Teknillinen korkeakoulu Tietotekniikan osasto

Tietojenkäsittelyopin laboratorio B

Helsinki University of Technology Department of Computer Science and Engineering

Laboratory of Information Processing Science B

Espoo 2004 TKO-B 154/04

# Matrix — A Framework for Interactive Software Visualization

Ari Korhonen, Lauri Malmi, Panu Silvasti, Ville Karavirta, Jan Lönnberg, Jussi Nikander, Kimmo Stålnacke, Petri Ihantola

Distribution:
Helsinki University of Technology
Laboratory of Information Processing Science
URL http://www.cs.hut.fi/english.html

This is an electronic version of the report. It can be found in Adobe Portable Document Format (PDF) at the following address:
http://www.cs.hut.fi/Publication/Reports/B154.pdf

There is no separate paper version of this report.

ISBN 951–22–7428–0
ISSN 1239–6893

Helsinki 2004

| | | | |
|---|---|---|---|
| **Authors:** | Ari Korhonen, Lauri Malmi, Panu Silvasti, Ville Karavirta, Jan Lönnberg, Jussi Nikander, Kimmo Stålnacke, Petri Ihantola | | |
| **Title:** | Matrix — A Framework for Interactive Software Visualization | | |
| **Publisher:** | Helsinki University of Technology Laboratory of Information Processing Science | | |
| **Report Code:** | TKO-B 154/04 | **Number of pages:** | 72 |
| **Date:** | November 25, 2004 | | |

Data structures and algorithms are essential topics in elementary computer science education. They include abstract concepts and processes, such as data types and the procedural encoding of programs, which people often find difficult to understand. Software visualization can significantly help in solving the problem.

In this paper, we describe the platform independent Matrix framework that combines algorithm animation with algorithm simulation, where the user interacts directly with data structures through a graphical user interface. The simulation process created by the user can be stored and played back in terms of algorithm animation. In addition, the user can use existing library routines to create illustrations for advanced abstract data types, or he can use Matrix to animate and simulate his own algorithms. Moreover, Matrix provides an extensive set of visual concepts for algorithm animation. These concepts include visualizations for primitive types, arrays, lists, trees, and graphs. This set can be extended further by using visualizations nested to an arbitrary level.

Furthermore, we present three applications built on Matrix framework. *TRAKLA2* is a web-based learning environment dedicated to distribute *visual algorithm simulation exercises* in a Data Structures and Algorithms course. *MatrixPro* is a tool for instructors for creating customizable algorithm animations in terms of algorithm simulation. *MVT* (*Matrix Visual Tester*) is a visual testing tool to make it easier for programmers to test and debug their code.

Keywords: Software visualization, automatic assessment, visual algorithm simulation, algorithm animation

## Contents

## 1 Introduction

Let us assume we can *visualize* a data structure and have an implemented algorithm to manipulate it. Obviously, the visualization can be used to animate the algorithm by showing a sequence of consecutive visual snapshots of the data structure. During the execution of the algorithm, the state of the data structure changes. These states and changes can be animated for the user, one after another. In addition, the user may have some kind of control over the process, so he or she can interact with the system in order to stop and continue or even step through the animation. The user may also have the option to change the values of program variables (or at least the input data for the algorithm), watch these values at certain breakpoints, and so forth. This kind of step-by-step animation and visualization of variables can be compared to *visual debugging* [44, 60] of an algorithm. Moreover, these animation steps can be stored in order to give the user control over traversing the animation sequence back and forth. We call such a traversal simply *algorithm animation*. Many systems [6, 8, 10, 11, 21, 39, 40, 45, 47, 49, 51, 53] have been developed for this purpose. Figure 1 illustrates the general overview of such traditional algorithm animation systems.

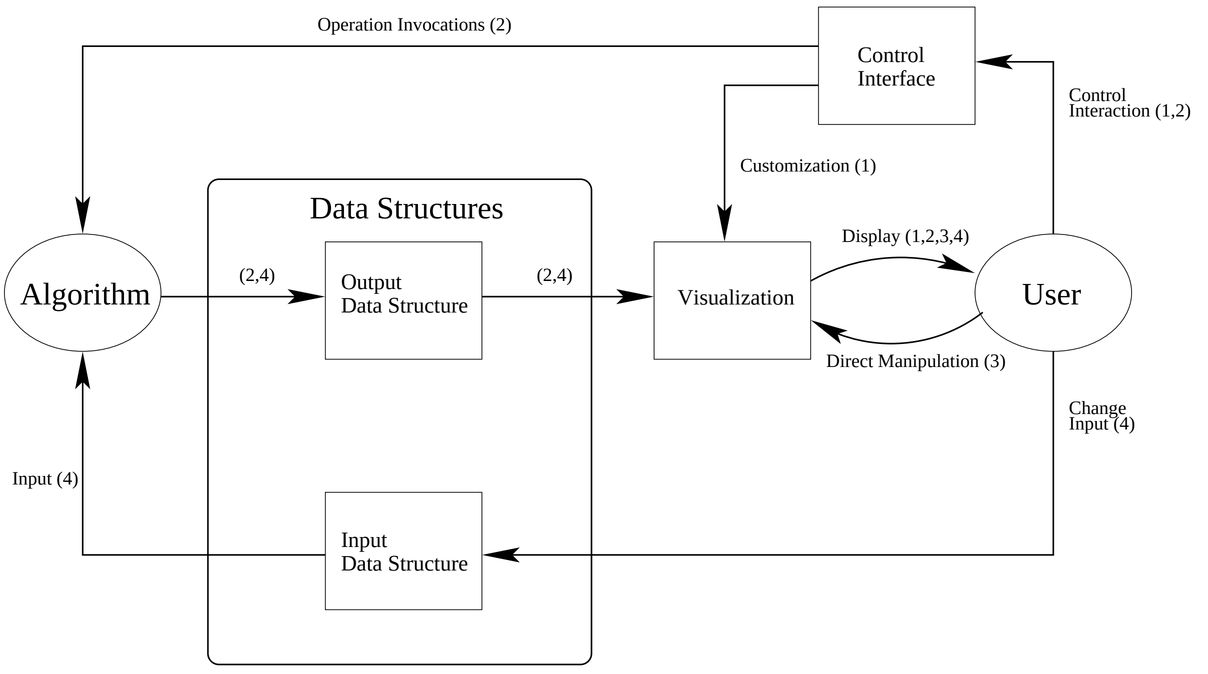


Figure 1: **General overview of traditional algorithm animation systems. Four interaction cycles can be identified. (1) The control interface allows the user to customize the layout, change animation speed, direction, etc., and (2) manipulate the data structures by calling predefined operations on them, such as insertion methods for abstract data types. (3) Direct manipulation enables interaction between the user and the graphical entities on the display. (4) The algorithm can be executed with different inputs.**

Furthermore, we can allow the user to perform his or her own operations. Thus, instead of letting the algorithm execute instructions and manipulate the data structure, we can allow the user to take control over the manipulation process. The user can directly change the data structure on the fly through the user interface and build an *algorithm animation by demonstration* similar to that of Animal [51] or Dance [54, 56]. Here the user not only has control over the visual representation (direct manipulation in Figure 1), but he or she can actually change the underlying data structure in the run-time environment. Thus, the user invokes actual operations by manipulating the graphical display and the visualization is automatically updated to match the changed structure. If the user processes the data structure as the algorithm did in the previous example, we say that the user simulates the algorithm [35, 36]. We refer to such a simulation process simply as *visual algorithm simulation* or *algorithm simulation* (see Figure 2).

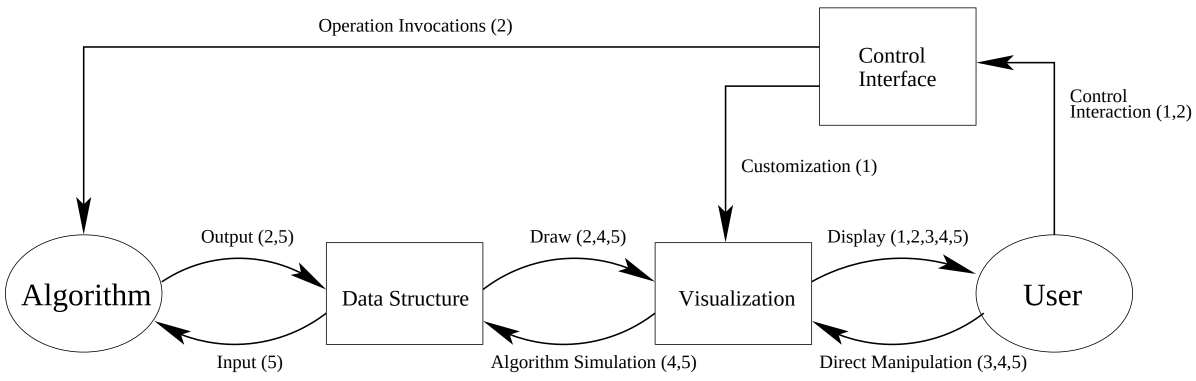


Figure 2: **General overview of algorithm animation and simulation. Five interaction cycles can be identified. The Control Interface (1) and (2) and its functionality remains the same as in Figure 1. (3) In addition, direct manipulation is allowed, but (4) the changes can also be delivered into the underlying data structures in terms of algorithm simulation. Finally, (5) the underlying data structure can be passed to the algorithm as an input by applying algorithm simulation functionality.**

Figure 2 is a simplified version of Figure 1 but it still allows more interaction between the user and the system. This is because we do not distinguish between input and output structures, and allow the user directly to interact with any data structure visualization. Thus, an output structure can be used as an input structure for the same or another algorithm. This is a fundamental difference with direct manipulation of graphical entities that essentially handles the visual representation of structures only. Moreover, the simulation process can always be continued with any data structure, for example, by performing some predefined operation on it.

Algorithm simulation is a traditional classroom technique: the teacher draws, step by step, how an algorithm changes the contents and/or the structure of a data structure. Similarly, corresponding exercises with adequate feedback can be used as an excellent tool for teaching and learning basic concepts of data structures and algorithms. Such exercises have the obvious advantage that many implementation-level details are omitted or hidden, which enables students to work on the conceptual level instead of the implementation level. Thus, it is easier to grasp the basic ideas behind algorithms. Such traditional techniques are implemented in the *Matrix* system, described in this paper. Moreover, Matrix maintains the actual data structures and therefore enables the possibility to assess automatically the simulation sequences generated by the user.

Naturally, algorithm simulation includes the idea of visual debugging, and it is straightforward to apply algorithm simulation to produce algorithm animations. However, algorithm simulation is a much more powerful concept than, for example, visual debugging. One way to distinguish between these two is to consider the source code. Visual debugging intrinsically requires that the source code exists. Algorithm simulation, instead, only requires that the user has a mental model of the algorithm to be simulated. We are not only able to change the values of variables and data structures but also, for example, the order of the operations the algorithm performs.

Another way to look at the situation is to think in terms of performed operations. We can consider an operation to be a primitive operation if we cannot divide it into lower-level operations. For example, adding unity into an integer is a primitive operation in this sense. On the other hand, the complexity of operations the user can perform during the simulation process has no limitations. Complex operations can be formed out of primitive ones in terms of algorithms, and the simulation can handle any level of operations. As an example, consider a case where the user is inserting a new key into a search tree. One way to do this is to attach a new node into the tree. In comparison, he or she can activate the insertion routine for the tree to carry out the same operation. Furthermore, a simulation has no problem inserting a whole set of keys into a dictionary, or merging two dictionaries on the fly. In visual debugging this is not possible, since on-the-fly changes in the data structures cannot be carried out without changing the actual code.

### 1.1 Motivation and Research Problem

Data structures and algorithms are important core issues in computer science education. In order to learn and understand data structures and algorithms, the student must understand many complex concepts. Algorithm animation, visual debugging and algorithm simulation are all attractive approaches to the problem of illuminating difficult concepts pertaining to data structures and algorithms. The key question is, however, how we should apply these methods in order to actually help the students to cope with the complex concepts they must understand and work with. A lot of research has been carried out to identify the great number of rules that we must take into account while designing and creating effective visualizations and algorithm animations for teaching purposes [3, 12, 17, 19, 43]. See, for example, the techniques developed for using color and sound [12] or hand-made designs [17] to enhance the algorithm animations. We argue, however, that these are only minor details (albeit important ones) in the learning process as a whole. In order to make a real difference here, we should change the point of view and look at the problem from the learner's perspective. How can we make sure the learner actually understands the concepts in question? It is not what the learner sees, but what he or she does. In addition, we argue that no matter how fancy visualizations the teacher has, the tools cannot compete in effectiveness with environments in which the learner must perform some actions in order to become convinced of his or her own understanding.

From the pedagogical point of view, for example, a plain tool for viewing the execution of an algorithm is not good enough [26]. Even visual debugging cannot cope with the problem because it is always bound to the actual source code. The system still does all the work and the learner only observes its behavior. We should at least ensure that some progress in learning has taken place. This requires an environment where we can give and obtain feedback on the student's performance. Many ideas and systems have been introduced to enhance the interaction, assignments, mark-up facilities, and so forth, including [1, 13, 20, 22, 42, 50, 55]. On the other hand, the vast masses of students in basic computer science classes have led us into a situation in which giving individual guidance to a single student is impossible even with semi-automated systems. Thus, a fully automatic instructor would be useful, such as presented in [4, 7, 9, 16, 25, 27, 28, 50, 52]. Nevertheless, the topics of data structures and algorithms are more abstract than those introduced on basic programming courses. Therefore, systems that grade programming exercises are not suitable here. We are more interested in the logic and behavior of an algorithm than its implementation details. The problem is to find a suitable application framework for a system that is capable of interacting with the user through general purpose data structure abstractions on the logical level (instead of the implementation level) and giving feedback on his or her performance. PILOT [9] comes quite close to this idea. In PILOT, the learner solves problems related to graph algorithms and receives a graphical illustration of the correctness of the solution, along with a score and an explanation of the errors made. However, the current tool covers only graph algorithms and focuses on the minimum spanning tree problem. Hence, our understanding is that PILOT does not provide a general purpose application framework that can easily be extended to other concepts and problem types.

The goal of the Matrix project is to develop such a general purpose platform for illustrating all the common data structure abstractions applied regularly to illustrate the logic and behavior of algorithms. Moreover, the platform should be able to allow user interaction in terms of algorithm simulation. As an application, we support training in which automatically generated visual feedback is possible for algorithm simulation exercises. We call such a process *automatic assessment of algorithm simulation exercises* [33].

## 1.2 Related Work

Next, we want to position Matrix more generally in the field of *software visualization*. This large area of research has been polarized toward two opposite domains, *algorithm visualization* which aims at visualizing the working of algorithms on a conceptual level, and *program visualization* where the execution of the actual algorithm code can be followed using various graphical tools. Especially, program animation allows considerable user interaction with the displays of the data structures by integrating visualization within a source-level debugger. Moreover, integration of program animation and concept animation allows visual debugging in such a way that the system is capable of displaying the dynamics of a program and visualizing the context in which the changes to data structures occur. The data structures can be examined both in the implementational and the conceptual level at the same time.

Although Matrix (and one of its applications) includes a number of program visualization features, the roots of Matrix belong to the first domain, the field of animating concepts, abstract data types (ADTs), and algorithms, instead of programs. Our belief is that understanding algorithms is easier if we first omit the details of the actual implementation and guide the student to tackle the code only after building a viable mental model of how the algorithm or the ADT works. Thus, Matrix supports a process where students first learn the abstract and then proceed to the concrete (implementation).

In Matrix, the user can start building animations by using only the simulation tools without implementing any code. He or she can take advantage of the implemented data structures available in Matrix, and operate on them within the simulation. In addition, he can implement new code that conforms to special concept interfaces, and thus facilitate visualization. The key motivation here is to provide several methods to generate visualizations for arbitrary programs with minimal effort on the part of the visualizer.

There are other tools without the simulation functionality that are dedicated to the easy creation of algorithm animations, for example, Amethyst [46], Jeliot [21], and UWPI [23]. These tools analyze the given Java or Pascal code to produce the visualization. The JDSL Visualizer [4] tool uses a similar interface approach to generate visualizations as Matrix. Moreover, Matrix, Jeliot, JDSL Visualizer, and UWPI can provide visualizations for high-level *conceptual displays* as opposed to the *concrete data structures* that appear in the implemented code. Finally, it is worth noting that both Matrix and JDSL Visualizer have the ultimate goal of allowing the system to give feedback about students' performance while working with exercises.

In order to help people gain a more thorough understanding of the working of program code, we have developed a Matrix-based application called *Matrix Visual Tester* or *MVT* that includes program visualization tools which allow the user to view the execution of the code concurrently with the data structure animation, and control the execution. These features can be seen as tools for *visual debugging* that is highly useful in implementing and testing new algorithms. This overlaps tools like DDD [60] and Amethyst [46] that can be considered debuggers with added features for visualizing data structures manipulated by the program.

Finally, we list some differences between Matrix and other systems. First, we do not know of any other system that is capable of algorithm simulation in the sense of direct graphical manipulation of visualized data structures. Astrachan et al. discuss simulation exercises while introducing the Lambada system [1, 2]. However, their context is quite different, because the students simulate models of practical applications, partly coded by themselves, and the system is used only to illustrate the use of primitive data structures without much interaction with the user.

Second, our approach is also quite different from the works of Stasko, Kraemer and Mukherjea [44, 53, 55, 57] where the general method is to generate animations by annotating the actual algorithm code. In Jeliot [21] the annotation method has been automated, but the starting point is essentially the given user code. This enables easy creation of visualization for arbitrary code, but still lacks the simulation functionality provided by Matrix.

