Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Bachelor of Science Thesis

Abstract Visualization of Algorithms and Data Structures

Authors

Johan Gerdin | Ivar Josefsson | Dennis Jönsson
Simon Smith | Richard Sundqvist

June 1, 2016

Abstract Visualization of Algorithms and Data Structures
DATx02-16-23
Johan Gerdin, Ivar Josefsson, Dennis Jönsson, Simon Smith, Richard Sundqvist

Supervisor: K.V.S. Prasad
Examiner: Niklas Broberg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden June 2016

Abstract Visualization of Algorithms and Data Structures
Chalmers University of Technology & Gothenburg University

Abstract

Within the field of visualization it is common to distinguish between Algorithm Visualization (AV) and Program Visualization (PV). AV uses high level abstractions to demonstrate how algorithms works, while PVs are debugger-like and display low level information about programs.

There exists a variety of visualization tools within both AV and PV today. However, few tools provide a good combination of the two. There is a lack of tools which can produce visualization with a high abstraction level from code in common programming languages. Moreover, most tools are either restricted to a certain programming language or a set of algorithms. Therefore users have to learn a variety of tools for different programming languages and algorithms.

The goal of this project is to combine AV and PV in a single system. The system is based around a communication contract which connects programs written in any language to any visualization tool. In this way, a programming language will immediately have a rich selection of visualizations available once an interface has been implemented for it. At the other end, new visualizations and visualization tools can be constructed without concern for how program execution will be recorded.

As a proof of concept, we have developed interface prototypes for Java and Python programs. Accompanying these are visualization tool prototypes, one written in Java and the other in HTML5.

Keywords: Algorithm Visualization; AV; Program Visualization; PV; Education; Visualization Tool

Abstract Visualization of Algorithms and Data Structures
Chalmers University of Technology & Gothenburg University

Sammanfattning

Det är vanligt att inom visualiserings-fältet att skilja mellan Algoritm Visualisering (AV) och Program Visualisering (PV). AV använder sig av en hög abstraktionsnivå för att visa hur en algoritm fungerar, medan Program Visualiseringar innehåller mer låg-nivå information på ett sätt som liknar en debugger.

Idag finns det existerande verktyg inom både AV och PV. Det är dock få som kombinerar dessa väl. Det saknas verktyg som kan producera visualiseringar av hög abstraktionsnivå från källkod, skrivna i vanligt förekommande programspråk. Utöver detta så brukar dessa verktyg vara begränsade till ett visst språk eller ett fixt antal algoritmer. Detta tvingar användare att lära sig en mängd olika verktyg för olika programspråk och algoritmer.

Målet med detta projekt är att kombinera både AV och PV i ett och samma system. Systemet bygger på ett kommunikationskontrakt (Communication Contract) som kopplar ihop ett program, skrivet i ett godtyckligt programspråk, med ett visualiseringsverktyg. Detta betyder att ett programspråk kommer direkt att ha tillgång till en stor mängd visualiseringar efter att ett gränssnitt har skapats för det. På samma gång kan visualiseringsverktyg byggas och användas, utan att behöva ta hänsyn till hur exekveringen av program ska spelas in.

Vi har utfört en konceptvalidering där vi har utvecklat gränssnitts-prototyper för Java och Python program. Vi har också byggt två prototyper av visualiseringsverktyg, ett skrivet i Java och det andra i HTML5.

Nyckelord: Algoritm Visualisering; AV; Program Visualisering; PV; Utbildning; Visualiseringsverktyg

Acknowledgements

First and foremost we would like to thank Jonathan Skårstedt, who had to leave us for health reasons, for being a part of the group and contributing to this project.

We would like to thank our supervisor K.V.S. Prasad for being with us during the entire semester; giving us guidance and comments on our report, brainstorming ideas for visualizations and extensions of our program as well as general non-related banter. Without the help and feedback from Prasad the project would be missing several unique and interesting features, in particular the multiset programming.

We would thank the EIRA (DATx02-16-39) group, for the interest they've shown in our project and for many rewarding discussions.

And finally, we would like to thank our opposition (Programmable Physical Toys, DATx02-16-17), whose feedback helped us make the report more comprehensible and well-structured, e.g. making the introductory part more potent in conveying what we did and why we did it.

Contents

1

Introduction

Courses within computer science are often programming intensive. It is usually expected of students to learn the practices of the field through hands-on experience. Beside the practical elements, students also have to acquire a fairly large body of knowledge. This is usually related to core knowledge and abstractions, like data structures, algorithms and underpinning theories of the field. More importantly students have to be able to grasp and implement the inner workings of algorithms such as sorting and searching, and their supporting data structures [1].

To succesfully implement algorithms in code, experience and knowledge is required. It is important to grasp the nuances and possible pitfalls of the algorithm as well as the technical implementation. Students often lack this knowledge and experience, which means that a lot of mistakes are usually made. Some tools to help fixing and finding these mistakes are low-level textual debuggers and industrial development environments. While these provide good language and platform dependent detail, they lack any abstraction for the algorithms. The behaviour of the algorithm is overshadowed by irrelevant program details [2].

In recent decades tutors as well as students of computer science have started turning to different kinds of visualization tools in an attempt to ease the learning process [3, 4, 2]. In the field of visualization, it is common to distinguish between two areas, namely AV (Algorithm Visualization) and PV (Program Visualization) [4, 5]. AV rarely has anything to do with code for the end user. It usually provides a visualization with high level abstraction which should be easy to understand. PV on the other hand is always related to code, usually in some programming language. The visualizations in PV tend to be debugger-like, providing a low abstraction level to expose details about the program execution.

Today there exists a variety of tools within both of these areas, easily available on the web. Thanks to significant advancements within the field in recent years, tools have quickly advanced and gone from outdated technologies like Java Applets to modern and powerful technologies like HTML5 [4, 6]. Regard-

less of recent advancements, decades old issues regarding usability and extendability remain. In addition to these issues, we have discovered that there is a lack of tools which provide a good combination of AV and PV. Especially tools that are simple to use and that provides a visualization with a relevant abstraction level.

1.1 Research background

In recent years there has been some research efforts devoted to improve the field of visualization [2]. The majority of this research have pinpointed a lot of existing problems. However, new visualization technology so far has largely been unable to resolve these issues.

There is currently no widespread adoption of visualization tools in computer science education. There are several issues that plague visualization tools and we have found no programs that adequately solve these issues. If these issues are solved we are much more likely to see widespread use of visualization tools [5]. Overall there are three identified areas in which visualization tools usually are lacking:

1. Algorithm prototyping mechanism
2. Provided visualizations
3. Level of visualization abstraction

1.1.1 Prototyping Mechanism

When using algorithm visualization it is important to be able to modify and change the code and see how it changes the visualization. Lack of good algorithm prototyping mechanism means that visualization tools do not provide a good way interface between code and visualization. The process of prototyping an algorithm can often be so complex or impractical that people simply refrain from using visualization tools [2, 4, 5, 7]. This problem usually embodies itself as one or a combination of three typical cases:

1. To generate visualizations from code, one is required to rebuild or clutter it with visualization specific constructs.
2. The whole visualization has to be built from scratch in an, to the user, unknown language or technology.

3. In order to visualize a program, parts of the program code must be migrated and built in foreign environments. This process may introduce new errors and cut off dependencies.

1.1.2 Provided visualizations

Many of the existing visualization tools are products of small scale scientific efforts [7, 6]. They are usually specialized in small set of visualizations and are limited in the way that they can be used. There exists some modern web based tools which provide a larger collection of visualizations. However, even these tools are limited to certain types of visualizations and are hard to extend. Because tools are limited, one has to learn a variety of tools to visualize different algorithms in different ways.

One way to solve the issue with lacking visualizations is providing a tool allowing for easy extendability, and there has been some previous efforts in overcoming the issue of extendability. ANIMAL is one prominent example. ANIMAL is a Java application that can produce visualizations from user defined scripts [8, 9]. Some applications have integrated the animal scripts and can produce visualizations on the ANIMAL system, most of which were produced in the late 90´s. The ANIMAL system, along with many other visualization tools from that time, seem to have faded and been replaced by modern and more engaging visualization tools [4, 6].

We believe that one of the main reasons why systems like ANIMAL and other tools gradually disappear after being put into use, is that they are tied to a certain implementation. Again, taking ANIMAL as an example: There are ways of producing the visualization scripts from different sources, but in the end, only ANIMAL is capable of producing visualizations from them.

1.1.3 Abstraction Level

In this project we have been working within the definitions of both AV and PV. As mentioned earlier AV rarely has anything to do with code. PV, on the other hand, generates visualization from program code. The difference between these types of tools becomes problematic because there exists no tools which can provide a visualization from a users own code with a high abstraction level, the closest we found was the tool PyAlgoViz. However, using PyAlgoViz, one has to merge the visualization code with the program implementing the algorithm. This is time consuming and introduces a lot of potential errors.

1.2 Purpose

Our purpose is to combine AV and PV in a single system that works for any programming language. This should enable programmers of any skill-level to easily visualize their programs in a variety of ways using the same system.

1.2.1 System Architecture

The system is constructed around a communication contract. In this system, the tools for prototyping an algorithm has been separated from the tools for visualization, making them independent modules. Overall, this system consists of four major modules:

Programming Language Interfaces: Interfaces can be developed for any imperative programing language. Using these interfaces, users should be able to specify how an algorithm should be visualized.

Communication Contract: The contract is the shared language of the different modules in the system. Programming language interfaces log the execution of a program according to a communication contract. This log can then be read by Visualization and Interpreter type modules.

Visualization Tools: Tools can be implemented in any programming language to consume a log and produce visualizations from the logged operations.

Interpreters: An interpreter combines groups of logged operations into operations of higher abstraction level. Interpreted logs will generally result in visualizations which are easier to understand than those produced by their uninterpreted counterparts.

The modules and their relationship is summarized in figure 1.1.

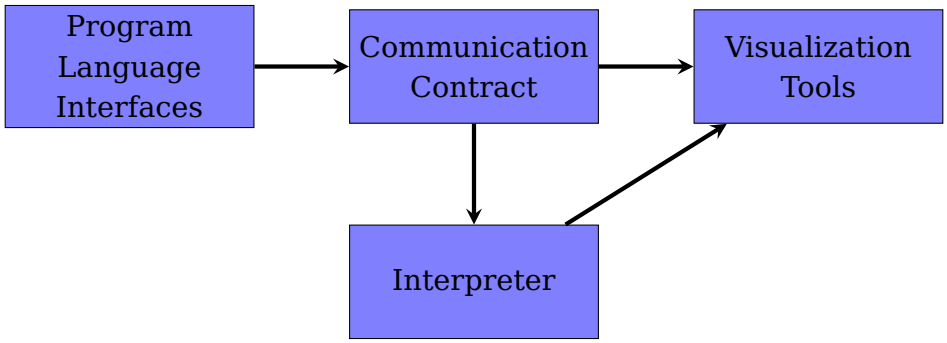

Figure 1.1: Overview of the main modules and how they relate to each other.

