Explaining Algorithms: A New Perspective

Tomasz Müldner
Jodrey School of Computer Science
Acadia University,
Wolfville, NS, Canada B4P 2R6
Tel: 1-902-585-1578
Fax: 1-902-585-1067
Tomasz.Muldner@acadiau.ca

Elhadi Shakshuki
Jodrey School of Computer Science
Acadia University,
Wolfville, NS, Canada B4P 2R6
Tel: 1-902-585-1524
Fax: 1-902-585-1067
Elhadi.Shakshuki@acadiau.ca
Explaining Algorithms: A New Perspective

Abstract
This paper presents a novel approach for explaining algorithms that aims to overcome various pedagogical limitations of the current visualization systems. The main idea is that at any given time, a learner is able to focus on a single problem. This problem can be explained, studied, understood and tested, before the learner moves on to study another problem. Towards this end, a visualization system that explains algorithms at various levels of abstraction has been designed and implemented. In this system, each abstraction is focused on a single operation from the algorithm using various media, including text and an associated visualization. The explanations are designed to help the user to understand basic properties of the operation represented by this abstraction, for example its invariants. The explanation system allows the user to traverse the hierarchy graph, using either a top-down (from primitive operations to general operations) approach or a bottom-up approach. The visualization system is based on Macromedia Flash FX. This paper provides the selection sort example of algorithm explanation.

Keywords: Algorithm implementation, visualization, implementation, abstraction, invariants.
1. Introduction

Today, an *algorithm* means a specific set of instructions for carrying out a procedure or solving a problem, and algorithms play an intrinsic role in Computer Science. Because of their importance, researchers have been trying to find the best way to teach and learn algorithms. One of the best known approaches has been to use visualization; especially its understanding as “the power or process of forming a mental image of vision of something not actually present to the sight” (Petre et al., 1998a).

Visualizations use various kinds of multimedia, including graphics to represent data, animation and video to convey the temporal evolution of a computer algorithm (Stasko & Lawrence, 1998), and voice for explanations, also called auralization (Brown & Hershberger, 1998). Visualizations often come in a form of local programs executed in a single machine. They also come as distributed web-based programs, using various kinds of server-side scripts to produce visualizations. In this case, the remote user can use applets or Java Server Pages to access these visualizations, choose the required algorithm, select input data, and so on.

There have been many papers describing the use of animation to software explanation. This paper will not review these usages, but interested readers are referred to the following two best known anthologies (Gloor, 1992; and Gloor, 1998). A typical approach used for algorithm visualization is:

1. Take the *description* of the algorithm (usually this description comes in the form of the code in a specific programming language, such as C).
2. Graphically represent *data* in the code using bars, points, etc.
3. Use animation to represent the *flow of control*.
4. Show the animated algorithm and hope that the learner will now *understand* the algorithm.

Step 2 is often automatically generated from the source code, possibly with the help of specifying “interesting events” (Brown & Sedgewick, 1998).

There are various problems with the above approach. First, providing a *graphical representation* of an algorithm is just another way to show the *code* of the algorithm – instead of using a textual programming language, we use a graphical language. Executing the visualization, we *simulate* the code.
written in a textual language, using a graphical language and the representation is typically at the low level of abstraction that shows the low level steps.

To explain an algorithm, too much emphasis may be placed on meta-tools (graphics and animation) rather than the problem at hand. Indeed, some studies found that the effect of using animation is either neutral or even negative (Stasko & Lawrence, 1998). Dijkstra (1989) even feared “permanent mental damage for most students exposed to program visualization software”. However, Crosby and Stelovsky (1995) found that students who interacted with an algorithm animation performed significantly better than students who listened to a lecture. Recently, an overview evaluation of the educational effectiveness of algorithm visualization has been given in Hundhausen et al. (2002).

In this paper, we describe an alternative and systematic procedure to explain algorithms, and this is why we talk about explaining algorithms. Our approach is based on results of evaluations of existing algorithm visualizations, some findings from Cognitive Psychology and Constructivism Theory regarding the active use of algorithm explanation systems, and experience from software engineering and verification regarding the use of multiple levels of abstraction and properties, such as invariants, to explain algorithms and justify their correctness. However, unlike some more radical believers of Constructivism, and following the experience with design patterns (Gamma et al., 1995), we believe that algorithm explanation should be prepared by experts, who have intimate knowledge of the algorithm and the best way of coding it.