Third, Matrix is designed for students working on a higher level of abstraction than, for example, JDSL Visualizer. JDSL Visualizer is intended to help students debug their own implementations of data structures, while Matrix is used to illustrate and grasp the logic and concepts of data structures and algorithms. Of course, both can be used when teaching data structures and algorithms.

Fourth, Matrix is implemented in Java, which gives us more flexibility in terms of platform independency than older systems such as Amethyst and UWPI.

## 1.3 History

At Helsinki University of Technology (HUT), we have a long tradition of using concept animation and algorithm simulation as the key techniques for teaching data structures and algorithms (DSA). The yearly enrollment on our DSA course is over 500-700 students. Because only one third of them study computer science as their major, we have chosen to use algorithm simulation in this course and have set up a smaller continuation course for the majors in which they actually implement algorithms.

In the basic course, each student has to submit some 20-25 simulation exercises which cover most basic algorithms in sorting, searching and graphs. Due to the vast mass of submitted exercises, we implemented and have been using the TRAKLA system [27] since 1991 until 2003, when it was replaced by the totally new system TRAKLA2, which is based on Matrix. TRAKLA generates algorithm simulation exercises with different input data (or other minor modifications) for each student, and sends them to the students by email. They solve the assignments by using manual algorithm simulation, present the answers in a predefined format, and submit the solutions to TRAKLA via email for automatic assessment. The automatic assessment is based on comparing of the submitted answer and the model answer generated by the actual implemented algorithm. In 1997, a graphical WWW-based user interface (WWW-TRAKLA[1]) for performing the simulation in terms of direct manipulation was added [33]. Then, instead of by manual simulation and using somewhat artificial formatting conventions, the students can concentrate on working in the conceptual level. During this over 10-year period we have obtained very good learning results: approximately half of the students get over 90 percent of maximum points and two thirds of them get at least 80 percent. The feedback from the students using these tools has been good and we have saved enormous amounts of work by not needing to check manually tens of thousands of assignments per course.

In spite of their success, TRAKLA and WWW-TRAKLA have reached their limits of development, because the original design has many limitations. First, WWW-TRAKLA is just a one-way front-end to TRAKLA for generating visual representations of data structures. There is no connection to the actual implemented data structures and algorithms inside TRAKLA. This has, for example, the disadvantage that although the student can generate an algorithm simulation sequence in WWW-TRAKLA, he cannot get the corresponding model sequence nor the animation back to the front end. Second, since WWW-TRAKLA does not include abstract data type functionality, it is more like a guided drawing tool instead of a DSA learning

[1]In some early documents or papers, we have also used the name TRAKLA-EDIT.

environment. Third, TRAKLA internally handles all exercises as separate modules without any true conceptual structure. Thus, adding new features and exercises has been very time-consuming process. Due to these limitations we set up a project to write a wholly new system called Matrix [31, 34][2] in 1999 which we will describe in the following sections.

### 1.4 Notes about Terminology

*Algorithm simulation* is the higher concept for manipulating data structures on the conceptual level. In case of performing the simulation on a paper, a blackboard, or some other non-computer equipment, we use the term *manual algorithm simulation*. As an opposite to this, we use the term *visual algorithm simulation* for operations where the simulation is carried out with a computer program which truly understands the context of the operations, *i.e.*, allows modification of underlying data structures. Between these two extremes there are simulation methods such as in WWW-TRAKLA in which the operations are carried out in terms of direct manipulation.

Finally, in the following, we use for simplicity the term algorithm simulation in the sense of visual algorithm simulation, unless otherwise stated.

### 1.5 Organization of This Paper

The rest of this paper is organized as follows. In Section 2, we describe the design principles used in Matrix and, in Section 3, we explain the features of Matrix in more detail. In Section 4, we describe three applications, *TRAKLA2*, *MatrixPro*, and *MVT* that all use the Matrix framework. In Section 5, we evaluate Matrix in the context set up by Price et al. in their taxonomy for software visualization [48]. Finally, in Section 6, we summarize the results and point out some future directions for development.

## 2 Theory and Design

In this section, we present the theory behind Matrix and explain how it affects the design and structure of the system. We start by briefly describing how Matrix is linked to different fields

[2]The original working name for Matrix was Object-TRAKLA.

of software visualization. After that we explain how Matrix models data structures, visualizes them, and allows operations on them in terms of algorithm simulation.

In Subsection 2.1, we describe the basics of the system and present some concrete examples of how the system could be used to provide methods for algorithm animation and simulation in general. In Subsections 2.2 and 2.3, we describe how to create and interact with a visualization for a particular concept and what kind of objects are involved during the animation and simulation process. Moreover, in Subsection 2.4, we give more concrete examples how the different conceptualizations provide tools for raising the level of abstraction while working with an AVL tree example.

## 2.1 Software Visualization

Matrix makes use of many different software visualization techniques. In this subsection, we describe these techniques and how they are applied by Matrix.

### 2.1.1 Algorithm Simulation and Animation

The first key design idea in Matrix is that we combine algorithm simulation and algorithm animation. Let us consider this more closely. In *algorithm simulation*, the user manipulates graphical objects according to the rules allowed for the structure in question (like an array or a binary tree) and creates a sequence of visualization steps. These steps include basic assignment operations that can be used to attach a value into a key field, to change references, or to invoke user defined operations such as an insertion or a deletion of a key. The primitive manipulation operations are called *primitive simulation*, and the method invocations *concept simulation*.

In *algorithm animation*, the user watches a display in which the changes in the representation are based on the execution of a predefined algorithm. Thus, it is the algorithm that alters the data structure, and the visualizations representing the structure are generated automatically. Matrix includes both of these methods and allows them to be combined seamlessly.

### 2.1.2 Algorithm Visualization

The second key design issue in Matrix is that it aims to work on the conceptual level instead of the code level. Thus, we promote *Algorithm Visualization*. The system automatically creates conceptual displays for data structures and allows the user to view and manipulate the structures on different levels of abstraction. Our objective is to provide the students a broader view of the different topics on data structures and algorithms than offered by most text books in the field. They provide — as it is feasible — a list of algorithms and data structures classified by typical application areas such as basic structures, sorting, searching, graph manipulation, and some other miscellaneous topics. In addition to this, we would like to help the students to build mental models in which hierarchies among data structures exist, for example, among different classes of search trees.

We stress the difference between *abstract data types* (*ADT*s) and their implementation, *i.e.*, we distinguish the semantic meaning of a data structure and its actual physical implementation. Abstract data types are a set of (abstract) items with a predefined set of operations. An example of an ADT is a *dictionary* that contains a set of items and has three operations: *insert*, *remove* and *search*.

*Fundamental data types* (*FDTs*) are data types that connect components (which may be other FDTs or primitive values in the form of *keys*) together in some way without constraints on type or value. The most common FDTs are array, linked list, tree, and graph. These FDTs are illustrated in Figure 3.

*Conceptual data types* (*CDTs*), on the other hand, are implementations for ADTs. CDTs are usually constructed from FDTs or other CDTs. A CDT may include constraints on the type and contents of the components it is constructed from, as well as the type for the components it can hold. For example, a *binary search tree* is a CDT that is constructed from a *binary tree* (an FDT). A clear distinction among the concepts, however, is required. We have observed too often that many students confuse binary trees with binary search trees. Making the difference between FDTs and CDTs more explicit should help to clarify this issue. Here, the binary tree FDT is a reusable concept employed by the binary search tree CDT that finally implements the operations defined in the dictionary ADT.

The overall design in Matrix for producing automatic visualizations for data structures is based on the idea that the data structure to be visualized should conform to predefined *visual concept interface(s)*. Matrix produces visualizations by calling the interface methods for fundamental data types. Usually, the implementation of a visual concept interface requires coding only a few methods. Thus, the actual data structure and its visual representation(s) are completely separated from each other, as depicted in Figure 2.

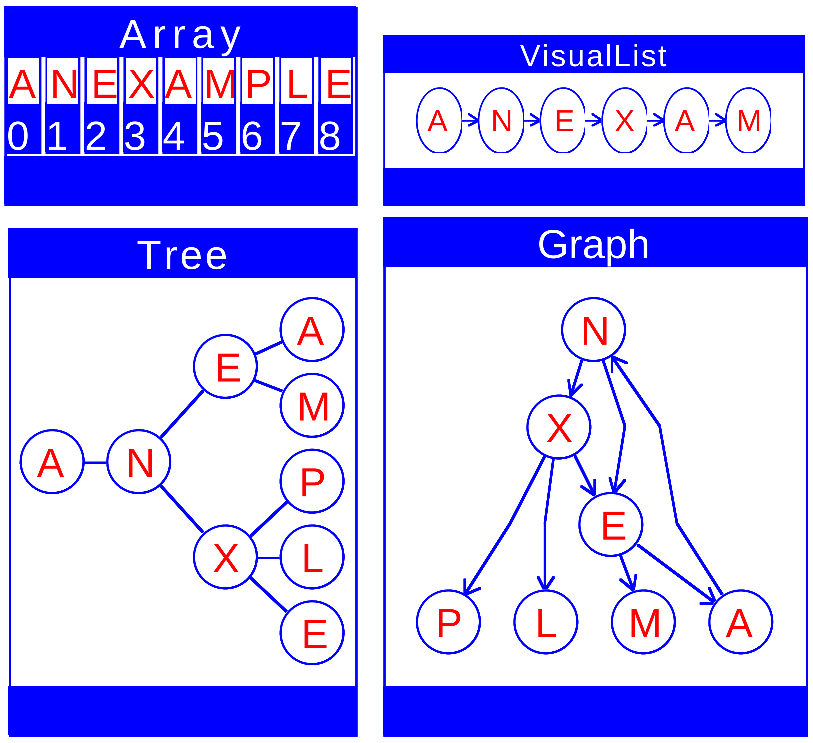


Figure 3: **Fundamental data types, such as arrays, lists, trees, and graphs are important reusable abstractions regularly used in computer science. These are printed from the Matrix system.**

From the user point of view, Matrix operates on a number of *visual concepts*. These include views for arrays, linked lists, binary trees, common trees, and graphs, as depicted in Figure 3. A CDT such as the heap, can be visualized by employing different visual concepts including the binary tree and the array representation. Moreover, each visual concept may have several *layouts* which control its visual appearance in detail.

The visual concepts can be nested to arbitrary complexity in order to generate more complex structures such as adjacency lists or B-trees. The user can manipulate the structures on the FDT level, *i.e.*, by assigning new values for keys, changing references and nodes, and so forth. He can also work on the CDT level by invoking, for example, the insert and delete routines for the structures.

### 2.1.3 Program Visualization

Our primary goal has been the development of tools in the algorithm visualization domain and most of our effort has been put into this. However, the other domain, program visualization, has been our secondary goal for a long time. We have investigated program visualization, debugging and testing in more detail and devised a combination of methods that we call *visual*

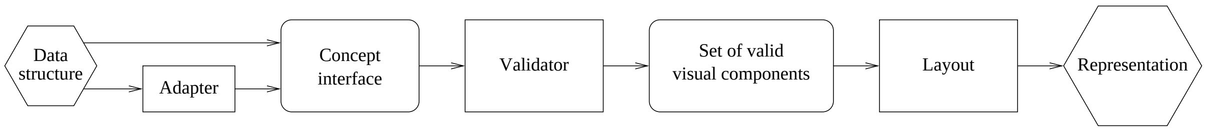


Figure 4: **Visual representation assembly line.**

*testing* [38, 41]. Visual testing combines program visualization at several levels of abstraction with interaction and animation methods similar to those used in algorithm simulation and animation. By utilizing Matrix, we have also developed a prototype visual testing tool called *MVT* (*Matrix Visual Tester*). MVT [41] can be described as a visual debugger with a configurable data view, execution logging, data modification through manipulation of the data view, and graphical execution control. MVT is described in more detail in Section 4.3.

## 2.2 Visualization Process

Matrix provides several methods for creating visualizations. The degree of hand coding needed to produce a visualization depends on the method used. First, it is possible to reuse the *Matrix library components*. These include several implementations of all supported concepts, called *Fundamental Data Types, FDTs*. Some authors refer to this method as the probing technique [39, 40]. Second, each FDT has the corresponding visual concept interface that can be implemented directly. Thus, it is possible to implement a data structure which directly conforms to the visual concept interface(s). Thereafter the visualization of the structure and the basic algorithm animation properties are automatically supported by Matrix. It should be noted, however, that this method requires extra work to gain the possibility to reverse the temporal direction of the animation [8, 48]. Finally, it is possible to produce the visualization manually in terms of visual algorithm simulation. Here the user manipulates the data structure through the graphical user interface, taking similar actions to what an actual algorithm would do. Note, however, that this third method involves no actual coding of programs. It is discussed further in Section 2.3.

The overall assembly line for producing a representation for a data structure is illustrated in Figure 4. The creation of visualization is divided into three phases that are *Adaptation*, *Validation* and *Laying out* the representation. In addition, the GUI provides a way to influence the functionality of these phases.

#### 2.2.1 Adapters

The system tries to treat each of the visualized data structures similarly through the visual concept interface. Unfortunately, this may lead to a situation in which it is almost impossible for the programmer to implement the required interface efficiently. Therefore, the system provides adapters for some of the concepts in order to make it easier to adapt a data structure to the concept. This adaption is transparent for the programmer and allows him to choose from a range of visual concept interfaces.

Let us consider the following example. Binary heap is implemented as an array and the intention is to visualize the structure both with an array representation and with a binary tree representation. The latter one, however, requires the structure to be dynamic, *i.e.*, there should exist a node to be visualized for each occupied position in the array. Thus, the adapter makes the conversion from the static array to the dynamic binary tree by creating the necessary intermediate nodes which do not exist in the original structure but are required for the visualization.

#### 2.2.2 Validation

For each concept, a single visual concept interface exists that is used to prepare the visualization. For example, every data structure $D$ that should have the option to be visualized as a tree should conform to the corresponding visual concept interface `Tree`. The next problem is to ensure that the procedure, which displays trees as its output, actually takes one as its input. We should therefore validate $D$ and remove all cycles. Thus, even if the input would be a graph, we can still accept the original structure and visualize it as a tree. In practice, this means that the visualization only covers the connected nodes of the graph starting from a given root node in depth-first search order. All back, forward and cross edges are visualized as connections to special terminal nodes whose layout differs from that of the ordinary nodes. Since such a representation may not be intuitive, the user can modify the behavior of the validation of trees in order to prevent the visualization of such edges.

#### 2.2.3 Layout

The final phase of creating the representation produces the layout for a concept. The layout procedure may assume that the input it receives really follows the conceptual principles discussed above. Thus, the procedure receives an input set of nodes and a set of connections between them that obey the rules of the concept. The output is a hierarchy of these nodes that should be as close as possible to what the user expects to see while working with the corresponding metaphor.

Unfortunately, there is no single layout that is adequate for, let us say, all trees. At the moment, the system provides two layouts for trees that differ slightly from each other. The GUI provides a way to change the layout at run time. However, we feel that it should be possible to add new layouts to the system in the future. Even though writing a new layout is not the sort of basic operation that ought to be done by an end user, we have taken into account the possibility of extending the set of possible layouts further.

### 2.3 User Interface Entities

In this section, we present the user interface concepts that form the basics of algorithm simulation in Matrix. The dynamic set (the hierarchy) $H = (N, R)$ consists of elements $N$ (nodes) and binary relations $R \cup N \times N$ (connections between nodes). Each element can hold a key $K$. In addition, a set may consist of other sets, thus allowing nested sets. Correspondingly a hierarchy may consist of nested structures by allowing the key of a node to be another hierarchy. By allowing the nested structures to be arbitrarily complex the system also allows the possible hierarchies to be arbitrarily rich in their design. As a consequence, we can allow visualization and interaction with arbitrarily complex symbolic models.

When we consider hierarchies in the context of the user interface, there are two equally important aspects that we should recognize. First, the hierarchy has a conceptual nature that allows it to arrange the nodes is such a way that the representation of the model appeals to the human intuition of the corresponding metaphor. The dynamic nature of the model leads to the idea of algorithm animation. This is due to the fact that in algorithm animation the challenge is to visualize the changes in the data structures for the user. Second, the interaction between the visualization and the user provides a meaningful way for exploration and understanding symbolic models, based on data structures, in terms of visual algorithm simulation. There are four different types of interactive components available in the GUI. The first three components behave similarly to each other. The fourth component, the binary relation between two other components, has somewhat different behavior.

The principles of how the interactive components interact with each other are discussed below. The interpretation of the semantic meaning of various user operations is, however, often a difficult or even ambiguous question when considered in general. While building the current set of data structures within Matrix we have made the necessary decisions. For future development, we provide the following guidelines.

### 2.3.1 Hierarchy, Node, and Key

A *hierarchy* refers to a set of *nodes*. Each node has a *key*. A key can be a primitive or some more complex structure. If the key is a primitive, it has no internal structure, and therefore it can not be examined further. If the key is a more complex structure, it is treated recursively as a hierarchy.

All these three entities can be moved around in the display. The simulation consists of *drag and drop* operations which can be carried out by picking up the *source* entity and moving it onto the top of the *target* entity. Each single operation performs the proper action for the corresponding entity, as follows.

Let us denote hierarchy, node and key as $H$, $N$, and $K$, respectively. In addition, the expression $X \rightarrow Y$ denotes the operation "X is dragged and dropped into Y". For simplicity, we say that we "drag X into Y".

Moreover, the simulation operations performed for a structure do not change the structure of the visual representation but carry out actual changes in the underlying data structure. The change is delivered first to the underlying data structure that is then visualized again for the user.