This separated architecture has enabled a PV/AV-system which is independent of the implementations of the communicating modules. In this way it is easier to extend the system for additional languages as well as adding new visualizations and keeping the system updated with the latest technologies.

1.3 Delimitations

This report argues for a form of visualization which combine the benefits of AV and PV. This is meant to serve as a proof of concept for the visualization community. The main goal is not to supply the field with a ready to use product. It is rather to supply the visualization community with an initial prototype which successfully incorporates our separated architecture and works as good combination of both AV and PV. In this section we will further discuss the limitations of each module in our system.

1.3.1 Communication Contract

The communication contract will only handle a basic set of operations on primitive types and primitive type arrays. The contract log will be strictly deterministic. If further functionality is to be added, the log has to be extended.

1.3.2 Programming Language Interfaces

Programming language interfaces have been implemented for Java and Python. The specifics of each implementation are dependant on the technical details of each language. Both will produce a log according to the contract. The way the users interacts with the interfaces will however differ. Further, our interfaces will only handle primitive types and primitive type arrays. Objects will not be supported by the system prototypes. This project is only concerned with imperative languages. No efforts have been made to produce interfaces for languages in any other paradigm.

1.3.3 Visualization Tools

The visualization tools have been implemented as a Java desktop application and an HTML5 application. These tools are only able to consume the log according to the contract and produce visualizations from the logged operations.

1.3.4 Interpreter

The interpreter has been implemented in Java. The prototype module does not provide the system with any core functionality. The purpose is strictly to show that it is possible to incorporate such a module.

2

Project Stages

Our main channels for communications were the web and one to two weekly meetings. During the weekly meetings we discussed our progress and handled planning and workload distribution. Extra workshops was set up to facilitate sharing of knowledge, collaborative work and handling of development related issues.

2.1 Research Stage

Research and initial requirements analysis made up the majority of the first stage of the project. The goal of this stage was to identify user needs and gain an overall understanding of the current state of the field. The results of this research served as a base for our initial vision and main high level features of the system.

2.2 Development Stage

Early stages of development consisted of small prototypes. We intended to get a firm grasp of the various technologies needed to complete the project. Many of these proto-applications were completely scrapped and others were built upon to create the final framework.

2.3 Usability Testing

Small scale user testing was conducted. The tests were summative in nature. The subjects were first exposed to another visualization tool and then our framework. They were then interviewed about what tool they preferred. The sample size of the user tests were too small to draw any conclusions. They served more as a way to gather feedback and indicator of interest in the project.

3

Technical Background

The following sections are about the general background of the tools and concepts we have used to achieve our goals. The practical usage might not be apparent at first, but this will be clarified further down in the report.

3.1 Communication Contract

This section briefly describes the technology behind the communication contract.

3.1.1 JavaScript Object Notation

JavaScript Object Notation (JSON) is a human-readable markup language. The highly extensible and widely used format stores data in key-value pairs. Because of its widespread use, libraries parsing JSON have been developed by most languages in use today. JSON is language independent, but developers experienced in C-family languages will likely recognize the format and syntax.

JSON supports objects, arrays, strings and values. Values is a string, number, boolean, object, or array [10]. This means that you technically could have an unlimited number of layers wrapping and organizing the data. In reality however, many parsers are unable to deserialize data which is stored "too deep".

3.2 Programming Language Interfaces

To keep the modularity it is important that the visualization can be done in several programming languages. The way the programming language interface is written is almost always unique to the individual language. This allows the

interface to utilize and adapt to the peculiarities of each language, allowing the addition of the visualization to be as unobtrusive as possible. The intent is for the interface to capture the intended operations without modifying the result of the program which is being executed.

This approach allows prototyping and execution in the user's own environment. Some tools (such as Python Tutor) require users to migrate, build and run code in foreign environments in order to produce a visualization. Depending on the project set-up, this may be difficult or even impossible. In some cases (especially commercially), users may not be willing or able to expose their program in this way.

Java has long been one of the most frequently used programming languages in computer science education [11], and it is the single largest language in all major fields in the industry [12]. However, studies have shown that Python is becoming increasingly popular in education and has surpassed Java as the number one programming language for beginners at US universities [13]. We chose to cover both Java and Python to reach as many users as possible.

3.2.1 Abstract Syntax Tree

In order to easier process source code one can generate what is called an Abstract Syntax Tree (AST). For example, an assignment of 1 + 2 to a varaible x would translate into the AST shown in Figure 3.1. The tree retains the structure of the code and could potentially be parsed back into source code given the tools.

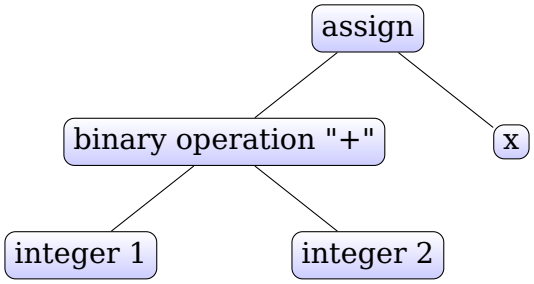

Figure 3.1: x := 1 + 2 processed into an AST

3.2.2 Java Annotations

Java provides a form of syntactic metadata called annotations [14, 15]. These are used to mark programming constructs for them to later be processed during compile or runtime. As it turns out, this is a fairly simple and efficient way

of providing the user with a method of extracting operations, thus allowing prototyping of visualizations.

Annotations are used for three main purposes:

1. Provide information to the compiler.
2. Compile-time and deployment-time processing.
3. Runtime processing

Annotations have a single '@' in the beginning to indicate to the compiler that what follows in an annotation. Annotations can be placed at declarations of classes, fields, methods, and other program elements.

There are three types of annotations in Java: Normal annotation, Marker annotation, and Single element annotation. A normal annotation specifies the name of the annotation together with a list of key value pairs as follows:

```
1     @TypeName( key_1 = value_1, .., key_n = value_n )
```

The other two are short-hand versions of normal annotation, marker being without a key-value pair list for example: $@TypeName$, and Single Element Annotation takes only one value so the key is discarded: $@TypeName(value)$.

There are some predefined annotations in the Java SE API. Some are used by the compiler and some apply to other annotations. For example, the predefined in the java.lang package are the following: $@Override$, $@Deprecated$, and $@SuppressWarnings$.

Custom annotation processing was standardized in Java 6 through JSR269 with the purpose of making annotation processing simple and available to the user through a standard Java API [16]. An annotation processor is just another Java program running on the Java virtual machine (JVM). The processor class has to implement the javax.annotation.processing interface and is run by the compiler [17]. Since the introduction of annotation processing users have been able to declare their own annotations. It is then necessary to create a program that will be used to process these annotations during compile time. This can for example be used to generate new source files and integrate user code into third party frameworks. Many large scale industrial Java APIs such as Java EE frameworks and JUnit are making heavy use of this [17, 18, 19].

3.2.3 Python Abstract Syntax Tree Module

The documentation for the Python Abstract Syntax Tree module explains that "The AST module helps Python applications to process trees of the Python abstract syntax grammar" [20]. To extend the meaning of this: An abstract syntax tree is generated from python source code [21]. This can then be used to analyze and/or modify the source code. See section 3.2.1 for more information on ASTs.

For example, if one has the given file called main.py detailed in listing 3.1. This source code would be processed into an AST containing a root node for the python Module, in this case named main [22]. This root node contains a body element which lists all statement-nodes in given order. In main.py the two statement-nodes would be, in order: An Assign node containing one target node Name with id "a" and a value List node with three Num child nodes (see figure 3.2). Another Assign node with one Subscript node as a target and a Num node containing the integer 5 as the value to be assigned. The Subscript node has two children, one Name node that is indexed by a Num node representing the integer 0 (see figure 3.3).

Listing 3.1: main.py

```
1    a = [1,2,3]
2    a[0] = 5
```

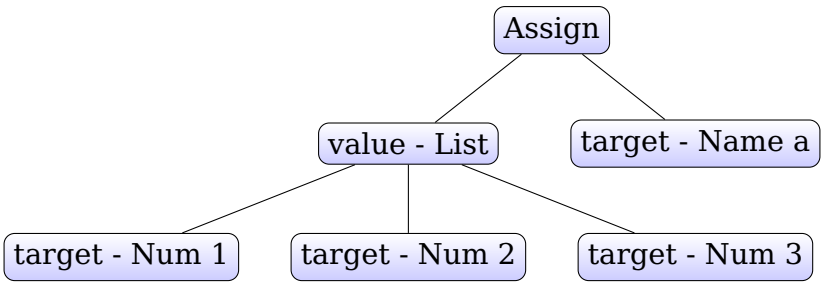

Figure 3.2: a = [1,2,3] processed into an AST

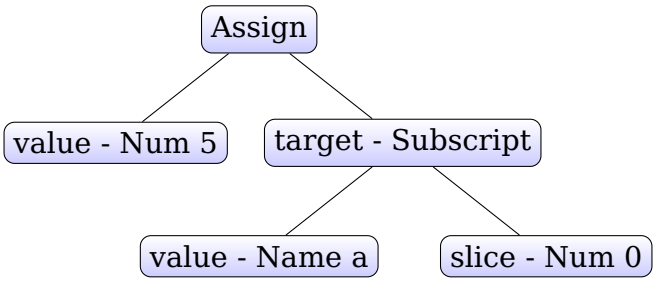

Figure 3.3: a[0] = 5 processed into an AST

The AST module contains two helper classes to traverse and visit nodes of these trees. One is called NodeVisitor and this class visits nodes of the tree without modifying it, this can be used to analyze the code structure without accidental modification. The other is a subclass of NodeVisitor called NodeTransformer that allows for modification of the tree on top of traversal.

3.3 Visualization Tools

This section briefly describes the technologies behind the visualization tools which have been implemented for the framework.

3.3.1 JavaFX

JavaFX is a Java API for building applications of many different types. The library offers rich functionality for building applications swiftly and easily while still providing a wide range of functionality. Among these are advanced 2D and 3D transformations such as affine transformations, rotation and scaling.

JavaFX uses the Prism engine, which has pipelines to high-performance APIs such as Direct3D and OpenGL. This allows JavaFX programs to render detailed 3D graphics in real time [23].

Our decision to use JavaFX was based on the the libraries' powerful rendering capabilities and Java's cross-platform support. The ability to deploy FX applications on browsers may also prove useful in the future, though it is not a concern at this stage.

3.4 Programming by Multiset Transformation

When learning programming it is a common issue that a programming language also has to be learned. Learning technical details that is specific to the language instead of focusing on the general concepts and ideas is not always a fruitful endeavour. When learning programming it is thus important that the language being learned is as easy and simple as possible, while still being powerful. If the language fulfills these criteria it makes it a lot easier for the student, not having to learn about an additional area.

Programming using multiset transformations was originally conceived of as a concept in a paper written in 1993 [24]. The authors of the paper developed

a language that they called GAMMA, allowing a user to express concepts and algorithms using relatively pure mathematical notation.

