

Frames | Objects

main
linha

string

0	1	2	3	4	5	6
"H"	"e"	"l"	"l"	"o"	"!"	"\n"

Figure 3.11: Visualization corresponding to trace 5

Chapter 4

Extended Python Tutor

Initially Online Python Tutor (OPT) was formed by a Python Debugger, an appserver and a browser. Besides these three components, the new OPT has a TutorStub, and a main script. For simplicity issues, this new solution will be named as Extended Python Tutor (EPTutor).

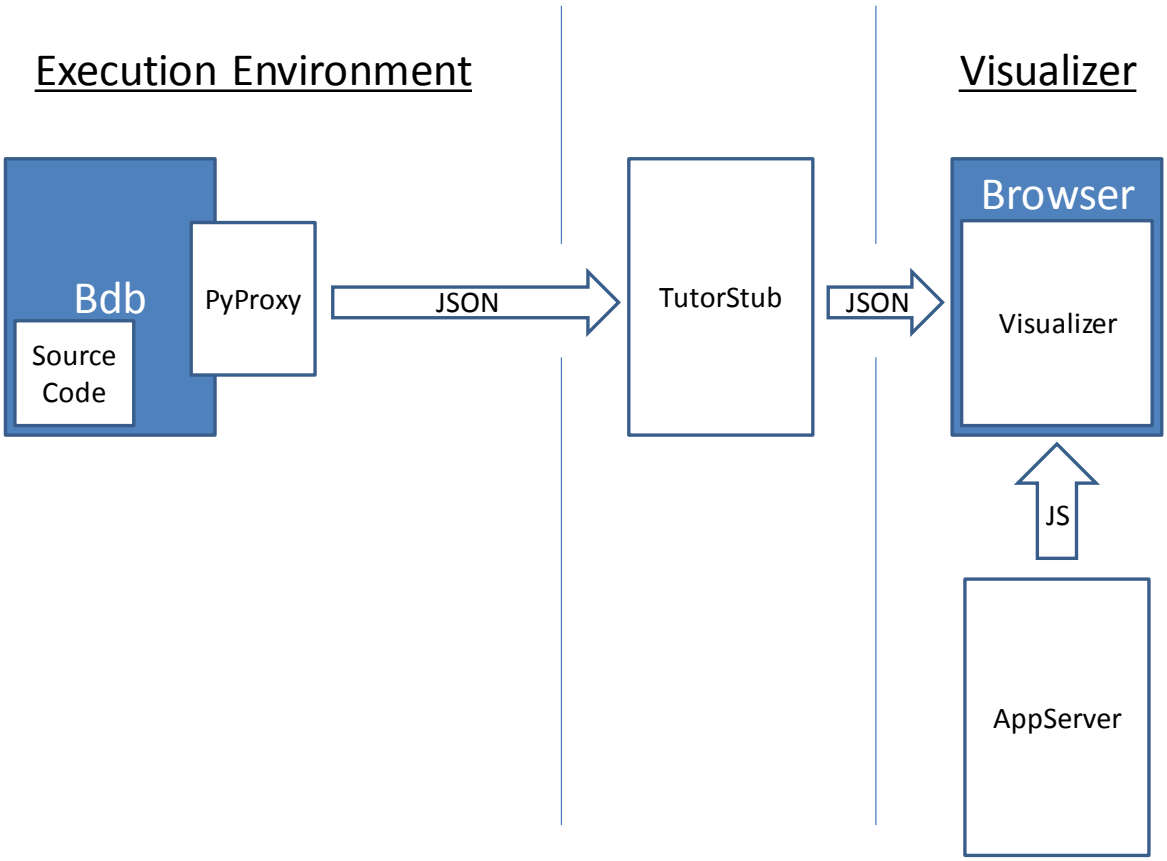

Figure 4.1: The architecture of Extended Python Tutor

In order to serve as example of this new architecture approach, EPTutor is based on OPT's components. The TutorStub is similar to the described in the CTutor's implementation.

The differences in the two TutorStub are in the the input that they receive from the execution environ-

ment. In CTutor is a string in the form of JSON format, and then exists a python method that converts this to a JSON object. In EPTutor the input is already a JSON object, so the transformation is avoided.