We are now ready to give the definitions for the following nine operations:

1. $H_1 \rightarrow H_2$ (drag a complex structure into another one)

   This first case is very interesting because it has several possible sub-cases:

   (a) If $H_1$ and $H_2$ are of the same type, we could give this operation the meaning of *Join(*$H_1$, $H_2$*)*. However, the case $H_1 = H_2$ (the structure is dragged into itself) is an unusual situation and this should raise an exception.

   (b) If $H_1$ and $H_2$ are of different types, the operation depends heavily on the actual data structure $H_2$. For example, if an array of keys is dragged into a binary search tree, it would be feasible to have each key inserted one at a time into the search tree. On the other hand, if the target structure is just an ordinary binary tree, we possibly cannot have any unambiguous operation to perform.

The bottom line is that it is up to the target to determine what this kind of operation should do. If $H_2$ conforms to the interface `CDT`, we know it has the method `insert` that could be invoked. The meaning of "Insert" here is determined by the person who implemented $H_2$; thus it is up to the programmer to choose whether the structure should do a "Join" operation or some other action. In the case of *probes* (predefined and implemented structures in the Matrix framework) this choice is already made for the user and it follows the way of thinking described above.

2. $H \rightarrow N$ (drag a complex structure into a node)

   This operation should connect $H$ to the node $N$ if such an operation is feasible. Otherwise the old key $K \in N$ is replaced by the hierarchy $H$. For example, we can attach a tree below a node in a common tree, but we cannot perform such an operation for a binary heap.

3. $H \rightarrow K$ (drag complex structure into key)

   This operation replaces the key $K$ with the hierarchy $H$ for the node $N, K \in N$.

4. $N_1 \rightarrow H$ (drag a node into a hierarchy)

5. $N_1 \rightarrow N_2$ (similar)

6. $N_1 \rightarrow K$ (similar)

   In these three cases the node $N_1$ usually represents some kind of substructure $H'(N_1)$ that is dragged instead of a single node and thus this operation should be treated as the operation $H' \rightarrow \{H, N_2, K\}$ as above. A new visualization is created for the substructure $H'$ that is initiated by $N_1$. For example, $N_1$ could be an internal node of some binary tree. The operation creates a new visualization (a new binary tree representation) rooted at $N_1$ and this hierarchy is therefore dragged into the target as we did previously. Another possible interpretation for this operation is to treat the node $N_1$ as a single representative of the whole hierarchy it belongs to; thus the operation is again the very same as $H'' \rightarrow \{H, N_2, K\}$. In this case, no new visualization is created. Instead, the whole structure $H'', N_1 \in H''$ is moved. This would be the same operation as if we chose the internal node to be the root of the whole tree in the previous example. This latter form of behavior is usually better when it is hard to define what kind of substructure should be moved along with a single node. For example, it is not always clear which part of an array the operation should affect, if we start to drag an array starting from the place positioned at some index $i$.

7. $K \rightarrow H$ (drag a key into a hierarchy)

   If $H$ conforms to the interface `CDT`, the corresponding `insert` method is invoked with the parameter $K$. Otherwise, we possibly cannot determine the meaning of the operation and this operation should raise an exception.

8. $K \rightarrow N$ (drag a key into a node)

   This operation resembles the two possibilities of case 2. In order to be able to connect the new key $K$ to $N$, however, we have to create a new node $N'$ for the key $K$ before the connection between the entity $K$ of hierarchy $H, K \in N' \in H$ and $N$ is possible. There is also the other option in which the old key of $N$ is replaced with $K$. The choice is made by the actual FDT implementation of $N$ (FDT). This is because the operation is completed in the corresponding FDT. This option may turn out to be very confusing if the decision on how the structure should behave in different kinds of circumstances is not clear.

   Unfortunately, there seems to be no clear default behavior, because even such basic concepts like trees may turn out to behave differently but still logically. It seems to be natural, for example, that free trees can grow and a new node $N', K \in N'$ is connected to some existing node $N$ by operation $K \rightarrow N$. On the other hand, for binary trees (k'ary trees) this is ambiguous because the operation does not explicitly determine the position of the new, possibly absent, (sub)tree, *i.e.*, there is a need for visualization of empty (sub)trees. In this latter case, it is convenient that the operation $K \rightarrow N$ replaces the old key of $N$ with the new key $K$ and no new nodes are created.

9. $K_1 \rightarrow K_2$ (drag a key into a key)

   This operation replaces the key of node $N, K_2 \in N$ with the key $K_1$.

#### 2.3.2 Connection Entity

The discussion above excluded the fourth entity necessary to fully demonstrate hierarchies and especially the case in which visible connections between the nodes in a hierarchy exist. Naturally, some operations are necessary for these binary relation entities. Let us denote the relations by $R$.

Basically we could just expand the discussion above and allow all operations

$$\{H, N, K, R\} \rightarrow \{H, N, K, R\}. \tag{1}$$

The functionality of the framework does not, however, support the operations $\{H, N, K, R\} \rightarrow R$ although there could be a meaningful interpretation at least for some of them. However, the framework does support the operations

$$R \rightarrow \{H, N, K\}. \tag{2}$$

In all of these operations the binary relation is supposed to be a directed edge and the operation moves the target of the edge to point to the new location, either to the new node $N$, $\{N|K \in N\}$ or to the head node of $H$ if it is defined.

## 2.4 Working with Different Levels of Abstraction

One design issue in Matrix was the separation of data structures and their conceptual nature. A new data structure can be created using fundamental data types (FDTs), which are used as building blocks for constructing conceptual data types (CDTs) which are used for implementing abstract data types (ADTs).

### 2.4.1 Working on the FDT Level

An FDT consists of nodes which are connected to each other in a specific way. A large number of applications can be covered by the five predefined FDTs: array, linked list, binary tree, tree, and graph. Thus, an FDT is conceptually a reusable low-level hierarchy that can be used to implement more complicated and sophisticated data structures. Furthermore, an FDT can be used as a basis for an even more sophisticated CDT, which in turn may implement an ADT.

The conceptual difference between the data types can be demonstrated as follows. We can choose to work on the FDT level and directly manipulate the contents of the nodes and the connections between them on display. This is called *primitive simulation*. As an example, the FDT can be a *binary tree*, and we can build an animation by illustrating how an array of keys is inserted into a *binary search tree*, one key at a time. An insertion is simulated by dragging the inserted key into an empty binary tree position below some leaf of the tree (case 2.3.1.8, $K \rightarrow N$). The actual data structure is updated and a new node is connected to it with the

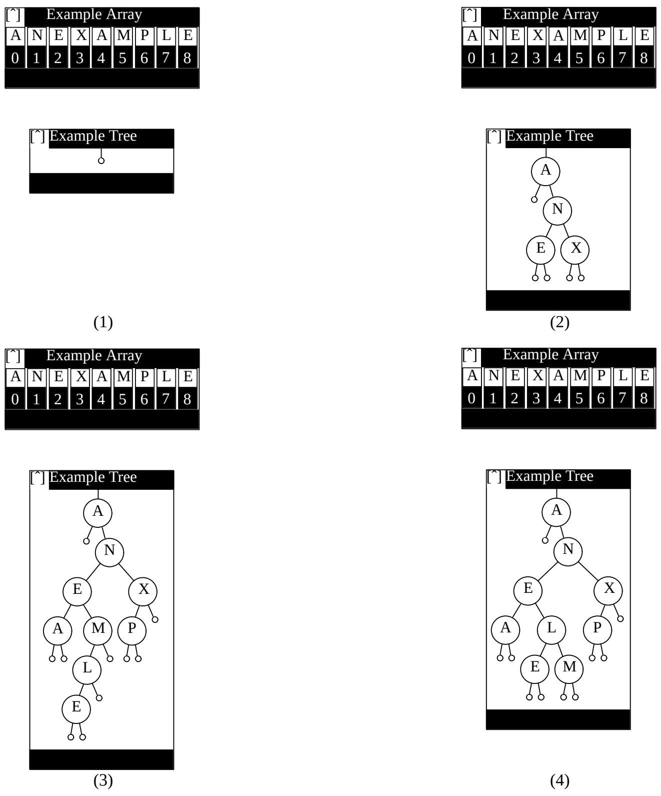


Figure 5: **Binary Tree Manipulation Example.**

inserted key as its label. Thereafter, the binary tree representation grows by one new node and two empty positions.

Similarly, if the user wants to simulate the balancing of a tree, he would change references at appropriate nodes to illustrate rotations. This is demonstrated in Figure 5 in which frame (1) shows the array of keys and the initially empty binary (search) tree before insertion of the key $A$. In addition, in frames (2) and (3), the tree is shown after the insertion of the key $X$ and when all the keys have been inserted. In frame (4), a single rotation to the right is demonstrated in the node $M$ after the user has updated the appropriate three references in the previous state.

### 2.4.2 Working on the CDT level

After learning how, for example, a binary search tree or an AVL tree functions, the user can move on to work on the CDT level and generate animations which show the behavior of the structure in different cases, such as in the worst case. In this case, manipulating the tree using FDT operations would be too clumsy. Instead, he or she can continue on the FDT level and use the existing library routines for inserting keys into a binary search tree or into an AVL tree. This is called *conceptual simulation*.

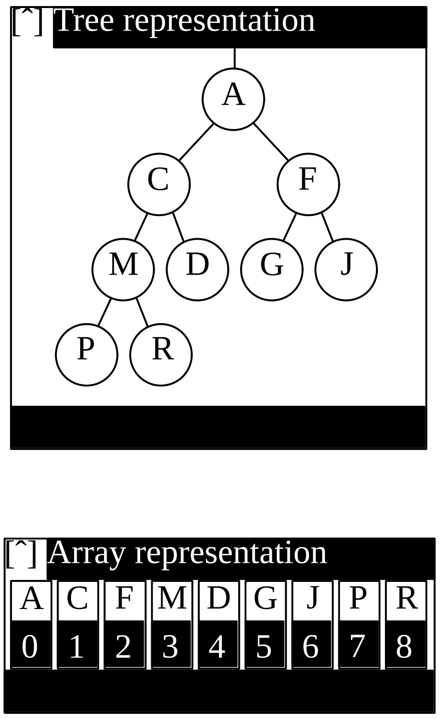


Figure 6: **Two representations for a single binary heap.**

In Figure 6 we show the visualization of a CDT employing two different representations. Heap operations are best visualized using the binary tree concept whereas the practical implementation is usually based on an array. Both of these representations are shown simultaneously as illustrated in the figure. Moreover, the user can perform concept simulation operations on one of them and see the corresponding changes in the other one immediately.

In the case of concept simulation, the interpretation for the drag and drop operation is changed if the shift key is held down during the operation. Dragging the source entity from its original location with the shift key pressed is interpreted as a deletion of the dragged entity from its original location. The deletion is performed after the entity has been dropped somewhere. Insertion (dropping) in the target entity is performed as discussed above.

The function of the delete operation depends on the dragged entity and its source entity. For example, if the source entity conforms to the interface `CDT`, then the source entity has a delete operation, which can be called after the dragged element has been dropped. An example of this is a binary search tree, which contains keys. When a key is dragged from the binary search tree while holding the shift key down, it is deleted from the tree. The key can be dropped into an empty place to delete it without inserting it anywhere. If the source entity is an FDT structure, then it doesn't have a delete operation and the drag and drop operation with the shift key held down doesn't delete anything.

# 3 The Matrix Framework

In Section 2, we discussed the concepts that the Matrix framework must support and the design and implementation of the framework on an abstract level. In this section, we will take a closer look at how the framework works.

The Matrix framework is implemented using the Java programming language. Initial versions of the framework used Java version 1.1. Now, however, the minimum requirements are Java 1.2 and a JAXP-compliant XML library, such as GNU JAXP (included with Matrix) or Crimson (included with Java 1.4). The XML library is required to parse the Matrix configuration file which is written in XML.

## 3.1 Visualization

The visualization of data structures in the Matrix framework is based on four *visual concepts*: *visual container*, *visual component*, *visual reference* and *visual data*.[32]. A *visual container* is a complex structure, which holds a number of variables (nodes, indexes, etc.) connected in a specific way. For each variable in a *visual container*, there is a *visual component* that is capable of visualizing the variable. The connections between variables may be visualized using *visual references* that are binary relations between two *visual components*. Furthermore, any part of a visualization may have *attributes* attached to it. In particular, each variable has a *key attribute*, which is a reference to *visual data* the variable represents. This *visual data* may be a primitive data type, or a more complex structure. A complex key attribute may be represented using a *visual container* nested inside the *visual component*.

The figure 7 shows the inheritance hierarchy of the classes used to implement visualization in the Matrix framework. The classes on the second level of the hierarchy correspond to the *visual concepts* discussed above. `VisualContainer` corresponds to *visual container*, `VisualComponent` to *visual component*, `VisualReference` to *visual reference*, and `VisualPrimitive` to *visual data* (*i*.e. primitive data type). However, since all objects required for the visualization have a lot of common functionality, the concepts are inherited from the abstract superclass `VisualType`. Functionality for debugging (of the framework itself), basic simulation, nested AWT component handling, etc., which are common to all visualizations, is implemented in `VisualType`. The `VisualType` class is a subclass of `java.awt.Container`, the Java AWT toolkit component that is capable of containing

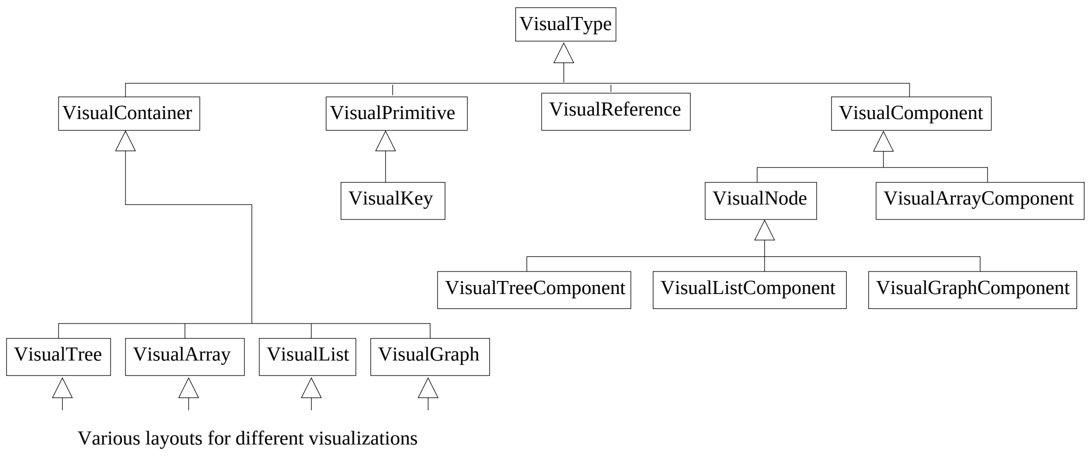


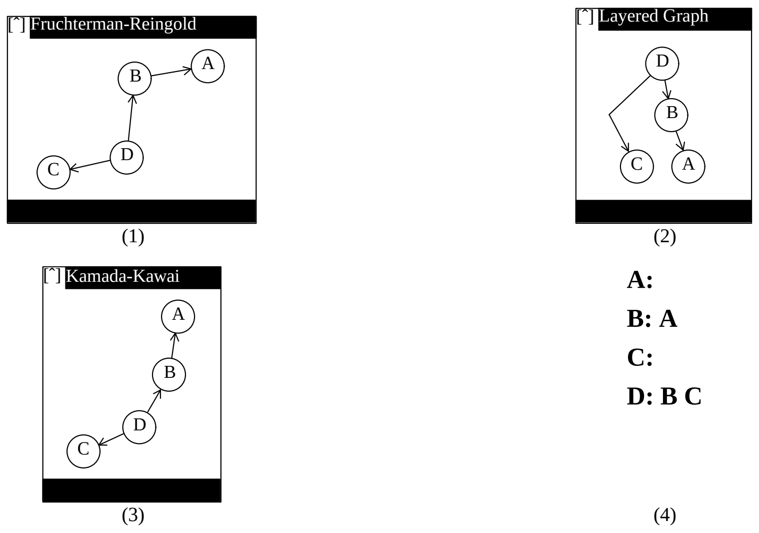
Figure 7: **Inheritance in visualization classes**

other AWT components.

The generic implementation of each *visual concept* is not, however, enough for efficient and useful visualization. The visualization of different data types such as *arrays*, *graphs*, *lists* or *trees* each require different functionality. Therefore, the class `VisualContainer` has four subclasses, `VisualTree`, `VisualArray`, `VisualList`, and `VisualGraph`. Each of these corresponds to one of the above-mentioned data types. These classes hold the functionality specific for the visualization and manipulation of a particular data type. For example, the `VisualGraph` class holds functionality that allows the user to add edges to a graph, and `VisualTree` makes sure that if a graph is visualized as tree, the shown structure has no cross- or back-edges visible. Moreover, each of the four classes corresponding to a specific data type has one or more subclasses that contain the functionality required for laying out and drawing a certain *layout*.

Each data type visualization has one or more layouts mentioned in Section 2.2. Each layout for a particular data type is contained in its own subclass. For example, the class `VisualGraph` has three subclasses: `VisualLayeredGraph`, `VisualKKGraph` and `VisualFRGraph`. `VisualLayeredGraph` implements a version of the directed acyclic graph algorithm supporting arbitrary graphs and variable-size nodes from Chapter 9 of [5], `VisualKKGraph` implements a version of the Kamada-Kawai graph drawing algorithm [30] and `VisualFRGraph` implements a version of the Fruchterman-Reingold graph drawing algorithm [18]. Figure 8 shows an example graph in adjacency list format as well as its visualization using each layout.