One of the goals when programming with multiset transformations is to allow the user to express algorithms and solve problems using a language that is widely known and understood; mathematics. This allows a user to explore and try out different algorithms and get an idea of how to solve a problem. This solution can then be used as a tool and help when implementing a solution in a language. This is a different way to express algorithms.

Programming with multisets is done using three different parts: the input, the result and a conditional. The input specifies two sets which share the same structure. The conditional is then applied to these two sets, either evaluating to true or false. If the conditional evaluates to false nothing happens, if it evaluates to true then the two input sets are removed and replaced by the contents of result. Looking at listing 3.2, the input is two elements, m and n. The output is m if the conditional is true, which it is if m is bigger or equal to n. If we let arbitrary collisions occur between elements in the starting set, we know that we will sooner or later end up with one element. This element will have the property that it is the biggest of the original elements.

Listing 3.2: The "max" algorithm implemented using multiset transformation.

```
1     m, n -> m if m >= n
```

Programming with multiset transformations has several similarities with both functional languages and parallel programming. By the very nature of an algorithm written using multiset transformations it is capable of being run in parallel. There is no determined sequential order that the algorithm processes data in, this is useful as it among other things allows the user to avoid being locked down in a style of thinking similar to that of imperative programming. In fact, this was one of the goals and purposes when the original authors conceived and developed programming using multiset transformations and their implementation of it [24].

4

Implementation

This chapter briefly describes how the different parts of the implementation come together. The appendix contains more detailed descriptions of some selected modules.

4.1 Implementation Overview

The implementation of the system can be separated into four major parts:

1. Programming Language Interface
2. Communication Contracts (PEL and IL)
3. Interpreter
4. Visualization Tool

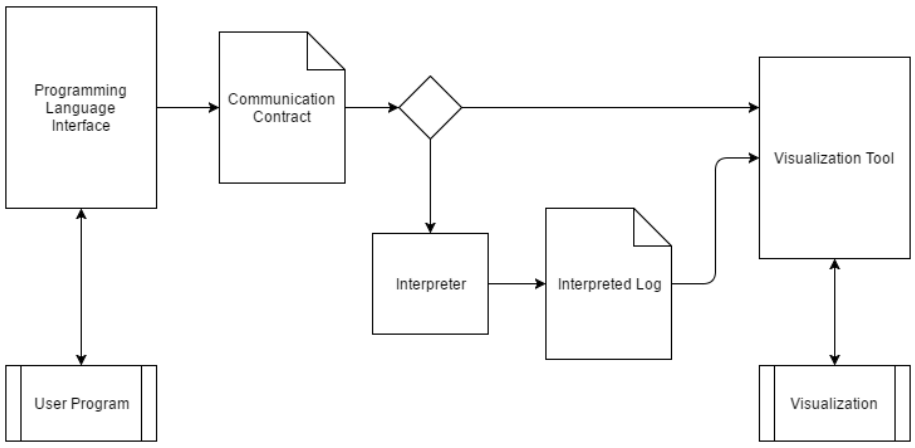

Figure 4.1: System overview.

The programming language interface observes the execution of a program, resulting in a log-file. This log-file is specified by the Communication Contract.

The Interpreter uses the contract to interpret the log-file. A Visualization Tool is used to visualize an interpreted or uninterpreted log-file. Figure 4.1 summarizes the relationships between these different modules.

4.2 Communication Contract

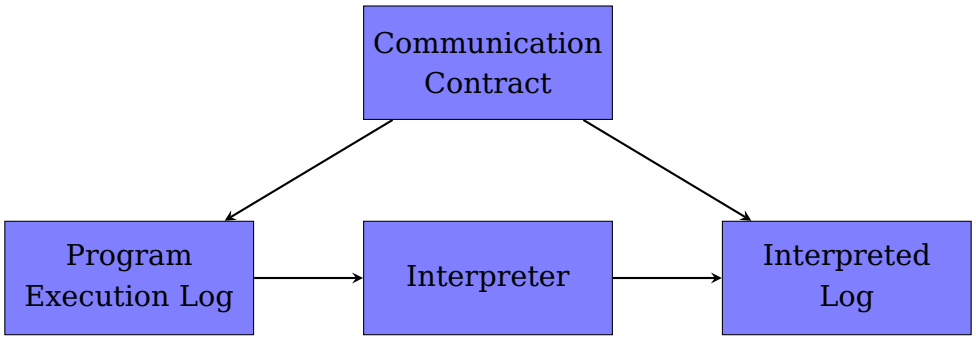

Figure 4.2: Communication Contract Overview. The Communication Contract specifies the format of the logs. The Program Execution Log can be interpreted into an Interpreted Log.

The Communication Contract specifies how the Programming Language Interfaces communicate with the Visualization Tools. This is done by specifying the format of a log-file based on a set of rules. Programming Language Interfaces generates this log-file from executed source code. This log can be visualized by any visualization tool that uses the Communication Contract. Figure 4.2 gives an overview of the Communication Contract and what it specifies. The components of this figure are explained further in this section.

There are two types of logs specified by the Communication Contract: The Program Execution Log (PEL) and the Interpreted Log (IL), as seen in Figure 4.2. The PEL is generated by the programming language interfaces in the form of a log-file. It stores the execution of a program as low-level operations. These operations contain information about some statement in the program, such as an assignment to a variable (see listing 4.1).

Listing 4.1: Example of a low-level operation

```
1    variable_x := 3
```

Metadata about the source code and the execution is also stored as a part of the log-file. This is, for example, information about variables contained within the logged operations. The log has been implemented using JSON (see subsection 3.1.1 for more information about JSON).

The IL is on the same format as the PEL, with some additional operations. It is created by *interpreting* some of the low-level operations in a PEL and creating operations of higher abstraction.

Listing 4.2: Three low-level operations creating a swap operation between x and y.

```
swap :=
    tmp := x
    x   := y
    y   := tmp
```

As an example, see listing 4.2. Here three sequential low-level operations form a swap of two values. This is stored in the IL as a single operation "swap". For more information about interpretation, see section 4.4.

To summarize: The Program Execution Log, or PEL, contains low-level operations capturing some statement in a program using a Programming Language Interface. The Interpreted Log, or IL, contains both low-level operations and operations of higher abstraction. These are created by grouping the low-level operations together forming one operation, again see Listing 4.2 for an example. The Communication Contract specifies what is contained within both logs. The logs are implemented using JSON.

See the Operation section (B.4.1) of the Appendix for a detailed description of the PEL and IL.

4.3 Programming Language Interfaces

Program execution must be logged without altering the original program. This section explains how we have implemented our interfaces for creating a PEL, specified by the Communication Contract (see section 4.2 for an explanation of the Communication Contract), in Java and Python.

4.3.1 Java Interface

The Java Interface is built on the Java annotation processing API and consists of three major parts:

1. A set of annotations (coding/prototyping)
2. Annotation processor (compile-time processing)

3. Generated visualization classes (runt-time processing)

The interface between a user and a visualization is strictly through a set of predefined annotations. This provides a simplistic and straightforward way of prototyping visualizations from a Java program. Annotated programming constructs are processed during compile-time and provides the API with information such as names and types of variables and data structure related metadata. This information is used to generate visualization classes which are the main program for generating a PEL of the program.

Because constructs in Java programs can take on a very complex behaviour such as side effects when manipulating objects, the initial version of the API can only handle primitive types and primitive type arrays. It will only be possible to visualize those constructs within a Java program.

The processing of a user program happens in three major stages: Annotation processing, program parsing, and code generation and compiling. During the annotation processing stage, annotated constructs are passed by the standard Java processing API to the processor. The processor records information about the constructs, such as identifiers, types and scope. This information is then used during parsing.

In the second stage, all annotated classes are reloaded by the API. In this stages we use a third party API called Java Parser to parse the source files and generate an in-memory representation of the program Abstract Syntax Tree (AST) [25].

In the third and final stage of the processing, the AST is traversed and manipulated using information provided by the annotations. This is done to capture the operations performed on the annotated constructs. After the entire AST has been traversed, a new source file is generated from the AST and written to the source folder alongside the original source files. The sources are then picked up by the compiler and compiled to class files. These files are identical to the originals, with four exceptions:

- Operations which access an annotated variable have been replaced with method calls to record the operation.
- Additional dependencies relating to the methods logging these operations have been added.
- The files have been given the Visual suffix. For example, Main.java becomes MainVisual.java
- The annotations have been removed.

To generate the PEL file, the user must run one of the newly generated Visual files as opposed to the annotated originals. So, if a user has chosen to visualize Main.java, she must run the MainVisual.java file for a PEL to be generated.

For this to work either one of the generated classes must have come from an executable class in the original program, or the user invokes the classes in some other way. If only a subset of all involved classes in a program are annotated, the program will still work through the original classes, however visualization data might be lost. Of course, this also means the user can choose not to annotate classes which perform operations the user does not wish to visualize.

The Java interface is made up of a set of two predefined annotations: @Visualize and @VisualClass. Both have to be applied to the right constructs in order to produce a meaningful visualization.

The **@VisualClass** annotation is applied to classes which are part of the visualization program. Classes annotated with **@VisualClass** will be picked up by and processed by the annotation processor. Dependencies to this class will also be swapped for the generated class in all parts of the visualization program.

The **@Visualize** annotation is the most interesting one from a user perspective. It is applied to the constructs that a user want to visualize. Due to limitations of the Java Annotation API, the annotation can only be applied to class fields and method parameters. Constructs which are marked with @Visualize are passed to the annotation processor and later tracked in the generated visualization program.

The **@Visualize** annotation is applied in the following way:

```
1    @Visualize(abstractType = "some_type")
```

The annotation takes one argument where the user can define the abstract type of the construct, for example array, matrix or tree.

4.3.2 Python Interface

The Interface in Python takes some user-specified settings as input and produces a PEL based on these settings. The settings used is specified by what is called a *Settings Variable*. This variable contains the following:

- *Root Directory* of program
- *Files* to be observed

- *Variables* to be observed
- A *Main File* or start-up script to be executed
- *Output Destination* the PEL file

The *Root Directory* contains all necessary files to run the program. *Files* contain the code that should be observed and produce the PEL. For example, a program with an observed file running a sorting algorithm would produce operations in the PEL from statements executed by this algorithm. The Python Interface will observe only statements containing *Variables* specified by the user. These *Variables* must be contained within previously specified *Files*.

The *Main File* is the file that when compiled and run by the Python Interpreter will run the user-program. An *Output Destination* is specified in the form of a path to some directory in the users local environment.

Generation of the PEL uses a specified Settings Variable to copy the entire program from given *Root Directory* and create a *temporary environment* which will be used to generate a PEL. The general procedure is outline below:

1. Generation of Abstract Syntax Trees using AST module
2. Modify Abstract Syntax Trees and translating back into source code
3. Execution of modified user program resulting in a PEL

Files from Settings Variable is transformed into Abstract Syntax Trees. The ASTs are traveresed and statements containing observed *Variables* are modified to generate an operation in the PEL. The trees are traversed and modified by using the Python Abstract Syntax Tree Module explained in subsection 3.2.3.