Explanation of the components and their major roles in EPTutor:

- Main script: This script is used for launch the two servers and the browser.
- AppServer: Loads all the contents (e.g. html, js and css files) to the browser.
- TutorStub: Handles all communication between the browser and the execution environment, and interacts with Python Debugger.
- Browser: Essentially is responsible for showing the visualizations, so it can be called as the visualizer.
- Execution Environment : In this case is a Python Debugger, that will runs the user's code.

These elements form an architecture very similar to the CTutor in the Figure 3.2, but instead of the GDB it is used Python Debugger, this architecture is presented in the Figure 4.1. Accordingly, the proxy exists but in a single box with the debugger, this happens because the debugger hasn't an interface.

Figure 4.2 and 4.3 illustrates the differences between OPT and EPTutor, and the interactions of its components.

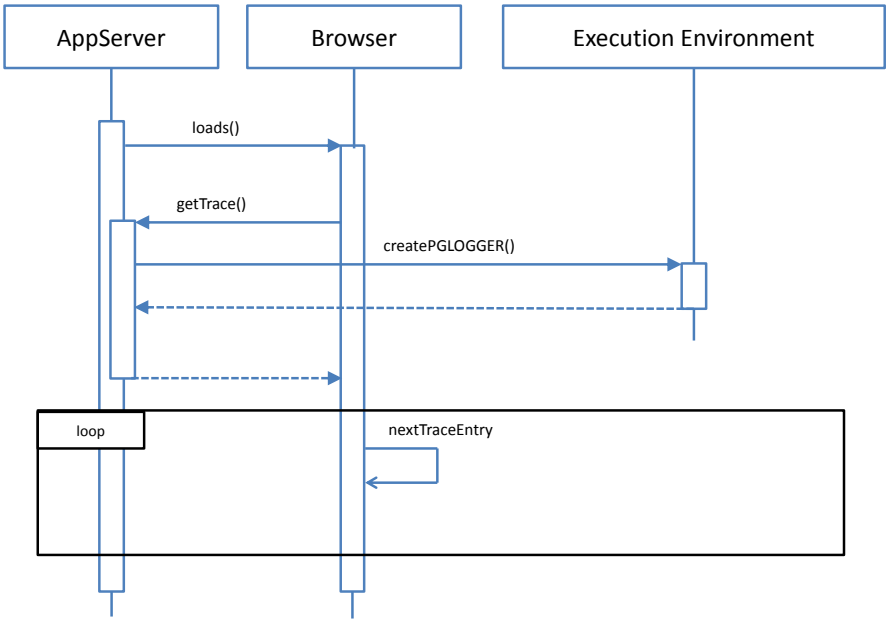

Figure 4.2: Interactions between components of Online Python Tutor

It is represented a loop in Figure 4.3, it means the visualizer of EPTutor is receiving each trace line separately, so in this case the visualizations are made in time. Whit this new architecture, EPTutor can surpass the inconvenience of the pre-execution of the OPT.