Similarly, the `VisualComponent` class has several subclasses that handle different kind of components. The most fundamental division is between *static* and *dynamic* data structures.



**A:**
**B: A**
**C:**
**D: B C**

Figure 8: **An example graph as (1) Fruchterman–Reingold graph, (2) layered graph, (3) Kamada-Kawai graph, and (4) adjacency list**

Static structures, such as an array, have their size (number of components) determined at the time of their creation and cannot be resized later. Dynamic structures, on the other hand, may grow (get more components) and shrink (remove components) during their existence. Components in *static* structures must be handled rather differently from *dynamic* components.

The only *static* data structure supported in Matrix is Array. Therefore `VisualArrayComponent` which handles the components (indexes) of an array is directly inherited from `VisualComponent`. However, since the framework supports several *dynamic* components, which share a lot of functionality, the class `VisualNode` is inherited from `VisualComponent`, and the visual components of different dynamic data types (list, graph and tree in Matrix) are inherited from `VisualNode`. Therefore, the `VisualNode` class exists because it is easier to implement the system that way.

## 3.2 Structures

The Matrix framework uses a number *concept interfaces* to define the different fundamental data types and abstract data types. The concept interfaces are implemented using Java interfaces. The hierarchy of the interfaces is seen in Figure 9. The `FDT` interface is the superclass of all concept interfaces. A Java object that implements the `FDT` interface is recognized by the Matrix framework as something that can be visualized. A non-FDT element (a Java object that does not implement the FDT interface) stored inside an FDT can be trivially retrieved and visualized as a primitive object (that is, a string representation of the corresponding element).

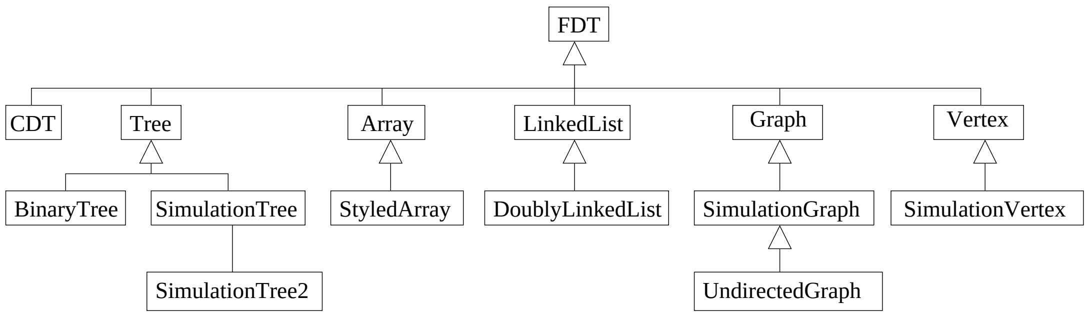


Figure 9: **The inheritance hierarchy of concept interfaces**

The FDT interface does not, however, contain any information on *how* to visualize the object implementing it. Therefore, no data structure should implement only the FDT interface, but use one of its' sub-interfaces seen in Figure 9.

The `FDT` interface has six sub-interfaces. Four of these (`Tree`, `LinkedList`, `Array` and `Graph`) correspond to the four data types mentioned in section 3.1. The `Vertex` interface represents a single vertex of a graph, and the `CDT` interface marks conceptual data types.

The `Tree`, `LinkedList`, `Array` and `Graph` interfaces also correspond to the four sub-classes of `VisualContainer` shown in Figure 7. The Matrix Framework can recognize an object that implements one or more of these four interfaces and is capable of visualizing it using layouts for that particular visualization. The `LinkedList` interface also contains methods that allow for *primitive simulation* of the structures that implements this interface.

In this context, *primitive simulation* is the ability to directly modify the variables of a data structure. That is, to add, remove, and change the contents of a variable. Depending on the type of the data structure it may also be possible to modify the references between variables (for example add, remove or move references in a graph).

For the `Tree` and `Graph` interfaces, a sub-interface containing simulation functionality is required for primitive simulation. For an `Array`, the concept of primitive simulation has no meaning, since the number of variable (array indexes) cannot be changed. It should be noted, that a data structure that implements, for example, the `Array` and `Tree` interfaces (but not `SimulationTree`, which is discussed below) can be manipulated using primitive simulation only when it is visualized as an array.

The `Tree` interface has two sub-interfaces, `BinaryTree` and `SimulationTree`. The `BinaryTree` interface includes functionality for binary trees. The interface forces each variable of a data structure to have a left and a right child, which makes it a binary tree. Without this interface, a variable's children are marked using numbers starting from 0. The `SimulationTree` interface contains functionality to allow *primitive simulation* of tree structures: creation and deletion of new tree nodes and manipulation of a node's successors. The `SimulationTree` interface has one sub-interface, `SimulationTree2`, which *marks* semantics for common trees. An object that implements the interface `SimulationTree` has a fixed number of child nodes (including empty ones), making the implementing structure a *k-ary tree*. An object implementing `SimulationTree2` can have an arbitrary number of children during the simulation process and is therefore a common tree.

The `SimulationGraph` sub-interface of the `Graph` interface is similar to `SimulationTree`, giving graphs additional functionality required for *primitive simulation*. It is possible to add, remove, and modify vertices of a `SimulationGraph`. The `UndirectedGraph` is a mere marker interface. The default visualization of objects that implement `UndirectedGraph` has all edges drawn without direction.

The vertices of a graph are defined by the `Vertex` interface, thereby making it possible to separate the implementation of a graph as a whole from the implementation of its vertices. Of course, it is possible to create a class that implements both `Graph` and `Vertex` interfaces. For `SimulationGraph`s, the `SimulationVertex` sub-interface of the `Vertex` interface should be used, otherwise it is not possible to connect vertices to each other.

Trees and graphs have separate interfaces for visualization (marking a particular structure as a tree or graph and forcing the implementation of methods required for visualization) and primitive simulation since in many cases, we might want to visualize tree or graph structures that should not be changed by the user. By leaving out the simulation functionality we can both simplify the implementation of such structures, and prevent the user from making undesired modifications. For `Array`s, primitive simulation does not exist in the same way it is possible for dynamic structures, and for `List`s the separation was deemed unnecessary.

The `Array` interface has one sub-interface, `StyledArray`. This interface implements functionality required for arrays that need array indexes other than the traditional numbers starting from zero. A class that implements `StyledArray` can have arbitrary strings as array indexes.

The `LinkedList` interface also has one sub-interface, `DoublyLinkedList`, which adds methods for making the list linked in both directions.

The last sub-interface of the `FDT` interface to be covered here is the `CDT` interface. All Java objects that implement the interface are considered to be abstract data types. An object that implements the `CDT` interface does not need to implement any of the four visualization interfaces mentioned above, as long as it can return an `FDT` that contains its internal representation. Of course, it is also possible to create structures that implement both the `CDT` interface as well as one or more visualization interfaces.

## 3.3 Implementation of the Assembly Line

Recall the visual representation assembly line shown in figure 4. We will now describe the implementation of the assembly line, using a binary search tree as an example. In this example, we will assume that no other data structures recognized by the framework can be used.

A binary search tree is a *tree* structure that has internal semantics guiding the positions of the variables contained inside it. Therefore, it is also a conceptual data type. Furthermore, each node of a binary search tree has at most two children, making it a *binary tree*. In order to visualize a binary tree as conceptual data type, we must implement at least two visualizations: `BinaryTree` and `CDT`. `BinaryTree` is a subclass of `Tree`, and `Tree` is a subclass of `FDT`. `CDT` is also a subclass of `FDT`. Therefore, we must implement methods of at least four interfaces. Furthermore, if we wish to be able to use *primitive simulation* to manipulate the tree, we will have to implement a fifth interface, `SimulationTree`.

All in all, these five interfaces have 14 methods that need to be implemented: two in `FDT`, two in `Tree`, two in `SimulationTree`, five in `BinaryTree` and three in `CDT`.

Through *concept interfaces*, the Matrix framework can examine and modify a given data structure. In the second phase of the assembly line, the framework examines the data structure and creates the required instances for visual concepts in order to visualize the data structure. If possible, the validation tries to reuse existing instances of visual concepts in order to make the validation process faster and more efficient. The basic validation code is in the class `VisualType`. This common code, located in method `void validate()` is called regardless of the type of visualization. It does the following:

1. removes unused subcomponents,
2. if this item is not used, return,
3. check for possible attributes and decoration,

4. validate all components and subcomponents of this visual concept, and
5. repaint the screen.

There is a great difference between validating all *components* and *subcomponents* of a visual concept.

Validating the *components* means that the current instance of `VisualType` being validated examines the data structure it represents and makes sure that for each part of the data structure there is an instance of the corresponding visual concept (a subclass of `VisualType`) that can be used to paint the visualization. Furthermore, if the current `VisualType` has some attributes or other components that do not directly correspond to any visual concept (for example, a button that shows a menu), these components are examined.

Since all instances used for visualization are subclasses of `VisualType`, and therefore subclasses of `java.awt.Container`, each of them may contain other instances of `VisualType`. The validation of *subcomponents* is the act of recursively going through each instance of `VisualType` contained in the instance currently examined. The `validate()` method is called to all visualization elements contained in the current instance.

The validation of a `VisualTree`, the class that handles the visualization of tree structures (like a binary search tree), goes as follows. First, there is an instance of `VisualTree`, which corresponds to the whole tree data structure. This is the *visual container* that holds the visualization of the tree. The `validate()` method is called for this object. To validate all its *components*, the `VisualTree` goes through the `Tree` it represents, creating the required `VisualComponent`s and `VisualReference`s. It also makes sure that if there are cross- or back-edges in the tree (the tree is actually a graph that is being visualized using a tree concept), these are interpreted as pointers to terminal nodes.

After the *components* of the `VisualTree` have been validated and, if necessary, new components created, the `VisualTree` calls `validate()` for each *subcomponent* it contains. Each *subcomponent* validates itself as well as its components and subcomponents (if it has any). After the validation is done, we have a set of valid instances of `VisualType` that can be used to draw a visualization of a data structure. The last step of the assembly line is to lay out these objects and to draw the image on screen.

Laying out the components is done in the specific class that handles layouts. Trees, for example, can be laid out using two layouts: `LayeredTree` and `LeafTree`. The layout part of the assembly line calculates a position for each instance of `VisualType` to be drawn on the screen. How the positions are calculated depends on the layout used. The `LayeredTree` layout tries to minimize the amount of wasted space in the tree visualization: it calculates the amount of space required by each node of the tree, and positions nodes so that they are as close to each other as possible. The `LeafTree` layout lays the tree out so that each leaf node has a its own "slice" of the x-axis of the visualization. Finally, the visualizations are drawn using the `paint(Graphics)` method, like all `java.awt.Component`s.

## 3.4 Animation

Animation in the Matrix framework is implemented on the *data structure level*. Animation does not record changes in the *visualization*, but in the underlying data structures.

Animation in the Matrix framework can be divided into two cases: forward animation, where the framework visualizes the changes as a data structure is modified, and backward animation, where the framework is used to rewind the data structure into an earlier configuration.

The system can automatically support forward animation for any data structure which conforms to one or more of the *concept interfaces* discussed in section 3.2. Support for backwards animation requires, however, some extra effort by the programmer of the data structure. An animation in the Matrix framework is stored as a series of changes in the data structure. Therefore, whenever an animation is rewound (moved backwards), the system actually restores the relevant data structure(s) into an earlier configuration, instead of just changing the visual representation(s). Unfortunately, it is very hard to store changes made to arbitrary Java objects. For this reason, the instance variables (variables which store the contents of the structure) must be stored in specialized `MemoryStructure` objects. These objects, which include representations for several Java primitives, a generic `Object` as well as an array of objects, have support for storage and retrieval of their change history. Without using these specialized objects, it is not possible for the framework to move the animation backward.

## 3.5 Simulation

Algorithm simulation in the Matrix framework is based on the fact that each data structure visualization is composed of several visualization objects, each of which represents an instance of one of the four visualization concepts discussed in section 2.3. Each of the concepts is mapped directly to a data structure, its component node, edge or key. It is therefore possible to interpret a user's manipulation of the visualization and reflect these changes in the underlying data structure. Each of the nine operations discussed in section 2.3.1 is implemented, and can be performed by dragging and dropping a visualization into another.

Furthermore, the system allows operations that cannot be implemented by dragging and dropping a visualization into another. These operations, like changing the name (label) of a visualization, changing the layout or the visualization concept, or changing the color of a node, can be implemented through a number of ways. The most often used method for implementing additional operations is through a pop-up menu. The contents of the menu depend on the visualization.

Each simulation operation a user makes is first interpreted by the visualization (an instance of (a subclass of) `VisualType`) where the operation was directed at. The `VisualType` decides how to interpret the operation (depending on the `VisualType`). In most cases the `VisualType` calls for one of the underlying data structure's methods. After the method has (possibly) modified the data structure, the `VisualType` marks itself invalid, since the underlying structure may have changed. The system notices the invalidation, and proceeds to validate and (possibly) lay out and redraw the structure.

## 3.6 Graphical User Interface

A graphical user interface (GUI) is the main method of interaction between the user and an application using the Matrix framework. The GUI consists of a number of `Panel`s containing one or more *data structure visualizations* and other components. The `Panel`s can be placed inside other GUI components. The framework supplies a `Frame`, which can be used to hold one or more `Panel`s. Furthermore, the functions of the framework and the features of the GUI can be customized using a *configuration file*. The configuration file can also be used to define and customize all `Menu`s used in an application.

Components other than `Panel`s can include taskbars, status or debug windows, etc.. Currently, the framework has one ready-made component, the animation control panel. The basic functions of the control panel are similar to those of a slide projector: you can step backwards and forwards step by step, go to the first or last position of the animation, or play the animation as a series of discrete steps. In addition to these "basic" functions, it is possible to examine the smaller microsteps (down to the manipulation of single variables) and see how large operations (for example, an insertion in a red-black tree) are composed of several smaller ones.

A `Panel` that holds *data structure visualizations* is the basic building block of a Matrix application. An arbitrary number of visualizations can be placed inside a single panel. However, the usefulness of the visualizations starts to suffer when too many data structures are shown simultaneously: all visualizations take up some space and the maximum amount of space that can be shown at one time is constricted by the size of the computer monitor.

The main advantage of placing several visualizations inside a single `Panel` is the possibility of interaction between data structures. When structures are visualized inside the same `Panel`, it is possible to move items from a visualization into another one. The Matrix framework includes its own `Panel`, the `StructurePanel` class, which is a subclass of `java.awt.Panel`. The `StructurePanel` contains the functionality required to implement the interaction between visualizations and the functionality required to interact with other parts of the Matrix framework, such as the `Animator`.

Several `Panel`s and other components can be included in a single `Frame`. Matrix has its own `Frame` class, which contains functionality common to many applications. Functions such as opening new data structures for visualization, changing the font, font size, saving the visualization on disk, etc. have been implemented in the `MatrixFrame` class.

Many of the functions in `Panel`s or `Frame`s can be accessed through `Menu`s. Menus in the Matrix framework are defined and customized in the *configuration file*. Using the configuration file, it is possible to create menubars to `Frame`s or pop-up menus to any visualization component. Because of the tree–like structure of menus (a menu or menubar may have sub-menus, which can hold sub-menus of their own), the menubars are defined using an XML-syntax.

The *configuration file* is also used to define the *visualizations* of various data structures. It is possible, for example, to change the possible and default layouts of a data structure using the configuration file. The configuration file can also be used to set initial values of variables (for example, to set the default background color used in `Panel`s and `Frame`s).

## 4 Applications

In this section, we present the most important applications built on Matrix. *TRAKLA2* is a web-based learning environment dedicated to distribute *visual algorithm simulation exercises* in a

Data Structures and Algorithms course. In this paper, we describe only the simulation exercise framework used in TRAKLA2[3]. *MatrixPro* is a tool for instructors for creating customizable algorithm animations in terms of algorithm simulation. *MVT* (*Matrix Visual Tester*) is a visual testing tool based on Matrix.

## 4.1 TRAKLA2

TRAKLA2 [37] is a web-based application for *automatically assessed visual algorithm simulation exercises*. The system offers developers many ways to *modify existing exercises, create new exercises* and *modify the GUI* provided for students. Therefore, the system itself can also be seen as an independent framework. The exercises in TRAKLA2 have the following characteristics:

- The student manipulates conceptual views of data structures, simulating actions a real algorithm would perform.
- All manipulation is carried out in terms of graphical user interface operations.
- The system records the sequence of actions prepared by the student and submits them to the server.
- The server compares the submitted sequence to a sequence generated by the real implemented algorithm and provides feedback about the comparison.
- The initial data for each exercise is personally tailored (e.g. randomly selected) for each student.

TRAKLA2 is a descendant of TRAKLA [27] and TRAKLA-EDIT [33], and was designed to solve the problems in those systems (see section 1.3). Compared to the old systems, TRAKLA2 has the following advantages:

- The evaluation of the submitted answer is based on comparing the sequences of states of data structures instead of comparing the final states only (or a few intermediate states only).
- The student can request grading of the solution unlimited number of times. However, after the grading action, the student cannot continue solving the exercise with the same data.

[3]For more information about the learning environment, visit `http://www.cs.hut.fi/Research/TRAKLA2/`

- After grading, the student can decide whether he/she submits the answer to the course data base.
- Students can also request the model answer for the personally tailored exercise. This answer is presented as an algorithm animation that the student can browse backward and forward. However — as with grading — the student has to reset the exercise after viewing the model answer.
- The system logs some information about the user interface actions for statistical analysis. This data aids us in improving the system itself, tuning individual exercises, and understanding students' learning better.