Specifically, we use a hierarchical, tree-based algorithm abstraction model. In this model, a single abstraction is designed to explain one operation using the Abstract Data Type (ADT) defined for this specific abstraction, with the top abstraction explaining the algorithm. For example, the root abstraction for a selection sort algorithm is designed to explain this algorithm, using the ADT operations. In this example, one of the ADT operations is a function smallest() that finds the smallest element in the sequence. A visual representation is used by the student to help him or her understand the basic properties of this abstraction; for example, invariants of the selection sort. The lower levels of abstraction focus on operations that are considered primitive at the higher levels. In the selection sort example, the low level
abstraction provides ADT with primitive operations to represent the function smallest(). Accordingly, some operations are implemented at lower levels of abstraction, while others are left as primitives. While the code is presented as text, there is also an associated visual representation, which is used by the student to help him or her understand the basic properties of this abstraction. Animation is not used because of experimental data showing that animations often confuse students (Stasko & Lawrence, 1998).

The student would study algorithms by interacting with a system for algorithm explanation. An algorithm explanation system is an interactive system, designed to include a student model to support evaluations and adaptations. It guides the student through levels of abstraction, and asks related questions; for example what are the invariants of the algorithm.

The rest of this paper is organized as follows. Section 2 briefly describes related work on algorithm visualization and animation. Section 3 describes our approach and section 4 provides two examples of algorithm explanations. Section 5 describes the implementation of the visualization system in Macromedia Flash MX. Finally, section 6 provides some conclusions and describes future research.

2. Related Work: Algorithm Visualization

For the sake of completeness, we start with several definitions that are mostly borrowed from Petre et al. (1998a). An algorithm animation (AA), also called algorithm visualization (AV), provides an abstraction of the algorithm data and operations, visualizes the current state of the algorithm, and animates transitions between successive states. There is an essential difference between AA and program animation (PA), or software animation (SA), in that the former visualizes algorithms while the latter visualizes the actual program execution. To illustrate this point, Fig. 1 shows a typical example of an algorithm animation - a screenshot from the animation of insertion sort using JAWAA, taken from Duke University (Whitley & King, 2001). The animation is accompanied by the following instructions:

- Start by comparing the first two rectangles in the group, and shift the second rectangle to the left if it is smaller.
• Look at the next rectangle, compare with the one on its left, and shift it to the left until it is smaller than the one to its right and larger than the one to its left.

• Repeat step 2 until finished.

![Figure 1: Animation of insertion sort.](image)

As you can see, the user has to map the problem domain (values to be sorted) to the graphical domain (bars), and then looking at the animation the user has to retrieve essential properties of the algorithm, such as maintaining a sorted prefix. This and many other existing algorithm animation systems resemble visual debuggers in that they show the execution of the algorithm by code-stepping, work at the lowest level of abstraction, and illustrate only the primitive code. This approach constrains users to view the code in the order of execution, which is the wrong information for understanding the algorithm. In addition, it has a poor cognitive fit with the plan-and-goal structures that users are trying to extract from the code (Petre et al., 1998a). Finally, these systems suffer from the lack of focus on relevant data (Braune & Wilhelm, 2000).

JSamba (1998) is the Java version of the well-known SAMBA (Stasko, 1996) AV system. Similar to JAWAA, it uses a scripting language to support graphic primitives and animations. On the other hand, Interactive Data Structure Visualization (IDSV) proposed in (Jarc et al., 1995) is not a general purpose system; instead it provides animations of various algorithms, for example sorting algorithms. IDSV is more interactive than JAWAA and JSamba. It also provides "I'll try" function to test users'
understanding of the algorithm. The tests are performed by asking users questions such as “What is the next step of the algorithm?”. Another system called “A New Interactive Modeler for Animations in Lectures” (ANIMAL) proposed by Roessling et al. (2000) is a general purpose AV system. In this system, animation can be created by scripting, calling Java-based API, or using a specialized editor. Finally, Algorithms in Action (AIA) system proposed in Stern and Naish (2002a; 2002b) does not use an abstract, language-independent pseudocode and has some features that belong to a visual debugger rather than a tool to learn algorithms.

In summary, most existing systems do not attempt to visualize or even suggest essential properties, such as invariants, which are essential for understanding algorithm correctness. One notable exception is the approach in Wilhelm et al. (2001), which uses a static source code analysis to abstractly execute the algorithm on “all possible sets of input data”, and visualize invariants.