NodeTransformers called OperationTransformers are used for traversing and finding statements that should be modified. They are modified in a way that captures the execution of specific statements in a function call. These are given here, function calls are within quotation marks:

- Write - an assignment, for example *x := 1* translates to *x := "write 1 to x, return 1"*
- Read - a read from a variable, for example *if x = 1* translates to *if "read x, return x" = 1*
- Pass - a reference to a variable is passed to a function parameter

ExpressionTransformers iterate through the found statements and translates the expressions within into something that can be read by a function call cre-

ated by OperationTransformers. The different expressions are detailed here:

- Name Node translates to ('var', name of node, evaluated value (during run-time) or 'Store' if the node is the target of an assignment). For example, variable *a* would become ('var','a',a).
- Subscript translates to ('subscript', list or dictionary, $index_1, ..., index_n$). For example, subscript to variable a with index 1 would become ('subscript',a,1).

The modified ASTs are then translated back into source code and written to the *temporary environment* (for more information on how the ASTs are translated to source code, see appendix C). When a program in this environment is executed, a PEL is produced.

See figure 5.1 of the Results section (5.2) for an example of how the Python Interface is used.

4.4 Interpreter

An Interpreter evaluates groups of primitive operations and attempts to consolidate them into higher abstraction level operations (see section B.4.9 of the Appendix for more information about the Interpreted Log). Operation groups are evaluated sequentially in the order in which they are received, generally from a log file. Our implementation of interpretation works on a fixed number of operations and cannot recognize patterns which are not explicitly tested for.

For example, a loop which runs for n iterations and increments a variable by one each time could not be consolidated, but it would be possible to detect a loop which runs exactly k times, if a consolidation grammar was created specifically for it. This behaviour also makes it unsuitable for interpreting programs which has not yet finished executing. While possible, behaviour may be erratic and the resulting visualisation will likely raise more questions than it answers.

It is important to note that interpretation may not be perfect, with a possibility of producing both false negatives and false positives depending on the implementation. *The Program Execution Log should always be considered the authoritative party if the final state of a model subjected to a PEL and an IL diverge.*

4.4.1 Example: Interpreting a Swap Operation

A swap is traditionally performed by storing the first operand in a temporary variable, overwriting it with the the second operand, then transferring the first operand from temporary storage to the second variable. This is summarized in figure 4.3.

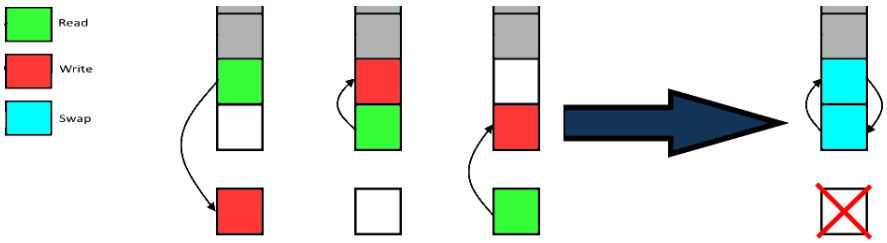

Figure 4.3: Interpretation of a Swap operation. Two elements in an array are swapped using the temporary variable at the bottom.

Our implementation uses a kind of hypothesis testing, that is the program makes a number of assumptions and then tests the veracity of these assumptions. If all tests pass, a Swap operation is constructed, replacing the original operations.

The code snippet shown in listing 4.3 is the "hypothesis testing" part of the consolidate operation in the Swap class. Checks for null pointers and so on are done beforehand.

Listing 4.3: Swap interpretation using "hypothesis testing".

```
Locator var1, tmp, var2;
// Operation 1: Set var1 -> tmp
var1 = rw0.getSource();
tmp = rw0.getTarget();
if (tmp.index != null) {
        return null; // tmp should not be another array.
}
// Operation 2: x -> var1?
if (rw1.getTarget().equals(var1)) {
        var2 = rw1.getSource(); // Set x = var2
} else {
        return null;
}
// Operation 3: tmp -> var2?
if (!(rw2.getSource().equals(tmp) &&
   rw2.getTarget().equals(var2))) {
        return null;
}
//Create and return Swap operation..
```

4.5 Visualization Tools

This section goes through the visualization tools we have implemented. Two tools have been developed, one made in Java as a desktop application and the other was made using HTML5 and Javascript.

The desktop application is more sophisticated, with additional features aimed at making the visualizations more engaging and easier to understand. It is also capable of drawing some operation types (such as Swap) that the web version does handle. Finally, only the Java version does animation.

4.5.1 Desktop Visualization

The desktop application is based on JavaFX, the spiritual successor to Swing. FX was chosen primarily because it is easy to use, and while performance is not good enough for heavy duty 3D work (though JavaFX performance is generally much better than Swing) it was assumed this would not be an issue with the type of drawing for which the framework will be used. In addition,

JavaFX applications have the capability of being deployed on the web. This is not something we're planning to do in the near future, but may be done in the future.

The guiding principle when creating animations is that they should make it as easy as possible for the user to understand what the program which produced the log is doing. This is primarily achieved using animation. For example, a Swap operation might be visualised by the two elements moving across the screen until they've reached their destination. This is shown in figure 4.4.

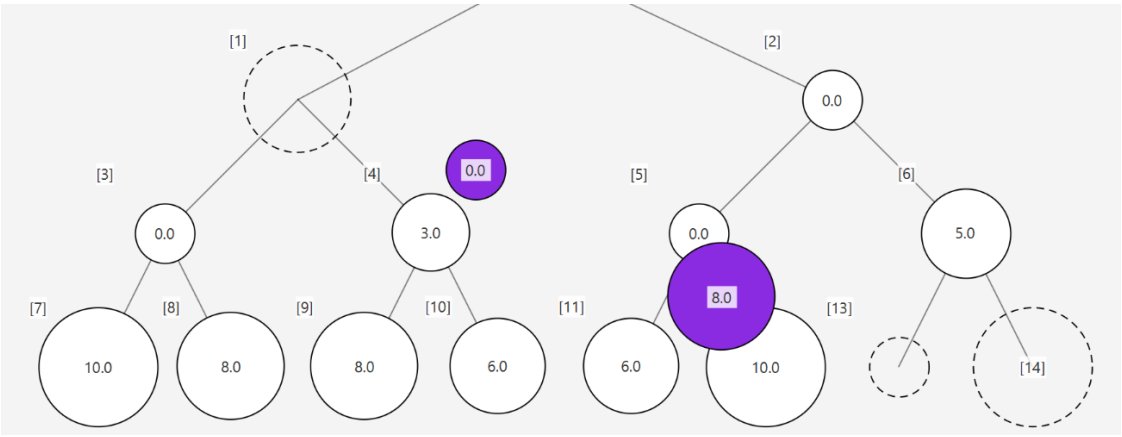

Figure 4.4: A Swap being performed on a K-Tree (with K=2) implemented using an array. The elements are traversing the canvas to reach their target locations ([1] and [13]) in the tree.

The nodes in the tree have been scaled according to their value relative to the other elements. In figure 4.4, the largest element, having a value of ten, is twice the radius of the smallest elements at zero. Both animation and relative node sizes can be turned on and off at will.

4.5.2 Web Visualization

The web-based visualization tool was created in HTML5 in combination with the *three.js* JavaScript 3D Library. HTML5 applications make great consumers as well as producers of the communication contract because it is implemented in JSON, which is basic JavaScript object notation. Due to the vast variety of graphics APIs, HTML5 is a powerful technology to building visualization tools on. Most modern visualization tools are now being developed as HTML5 or other types of web applications [4, 6].

We have made the web interface as simple as possible, only focusing essential details. The only initial functionality provided to the user, is uploading files in a drag and drop manner.

Because of the wide support of graphics in HTML5, 2D as well as 3D visualizations are possible in the web tool.

4.6 Programming by Multiset Transformation

We feel confident in the system we have built. However, it does require a fair amount of programming knowledge. As a bit of a sidetrack we decided to build a standalone application that had basic support for programming with multiset transformations. Our implementation allows a user to visualize basic algorithms without a need for any programming languages. All that is required by the user is some basic knowledge of mathematics.

There is a stark difference between the programming with these multiset transformations and programming using the Java language. Especially as a beginner programmer there is a lot of terms, concepts and ideas around the Java language that need to be learned. In comparison, with Multiset programming there is very few concepts, and those that need to be learned are expressed in a language that is already known.

When using what we refer to as the multiset program, the user is required to fill in four fields before a simulation can be started. We call these fields: input, output, conditional, and range. The purpose of range is to allow the user to define the ranges that define how values are generated. The input, output and conditional fields are three different split up parts of the

In order to visualize the multiset transformations we have a visualization where each entry in the starting set is represented by a ball. It was inspired and is very similar to a type of visualization done in chemistry when modelling chemical reactions [26]. Whenever two balls collide, we solve the collision and then send the two balls as entry values to the multiset representation. The multiset transformation returns any elements to be kept, and any balls not returned in this way will be removed from the visualization. The implementation supports basic simple comparisons in the conditional such as "greater than", "equals" etc. The visualization of the balls is done on a canvas in JavaFX, the balls are modelled using simple physics based collision calculations. At start each element (or ball) recieves a random movement vector at their preset positions, independent of the values of any elements contained within the ball.

5

Results

We have created a framework which combines AV and PV and is capable of producing visualizations from programs implemented in Java or Python. The way the framework has been constructed, it is possible to add interfaces for additional programming languages. This enables users to easily visualize an algorithm implementation using familiar constructs within the programming language of choice. Users work in their own environment, keeping their programs and dependencies intact. The Java and Python interfaces produces execution logs of annotated programs. These logs can in turn be consumed and visualized by any visualization tool within the framework, regardless of their implementation language. This enables easy visualization of the user's own code at a variety of abstraction levels using different visualization tools.

5.1 Java Interface

The Java interface enables users to visualize an algorithm implementation using annotations within the Java language. Because of the common occurrence of annotations, users familiar with the language are able to quickly adopt the interface. Listing 5.1 shows an example of an annotated Java program. Here the class as well as the parameter "intArray" in the sort function have been annotated. The user have chosen to visualize the parameter as an array by passing a string to the @Visualize annotation.