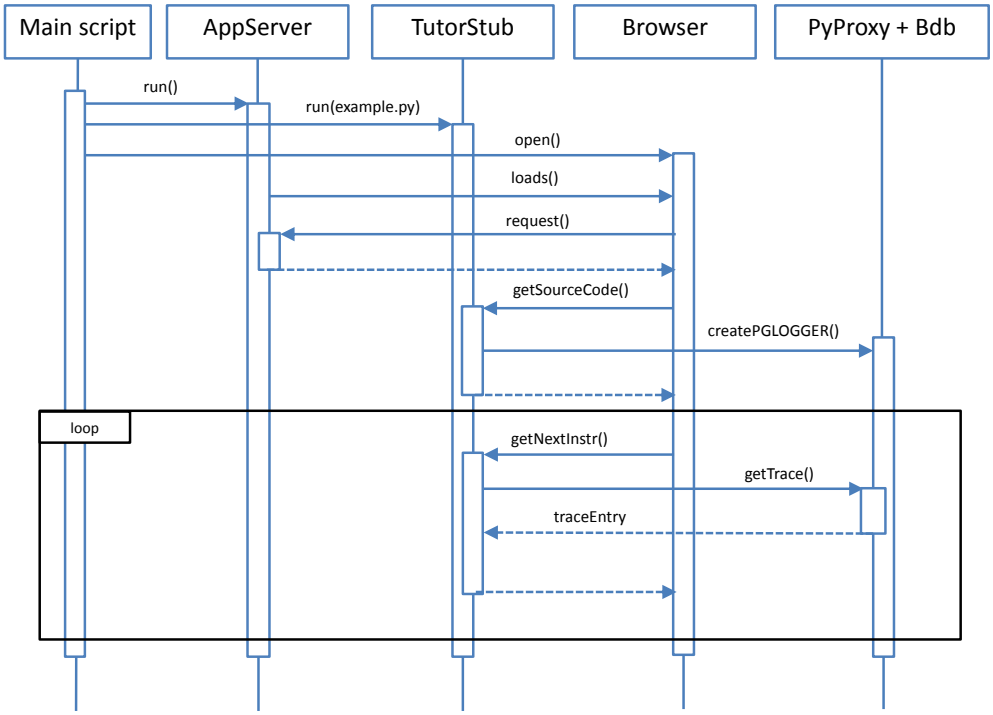

Figure 4.3: Interactions between components of Extended Python Tutor

Chapter 5

Solution Evaluation

The evaluation of CTutor was based on the programming examples of the book ”Linguagem C” [54]. This book is usually the manual of C in first courses.

The practicability of using a programming visualization tool as a complement of the learning process was demonstrated using the programs presented in the book. With this it is possible to evaluate CTutor.

The Table 5.1 presents the elements of the C language supported by CTutor. These elements are divided in groups following the book's chapter.

Elements of language	
Basic Data Types (char, int, float and double)	✓
Operator Types	✓
Constants	✓
Control Statements (if..else, switch, loops)	✓
Functions	✓
Arrays	✓
Multidimensional Arrays	✓
Strings	✓
Input and Output (printf, scanf)	✓
Pointers	+/-
Command Line Arguments (argc, argv)	✓
Files	✓
Structured Datatypes	++/-
Dynamic Memory Allocation	X
Pre-Processors and Macros (#include, #define)	✓
Multifile program	✓

Table 5.1: CTutor's supported elements of language C

The first conclusion reached with the information presented in the Table 5.1 is that CTutor supports all the C language. But, there are some elements that are not visualized, this means that CTutor runs the program examples, yet the visualization is not visible as expected.

For example, in the chapter about Dynamic Memory Allocation, it would be interesting when a student uses the function **malloc**, the visualization demonstrates the memory reserved, instead of hiding this

important concept.

As CTutor uses an adapted visualizer from Online Python Tutor and the concepts of some elements in C are hard to divide, some issues occur on the visualizations.

The visualization of multidimensional arrays is not the expected, instead of a matrix represented by a table, it is shown a graphic that represents a variable with type **int*[]**. In Figure 5.1 it is represented a matrix of type integers.

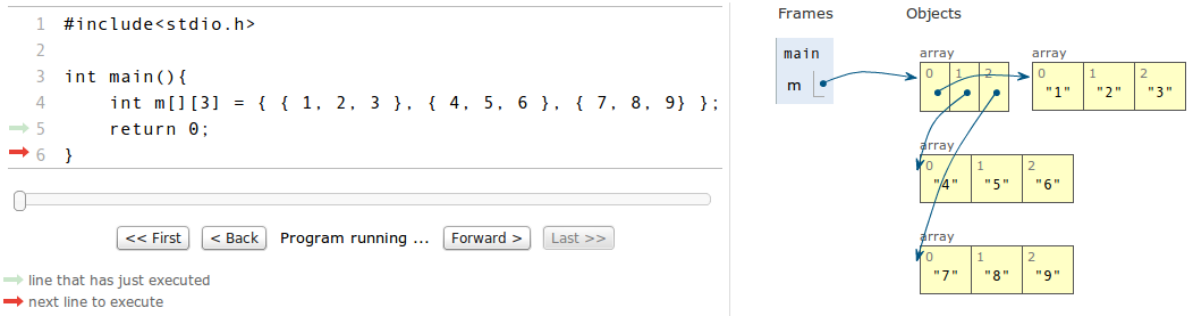

Figure 5.1: Example of a multidimensional array

The value of a pointer in C, is a memory address, but this address can belong to other elements so how to represent this in suitable way. In order to facilitate the visualization for the student, a pointer is presented in the visualizer with its address when it is created, when its address is the same of basic data types, Figure 5.2. The pointer is represented as an arrow to other elements, when its address is the same as a complex data types (string or array), Figure 5.3.