TRAKLA2 was brought into production use in Spring 2003. The system was used in the basic data structures and algorithms courses[4] with about 600 students. The attitude toward the system among students was very positive [37] and currently we are implementing more exercises, so that we can completely abandon old TRAKLA and TRAKLA-EDIT. For a list of the currently implemented exercises see Appendix C.

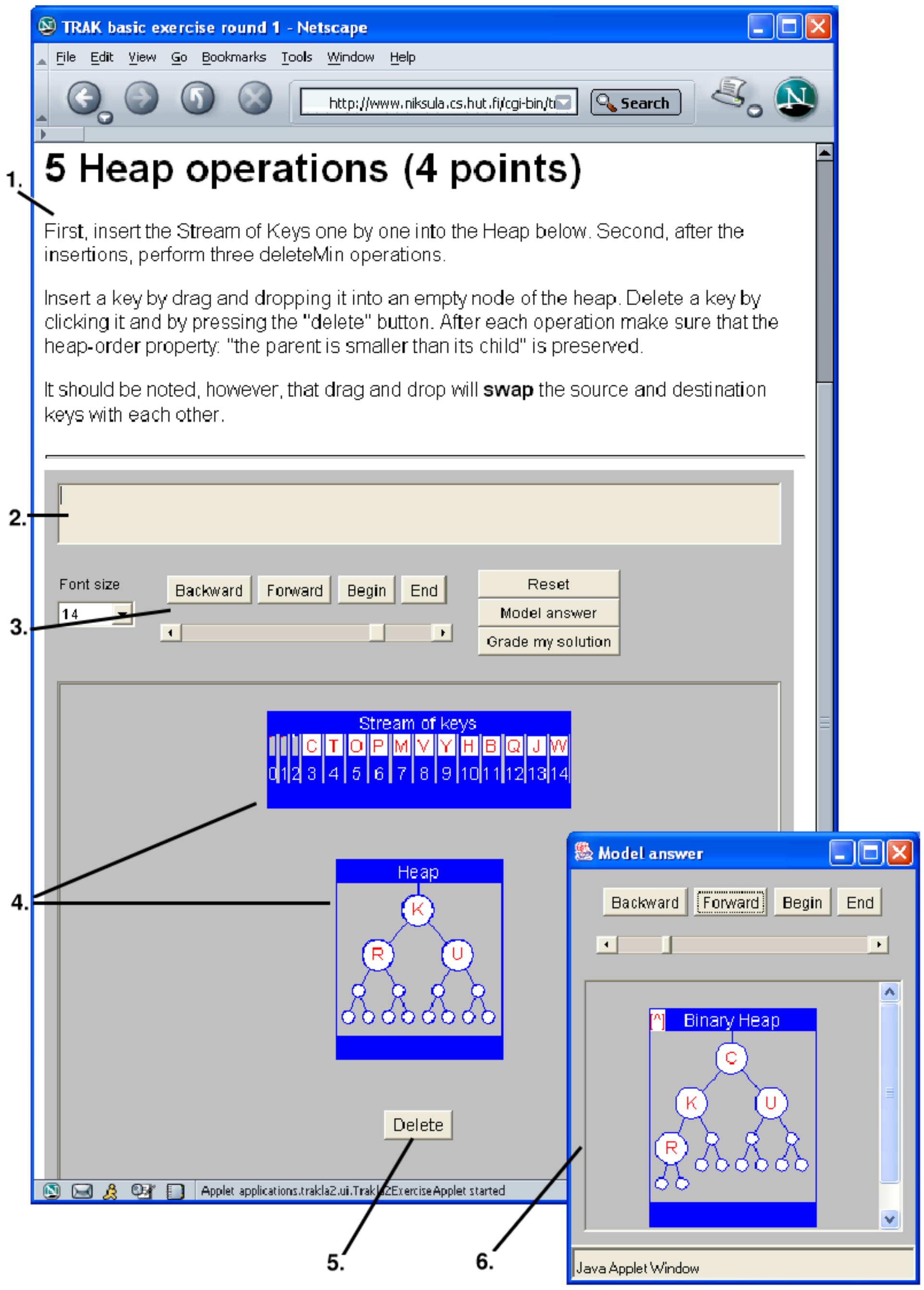
#### 4.1.1 Point of View of a Student

Let us consider the exercise in Figure 10. The exercise page contains the following elements: At the top of the page is the **Textual exercise description** (marked with number 1 in the Figure). This description can also contain links to the additional material (e.g. electronic books or real implementations of the algorithm). Directly below the textual exercise description is the TRAKLA2 applet. Topmost in the applet is another exercise description (2), which is used to show information about the exercise that changes when the exercise is initialized with new data. The next element in the exercise applet is the **Animator panel** (3) that can be used to navigate backward and forward through the solution the student is creating. In addition to the animator panel that the Matrix framework provides, the panel also has some buttons related to submission and initialization of the exercise. At the bottom of the applet (and the page), we have a set of **visualized data structures** (4) that the student manipulates.

The student creates his solution through *algorithm simulation*, that is, direct manipulation of the data structures, through GUI operations of Matrix. Therefore, in these exercises, students directly manipulate conceptual visualizations of data structures in order to simulate the running

[4]One course version was for CS majors — `http://www.cs.hut.fi/Studies/T-106.250/` — and one for students of other engineering curricula — `http://www.cs.hut.fi/Studies/T-106.253/`.

of given algorithms. In this particular exercise, keys can be moved from the key array to the heap with simple drag and drop operations. The swap operation that is needed when constructing the heap is performed by dragging and dropping a key in the heap on top of another key (e.g. dragging and dropping R on top of the K in Figure 10 will swap these two keys). The delete operation is performed by first selecting the target node and then pressing the corresponding push button (5). The animator panel can also be used during the simulation process as described in Section 3.6.


TRAK basic exercise round 1 - Netscape

5 Heap operations (4 points)

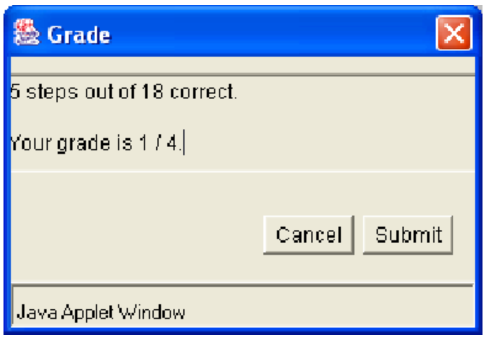
First, insert the Stream of Keys one by one into the Heap below. Second, after the insertions, perform three deleteMin operations.

Insert a key by drag and dropping it into an empty node of the heap. Delete a key by clicking it and by pressing the "delete" button. After each operation make sure that the heap-order property: "the parent is smaller than its child" is preserved.

It should be noted, however, that drag and drop will **swap** the source and destination keys with each other.

Applet applications.trakla2.ui.TraklaZExerciseApplet started

Figure 10: **TRAKLA2 applet page and the model answer window.**



5 steps out of 18 correct.

Your grade is 1 / 4.

Figure 11: **TRAKLA2 feedback window. Here, the exercise can be submitted or one can try to solve the exercise again with new input data.**

After solving the exercise, the student can ask for feedback on his solution by pressing the `Grade` button. The answer is checked and textual feedback is provided (See Figure 11). When the grading is done, the student can submit his solution to the course database by using the `Submit` button. It should be noted that, if grading has been requested but the answer was not submitted, the answer cannot be graded or submitted any more. However, the exercise can be reinitialized with new data by pressing the `Reset` button. The binary heap exercise, for example, is initialized with 15 alphabetic keys (Stream of keys), that do not contain duplicates. This means that the exercise can be initialized in more than $10^{19}$ different ways. The student can *reset the exercise* at any time. As a result, the exercise is reinitialized with new random keys. Reinitialization always restores the possibility of grading and submitting.

By pressing the `Model answer` button, the student gets the model answer for the assignment. The model answer is presented as an algorithm animation (see number 6 in Figure 10). After the student requests the model answer, he cannot continue with the same data. In general, an exercise can be solved an unlimited number of times, but each time the input for the algorithmic exercise is different. However, the algorithm itself remains the same.

### 4.1.2 Implementing Exercises

The process of creating the a new exercise consists of several steps: writing a manuscript, implementing the exercise, testing and taking the exercise into production.

The first thing to do is to write a textual description, a manuscript of the exercise. The manuscript should contain at least the name of the exercise, exercise description and instructions given to the user, as well as the input, auxiliary and output data structures and the way the exercise is graded.

Based on the manuscript, the the exercise can be implemented in Java using the features and concepts of TRAKLA2 and the Matrix framework. Each exercise has a corresponding Java class, which defines the elements of the exercise by implementing several TRAKLA2 interfaces. The source code for the exercise in Figure 10 is presented in Appendix A. TRAKLA2 exercises can be implemented by defining at least the following elements:

1. Data structure types used in the exercise (e.g. array, binary tree)
2. Names for the visual representations (e.g. “Stream of keys”, “Heap”)
3. Methods for creating randomized initial values for data structures. In general, arbitrary random data is not suitable and input values have to be tailored for the exercise.
4. An algorithm for creating the model answer.

There is often a need to change the default GUI behavior provided by the Matrix framework. This can be achieved by implementing additional interfaces. For example, by default the drag and drop operations provided by Matrix can provide too much freedom for the student. The Matrix framework allows many operations on the representations. Therefore, it is important to define which operations are enabled and which are disabled for each structure in the exercise. For example, in the Heap exercise, dragging keys from the table is enabled, whereas dropping a key into the table is disabled (see the implementation of the `ConfigureVisualType` interface in Appendix A for more details). If hierarchical structures (see Section 2.3.1) are used, it is also possible to define different constraints for each visualization type used to visualize a hierarchical structure.

Sometimes the default user interface operations are not adequate for performing all the operations needed in the exercises and thus adding new functionality to the user interface is needed. This can also be achieved by implementing interfaces. For example, the drag and drop operations that the Matrix framework provides are not suitable for all kinds of data structure manipulations. In the heap exercise, deletion of a tree node is performed by selecting the node to be deleted and pressing the `Delete` button. In general, buttons can added to the exercise by implementing the `ButtonExercise` interface (see source code for more details).

Testing and taking into production use are not examined here closer, since they are straightforward steps and thus not interesting.

#### 4.1.3 Server-side

All TRAKLA2 exercises are presented within an applet that is merely a user interface for an algorithmic exercise. The applet should be embedded into a web page, which contains the actual description of the exercise. The surrounding web-based environment takes care of authentication and selecting exercises. Among many other parameters, the exercise applet takes the IP address of the server and the communication between the applet and the server is carried out over the RMI protocol.

The server stores all the submitted answers and keeps track of students' grades. The server also logs data about the UI operations the student performs in the applet. All this information can later be used to study students' learning and behavior while using the system.

#### 4.1.4 Discussion

We have noticed that the actual implementation of the exercise is not the most challenging part of making a new exercise. The implementation is so straightforward that there is no need for a deeper understanding of the underlying Matrix framework — knowing the basic concepts and interfaces is enough. The work required to implement a new exercise has typically been 2–3 workdays for a competent programmer (if no new data structures or visualizations are needed). Now, the effort should be aimed toward designing good manuscripts for the exercises and making the user interface intuitive for the learner.

However, implementing a new exercise may require implementing new data structures and algorithms. This will increase the workload and requires understanding of the framework. However, according to our experience, it is quite unlikely that one would need to add new visualizations to the framework. A list of the data structures and visualizations in the framework is represented in Appendix D.

## 4.2 MatrixPro

MatrixPro (see Fig. 12) is a tool for instructors for creating algorithm animations in terms of algorithm simulation. The animations can be prepared prior to a lecture or on-the-fly during the lecture in order to demonstrate different algorithms and data structures at hand. Thus, the tool allows the instructor to ask what-if type of questions in a lecture situation and make the lecture more interactive. Moreover, there is an option to introduce exercises for students [37].

For computer science students, MatrixPro can be a tool for figuring out how different algorithms work. Solving the exercises the student can test his or her knowledge about a topic.

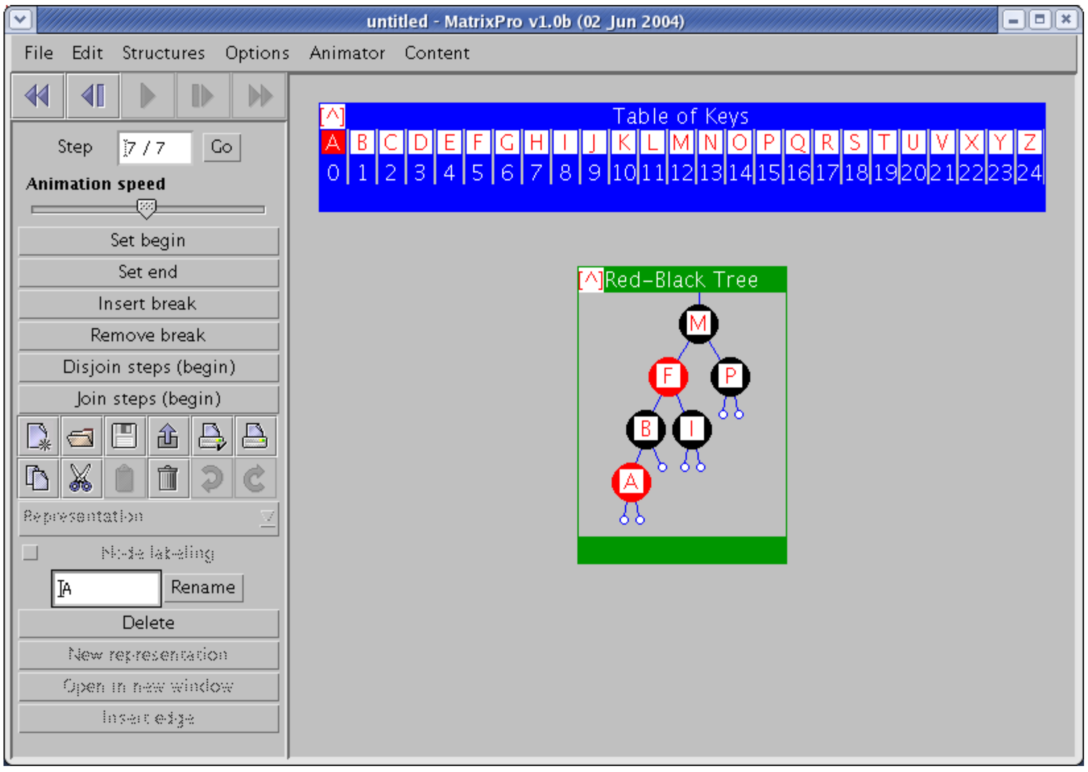


Figure 12: **The MatrixPro main window. The main components are the menubar at the top of the window, toolbar on the left and data structure visualizations on the right.**

### 4.2.1 Overview

The MatrixPro main window is represented in Figure 12. The main components of the window are the menubar at the top, toolbar on the left and data structure visualizations on the right.

The menubar contains the general file commands like open, save, close and exit. There are also commands to insert new data structures and to move the animator backward and forward. One menu contains different options and one the TRAKLA2 exercises.

The toolbar contains components to quickly access the most used features and also some features used to customize the animation that are not available elsewhere. The most important component is the topmost visual animator that makes it possible to move backward and forward in the animation.

#### 4.2.2 Features

**Ex tempore usage.** One of the most important features is the ability to use the system on-the-fly basis. MatrixPro offers an easy way to create algorithm animations by either applying the automatic animation of CDT structures or by simulating an algorithm by hand. All the ready-made CDT structures can be animated on-the-fly by invoking operations on them. Moreover, the user can freely modify the data structures as an FDT by making primitive operations.

**Customized animations.** The system supports customization of animation in two ways. First, the user can build animations using algorithm simulation. The second customization feature allows controlling the granularity of the visualized execution history, *i.e.*, how large steps are shown when browsing the animation sequence. For example, primitive FDT modifications usually correspond to a couple of micro steps that form a single animator step. CDT operations, however, may consist of many animator steps. Thus, a number of animator steps can be combined to a macro step that is performed with a single GUI operation. In general, one animator step can have any number of nested steps.

**User-Made Operations** The system aids the algorithm simulation process by making it easy to invoke operations for structures from the toolbar. Any method of the underlying structure can have a corresponding interface functionality (*e.g.*, push button) that appears both in the toolbar and in the pop-up menu of the structure. For example, for AVL trees and Red-Black trees the automatic balancing after insertion can be turned off and the rotations and color changes can be simulated by push buttons when appropriate during the simulation process.

**Automatic labeling of nodes** The nodes in a structure can be automatically labeled, *i.e.* a unique number appears beside each node (see Figure 13 for an example). With this feature turned on, one can explicitly refer to a node while explaining or asking something. This is useful especially in a lecture situation as the instructor can ask questions concerning the view.