Listing 5.1: An annotated bubble sort implementation in Java

```
1
2  @VisualClass
3  public class BubbleSort {
4
5      public static void sort(
6          @Visualize(abstractType="array") int intArray[]) {
7          ...
8      }
9
10     public static void main(String [] args){
11        sort(new int []{1,5,9,11,5,8,3,14,4});
12     }
13 }
```

5.2 Python Interface

```
from context import sample
from sample import pylogger
from os.path import abspath

if __name__ == '__main__':
        # API for create_settings and Variable needed
        # to create PEL.
        #
        # pylogger.create_settings Parameters
        #       root_directory, — root directory of program
        #       files,                  — files to observe
        #       variables,              — variables to observe
        #       main_file,              — main file of program
        #       output                  — where to wite PEL output
        # pylogger.Variable Constructor
        #       name,                   — name of variable
        #       rawType,                — type of variable
        #       attributes=None,
        #       abstractType=None
        settings = pylogger.create_settings(
                abspath('./test'),
                ['/main.py'],
                sample.pylogger.Variable('array','list',attributes={'size' : [10]}),
                '/main.py',
                abspath('./test')
        )
        pylogger.run(settings)
```

Figure 5.1: Wrapper script for example-program located in /test

Usage of the resulting python interface is shown in Figure 5.1. This wrapper script creates a settings variable by calling create_settings supplied by the pylogger module. Then it uses the run method to generate a PEL file in specified output-folder based on the other paramters given about the source code. For a more detailed explanation of the settings variable, see subsection 4.3.2.

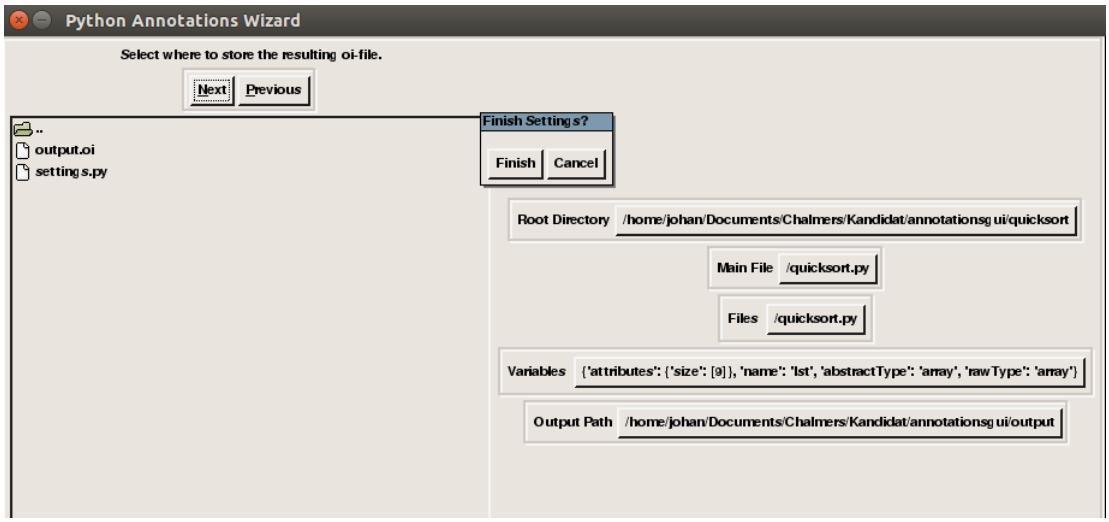

Figure 5.2: Image of pylogger wrapper gui, containing a fully specified settings variable

A prototype of a GUI wrapping the pylogger-module is shown in Figure 5.2. To the right all parameters of a settings variable has been specified for a program called *quicksort.py*. The program is contained within the directory *quicksort*. One variable is observed, called "lst" of type array. The output location is specified to a folder called *output*.

The interface is fully operational, however, user-interaction has not been taken into full consideration. The prototype exemplifies how usage of the pylogger module could be simplified over the script written in figure 5.1.

5.3 Desktop Visualization

The visualization module for the desktop is built on JavaFX. Figure 5.3 shows the animation of a read operation without a target. Node sizes have been set relative the other elements in the data structure. In the picture, the largest element is twice the size of the smallest element. This can be turned on and off, just like animation.

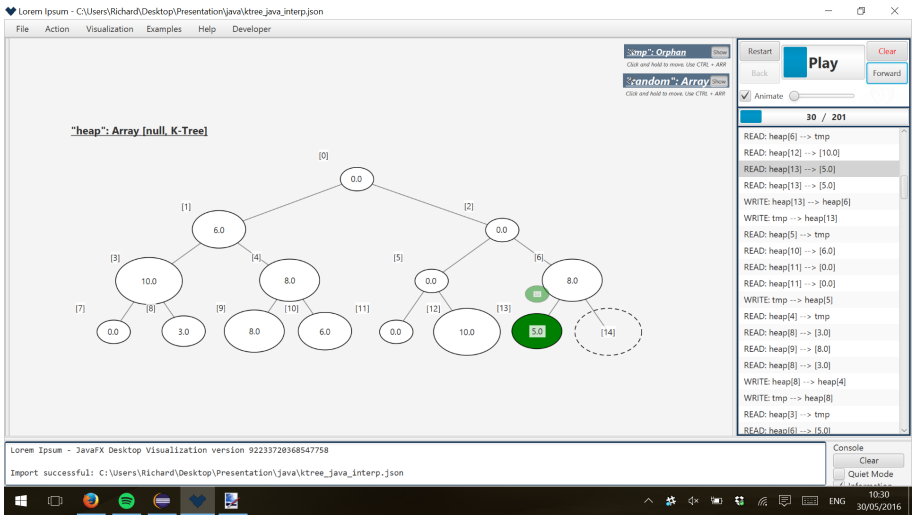

Figure 5.3: A Read operation without a target being visualized in the desktop application. The animated element will slowly shrink and fade until it is no longer visible.

Data structures can be scaled and moved where ever the user prefer them to be, or she may let the program place the visualizations for her. Two minimized data structures are visible in the upper right corner. To the right is a control panel, along with an operation queue. The operation which is currently being animated has been highlighted by the program. The blue filling of the play/-pause button indicates the time left until the next operation, while the one below it indicates overall model progression. The slider is used to set automatic playback pace; it has been turned all the way down to make it easier to take screenshots of the animation.

5.4 Web Visualization

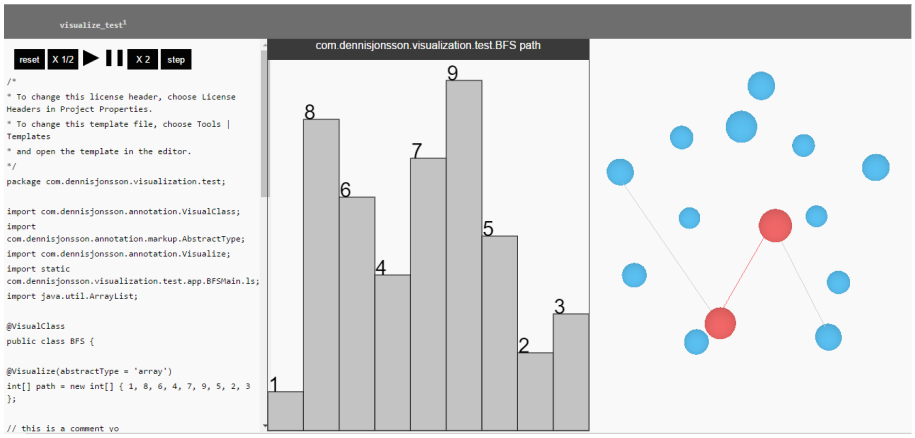

Figure 5.4: Breadth First Search visualized with the web application.

We have implemented a visualization applet for the web which is shown in figure 5.4. The component to the left is the source code and controls used to control model progression. In the middle column the array that contains the resulting path of the algorithm is displayed. To the right the graph is drawn with current path is represented by lines between the nodes.

5.5 Multiset Implementation

We have built a standalone application that serves as a prototype of programming using multiset transformations (see figure 5.5). It is a rather rudimentary implementation, but it serves as a proof of concept giving the user an idea of the possibilities. It is possible for the user to implement and define simple algorithms such as Max, Min, Primes etc. While more advanced algorithms such as sorting is currently unavailable. The application has several different examples that can be loaded, to help the user understand how the application is used.

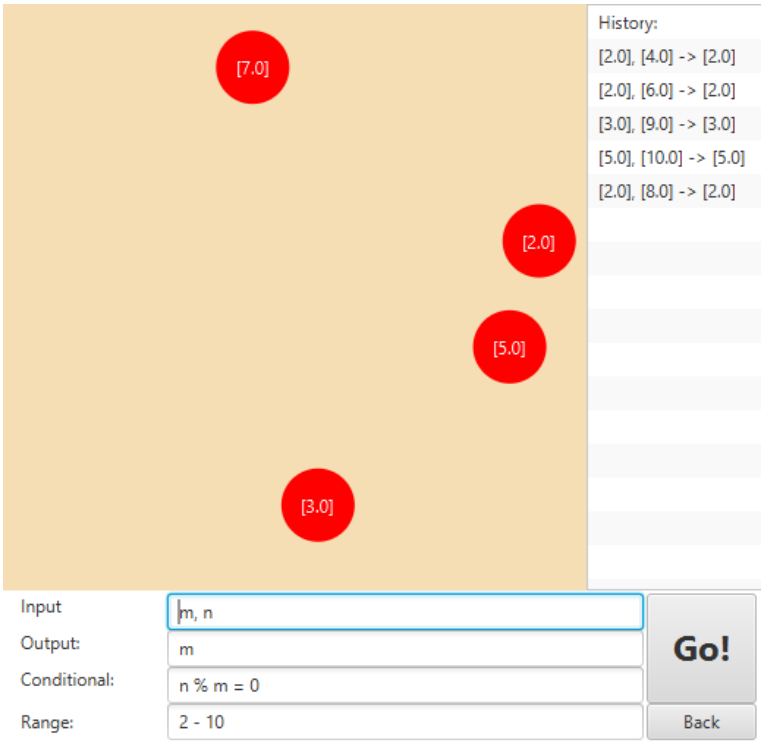

Figure 5.5: Programming using multiset transformations visualized as moving balls in our standalone application. The image shows an algorithm that finds every prime number between 2 and 10.

5.6 User Evaluation

We have performed small-scale summative testing of our framework on students within computer science. This included that subjects got to compare our tool to Python Tutor. We found that subjects preferred our framework over Python Tutor.

Working with python Tutor, subjects often experienced build failures. This was mainly due to lost dependencies or constraints imposed by the foreign system. Subjects often had a hard time understanding and fixing the errors because of their unusual character. Examples of these error are shown in Figure 5.6

(a) Timeout Error in Python Tutor. **(b)** Unknown Error in Python Tutor.

Figure 5.6: Examples of errors encountered in Python Tutor.

Subjects also found visualizations with a higher level of abstraction easier to understand with our framework. Python Tutor, as most other PV tools, provides a very low level of abstraction. This is a good way to describe the inner workings of the program, but not always the algorithm. Subjects were able to see the different values of all variables for each step of the algorithm in Python Tutor. They were however unable to determine what an algorithm was actually doing. Important changes were often lost in the mist of detail.

Using our framework, it is possible to choose which variables to visualize in a program as well as how they should be visualized. This can be exemplified by comparing the representation of a bubblesort in Figure 5.7 and Figure 5.8 and a Knapsack implementation in Figure 5.9 and 5.10. The Python Tutor versions are shown in Figures 5.7 and 5.9. Our web-based bubblesort visualization is shown in Figure 5.8 and our desktop application knapsack visualization is shown in Figure 5.10.