3. Algorithm Explanation

This section describes in details our proposed approach, which is called Algorithm Explanation (AE). The main goal of AE is to support the task of algorithm comprehension (we are interested in learning, rather than teaching). According to Petre et al. (1998b) to make this task possible, the learner has to build a mapping from the domain consisting of the algorithm entities and temporal events to learner’s conceptions of these entities and events.

The target audiences are students who know well programming in at least one programming language, such as C, C++ or Java and are willing to learn algorithms. These students need to learn our conventions that are used for writing the pseudocode and showing visualizations.

3.1 Goals

This section describes what exactly we want to achieve by using AE, i.e. what we expect the student will learn. This can be described by the following goals:

G1. Understanding of both, what the algorithm is doing and how it works.
G2. Ability to justify the algorithm correctness (why the algorithm works).

G3. Ability to program the algorithm in any programming language.


In order to describe a system that satisfies these goals, we now list the requirements that an algorithm explanation should satisfy.

3.2 Requirements

To achieve the above goals, the following requirements must be satisfied:

R1. The algorithm is presented at several levels of abstraction.

R2. Each level of abstraction is represented by the pseudocode, and optionally by visualizations.

R3. Active learning is supported.

R4. The design helps to understand time complexity.

R5. The presentation uses multiple views.

R6. Presentations are designed by experts.

Now, we elaborate on each of the above requirements and explain how they are related to our goals.

Re R1. There are several advantages of using multiple levels of abstraction. First, the research in cognitive psychology on knowledge organization supports using multiple levels of abstraction when dealing with complex tasks (Anderson, 1980). The idea of using more than one level of abstraction is also supported by Petre et al. (1998b) who claim that in general, it is hard to determine a single suitable level of abstraction. Second, the research has shown (Petre et al., 1998b) that if the presentation is designed to highlight some kind of information, then it is likely to obscure other kinds. In our approach, each level of abstraction is used to highlight a single kind of information; for example a single invariant, and so the student can focus on this kind of information. Third, to reason about a process in the world requires setting up a mental model of each state of an algorithm (Petre et al., 1998b).

There are two possible approaches to defining levels of algorithm abstraction. For both approaches, the top level is first defined, using operations considered to be primitive at this level. Then,
with the first approach, each top level primitive is defined at the single lower abstraction level, possibly using other primitives (those primitives would be defined at the next lower abstraction level). With the other approach, the top level primitives are replaced (rather than defined) by their definitions, which again can use some lower-level primitives. Thus, the second approach resembles code inlining. We have chosen to take the former approach, which helps to concentrate at issues pertinent to one abstraction level. We disagree with the criticism in (Faltin, 2001), which claims that structuring algorithms using this approach produces code that is longer and less efficient. The recent advances in compiler technology and the right mapping of the correctly designed pseudocode to a programming language make the code sufficiently efficient (see examples in Section 4). The latter approach was investigated in Feng (2003).

It should be noted that, in our approach, each level of abstraction has a small number of states. This is important because the large number of states makes it difficult to reason about properties of the algorithm. There are two possible ways to studying an algorithm; top-down and bottom-up. We advocate top-down approach, starting from the top level of abstraction and then moving to lower levels. The reason for using a top-down approach is that it helps students to understand various special cases, such as various ways to compare integer values in a sorting algorithm, whereas a bottom-up approach hardcoded specific cases. This approach differs from the one used in “Algorithms in Action”, AIA, from Stern & Naish (2002a and 2002b), where students were allowed to choose the starting level of abstraction, and often chose a bottom-up approach. Therefore, this requirement contributes to G1, in both, “what” and “how”.

Re R2. The pseudocode is a model of the algorithm, and it includes the high-level abstract data structures and operations. These operations are designed so that they can be directly mapped to most procedural and object-oriented programming languages. Using pseudocode, the algorithm can be studied independently of any programming language (Fleischer & Kucera, 2001). The pseudocode given at each level of abstraction has an optional visual representation, which exposes its properties, in particular its invariants. This approach is supported by findings provided by Petre et al. (1998b) that proper explanation supports a display-based reasoning; that is the display becomes a focus for reasoning and supports creating the mental image of things that do not really appear there. For example, the visualization in Fig.
5 helps to recognize two invariants of this algorithm. Note that our concept of a pseudocode differs from that used in Stern & Naish (2002a and 2002b), where the pseudocode is based on C, with some abstract procedure calls. By exposing the algorithm’s properties, in particular its invariants, at various levels of abstraction, this requirement contributes to G1 and G2. In addition, the pseudocode is written in generic terms so that it can be used not only to write concrete implementations in specific programming languages such as Java, but also to produce different concrete implementations for example using linear or binary search to implement insertion sort. This contributes to G3.