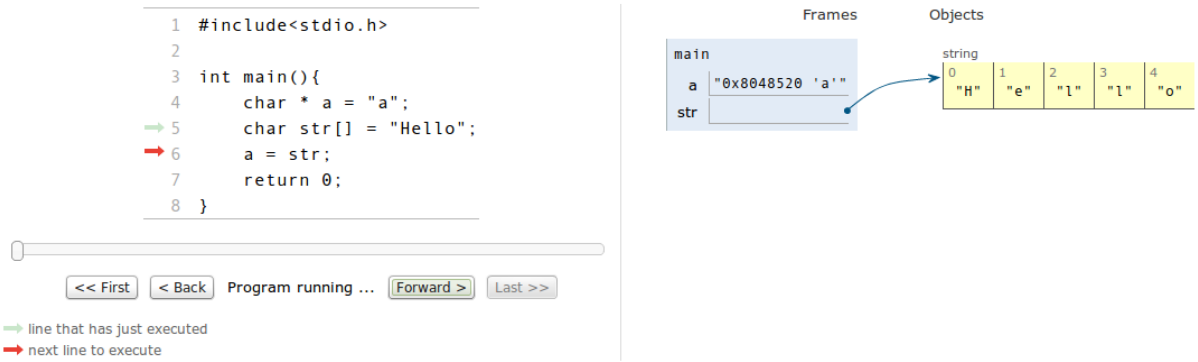

Figure 5.2: Example of a pointer to a char

Usually the structs in C are dependent on pointers, for example the use of lists in C. As seen in Figure 5.4, the visualization of this element can be misunderstood, because the visualizations of a struct and a pointer to it are the same. Ideally, it is better to change this issue, one way to solve it is using a brace "{" to associate the struct to the variable created in the frame main.

```
1  #include<stdio.h>
2
3  int main(){
4      char * a = "a";
5      char str[] = "Hello";
6      a = str;
7      return 0;
8  }
```

<< First | < Back | Step 6 of 6 | Forward > | Last >>

line that has just executed
next line to execute

Frames: main — a, str

Objects: string — 0 "H", 1 "e", 2 "l", 3 "l", 4 "o"

Figure 5.3: Example of a pointer to an array of chars

```
1  #include <stdio.h>
2
3  typedef struct {int Dia,Mes,Ano;} DATA;
4
5  typedef struct pessoa
6  {
7      char  Nome[100];
8      int   Idade;
9      float Salario;
10     DATA  Nasc;
11 } PESSOA;
12
13 int main()
14 {
15     struct pessoa p = {"Carlos",60,12345.67,{23,5,1954}};
16     struct pessoa *ptr = &p;
17
18     return 0;
19 }
```

<< First | < Back | Program running ... | Forward > | Last >>

line that has just executed
next line to execute

Frames: main — p, ptr

Objects: struct — "Nome" "Carlos"; "Idade" "60"; "Salario" "12345.6699"; "Nasc" struct — "Dia" "23"; "Mes" "5"; "Ano" "1954"

Figure 5.4: Example of a struct and a pointer to it

Chapter 6

Conclusions

Programming is a complex domain in which students need a constructive support from their learning environments, to help students to understand programming, with this purpose some tools such as program visualization tools exists.

This program visualization tools can be a great aid for the usual learning process of a new programming language, being a complement of the classes or the e-learning courses. Also, if a student has the need to learn by himself this kind of tools provide an opportunity to learn by simple examples.

After doing some research on existing program visualization tools, it is possible to identify the three components of these tools: the execution environment, the visualizer and the language supported. The problem in the existing tools is that we cannot split the first and second components and reuse it in another tool. For a novice student, which is already attached to a particular visualizer, it will be difficult to change when he needs to learn another language, he will forced to use another tool.

The professor can also have some problems when switching between tools for teach each language, for example the examples created can be better visualized in certain visualizers and thus prevent the best explanation for the student.

CTutor solves this problem with a new architecture. The architecture of CTutor is constituted with a visualizer and an execution environment but separated in diferent processes and with a well defined interface.