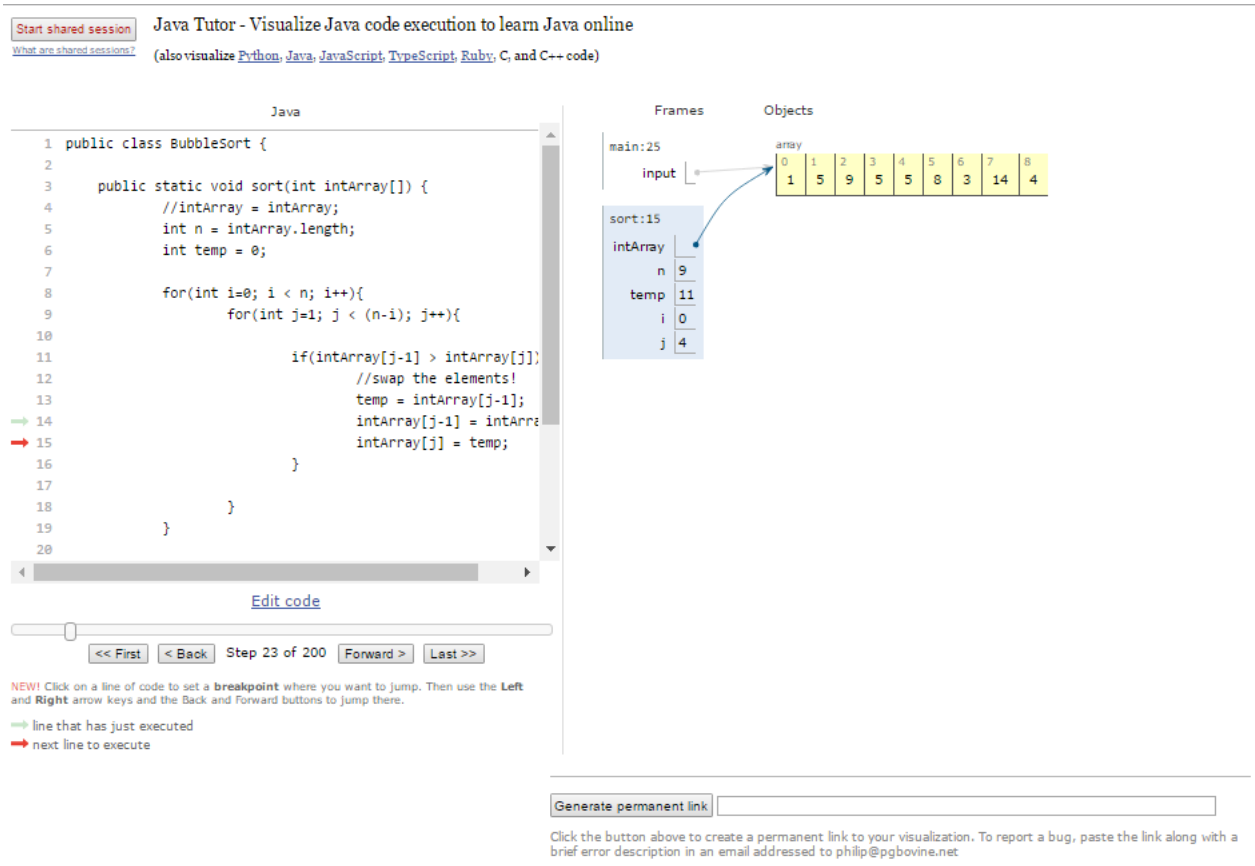

Figure 5.7: Bubble sort implementation visualized in Python Tutor.

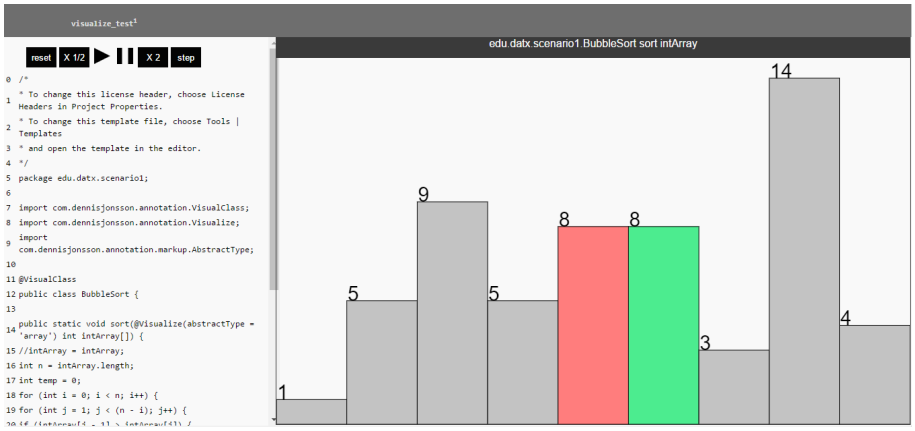

Figure 5.8: Bubble sort implementation visualized in our web based visualization module.

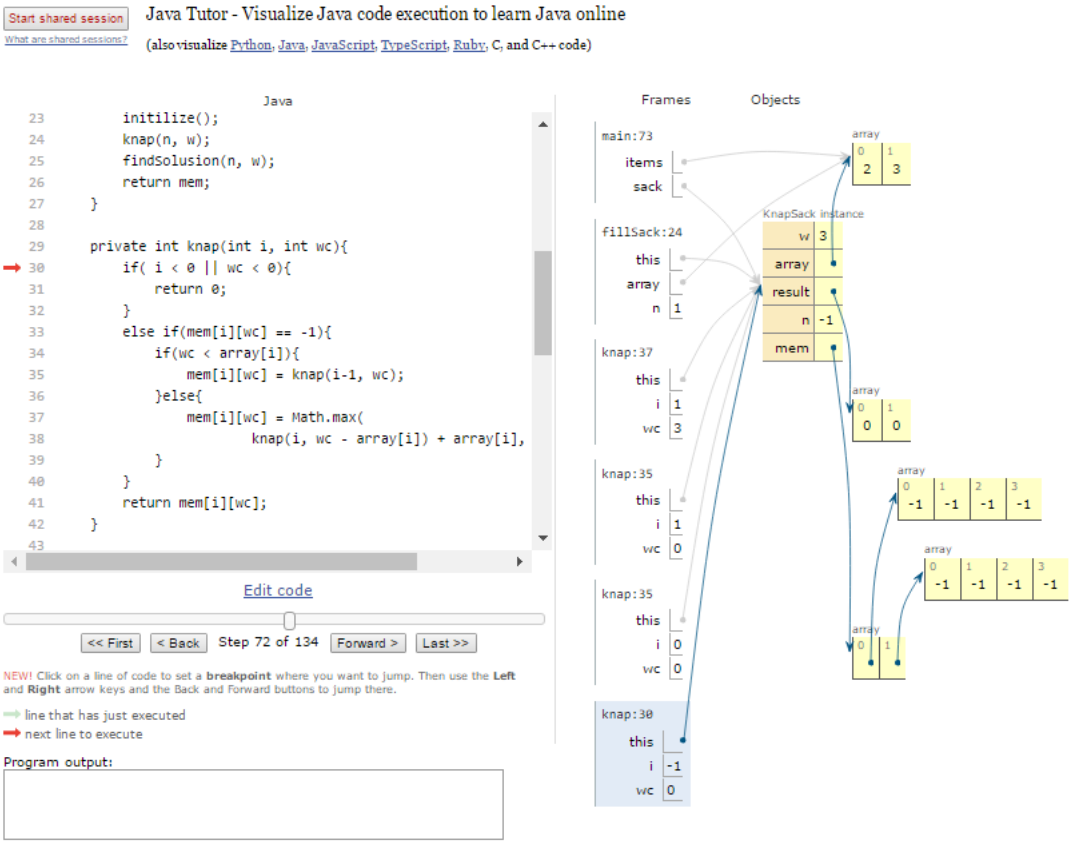

Figure 5.9: Knapsack implementation visualized in Python Tutor.

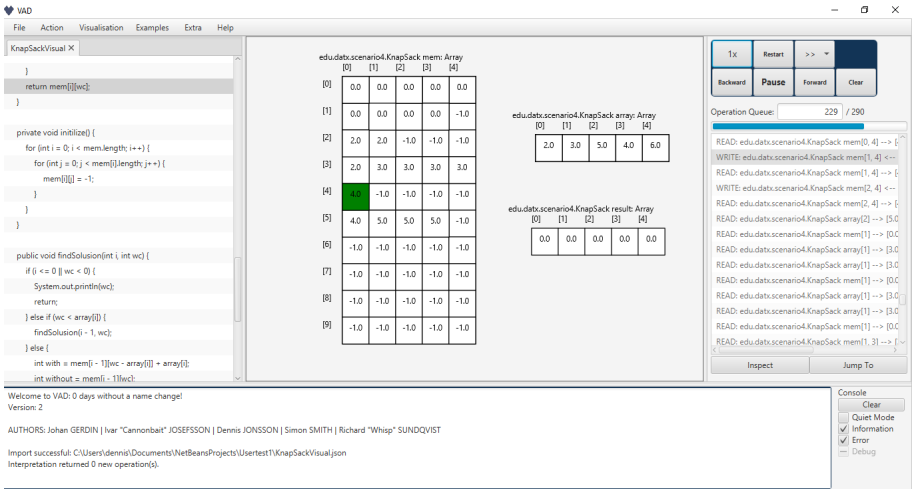

Figure 5.10: Knapsack implementation visualized in an early version of our desktop based visualization module.

6

Discussion

This chapter discusses the results of the project with regard to what has been done but also what can be done. First topic is a comparison of our results to what already exists within the field. Then, current issues and various possible extensions are presented. Lastly, a short conclusion.

6.1 Comparison With Other Visualization Tools

There already exists a lot of visualization tools. What we felt was lacking among those tools was a solid connection between the abstract concepts and the practical implementation. Our framework can, given any source code, create a customizable visualization of a program with minor to no modification of the original source files.

Pure AV tools are capable of explaining the different steps of an algorithm to a degree that our tool cannot. We receive very little information about what the visualized algorithm is supposed to do beforehand. Because of this its hard to predict how to best visualize the algorithm; that decision is left to the user.

However, this is an effect of what we wanted to create: A combination of AV and PV, user code visualized with little effort by the user. An abstract and pure AV say nothing about what the users code does, only what it should do. PV has a heavy reliance on the user code and presents little abstraction of the execution, aiming to be as exact as possible.

6.2 Issues

While we think that our system fills a needed niche in the visualization toolkit there are several issues with the system that we have found. Some of these issues are impossible to solve without making major changes to the system

architecture itself.