**Customizable user-interface.** The user interface can be customized to fit the needs of various users as the initial set of toolbar objects can easily be modified. Components can be moved up or down in the toolbar or they can be removed from or added to the toolbar. All such changes can also be made by modifying the configuration file. However, in this case, several other issues can be customized, as well, *e.g.*, by making different configurations for different user groups. For example, if the system is used in a laboratory settings by students, they probably will not need all the GUI functions, and thus the corresponding GUI components could be excluded from the toolbar, or even the whole toolbar could be left out. In addition, by

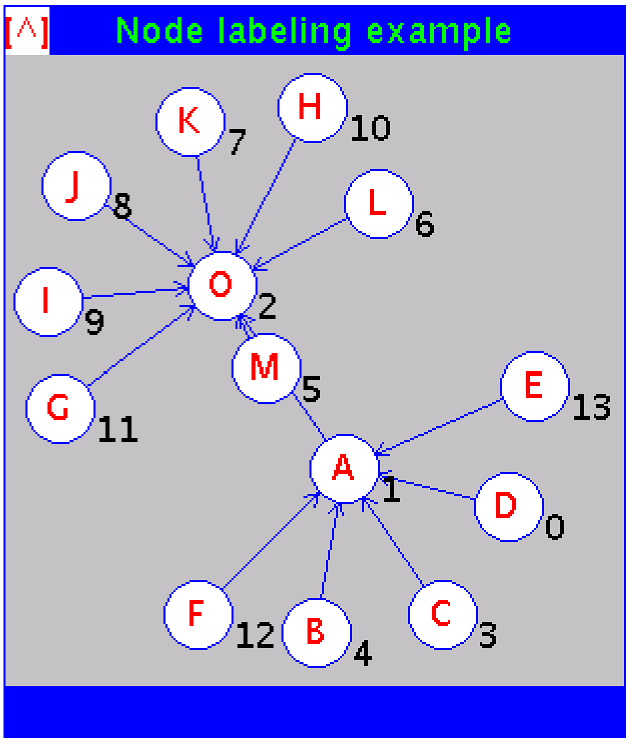


Figure 13: **Example of the automatic node labeling. A unique number appears beside each node**

modifying the configuration file, the contents of the menubar and pop-up menu can be changed. For example, the default font, font size, and background color can be set. Moreover, the default representation (layout) for each data structure can be changed.

### 4.2.3 Use of Matrix framework features

MatrixPro uses many of the features provided by the Matrix framework.

**Algorithm simulation and animation** The most important of the features offered by the Matrix framework are the algorithm simulation and animation capabilities.

**User interface** The user interface is based on the user interface components included in the Matrix framework. Although these components have been extended to add some behavior to the application, using them to develop a Matrix-based application is much easier than starting the GUI from scratch.

**Configuration** MatrixPro uses the Matrix configuration system to create the menubar and the pop-up menus, as well as to initialize the application. The available structures, visualizations, and the default visualizations for structures are configured with the system.

**TRAKLA2 exercises** The exercises created especially for TRAKLA2 are also included in the Matrix framework, and they can be used within MatrixPro without the possibility to submit the answers. List of the exercises can be found from Appendix C.

**Library and visualizations** MatrixPro includes a library of data structures, which the user can operate on. These structures have several different visualizations. Current selection is listed in Appendix D.

**Storing and Retrieving Animations.** Although the automatically created animations may serve as visualizations to illustrate an algorithm without any modifications, some instructors certainly want to customize and save examples for later use. For this use the Matrix framework provides the possibility to export and import the *underlying data structures* as serialized Java objects. Thus, the corresponding animations can be recreated from this file format. Moreover, the *animations* can also be exported to Scalable Vector Graphics (SVG) [59] form. In the latter case, the generated SVG animations also include a control panel that enables moving the animation backward and forward in a similar way as in MatrixPro itself. The speed of the animation can also be changed. Furthermore, although the animations in MatrixPro are discrete (the steps are intended to be explained while shown), the exported SVG animations use smooth animation to visualize the changes in the structures. They are intended to be more self-explanatory, and thus can be inserted, for example, in the course web material, possibly accompanied with some additional text material.

In some cases teachers need to prepare simple figures of data structure, *e.g.*, to illustrate textual material. Of course, general purpose drawing tools and formats can be applied, but they lack the ability to automate the process of creating data structure representations by directly executing the corresponding algorithms that produce the result. For example, the TEXdraw [29] macros can be considered to be such a format that the user or a tool can produce. However, creating a complex conceptual visualization (for example, a red-black tree with dozens of nodes and edges or even a directed graph) is a very time consuming process without a tool that can be automated (*i.e.*, programmed) to produce valid displays. With Matrix this process can be easily automated and the produced visualizations can be exported directly in TEXdraw format to be included in LaTeX documents.

Finally, we note that data structures can also be loaded from ASCII files which is practical, for example, for graphs, because the adjacency list representation is easy to write using any text editor. Moreover, the data structures can be saved into ASCII files, as well.

One possible use for the ASCII file formats is importing data generated by third party applications. The current Matrix framework, however, cannot import *animations* in ASCII format, only visualizations. Of course, the visualization loaded from a file can be animated further in terms of algorithm simulation. Nevertheless, one interesting future direction could be to combine the abilities of several animation and visualization tools that share a common import and export format. For example, JAWAA [47] has its own scripting language that is capable of determining animations. The field, however, lacks mutual agreement on using a general purpose common animation description language that many systems could support.

Figure 14 summarizes the working process when preparing examples.

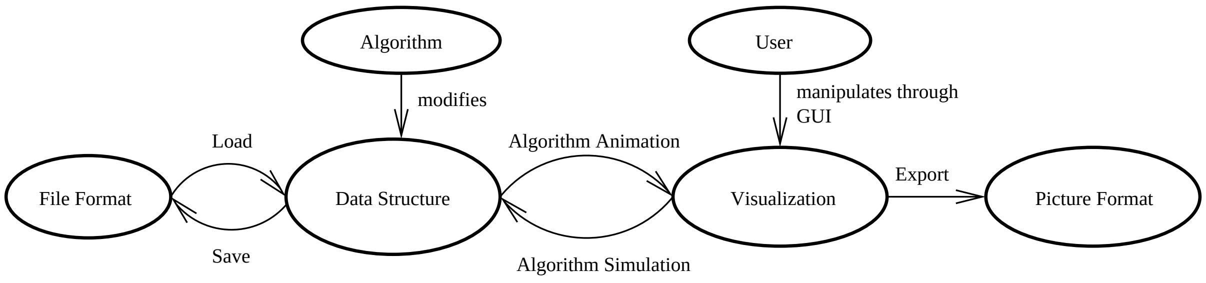


Figure 14: **The process of creating algorithm animations in terms of algorithm simulation.**

### 4.2.4 Future work and ideas

Future development ideas for the system:

1. A layout in which the nodes could be freely positioned by the user. This would make it possible to create more customized pictures and animations.
2. More structures that have different operations which help with the simulation of an algorithm.
3. More export formats could be provided. One important format would be an ordinary picture format (for example PNG or JPEG).
4. The possibility to import visualizations made with some other visualization system could be very useful. In addition, importing some of the developed graph description languages (for example GraphXML [24], or GraX [15]) could be implemented.
5. History view which would show several steps simultaneously thus making it easier to visually compare them.

6. The possibility for creating visualizations from the command line. It could be handy to be able to create pictures from a structure defined in the Matrix ASCII file format. This would enable script-based creation of visualizations.

## 4.3 MVT

Unlike TRAKLA2 and MatrixPro (both of which are designed to help teach data structures and algorithms), *MVT* (*Matrix Visual Tester*) [41] is intended for use in software development. The goal of MVT (and of *visual testing of software* in general) is to make it easier for programmers to test and debug their code and study the execution of code written both by themselves and others. This has also been the long-standing goal of *debuggers* such as FLIT [58] and GDB. *Visual debuggers*, such as DDD [60], Amethyst [46], and Lens [44], combine program visualization and debugging. Visual testing (introduced in [41]) is similar to visual debugging, but it also allows the user to:

- Manipulate data structures of a running program through a graphical user interface, as in user-controlled algorithm simulation.
- Invoke methods graphically or automatically to test program code or extract information.
- Step through the previous states of a program as in algorithm animation.
- Examine the execution history of a program using various graphical views.

As its name suggests, MVT is a visual testing tool based on Matrix. However, MVT is only a limited prototype. For example, MVT currently does not visualize the execution history in any other way than allowing stepping through the execution history as in Matrix.

MVT visualizes the state of a running Java program (which we will refer to as the *debuggee*). Most of this state consists of the data structures of the program. This places MVT firmly in the domain of program visualization even though Matrix is primarily designed for algorithm visualization.

MVT also allows the user to modify data values in the debuggee, invoke methods (including constructors) and start and stop the execution of the debuggee and the individual threads in it. In other words, MVT allows the user to manipulate a Java virtual machine (JVM) and arbitrary Java classes in much the same way Matrix on its own allows the user to manipulate data structures implemented specifically for Matrix.

### 4.3.1 Visualization

The most important view in MVT is the data view, which shows the *variables* (memory addresses that contain a value that can be read and written by a Java program) in the debuggee grouped together in *data containers*. The variables are *local variables*, *array elements* and *fields*. Local variables belong to *frames* (each of which corresponds to an invocation of a method) on the *execution stack* of a *thread*. Array elements form *arrays* and fields belong to *classes* or *objects*, which are instances of classes. The data view in MVT is based on showing the data containers as tables (similar to the way Matrix visualizes arrays) and the variables they contain as elements in the tables.

Primitive values are simply shown as text. Object references are represented by nesting the referenced object inside the referring object or using an arrow from the referring variable to the object that is referred to.

The current execution position is shown in two ways. Each stack frame is labeled with the currently executing method and line number in the source code. Also, the current line in the topmost frame in the running thread is shown highlighted in the source code view below the data view.

MVT logs the execution of the debuggee. The user can step backward and forward through the execution history using the animator controls above the data view. The data and code views in the main MVT window show the state of the program at a specified time. The user can also easily rewind the view to the last modification to a variable. Stepping back and forward through the log does not affect the actual state of the debuggee.

### 4.3.2 Elision and Abstraction

Most Java programs contain a lot of data. Being able to find the important parts of the data and present it as clearly as possible is therefore necessary. *Elision* (hiding unwanted information) and *abstraction* (hiding nonessential aspects of information such as implementation details) make it easier for the user to find and understand the information he is interested in.

MVT ignores a lot of information that is usually uninteresting or hard to keep track of efficiently. This includes the values of variables defined in standard library classes, as this information is seldom of interest to the user. Not monitoring standard library data also improves the performance of MVT by decreasing the amount of data it must process.

MVT allows the user to select the information to be shown in a variety of ways. The user can choose whether the threads of the program and their stack frames are visible. The user can also show and hide classes and their instances individually, all at once, or all instances of all the classes in a package at once. The user can choose which fields of an object are shown separately for each object or for every instance of a class or interface at once. Referenced objects are always shown if a variable containing a reference is visible.

#### 4.3.3 Controlling the Debuggee

When MVT has started the debuggee JVM, no code is running in the JVM (aside from the system and debugging threads). Typically, the user will load a class and then choose a method to execute from this class.

Choosing a method produces a method invocation object (shown as a table) into which arguments for the method call should be inserted. The method invocation can then be executed. By stepping, using breakpoints or interrupting the debuggee, the user can suspend the execution of the method and examine the state of the debuggee. Several method invocations can be active at once in separate threads (each of which can be separately suspended and resumed).

MVT allows the user to change the values of variables by dragging data values into them. New primitive variable values can be created by entering a new value as text. New objects can be created by invoking a suitable constructor. New values and values returned by method invocations are collected in a special container.

By providing the ability to invoke methods and modify data values, MVT allows the user to try out methods with different arguments without writing additional test code and recompiling.

#### 4.3.4 Implementation

MVT consists of several parts. The parts and their relationships to each other and other software are shown in Figure 15, and described in more detail in this subsection.

**Instrumentation of Debuggee** MVT uses *JDI* (*Java Debugger Interface*), the high-level API of JPDA, to connect to the debuggee and monitor the data in it. However, JDI does not provide any notifications for changes to arrays and local variables. Also, there is no easy way to identify stack frames in JDI, which makes it hard to keep track of the local variables in the stack frames.

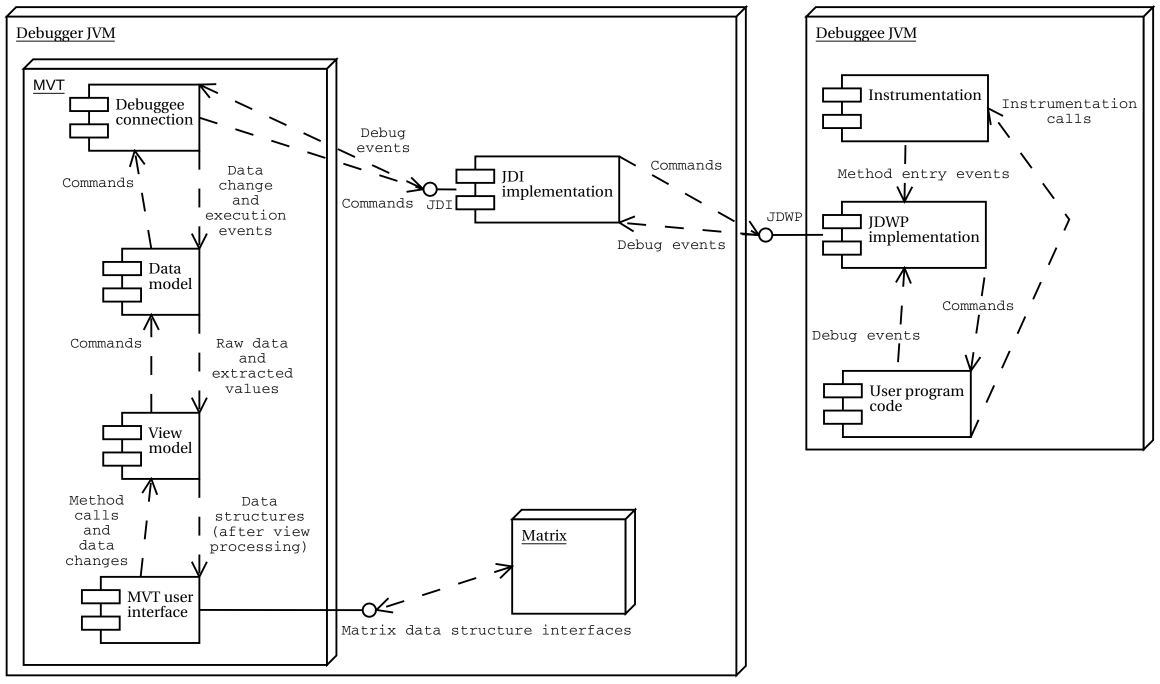


Figure 15: **Structure of MVT. In** *Debugger JVM* **Data Model logs events from the** *Debugger JVM***, View Model keeps track of visualization settings and user interface visualizes and provides means to interact with the View Model.**

Monitoring data changes and the identification of stack frames are done by inserting additional method calls in the bytecode. These method calls can be observed using JDI and the data values passed to MVT as parameters to the method calls. Unfortunately, relying on bytecode instrumentation means that MVT can not properly detect data changes made by uninstrumented code such as standard libraries and native methods.

The instrumentation code is added to the debuggee using the *instrumenter*, which should be run before starting MVT. The instrumenter uses *BCEL* (*Byte Code Engineering Library*) to parse class files, add instrumentation code to the bytecode and save the instrumented class files.

BCEL is also used to extract from the class files some additional information required by MVT while the debuggee is running that is not available through JDI, such as the mapping between local variable slots in the JVM and their names.

**Debuggee Connection** The *debuggee connection* receives JDI events (including those generated by the instrumentation) and converts them into a simpler and more consistent form. The debuggee connection part also circumvents some of the limitations of JDI. For example, the lack of multiple simultaneous user method invocations is compensated for by invoking a method that runs the user method in a new thread.

**Data Model** The *data model* uses the debuggee connection to log the events in the debuggee and create a copy that can be examined at different points in the execution history. The data model logs the executed lines of code, the values of variables (including values received from extractors) and the contents of the data containers.

**View Model** The *view model* keeps track of visualization settings, and uses these settings to build data structures that are visualized using extended versions of the array and graph visualizations of Matrix.