Re R3. Active learning follows Cognitive Constructivism principles, see Hundhausen et al. (2002). This style of learning includes various kinds of interactions with the student. For example, students are able to use their own input data sets; use a do-it-yourself mode, that is predict the next of the algorithm (Faltin, 2001; Stern, 2002a; Stern, 2002b), and determine the essential algorithm properties. AE always comes with several sets of representative sample input data; the learners can also use their own input data. AE includes student evaluation, which consists of programming the target algorithm, a textual programming language. This ability is missing from all algorithm animation systems, but in our opinion it is absolutely essential. The ultimate goal of teaching algorithms is to educate programmers, who will be able to implement various algorithms. Finally, AE includes post-test (Stasko & Lawrence, 1998) with tasks such as hand-execution of the algorithm on sample sets of input data, and answering various questions about the algorithm.

Re R4. AE helps the learner by providing tools that help to understand time complexity, described in the next section.

Re R5. Multiple views showing algorithm states are used to avoid forcing the viewer to remember the previous states (two consecutive frames are shown in the “comic strip” approach (Biermann & Cole, 1999). The decreased cognitive overhead contributes to understanding how the algorithm works, which contributes to goal G1.

Re R6. AE presentations are designed by experts who have a complete understanding of essential properties to be exposed, such as invariants. Essentially, our approach is similar to that used in design
patterns, which are created by experts and used by novices. It is also similar to the concept of “algorithmic design patterns” described in (Goodrich & Tamassia, 2001). The design by experts leads to a better code. Note that our proposal differs from that proposed by Hundhausen and his co-workers (Hundhausen et al., 2002) who recommend that students should design algorithm animations. We strongly believe that designing algorithm animations concentrates on meta-tools and distracts students from their primary goal, which is learning algorithms.

AE uses a variety of tools to help the students to learn algorithms, and visualization is just one possible tools. Indeed, we provide both textual and visual representation and use the latter to help students recognize and understand algorithm properties. Note that Petre et al. (1998b) stated that two representations are not necessarily better than one. In this work, we use a visual representation to help student derive essential properties, such as invariants. However, we leave the option of using just the text, and if the students can successfully derive invariants from the text they do not have to see the visual representation. Thus, the visual representation is used to help reason about the textual representation. It is available to the user, because in some cases this representation provides the so-called “gestalt” effect; it provides an overview making a structure clearer (Petre et al., 1998b).

3.3 AE Descriptions

Our goal is to develop an AE catalog consisting of descriptions of many well known algorithms. Each description, or catalog entry, consists of the following four parts:

1. A hierarchical Abstract Algorithm Model (AAM) consists of abstractions representing operations. Each abstraction explains a single operation \( \text{op}() \), and consists of a textual representation and an optional visual representation. The textual representation includes an ADT that provides data types and operations. It also provides a representation of the operation \( \text{op}() \) using the ADT from this abstraction. All these abstractions form a tree, rooted at the abstraction representing the algorithm operation. The AAM tree is used only by the implementation of the explanation system and its complexity is transparent to the learner, who is guided by the AE
There are two possible modes that can be considered for explaining an algorithm. In the first mode, the explanation can start from the algorithm explanation and proceed towards more primitive operations; this mode is represented by the top-down traversal of the AAM. In the second, the explanation can start from primitive operations and proceed towards the algorithm explanation; this mode is represented by the bottom-up traversal of the AAM.

2. An intermediate representation of the so-called primitive operations from the ADTs, which are not implemented in the AAM. This representation is designed to help the student to write concrete implementations.

3. Tools that can be used to help predicting the algorithm complexity.

4. Questions for students, including “do it yourself” mode.

In order for us to explain part 1, we assume that \( f() \) is an operation. The abstraction that explains \( f() \), \( \text{abst}(f) \) is a pair (ADT, representation of \( f() \) in the ADT), where ADT consists of data types and primitive operations. An abstraction \( \text{abst}(f) \) is a parent of an abstraction \( \text{abst}(g) \) provided that \( g \) is one of the primitive operations from the ADT \( \text{abst}(f) \). Therefore, a child abstraction provides a partial implementation of the operation from the parent abstraction. Typically, there are only few operations from any abstraction’s ADT that are implemented in a child of this abstraction; others are considered primitive operations. An Abstract Algorithm Model (AAM) of an algorithm \( f() \) is a tree rooted at \( \text{abst}(f) \).