When we started implementing the multiset we quickly realised that using the communication contract for this feature was almost completely pointless. The log does not have the capacity to handle stochastic algorithms in a good way. We can save runs of the algorithm, but when the user wants to run the same algorithm again there is a lot of overhead for what should be a simple process. Because of these issues our implementation of multiset programming is built directly into the Java application. We've also had discussions if it would not be better to make the multiset application into a completely standalone application.

Another issue is the fact that currently there are several steps that need to be taken before a visualization appears. This is a problem since it breaks up the flow of the programmer and causes a loss in productivity [27]. However, some of the possible extensions that we suggest in the next section could solve this issue.

6.3 Possible Extensions

These are possible extensions we would attempt to make if given more time. They are generally focused on making our framework smarter and more competent at predicting and interpreting what is happening in the execution.

6.3.1 Interface Extensions

One possible extension is to let the user specify what algorithm she has implemented and provide information about the different components. For example, lets say a person called Lisa has implemented a quicksort as shown in Listing 6.1. She can then tell the interface that the variable on line 10 is the pivot and should be visualized as such by creating annotations as shown on lines 8 and 9. This gives the visualization more parameters to specialize the representation of the execution.

Listing 6.1: Implementation of Quicksort with suggested extended Java annotations.

```
1
2  @Algorithm("quick sort")
3  @VisualClass
4  public class QuickSort {
5
6      public void quickSort() {
7              ...
8              @Component("pivot")
9              @Visualize(abstractType="variable")
10             int pivot = arr[middle];
11             ...
```

```
12      }
13  }
```

6.3.2 Adding New Programming Language Interfaces

At the moment, only Java and Python is in any way supported. Because the number of supported languages is an important metric for us, we would prioritize wide-spread languages like Ruby and C++.

6.3.3 Observing Objects

For both Java and Python, only primitives and primitive type arrays can be observed. We wish to build on the current implementations to be able to watch objects as well. At a minimum, we would like to be able to observe common, standardized data structures such as array lists in Java.

This is important since most data structures are baked into bigger classes as per object oriented programming.

6.3.4 Interpreter Development

The current iteration of our Interpreter is only capable of recognizing one type of operation — the Swap. We would like to add more high level operations, such as Zig and Zag for Splay Trees.

This is a crucial part of making the visualizations more abstract. Given that a broad array of these higher-level operations could be found in the PEL, our tool would be well on the way to creating intuitive visualizations.

Once the Interpreter can detect operations such as ZigZig, we can let the user mark sections of the original source as part of a ZigZig. We can then compare the users implementation of the operation to a correctly implemented one, showing them where they went wrong or just how it can be done in a different way.

6.3.5 Networking and Algorithm Sharing

Allowing users to share and upload their visualized algorithms, would have big use cases and possibilities. You could easily create a competition or ask

students to send in interesting visualizations. It could be used to allow teachers and assistants to more easily help students that have gotten stuck on their assignment. It would also help assistants that grade and correct assignments in programming courses.

6.3.6 Concurrent And Parallel Algorithms

Operation recording is strictly sequential. We would like to build on the operation body to allow execution and visualisation of multiple operations at once. If two operations were executed at the same time they should be visualized as such. In doing so, more interesting and informative visual properties are achieved.

Once we know the logical order in which operation should be executed, we can compare program execution flows. For example, a student trying to implement bubble sort could compare her own implementation to a correctly implemented bubble sort to see where theirs differ.

6.3.7 Programming With Multisets

The implementation of multiset is rather basic and lacks several important features that a complete implementation should have. The most important of these lacking features is probably the ability for each element to be a set containing an undefined amount of variables. Currently, each element can only consist of one variable, which prevents us from implementing more advanced algorithms such as sorting. The system should also be able to accept more complex and advanced conditionals, comparing multiple different values and linking different statements together using boolean logic. With these features we believe that the multiset part of our implementation could be used with great effect in getting students of programming interested in the field of algorithms. Our implementation is considerably less capable than to the GAMMA implementation which the original multiset concept supplied [24]. This is, in part, intentional as GAMMA focused on a variety of languages while our focus has been ease of use and quickly producing visualizations.

For example, a program that finds all primes between 2 and a finite number is implemented with both multiset and Java (see listing 6.2 and 6.3, respectively). When comparing these two implementations, multiset has a clear advantage in terms of readability; especially for persons who are more familiar with mathematics than imperative languages. However, the comparison is not completely fair as the Java program can be simplified or use another algorithm.

Listing 6.2: An example algorithm for finding prime numbers implemented using multiset transformation. Note that this implementation needs a set of numbers from 2 to b, where b is an integer.

```
    m, n -> m if n % m = 0
```

Listing 6.3: A program for finding prime numbers from 2 to and including n, implemented in Java.

```
import java.util.*;

public class Sieve {
    /**
     * Calculate all primes in [2, n].
     */
    public static List<Integer> sieve (int n) {
        // Generate interval numbers.
        List<Integer> numbers = new ArrayList<Integer>();
        for (int i = 2; i <= n; i++) {
            numbers.add(i);
        }

        // Sieve for primes.
        List<Integer> primes = new ArrayList<Integer>();
        numbersLoop: for (Integer candidate : numbers) {
            for (int i = 2; i < candidate; i++) {
                if (candidate % i == 0) {
                    continue numbersLoop; // Candidate non-prime.
                }
            }
            primes.add(candidate); // Found a prime!
        }

        return primes;
    }
}
```

6.3.8 Integration with IDE

Our system currently suffers from the issue that there is a significant delay between writing code and being able to see the result of this code in a visualization. If the system was adapted into a plugin for popular IDEs it would allow a more seamless experience. When programming it is important to have a fast

and quick feedback loop, allowing the user to quickly iterate and experiment without having to wait [27].

6.4 Closing Thoughts

If we want our tool to be competitive we need to further develop it in the areas discussed in the previous section. But it serves as a proof of concept of what can be done. A more fleshed out interpreter could lead to powerful visualizations with little effort required by the user. Our assessment, based on what we have seen in the current field, is that something like our tool is what needs to be done if AV and PV are to become mainstream in computer science education.

Bibliography

[1] E. Fouh, M. Akbar, and C. A. Shaffer, "The role of visualization in computer science education," *Computers in the Schools*, vol. 29, no. 1, 2012.

[2] C. Demetrescu, I. Finocchi, G. F. Italiano, and S. Näher, "Visualization in algorithm engineering: Tools and techniques1," *Lecture Notes in Computer Science*, vol. 2547, pp. 1–3, 2002. [Online]. Available: http://wwwusers.di.uniroma1.it/~finocchi/papers/dfin02.pdf

[3] T. Shimomura, "Semantic program visualization with attachable display classes," *International Journal of Computers and Applications*, vol. 35, no. 2, pp. 70–78, 2013. [Online]. Available: http://www.cs.utep.edu/makbar/papers/TOCE.pdf

[4] V. Karavirta and C. Shaffer, "Creating Engaging Online Learning Material with the JSAV JavaScript Algorithm Visualization Library," *IEEE Transactions on Learning Technologies*, pp. 1–1, 2015. [Online]. Available: https://people.cs.vt.edu/~shaffer/Papers/p159-karavirta.pdf

[5] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, p. 15, 2013. [Online]. Available: http://demo.villekaravirta.com/PVreview.pdf

[6] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards, "Algorithm visualization: The state of the field," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 3, p. 9, 2010. [Online]. Available: http://www.cs.utep.edu/makbar/papers/TOCE.pdf

[7] M. L. Cooper, C. A. Shaffer, S. H. Edwards, and S. P. Ponce, "Open source software and the algorithm visualization community," *Science of Computer Programming*, vol. 88, pp. 82–91, 2014.

[8] AlgoViz, "Animal," accessed: 2016-05-16. [Online]. Available: http://algoviz.org/node/535

[9] D. G. Rößling, “Animal,” accessed: 2016-05-16. [Online]. Available: http://www.algoanim.net/

[10] “Json,” accessed: 2016-05-16. [Online]. Available: http://www.json.org/

[11] “Pypl popularity of programming language,” accessed: 2016-05-16. [Online]. Available: http://pypl.github.io/PYPL.html

[12] “The 2015 top ten programming languages,” accessed: 2016-05-16. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

[13] “Python is now the most popular introductory teaching language at top u.s. universities,” accessed: 2016-05-16. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-unive fulltext

[14] “Java language specification, chapter 9. intefaces: Annotation,” accessed: 2016-05-16. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7

[15] Oracle, “Java documentation, lesson: Annoatations,” accessed: 2016-05-16. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/annotations/index.html

[16] “Jsr 269: Pluggable annotation processing api,” accessed: 2016-05-16. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=269

[17] Oracle, “Java platform standard edition 7 documentation: Interface processor,” accessed: 2016-05-16. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html

[18] Oracle, Project Kenai, and Cognisync, “Java ee platform specification,” accessed: 2016-05-16. [Online]. Available: https://java.net/projects/javaee-spec/pages/Home

[19] Apache Maven, “Junit,” accessed: 2016-05-16. [Online]. Available: http://junit.org/junit4/index.html

[20] “Python ast module,” accessed: 2016-03-20. [Online]. Available: https://docs.python.org/2/library/ast.html

[21] A. Ranta, “Abstract and concrete syntax,” accessed: 2016-04-19. [Online]. Available: http://www.cse.chalmers.se/edu/year/2011/course/TIN321/lectures/proglang-02.html

[22] T. Kluyver, “Meet the nodes,” accessed: 2016-03-22. [Online]. Available: http://greentreesnakes.readthedocs.org/en/latest/nodes.html

[23] C. Castillo and Oracle, “Javafx architecture,” accessed: 2016-05-27. [Online]. Available: https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm

[24] J.-P. Banâtre and D. Le Métayer, “Programming by multiset transformation,” *Communications of the ACM*, vol. 36, no. 1, pp. 98–111, jan 1993. [Online]. Available: http://portal.acm.org/citation.cfm?doid=151233.151242

[25] J. Long, “Java parser,” accessed: 2016-05-16. [Online]. Available: http://javaparser.github.io/javaparser/

[26] J. P. Banâtre, P. Fradet, and Y. Radenac, *Unconventional Programming Paradigms: International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Higher-Order Chemical Programming Style, pp. 84–95, accessed: 2016-05-30. [Online]. Available: http://dx.doi.org/10.1007/11527800_7

[27] M. Lafeldt, “Fast feedback is everything,” accessed: 2016-06-01. [Online]. Available: https://mlafeldt.github.io/blog/fast-feedback-is-everything/

[28] A. Ronacher, “codegen,” accessed: 2016-05-16. [Online]. Available: https://github.com/andreif/codegen

A

Supporting Library

This appendix briefly describes the contents of our Java supporting library.

A.1 Log Stream Manager (input/output)

The LogStreamManager class provides methods for producing and intercepting data streams produced by the various components of this project. Data sent is assumed to be on either a JSON or native (serialized Java objects) format. The class also reads and writes log files. It is used by both the Java and Python interceptors to stream data to the desktop based visualization, as well as in the visualization module itself.