The implementation of a new component, TutorStub, between the execution environment and the visualizer leads to a reduced coupling between them. The developing of a CProxy to interact with the execution environment, in this case GDB with its GDB/ Machine Interface, enables the construction of a trace for every execution point and outside of the GDB, avoiding its modification.

This architecture can facilitate the development of new tools and also a mix between visualizers and execution environments. With a minor effort, comparing of building a tool from scratch, the developer can implement a component adjusted to his needs.

Extended Python Tutor appears as a consequence of adapt the architecture of CTutor to Online Python Tutor, with that the greater inconvenience of Online Python Tutor is surpassed. Besides, EPTutor has some limitations it is an evidence that this architecture is a good decision to make, when built a

program visualization tool.

The evaluation of this solution was made using programming examples from a book, the result is that CTutor can support most elements from the C language but has some issues in the visualization of pointers.

After all, CTutor is a program visualization tool that can visualize almost all the examples used to teach C, as well its architecture can facilitate the developing of other visualization tools, visualizers and execution environments separately.

6.1 Future Work

In the future, developers have multiple ways to proceed the work done until here, such as:

- Incorporate CTutor in different learning environments or to support other languages;
- Enhance the portability of CTutor, porting it to the web, using the solution exposed in complementary tools;
- Improve its visualizer to face the issues, regarding the visualization of structs, dynamic memory and pointers;
- Use CTutor's architecture to create other program visualization tools, or more visualizers.

Bibliography

[1] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[2] Eclipse ide. `http://www.eclipse.org/`. accessed 20.11.2014.

[3] Netbeans ide. `http://www.netbeans.org/`. accessed 20.11.2014.

[4] Programming language popularity. `http://www.langpop.com/`. accessed 20.11.2014.

[5] Guo, Philip J. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages, 579–584, New York, NY, USA, 2013. ACM.

[6] Sorva, J. Visual program simulation in introductory programming education. *Zhurnal Eksperimental'noi i Teoreticheskoi Fiziki*, 2012.

[7] Haibt, Lois M. A program to draw multilevel flow charts. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, pages, 131–137, New York, NY, USA, 1959. ACM.

[8] M. Ben-Ari, N. Myller, E. Sutinen and J. Tarhio. Perspectives on Program Animation with Jeliot. *Zhurnal Eksperimental'noi i Teoreticheskoi Fiziki*, 2012.

[9] Ahtinen, S L and Utinen, E S and Arhio, J T R. Automated Animation of Algorithms with Eliot. pages, 337–349, 1998.

[10] Haajanen, J. and Pesonius, M. and Sutinen, E. and Tarhio, J. and Terasvirta, T. and Vanninen, P. Animation of user algorithms on the Web. pages, 356–363, 1997.

[11] Ben-Bassat Levy, Ronit and Ben-Ari, Mordechai and Uronen, Pekka a. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, January 2003.

[12] Moreno, Andrés and Myller, Niko and Bednarik, Roman. Jeliot 3 , an Extensible Tool for Program Visualization. pages, 3–4, 2005.

[13] J. Stasko. *Polka Animation Designer's Package*. Polka software documentation, 1994.

[14] J. Smart. *Reference manual for wxWindows 1.60: a portable C++ GUI toolkit*. Artificial Intelligence Applications Institute, 1994.

[15] Levine, John and Mason, Tony and Brown, Doug. *Lex & Yacc, 2Nd Edition*. O'Reilly, second edition, 1992.

[16] Hillion, S. Dynamicjava. `http://old.koalateam.com/djava/`. accessed 20.11.2014.

[17] García, AM. The design and implementation of intermediate codes for software visualization. 2005.

[18] Moreno, Andrés and Myller, Niko and Sutinen, Erkki and Ben-Ari, Mordechai. Visualizing programs with Jeliot 3. page, 373, 2004.

[19] Gajraj, R and Bernard, Margaret. Transforming source code examples into programming tutorials. *ICCGI 2011 : The Sixth International Multi-Conference on Computing in the Global Information Technology*, (c):160–164, 2011.

[20] Pritchard, David and Vasiga, Troy. Cs circles: An in-browser python course for beginners. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages, 591–596, New York, NY, USA, 2013. ACM.