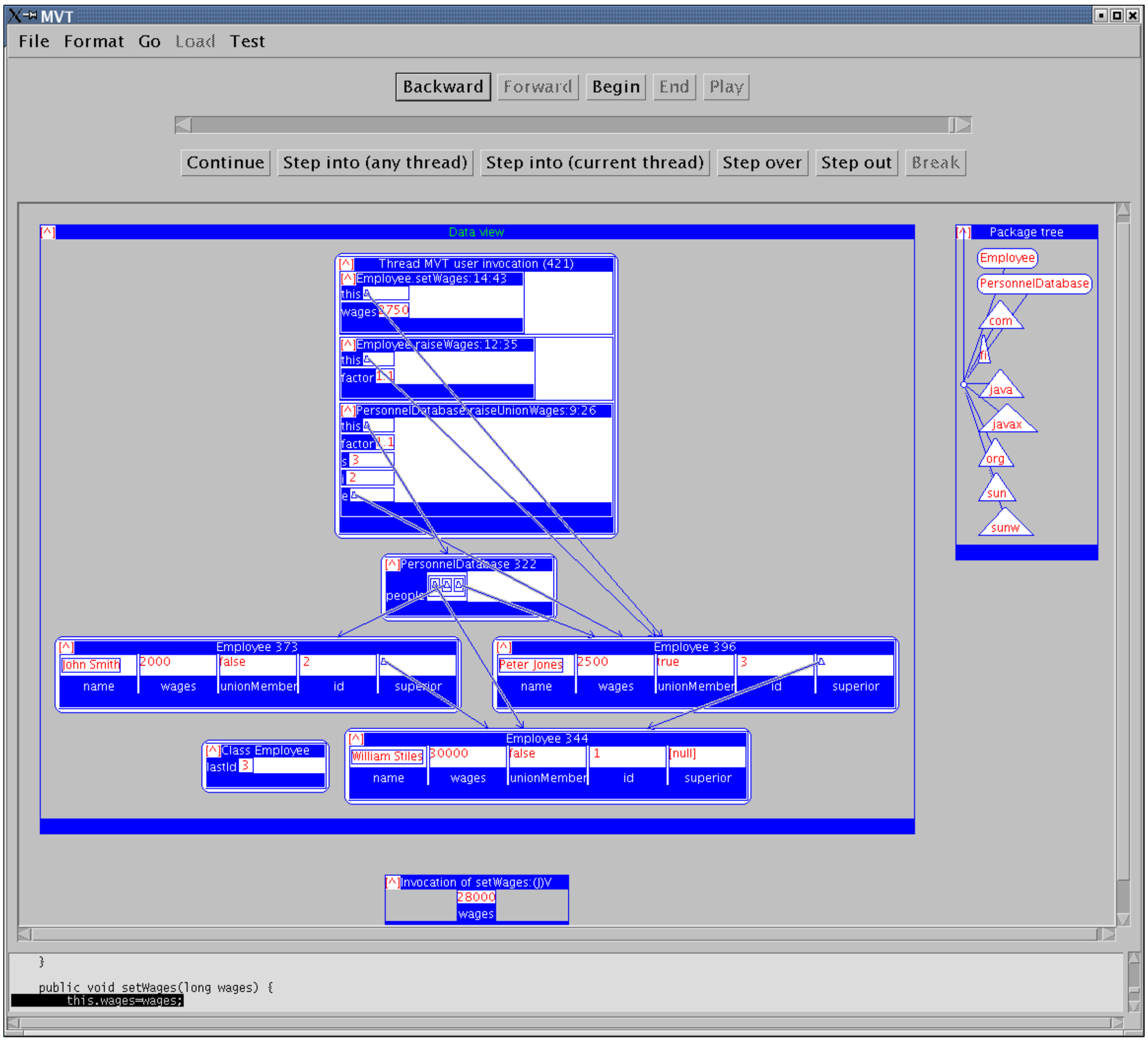
Each class has a set of associated properties that describe the desired visualization (e.g. which fields of the object are shown, whether the object should be nested inside the referring data container) and lists of extractors and converters to use. Classes inherit extractors and converters from their superclasses and implemented interfaces. If a view property is not defined for a class, the corresponding property of the most similar (based on defined methods and proximity in the inheritance hierarchy) implemented interface or ancestor class will be used.

**User Interface** The *user interface* is based on the Matrix prototype UI and extends it with additional AWT user interface components to access the features of MVT.

The main window is similar to that of the Matrix prototype UI. An example is shown in Figure 16. However, several features that are not relevant to MVT have been disabled. Instead, a source code view, debuggee execution buttons and two new menus have been added. The **Load** menu contains settings for class path and source path and the option to start the debuggee. When the debuggee has been started, the **Test** menu is enabled. This menu contains execution control options for the debuggee, some global view options and commands to create new primitive values and show the primitive array classes.

The structure panel contains the package tree (which shows the packages and classes available to the debuggee) and the data view (which shows the data containers in the debuggee and any user-created data). The package tree provides access to view settings for classes and packages and allows the user to load classes manually. The visualization of individual data containers can be adjusted using their pop-up menus. The pop-up menus also provide access to the methods of classes and objects.

The MVT window in Figure 16 contains (from top to bottom): menus, animator control buttons and bar, stepping control buttons, the structure panel and the source view with the current line


MVT
File Format Go Load Test
Backward Forward Begin End Play
Continue Step into (any thread) Step into (current thread) Step over Step out Break
Data view
Package tree
Thread MVT user invocation (421)
Employee.setWages:14:43
Employee.raiseWages:12:35
PersonnelDatabase.raiseUnionWages:9:26
PersonnelDatabase 322
Employee 373
Employee 396
Employee 344
Class Employee
Invocation of setWages:(J)V

```
}
public void setWages(long wages) {
    this.wages=wages;
```

Figure 16: **A screenshot of MVT. The components in the user interface are (from top to bottom): menus, animator control buttons and bar, stepping control buttons, the structure panel and the source view with the current line highlighted.**

highlighted.

The structure panel contains the data view, the package tree and a user-specified invocation of `setWages` that has not yet been started.

The data view shows a simple database of employees (implemented as an `ArrayList` of `Employee`s). The user has invoked a method (`raiseUnionWages`) that raises the wages of every union member in the database by 10% by iterating through the list of employees, calling `raiseWages` on every `Employee` that is a union member, which calls `setWages` to change the wages. A breakpoint has been used to interrupt `setWages` at the point where Peter Jones gets his raise. The execution stack frames corresponding to the methods that are running at this point of execution are shown inside the representation of the thread to which they belong (the innermost call is at the top). Each frame is labeled with the method name, a unique identifier (the number of the call in chronological order) for the call and the current line number. Each frame shown contains some local variables. The data view also shows the `Employee` class, which keeps track of the last (unique, increasing) `id` assigned to an employee.

The package tree shows the classes in the root package and the other packages available to the debuggee VM (the standard Java library and MVT). All packages except the root package are minimized to a single node for clarity.

The invocation structure at the bottom of the structure panel is a temporary structure used to specify the arguments for a method call. When the user chooses a method to execute (from the pop-up menu of an object or class), an invocation structure is shown. When the user has specified the arguments (by dragging values into the invocation structure), he can tell MVT to perform the method call specified by the invocation structure. In this case, the user has already specified the parameter for the `setWages` call, so he can start the method call whenever he likes.

#### 4.3.5 Conclusion

MVT is a visual testing tool; it allows a programmer to try out Java program code and examine its execution and the data it produces. However, MVT does not implement all of the ideas of visual testing; most notably, the execution history visualizations have not yet been implemented. Also, MVT is currently very slow, both in executing the debuggee and visualizing the data that it has logged. This noticeably affects the usefulness of MVT, as test runs on nontrivial real-life programs may take hours to complete instead of a few seconds and stepping through logged data is quite slow even on a fast workstation.

Future development in the field of visual testing will probably concentrate on improved performance in logging and visualization and adding execution history views that summarize the method calls and other operations performed by the program. The suggested execution history views (described in more detail in [41]) include:

- Call trees containing the operations performed by a thread structured as a tree of method calls. Such trees are used, for example, in the History view of RetroVue [14].
- Adding execution history information to the data view by adding arrows or annotations to indicate variable accesses, calls, etc.

- Dynamic dependence graphs that show the operations executed by a program and the data and control dependencies between the operations.

Performance issues may force us to use a specially modified JVM instead of accessing a standard JVM using JPDA, although compatibility and legal issues complicate this. We may also base our future visual testing tool on a different graphics or visualization framework than Matrix in order to produce more suitable visualizations or to improve performance.

## 5 Evaluation

In this section, we evaluate the key characteristics of Matrix. We use the taxonomy defined by Price, Baecker and Small [48] to classify the framework in a number of categories: *scope*, *content*, *form* and *method* used, *interaction* provided for the user, and the *effectiveness*. The taxonomy also defines a number of subcategories and even more detailed issues. Most of the subcategories are covered in this text, and in addition, we have added some new points of view to evaluate Matrix in detail.

### 5.1 Scope of the System

Category Scope specifies the range of programs that the system may take as input for visualization. There are two subcategories: Generality, i.e., how wide a range of programs can be visualized, and Scalability, i.e., how well the system scales up to large examples.

Matrix can be ranked high in Generality. It is written in Java and thus it is platform and operating system independent. Moreover, it is capable of visualizing data structures and animating algorithms written in Java (assuming the data structures conform to the interfaces in Matrix; our prototype visual testing tool MVT (described in Section 4.3) removes this restriction). It should be noted, however, that Matrix is also “programming language independent” in the sense that it is capable of visualizing and animating algorithms in terms of user controlled simulation. Thus, the actual “implementation” of the visualized algorithm is irrelevant and could be expressed by using any programming language or even some kind of *pseudocode*. Finally, by supporting the combination of language independency and ordinary animation facilities we give the user very wide possibilities for creating visualizations.

The second subcategory, Scalability can also be ranked high. The design of Matrix does not place any restrictions on program and dataset size. However, some implementation decisions and limitations may limit the usefulness of Matrix with large datasets. For example, there aren't very good layouts for large data structures.

## 5.2 Content of Visualization

The content of visualization defines what subset of information about the software is visualized. The two main subcategories are the support for algorithm animation and for program visualization. An additional aspect is how accurately the visualization reflects the actual concepts and behavior of the underlying virtual machine.

Matrix is primarily intended to support working on the conceptual level. We want, however, to remove the burden of laborious creation of concept visualizations. Another important aspect is to provide pedagogically sound default visualizations for basic data structures. This has led to the idea of having a system in which the representations correspond to the visualizations a teacher might draw on a blackboard in class. Because the visualization can be manipulated, either by an algorithm or by the user, the natural extension to this has been to include algorithm animation. We refer to this approach as *dynamic concept animation*.

Dynamic concept animation allows several important features. First, the graphical user interface includes examples of data structures with predefined data that can be visualized and animated easily. Moreover, the user can add data of his own to simulate the example even further. This is due to the fact that the user can interactively specify the datasets using the graphical user interface. Second, the chosen visual concept and its layout can be interactively changed during the simulation. Third, there can be multiple synchronized windows simultaneously open to visualize a data structure or portions of it by using some other visual concepts, thus providing a deeper insight to what is going on.

Due to the primary goal of supporting concept visualization and concept animation, visualizations in Matrix do not concentrate on illustrating the accurate information about the state of the underlying virtual machine. In general, we do not want to blur the default visualizations by including very detailed data about the variables. However, we have implemented an additional visual debugger (Matrix Visual Tester, see Section 4.3) that allows the user to examine these variables, if required.

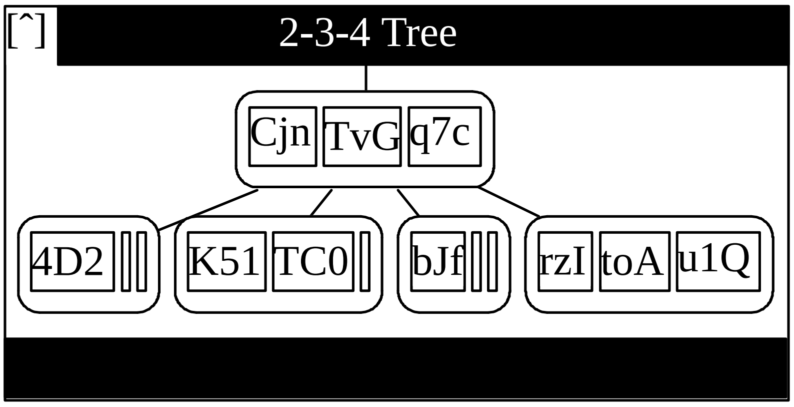


Figure 17: Visualization of 2-3-4 tree.

## 5.3 Form

Category Form describes the characteristics of the system output. It has a number of subcategories of which we cover Medium, Presentation style, Granularity and Support of multiple views.

The primary target medium of Matrix is the monitor but the system is also capable of printing colored still pictures or series of pictures that could be used as ordinary slides. The system can also be used for exporting one animation step as a TEXdraw picture or the whole animation as a Scalable Vector Graphics (SVG) animation. The graphical vocabulary used in the presentations consists of arrays, lists, trees, and graphs, which can be nested to arbitrary depth to produce more advanced visual concepts.

For example, an adjacency list is shown as an array with nested linked lists, and a B-tree is a tree with arrays in its nodes (Figure 17). The elements of any visual concept can be of any type, since the system uses a recursive algorithm to visualize the elements.

The animations generated by Matrix are sequences of snapshots of the data structures. We have not included smooth animation, i.e., smooth transitions from state to another, since Matrix provides other means for highlighting the changes in the state of the structure. The exported SVG animation, however, use smooth animation in addition to the highlighting.

The system can visualize fine-grained details to arbitrary level[5] in a manner similar to that of a debugger and it is up to the visualizer to determine how much of the details should be visible at a time. If the visual concept is too big to fit into the frame, it is still possible to filter out details to get the big picture. This is because the system provides facilities for eliding information.

[5]Down to the *primitives* that refer to any objects in which internal structure is not a subject of examination.

As mentioned before, Matrix provides tools for generating multiple synchronized views of the same data structure. Multiple views can also be used for focusing on some substructure of a dynamic data structure like a search tree.

## 5.4 Method

Method defines how the visualization is specified. It has two aspects. First, what is the style of visualization used, and second, how the visualized software and the actual visualization are connected.

Matrix provides several methods for specifying a visualization. First, the user can design and implement a new application by reusing the existing FDT classes (probes) available in Matrix. Second, the user can start from his own Java class implementing the algorithms and the data structures. To enable the animation in this case, the data structures have to implement the appropriate visual concept interfaces specified in Matrix. Of course, the structure can implement several interfaces, thus having several possible visualizations. In Matrix, any data structure that conforms to some visual concept interface can be visualized as the appropriate visual concept. The system chooses automatically the visual concept based on the type of the data structure, allocates space in the window and displays the values and the relations of single data structure objects (nodes).

In both methods above the code has to be linked to Matrix either by compiling the whole system or by using the dynamic class loader. The visualization is thereafter generated automatically. In the third method no code is needed at all and the user builds the whole animation in terms of user controlled simulation. This method is therefore based more on interaction between the system and the user (see 5.5).

Moreover, the user can tailor the visualization in different ways. First, the system includes particular *decoration interfaces*, that can be used to make additional information visible, such as extra labels, coloring etc. Second, the Matrix GUI can be used to change the layout of the visual concept, because there can exist several layouts for one single concept. For example, there are two different kinds of tree layouts in the system at the moment. One is more suitable for representing large trees because of its more economical layout and the other has the specialty to show all leaves ordered horizontally from the left to the right. Third, the layout can be rotated, so that most visual concepts can be shown in different orientations, for example, *top-down* or *left-to-right*. In addition, the display can be customized by renaming, resizing or minimizing an entity.

The visualization of Matrix is loosely coupled with the algorithm code. The idea was to separate the code and its presentation. The implementation could be described by saying that it follows the Hollywood principle: “Don’t call us. We’ll call you”. This is achieved by offering the visual concept interfaces described above.

## 5.5 Interaction

The category Interaction includes topics like how commands are given, how the user can navigate through a visualization and is it possible to manage the recording and replaying of animations. We add a new subcategory, not mentioned in the taxonomy of Price *et. al.* [48]: the possibility to manipulate the data structures directly in order to simulate algorithms.

Matrix is used through a graphical user interface based on menu techniques and graphical interaction for manipulating the structures visualized on the screen. The user’s own code can be loaded as compiled Java classes into the system.

The user can control the visualization in different ways. As mentioned, he can simulate algorithms by directly modifying the contents of nodes and the connections between them, or by using existing CDT methods to perform desired operations.

A Matrix animation is a sequence of states of the data structure regardless of how this sequence is generated. The user can replay the animation either step-by-step or continuously. In addition, the temporal direction of the animation can be reversed. This facility is very important, particularly in classroom situations where demonstrations are required.

The user may also create custom visualizations by using the existing predefined data structures. For each visual concept, there is at least one actual implementation. An instance of the data structure can be constructed from the menu, and when appropriate, the structure can be visualized with several visual concepts. For example, the binary heap implementation can be visualized both as an array and as a tree. Moreover, the supported data structures can be nested to create custom visualizations to arbitrary level.

Finally, the level of interaction is not restricted only to the control over the animation process but also real-time modification of the actual data structures is possible in terms of user controlled simulation. The interaction with the representation offers an experimental technique for exploring “possible worlds” through simulation process. Any exploration of the states of the data structure under the chosen setup is a simulation experiment in this sense. This kind of simulation experiment may serve as a vehicle to gain better understanding of the underlying concepts that are subjects of examination.

### 5.6 Effectiveness

By Effectiveness we mean the ability to communicate information to the user.

Matrix is designed to illustrate data structures and algorithms for educational purposes, thus it is important that the system is easy to use. The user can, for example, display the visualizations of basic data structures simply by selecting proper concepts from the menus and manipulating them in terms of user controlled simulation. However, the appropriateness and clarity of these representations are highly dependent on individual examples.

The Matrix-based application TRAKLA2 [37] was in production use for the first time in the basic data structures and algorithms courses at HUT in spring 2003. The exercises were a gradable and compulsory part of the courses. In total, some 600 students participated the courses and there were 14 different TRAKLA2 exercises. The attitude to the system among the students was very positive: 94% of the students thought that the system was very easy to use. In addition, they considered the TRAKLA2 exercises to be a good learning aid.

## 6 Conclusion

In this paper, we have presented the Matrix framework for visualizing data structures and for animating algorithms. Matrix provides new flexibility for creating visualizations. It combines ordinary code-based algorithm animation with algorithm simulation, in which the user directly manipulates data structures by using the graphical user interface facilities without writing any code.

Matrix has been designed to support easy building of concept visualizations and concept animations similar to textbook examples. Our experience from the TRAKLA system [27] supports our view that this approach, especially algorithm simulation exercises, improve the learning curve. The students can achieve a clear understanding of data structures and algorithms without pondering sometimes blurring implementation details.

Matrix, however, also supports automatic generation of algorithm animation for use in classes that concentrate on implementation issues. The users can write code which reuses the fundamental data types available in the system that can visualize themselves automatically. Alternatively, their own data structure classes can implement one or more visual concept interfaces which provide the automatic visualization for the visual concepts. Such code can be dynamically imported into Matrix, after which it can be simulated just like the standard system components. Thus, the user can implement a data structure of his own and survey its behavior using the simulation tools.