The class provides methods to stream strings, used primarily when using the manager with languages other than Java, or members of the wrapper package (see A.2).

A.2 Wrapper (Communication Contract Java Implementation)

The Wrapper package is our Java implementation of the contract. It is used by the LogStreamManager class as well as by the desktop based visualization. Most fields are implemented using enumeration rather than strings to make it easier to use, as well as wrapping additional data about what the different enumerated values.

B

Communication Contract

This appendix specifies what fields are defined by the contract and how they are used. It is intended for persons who wish to implement the contract themselves. Please note this specification is *not* yet final.

B.1 Introduction

All data is encoded using JavaScript Object Notation (JSON). JSON does not require any white spaces or line breaks, these are included to increase human readability.

B.1.1 How To Read the Specification

All field names are written with lower case letters. Names and field values are case sensitive. The log files use standard JSON notation.

B.1.2 Mandatory fields

Field names in **bold** must be included (non-null).

B.1.3 Specific Format

Some fields require values on a special format. These are referenced by the "<>" tags. For example, <Header> refers to data on the Header (see B.3) format.

B.1.4 Whitelisted Values

The '|' character indicates that one of the declared options must be chosen. For example,

```
"visual": "box" | "bar" | "tree"
```

indicates that the visual field must contain a String with the value *"box"*, *"bar"*, or *"tree"*.

If the field name is bolded, unlike our example here, the field must be present and non-null. Otherwise it may contain one of the permitted values, null, or it may be excluded entirely.

B.2 Root

This is the top-level object for the serialized log. The header contains information about version, variables etc. The body contains a list of Operation (see B.4.1) items.

Note especially that neither the header nor the body are required. That is, you can parse a log file with no data at all. This is especially useful when streaming operations as soon as they are produced. Including a complete header every time a single operation is streamed is, in fact, strongly discouraged. This is because the header will likely be several times larger than the operation itself, causing vast amounts of useless overhead to be sent.

```
{
    "header":<Header>,
    "body":<Body>
}
```

B.3 Header

The header contains basic information about the contents of the log.

```
{
    "version":<int>,
    "annotatedVariables": Map(<AnnotatedVariable.identifier>,
       <AnnotatedVariable>),
    "sources": Map<String, List(String))
```

```
}
```

B.3.1 Version

The version is an integer which denotes version number for the IO-file. It is included to ensure the file is parsed correctly, as field names and values may differ between versions.

B.3.2 Sources

Sources are source files stored in a Map. The key is the name of the source file, the value is a list of strings containing the lines of the source code. The Example section (B.5) demonstrates sources and their use by the Operation object class.

B.3.3 annotatedVariables

An Annotated Variable is a data structure which is to be drawn when visualising. An undeclared data structure may still be part of an operation, however the operation may be rendered differently than intended, or not at all.

```
{
    "identifier":<String>,
    "rawType": "array" | "independentElement"
    "abstractType": "tree"
    "visual": "box" | "bar" | "tree"
    "attributes": Map(String, Object)
}
```

B.3.4 Identifier

The literal name of the data structure. For example, the declaration

```
int myVar = 1337
```

would result an an Annotated Variable with identifier *myVar*, if it were observed. The value cannot be stored in the Annotated Variable itself. Instead, an operation is used to initialise the value. Identifiers must be unique.

B.3.5 Raw Type

The Raw Type indicates the actual type of the data structure. This determines how the structure is parsed and how it may be accessed when applying Operations to the data structure.

Permitted values: "array" | "independentElement"

B.3.6 Abstract Type

The Abstract Type determines the logical view of the data structure. For example, a binary tree implemented using an array would have *"rawType"* set to *"array"* and *"abstractType"* set to *"tree"*.

B.3.7 Visual

Determines how the data structure is drawn. If null or unknown, the default visual for the Raw Type will be used.

Permitted values: "box" | "bar" | "tree"

B.3.8 Attributes

The Attributes field contains meta data for the data structure, generally out of convenience. Currently, the only value for this field is the "*size*" field, indicating the length of arrays. For example, the data structure generated when observing *int[][] myVar = new int[4][2]* would declare *"size": [4, 2]*.

B.4 Body

The body is a list containing Operation items. These can be executed by the visualizer by applying them to the affected Annotated Variable(s) declared in the header.

B.4.1 Operation

An operation is typically a function which is applied to one or several Annotated Variables to drive the visualisation forward. A notable exception is the Message operation, which simply displays a text message.

The Operation object class contains the following fields:

```
{
    "operation":<String>,
    "source": <String>
    "beginLine": <int>,
    "endLine": <int>,
    "beginColumn": <int>,
    "endColumn": <int>,
    "operationBody": Map(String, List<Number> | <Locator>)
}
```

B.4.2 Operation Body

The Operation Body is a map containg the Locators, specifying affected elements and data structures, and the value which was the result of the operation (if applicable). The value is always identified by the *"value"* key. The value is a list of doubles, though there is only one element present in most cases.

See also: Locator (B.4.3).

B.4.3 Locator

This object is used by operations to identify an element in a specific data structure. In some cases, a Locator which specifies an identifier but not an index will apply the operation to the entire data structure.

The keys for the Locators vary depending on the operation type. However, most operations use the keys *"source"* and *"target"*.

See also: Operation Body (B.4.2).

```
{
    "identifier": <String>,
    "index": List(int)
}
```

B.4.4 Source Code References

These fields indiciate the source file and lines/columns which produced this Operation. Note that line numbers and columns begin counting at one, not zero.

The fields beginLine, endLine, beginColumn and endColumn are ignored if source is null.

The beginLine is considered the master value, meaning endLine, beginColumn and endColumn are ignored if beginLine is set to -1.

If the beginColumn and endColumn are set to -1, all rows specified by the [beginLine, endLine] interval will be highlighted. Note also that beginLine will always result in a row highlighting unless it is set to -1.

B.4.5 Atomic Operations

These are the operation types which programming language interfaces are expected to be capable of producing. Simply producing read and write operations would be enough for most applications though.

B.4.6 Read

The Read operations sets the value of the target element to the value supplied by the operation. This should be the same value as the value of the element identifier of the "source" Locator in a correctly implemented IO, but no such correcting checking is performed.

```
{
    "operation": "read",
    "source": <String>
    "beginLine": <int>,
    "endLine": <int>,
    "beginColumn": <int>,
    "endColumn": <int>,
    "operationBody": {
        "source": <Locator>,
        "target": <Locator>,
        "value": List<Number>
    }
}
```

B.4.7 Write

The Write operation sets the value of the target element to the value supplied by the operation. This should be the same value as the value of the element identifier of the "source" Locator in a correctly implemented IO, but no such correcting checking is performed.

```
{
    "operation": "write",
    "source": <String>
    "beginLine": <int>,
    "endLine": <int>,
    "beginColumn": <int>,
    "endColumn": <int>,
    "operationBody": {
        "source": <Locator>,
        "target": <Locator>,
        "value": List<Number>
    }
}
```

B.4.8 Message

The Message operation prints a text message.

```
{
    "operation": "message",
    "source": <String>
    "beginLine": <int>,
    "endLine": <int>,
    "beginColumn": <int>,
    "endColumn": <int>,
    "operationBody": {
        "value": <String>
    }
}
```

B.4.9 Interpreted Operations

Interpreted operations are operations consisting of several Read and Write operations. They can be constructed by an Interpreter (see 4.4) in a voluntary

step before visualizing program execution.

B.4.9.1 Swap

A Swap changes the values of the two variables identified by var1 and var2. The value field denote the values of var1 and var2 after the Swap has been completed. Not especially that both var1 and var2 must be present.

```
{
    "operation": "swap",
    "source": <String>
    "beginLine": <int>,
    "endLine": <int>,
    "beginColumn": <int>,
    "endColumn": <int>,
    "operationBody": {
        "var1": <Locator>,
        "var2": <Locator>,
        "value": [Number, Number]
    }
}
```

B.5 Example

The following is a simple example showing the Read, Write, Message, and Swap operations. You may copy and paste the contents into a file called *"something.json"* and load it into a visualizer to render it.

```
{
  "header": {
    "version": 0,
    "annotatedVariables": {
      "myArray": {
        "identifier": "myArray",
        "rawType": "array",
        "attributes": {
          "size": [2, 3]
        }
      },
      "myTempVar": {
        "identifier": "myTempVar",
        "rawType": "independentElement",
```

```
        "attributes": {}
      }
    },
    "sources": {
      "Composite.psuedo": [
        "SWAP: myArray[0, 2] <-> myTempVar"
      ],
      "Atomic.psuedo": [
        "WRITE: myArray <- [0, 0, 0, 1, 1, 1]",
        "READ: myArray[0, 2] -> myTempVar",
        "MESSAGE: \"JavaFX is the future!\"",
        "REMOVE: myArray[1, 1]"
      ]
    }
  },
  "body": [{
    "operation": "write",
    "operationBody": {
      "value": [
        [0, 0, 0],
        [1, 1, 1]
      ],
      "target": {
        "identifier": "myArray"
      }
    },
    "source": "Atomic.psuedo",
    "beginLine": 1,
    "endLine": 1,
    "beginColumn": -1,
    "endColumn": -1
  }, {
    "operation": "read",
    "operationBody": {
      "value": [0],
      "source": {
        "identifier": "myArray",
        "index": [0, 2]
      },
      "target": {
        "identifier": "myTempVar"
      }
    },
    "source": "Atomic.psuedo",
```

```
    "beginLine": 2,
    "endLine": 2,
    "beginColumn": -1,
    "endColumn": -1
  }, {
    "operation": "message",
    "operationBody": {
      "value": "JavaFX is the future!"
    },
    "source": "Atomic.psuedo",
    "beginLine": 3,
    "endLine": 3,
    "beginColumn": -1,
    "endColumn": -1
  }, {
    "operation": "swap",
    "operationBody": {
      "var1": {
        "identifier": "myArray",
        "index": [1, 2]
      },
      "var2": {
        "identifier": "myTempVar"
      },
      "value": [0, 1]
    },
    "source": "Composite.psuedo",
    "beginLine": 1,
    "endLine": 1,
    "beginColumn": -1,
    "endColumn": -1
  }, {
    "operation": "remove",
    "operationBody": {
      "target": {
        "identifier": "myArray",
        "index": [1, 1]
      }
    },
    "source": "Atomic.psuedo",
    "beginLine": 4,
    "endLine": 4,
    "beginColumn": -1,
    "endColumn": -1
```

```
103     }]
104 }
```

C

Codegen

The codegen module is an open-source project aimed to complement the native AST module in python. It takes an AST node and converts it back into the source code as a string. This string can in turn be written to a file, generating a file that can be run by the python interpreter [28].

It does this by visiting each node of the AST and by building a string buffer based on the type of node and its children. A Name node would for example appended to the buffer as its id. The indentation level is kept track of and is increased as nodes such as function definitions are visited.

The end result is a string object containing source code parsed from the AST. This string can now be written to a file as is or compiled and executed by native provided functions.