[21] J. DeNero. Structure and interpretation of computer programs. `http://www-inst.eecs.berkeley.edu/~cs61a/`. accessed 20.11.2014.

[22] Miller, Bradley N. and Ranum, David L. Beyond pdf and epub: Toward an interactive textbook. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, pages, 150–155, New York, NY, USA, 2012. ACM.

[23] bdb - debugger framework. `https://docs.python.org/2/library/bdb.html`. accessed 20.11.2014.

[24] Introducing json. `http://json.org/`. accessed 20.11.2014.

[25] Online java tutor. `http://cscircles.cemc.uwaterloo.ca/java_visualize/`. accessed 20.11.2014.

[26] Online ruby tutor. `http://www.basicruby.com/tutor`. accessed 20.11.2014.

[27] Online javascript tutor. `http://jstutor.herokuapp.com/`. accessed 20.11.2014.

[28] Java jail. `https://github.com/daveagp/java_jail`. accessed 20.11.2014.

[29] Node.js. `http://nodejs.org/`. accessed 20.11.2014.

[30] Stallman, R.M. and Pesch, R.H. *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 1992.

[31] Zeller, Andreas and Lütkehaus, D. DDD—a free graphical front-end for UNIX debuggers. *ACM Sigplan Notices*, 1996.

[32] Perl debugger. `http://debugger.perl.org/tools.html`. accessed 20.11.2014.

[33] Bash debugger. http://bashdb.sourceforge.net/. accessed 20.11.2014.

[34] Pydb – extended python debugger. http://bashdb.sourceforge.net/pydb/. accessed 20.11.2014.

[35] Remake – gnu make with comprehensible tracing and a debugger. http://bashdb.sourceforge.net/remake/. accessed 20.11.2014.

[36] *Debugging a Program With Dbx*. iUniverse.com, Incorporated, 2004.

[37] Ladebug. http://physics.ust.hk/compudoc/ladebug/ladebug-manual-primary.html. accessed 20.11.2014.

[38] Lessa, Demian and Czyz, JK and Jayaraman, Bharat. JIVE: A Pedagogic Tool for Visualizing the Execution of Java Programs. 2010.

[39] Gestwicki, Paul and Jayaraman, Bharat. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, pages, 95–104, New York, NY, USA, 2005. ACM.

[40] Rumbaugh, James and Jacobson, Ivar and Booch, Grady. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[41] Hongwarittorrn, Nuttanont and Krairit, D. Effects of program visualization (jeliot3) on students' performance and attitudes towards java programming. *The spring 8th International conference on Computing, Communication and Control Technologies*, 2010.

[42] Maravić Čisar S., Radosav D., Pinter R., Čisar P. Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3. *International Journal of Computers Communications & Control*, 2011.

[43] Picoc: A very small c interpreter. https://code.google.com/p/picoc/. accessed 20.11.2014.

[44] Stan Shebs. The architecure of open source aplications, volume ii: Structure, scale, and a few more fearless hacks. http://www.aosabook.org/en/gdb.html. accessed 20.11.2014.

[45] The gdb/mi interface. https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI. accessed 20.11.2014.

[46] The gnu readline library. http://cnswww.cns.cwru.edu/php/chet/readline/readline.html. accessed 20.11.2014.

[47] Yee, Bennet and Sehr, David and Dardyk, Gregory and Chen, J. Bradley and Muth, Robert and Ormandy, Tavis and Okasaka, Shiki and Narula, Neha and Fullagar, Nicholas. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages, 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[48] The llvm compiler infrastructure. http://llvm.org/. accessed 20.11.2014.

[49] Gcc - the gnu compiler collection. https://gcc.gnu.org/. accessed 20.11.2014.

[50] cjson. http://cjson.sourceforge.net. accessed 20.11.2014.

[51] Gdb/machine interface library. http://libmigdb.sourceforge.net. accessed 20.11.2014.

[52] Bottle: Python web framework. http://bottlepy.org/docs/dev/index.html. accessed 20.11.2014.

[53] Python bindings for the webkit gtk+ port. https://code.google.com/p/pywebkitgtk/. accessed 20.11.2014.

[54] Luís Damas. *Linguagem C*. FCA - Editora Informática, 17nd edition, 1999. ISBN:978-9727221561.