In addition to this functionality, there are certain features in Matrix which we consider important because they clarify the relations between the concepts of data structures and algorithms. First, one data structure can have many visualizations. For example, a binary heap implementation can be represented both as a tree and as an array. Second, one visualization can be used for displaying many different data structures. The design of the system is based on reusable visual concepts that can be mapped to any proper data structure to be visualized. Third, the visual concepts can be nested to arbitrary level. Thus, the system provides the possibility to construct arbitrarily complex structures by reusing these visual concepts.

After describing the different details of the framework and its overall structure we presented also several applications that make use of this framework and its functionalities.

TRAKLA2 [37] is a web-based application for automatically assessed visual algorithm simulation exercises. The students can do number of algorithm and data structure exercises with it by directly manipulating conceptual views of data structures and in this way simulating the actions of a real algorithm. The system is also capable of evaluating the student's submitted answer by comparing a sequence of states of the answer to the sequence of states which is produced by the actual algorithm, so that the student can be given an immediate feedback about the answer.

MatrixPro is a tool for instructors to create algorithm animations in terms of algorithm simulation to be used as examples during the lectures. The animations can be prepared prior to the lecture or on-the-fly during the lecture. For computer science students, MatrixPro can be a tool for figuring out how different algorithms work by, for example, manipulating the ready-made structures of the Matrix framework. MatrixPro includes a number of useful features like customized animations, automatic labeling of nodes, and customizable user-interface besides the features of the Matrix framework.

Third application, MVT (Matrix Visual Tester), is different from the two previous applications which are designed to help teach data structures and algorithms. Instead, MVT is intended for use in software development. The goal of MVT is to make it easier for programmers to test and debug their code. This visual testing allows users to manipulate data structures in a running program through a graphical use interface, as in user-controlled algorithm simulation. It also makes it possible, for example, to invoke methods graphically and step through the previous states of a program as in algorithm animation.

## 6.1 Future Directions

Matrix currently includes the properties described above. However, many new ideas and features are still under development. In the future releases at least some of the following aspects should be developed further:

1. establishment of global library of exercises and demonstrations (requires possibly a web file system),
2. implementation of internal pseudo-programming language interpreter (in order to provide on-line programming without compilation),
3. combining the abilities of several animation and visualization tools in terms of shared common import and export format (possibly by sharing an XML format definition for common data structures),
4. further customization of representations,
5. including smooth animation in the Matrix to present more clearly the changes in the structures during the animation, and
6. more research on program animation capabilities.

## References

[1] O. Astrachan and S. H. Rodger. Animation, visualization, and interaction in CS1 assignments. In *The proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 317–321, Atlanta, GA, USA, 1998. ACM.

[2] O. Astrachan, T. Selby, and J. Unger. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of Frontiers in Education*, pages 130–134, 1996.

[3] R. M. Baecker. *Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science*, chapter 24, pages 369–381. The MIT Press, Cambridge, MA, 1998.

[4] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, pages 261–265, New Orleans, LA, USA, 1999. ACM.

[5] G. D. Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[6] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

[7] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, 1993.

[8] C. M. Boroni, T. J. Eneboe, F. W. Goosey, J. A. Ross, and R. J. Ross. Dancing with Dynalab. In *27th SIGCSE Technical Symposium on Computer Science Education*, pages 135–139. ACM, 1996.

[9] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 139–143. ACM, 2000.

[10] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, Massachussets, 1988.

[11] M. H. Brown. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan, Oct. 1991.

[12] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, 1992.

[13] M. H. Brown and R. Raisamo. JCAT: Collaborative active textbooks using Java. *Computer Networks and ISDN Systems*, 29(14):1577–1586, 1997.

[14] J. Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[15] J. Ebert, B. Kullbach, and A. Winter. Grax: Graph exchange format. In *Workshop on Standard Exchange Formats (WoSEF) at (ICSE'00)*, Limerick, Ireland, 2000.

[16] J. English and P. Siviter. Experience with an automatically assessed course. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 168–171, Helsinki, Finland, 2000. ACM.

[17] R. Fleischer and L. Kucera. Algorithm animation for teaching. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 113–128, Dagstuhl, Germany, 2001. Springer.

[18] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experiment*, 21(11):1129–1164, 1991.

[19] P. A. Gloor. *User interface issues for algorithm animation*, chapter 11, pages 145–152. The MIT Press, Cambridge, MA, 1998.

[20] O. Grillmeyer. An interactive multimedia textbook for introductory computer science. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 286–290. ACM Press, 1999.

[21] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the Web. In *Proceedings of Symposium on Visual Languages*, pages 360–367, Isle of Capri, Italy, 1997. IEEE.

[22] S. R. Hansen, N. H. Narayanan, and D. Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1), May 2000.

[23] R. R. Henry, K. M. Whaley, and B. Forstall. The University of Washington illustrating compiler. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

[24] I. Herman and M. S. Marshall. GraphXML - an XML-based graph description format. In *Graph Drawing*, pages 52–62, 2000.

[25] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 46–50. ACM Press, 2002.

[26] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

[27] J. Hyvönen and L. Malmi. TRAKLA – a system for teaching algorithms using email and a graphical editor. In *Proceedings of HYPERMEDIA in Vaasa*, pages 141–147, 1993.

[28] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education*, pages 335–339, San Jose, California, USA, 1997. ACM.

[29] P. Kabal. TEXdraw – PostScript drawings from TEX. Web page, 1993. `http://www.tau.ac.il/cc/pages/docs/tex-3.1415/texdraw\protect\T1\textu`

[30] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[31] A. Korhonen. *Algorithm Animation and Simulation*. Licenciate's thesis, Helsinki University of Technology, 2000.

[32] A. Korhonen. *Visual Algorithm Simulation*. Doctoral thesis, Helsinki University of Technology, 2003.

[33] A. Korhonen and L. Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, 2000. ACM.

[34] A. Korhonen and L. Malmi. Proposed design pattern for object structure visualization. In *Proceedings of the First Program Visualization Workshop*, pages 89–100, Porvoo, Finland, 2001. University of Joensuu.

[35] A. Korhonen and L. Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

[36] A. Korhonen, L. Malmi, J. Nikander, and P. Silvasti. Algorithm simulation – a novel way to specify algorithm animations. In M. Ben-Ari, editor, *Proceedings of the Second Program Visualization Workshop*, pages 28–36, HorstrupCentret, Denmark, June 2002.

[37] A. Korhonen, L. Malmi, and P. Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[38] A. Korhonen, E. Sutinen, and J. Tarhio. Understanding algorithms by means of visualized path testing. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 256–268, Dagstuhl, Germany, 2002. Springer.

[39] J. F. Korsh and R. Sangwan. Animating programs and students in the laboratory. In *Proceedings of Frontiers in Education*, pages 1139–1144, 1998.

[40] P. LaFollette, J. Korsh, and R. Sangwan. A visual interface for effortless animation of C/C++ programs. *Journal of Visual Languages and Computing*, 11(1):27–48, 2000.

[41] J. Lönnberg. Visual testing of software. Master's thesis, Helsinki University of Technology, Oct. 2003.

[42] D. V. Mason and D. M. Woit. Providing mark-up and feedback to students with online marking. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 3–6, New Orleans, LA, USA, 1999. ACM.

[43] B. P. Miller. What to draw? When to draw? An essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, 18(2):265–269, 1993.

[44] S. Mukherjea and J. T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *Transactions on Computer-Human Interaction*, 1(3):215–244, 1994.

[45] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[46] B. A. Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice Pascal programmers. In *IEEE Workshop on Visual Languages*, pages 192–198. IEEE, Oct. 1988.

[47] W. Pierson and S. Rodger. Web-based animation of data structures using JAWAA. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, Atlanta, GA, USA, 1998. ACM.

[48] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[49] R. Rasala. Automatic array algorithm animation in C++. In *The Proceedings of the 30th SIGCSE Technical Symposium on Computer science education*, pages 257–260, New Orleans, LA, USA, 1999. ACM.

[50] K. A. Reek. The TRY system or how to avoid testing student programs. In *Proceedings of SIGCSE'1989*, pages 112–116. ACM, 1989.

[51] G. Rößling. The ANIMAL algorithm animation tool. In *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 37–40, Helsinki, Finland, 2000. ACM.

[52] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, United Kingdom, 2001. ACM.

[53] J. T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[54] J. T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of Conference on Human Factors and Computing Systems*, pages 307–314, New Orleans, Louisiana, USA, 1991. ACM, New York.

[55] J. T. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, San Jose, CA, USA, 1997. ACM.

[56] J. T. Stasko. *Building Software Visualizations through Direct Manipulation and Demonstration*, chapter 14, pages 103–118. MIT Press, Cambridge, MA, 1998.

[57] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[58] T. G. Stockham and J. B. Dennis. FLIT — Flexowriter Interrogation Tape: A symbolic utility program for the TX-0. Memo 5001-23, MIT, July 1960.

[59] W3C. Scalable Vector Graphics (SVG) 1.0 specification. `http://www.w3.org/TR/SVG`, Sept. 2001.

[60] A. Zeller. Animating data structures in DDD. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 69–78, Porvoo, Finland, 2001. University of Joensuu.

## A Source Code for the Heap Exercise

This is the source code for the heap exercise discussed in Section 4.1.

```
package content.exercises;
import matrix.util.*;
import matrix.structures.FDT.*;
import matrix.structures.FDT.probe.*;
import matrix.structures.CDT.probe.*;
import matrix.animation.Animator;
import matrix.simulation.VisualTypeConf;
import matrix.structures.memory.*;

public class Heap_Insert_Delete extends AbstractSimulationExercise
    implements SimulationExerciseModel, ModelAnswerNames,
               ButtonExercise, ConfigureVisualType {

    private ExerHeap studentHeap;
    private String randomInput;

    public Heap_Insert_Delete() {}
```

from the SimulationExerciseModel interface

```
    // Initializes and returns all the structures that are provided for
    // the student. Creates a new random input for the exercise and
    // stores this random input, so the model solution can be later
    // created. Uses the inherited field: ‘‘seed’’ to initialize the random
    // number generator.
    public FDT[] init() {

        randomInput =
            RandomKey.createNoDuplicateUppercaseRandomKey(
                new java.util.Random(seed),15);
        studentHeap = new ExerHeap(15);

        return new FDT[] { new Table(randomInput), studentHeap };
    }


    // Gives the titles for the structures that init() returns.
    public String[] getStructureNames() {
        return new String[] { "Stream of keys", "Heap" };
    }


    // Returns an array of structures returned by init(). If all
    // structures are not needed when assessing the answer only a subset
    // of those structures are provided. Length of the array has to be same
    // as in the model answer returned by solve().
    public FDT[] getAnswer() {
        return new FDT[] { studentHeap };
    }


    // Returns additional textual information that can be included in the
    // exercise. There is no additional information in this specific exercise.
    public String getDescription() {
        return "";
    }
    // Returns the model solution (array of matrix structures) for
    // this exercise. Animator is used to separate steps that are going
    // to be graded - normally one point is given for each step
    // found in the model answer.
    public FDT[] solve() {
        ExerHeap modelAnswer = new ExerHeap(15);
        Animator animator = Animator.getActiveAnimator();
        Table tbl = new Table(randomInput);

        // First we insert all keys to the heap
        for(int i = 0; i < tbl.size(); i++) {
            animator.startOperation();
            modelAnswer.insert(tbl.getObject(i));
            animator.endOperation();
        }

        // Then we perform 3 deleteMin/Max operations to the heap
        for(int i = 0; i < 3; i++) {
            animator.startOperation();
            modelAnswer.delete(null);
            animator.endOperation();
        }

        return new FDT[] { modelAnswer };
    }
```

from the ModelAnswerNames interface

```
    // Just like getStructureNames() but these are titles for model
    // answer structures
    public String[] getModelAnswerNames() {
        return new String[] { "Binary Heap" };
    }
```

from the ButtonExercise interface

```
    // Button labels
    public String[] buttonNames() {
        return new String[] {"Delete"};
    }


    // We call deleteRoot method for the currently selected structure
    // if the button is pressed
    public String[] buttonCommands() {
        return new String[]
            {"reflectSelectedVisualType(reflectEDT(deleteRoot))"};
    }
```

from the ConfigureVisualType interface

```
    // Some restrictions to structures that were returned by the
    // init(). e.g. VisualKeys can be dragged and dropped - but you
    // dragging is not allowed for other structures.
    public VisualTypeConf[] conf() {

        VisualTypeConf tabl = new VisualTypeConf();
        tabl.enable("matrix.visual.VisualKey",
                    VisualTypeConf.HIGHLIGHT_OPERATION);
        tabl.enable("matrix.visual.VisualKey",
                    VisualTypeConf.DRAG_OPERATION);

        VisualTypeConf tre = new VisualTypeConf();
        tre.enable("matrix.visual.VisualKey",
                   VisualTypeConf.HIGHLIGHT_OPERATION);
        tre.enable("matrix.visual.VisualKey",
                   VisualTypeConf.DROP_OPERATION);
        tre.enable("matrix.visual.VisualKey",
                   VisualTypeConf.DRAG_OPERATION);
        tre.enable("matrix.visual.VisualKey",
                   VisualTypeConf.POP_UP_MENU_OPERATION);
        tre.enable("matrix.visual.VisualLayeredTreeComponent",
                   VisualTypeConf.HIGHLIGHT_OPERATION);
        tre.enable("matrix.visual.VisualLayeredTreeComponent",
                   VisualTypeConf.DROP_OPERATION);

        return new VisualTypeConf[] { tabl, tre };
    }


    // Trick how to handle version problems in serialization
    static final long serialVersionUID = -7671756502477641250L;
}
```

## B Example of the ASCII file format

Here is a short example of the ASCII file format. Figure 18 presents the corresponding structure after importing the ASCII file into MatrixPro.

```
#matrix structures                        //heading of the file
example#1                                 //name of the main structure
#matrix graph adjacency-list              //type of the structure is graph
example#1_1:example#1_2 example#1_3 example#1_4 example#1_5 //nodes in adjacency-list
#EOS                                      //end of structure -character
  example#1_1                             //description of the first inner structure
  #matrix array                           //type of structure is array
  Cjn                                     //first key of the array
  TvG
  q7C
  #EOS                                    //end of the inner structure
  example#1_2                             //another inner structure
  #matrix array
  4D2
  ’
  ’
  #EOS
  example#1_3
  #matrix array
  K51
  TC0
  ’
  #EOS
  example#1_4
  #matrix array
  bJf
  ’
  ’
  #EOS
  example#1_5
  #matrix array
  rzI
  toA
  u1Q
  #EOS
```

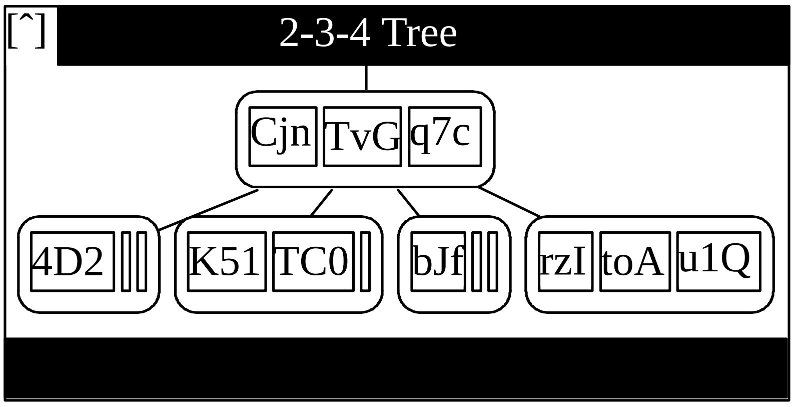


Figure 18: Visualization of the structure described in the ASCII file.

## C List of TRAKLA2 exercises

**Search algorithms:** Binary search, Interpolation search

**Tree traversal:** Preorder traversal, Inorder traversal, Postorder traversal, Levelorder traversal

**Sorting algorithms:** Quicksort, Radix Exchange Sort

**Search trees:** Binary search tree insertion, Binary search tree deletion, Digital search tree insertion, Radix search trie insertion, Single rotation, Double rotation, AVL tree insertion, Red-black-tree tree insertion

**Priority queues:** Heap building, Heap deleteMin

**Hashing:** Linear probing, Quadratic probing, Double hashing

**Graph algorithms:** BFS, DFS, Prim’s algorithm, Dijkstra’s algorithm

## D List of structures and visual representations

| **Structure** | **Default representation** | **Possible representations** |
|---|---|---|
| *Fundamental data types* | | |
| Array | array | array |
| Linked list | list | list |
| Binary tree | layered tree | layered tree, leaf tree, array |
| Common tree | layered tree | layered tree, leaf tree, layered graph vertex |
| Undirected graph | Fruchterman-Reingold graph | layered graph, Kamada-Kawai graph, Fruchterman-Reingold graph, dummy graph, array |
| Directed graph | Fruchterman-Reingold graph | layered graph, Kamada-Kawai graph, Fruchterman-Reingold graph, dummy graph, array |
| *Conceptual data types* | | |
| Queue | list | list |
| Stack | array, list[6] | array, list |
| Binary heap | layered tree | array, layered tree, leaf tree |
| Binary search tree | layered tree | array, layered tree, leaf tree, layered graph vertex |
| AVL tree | layered tree | array, layered tree, leaf tree, layered graph vertex |
| Red-black tree | layered tree | array, layered tree, leaf tree, layered graph vertex |
| 2-3-4 tree | layered tree | layered tree, leaf tree |
| Digital search tree | layered tree | layered tree, leaf tree |
| Radix search trie | layered tree | array, layered tree, leaf tree, layered graph vertex |
| Splay tree | layered tree | array, layered tree, leaf tree, layered graph vertex |