For example, the AAM of a selection sort is a tree of abstractions rooted at \( \text{abst}(\text{selection}) \). To explain an algorithm, we construct an AAM tree with sufficient number of levels so that the student is able to understand how and why the algorithm works. In particular, the student can form and justify invariants of the algorithm.

Part 2 provides the intermediate representation of all AAM’s primitive operations. To implement the algorithm in a specific programming language, the student has to map all primitive operations that do not have implementations in the AAM to the selected language. This representation is called an Abstract
Implementation Model (AIM). The representations in AIM are generic in that they are not using any specific programming language; instead they use high-level concepts that can be mapped to many languages. Once all primitive operations are represented in the AIM, the code for the algorithm and possibly some of its operations can be easily represented using this AIM.

Let’s add that more than one explanation of a single algorithm can be made available in our system, for example the graph shown in Fig. 2 can be expanded by adding explanations from the INSERT ADT. Then, the learner can choose a version with fewer or greater number of explanations. Instead of the learner making this decision, it is better to have a system that reacts to the learner’s progress.

Figure 2: The AAM for insertion sort.

Part 3 deals with an explanation of algorithm complexity, which is one of the most difficult goals of algorithm visualization. This is because it requires mathematical notions that are hard to visualize. There may be three kinds of tools designed to help the student to derive the complexity of the algorithm being studied. The first tool, based on Horstmann (2001), gives the student a chance to experiment with various data sizes and plot a function that approximates the time spent on execution with these data. The second tool, based on Goodrich & Tamassia (2001) p. 477, provides a visualization that helps to carry out time analysis of the algorithm. The third tool, based on part 4 above, asks students various questions regarding the time complexity, and questions specific to the algorithm being studied, and evaluates their answers (for the example, see section 4.1.4).
3.4 Graph Representation of AAM

To explain an algorithm, it is often necessary to explain various Abstract Data Types, ADTs. As an example, consider a Kruskal algorithm to find a minimum-cost spanning tree (Aho, 1983). This algorithm uses three ADTs: SET, PRIORITY-QUEUE and MERGE-FIND. Each of these ADTs includes a number of operations that have to be explained. Note that there are two kinds of ADTs: “pure” ADTs that define operations but no implementations of these operations, and “concrete” ADTs that define specific implementations. For example, a pure ADT PRIORITY-QUEUE can be implemented as a concrete ADT that uses singly-linked lists. Using Java terminology, a pure ADT is an interface, whereas a concrete ADT is the implementation of this interface.

To be able to reuse ADTs, we introduce two kinds of nodes in the AAM, namely: a type node and an operation node. The explanation provided by the AAM graph shown in Fig. 3 gives the textual and visual representation of the Kruskal algorithm using three pure ADTs, shown in uppercase. Each of these ADTs may have different implementations, shown in lowercase. For example, PRIORITY-QUEUE can be implemented using a linked list, and MERGE-FIND can be implemented using trees. Note that in Fig. 3 there are five type node (shown in boxes with single frames), and three operations nodes (shown in boxes with double frames). The “Linked List” type node has two parents, indicating that it can be used to implement ADTs from both of these parents.

![Diagram](https://via.placeholder.com/150)

Figure 3: Graph representation of the Kruskal algorithm.
3.5 Design of AE Systems

An AE system should help the student to select one algorithm, and guide him or her to go through its explanations. These explanations are stored in the AAM, and the student will follow these steps (recall that the actual representation of AAM is transparent to the student who is guided by the AE system):

1. The root abstraction is explained:
   (a) The ADT associated with the root abstraction is shown.
   (b) The implementation of the operation from this abstraction, using the ADT operations is shown.
   (c) The student is asked to explain basic properties of the implementation from (b), including invariants.
   (d) The student may choose to see the visualization associated with the implementation from (b); if so this visualization is made available, and the student can enter specific input data and watch the visualization, or use a “do it yourself” mode to test their understanding of the algorithm.
   (e) Explanations as to which ADT operations are primitive operations.

2. For any child abstraction, associated with the higher-level operations that are not primitive, the explanation process from 1 above is repeated.

3. Now, the student is supposed to implement an algorithm in a selected programming language. First, AIM is shown, and the student is asked to implement an algorithm in one of several available programming languages. Sample input data are provided, including boundary cases that can be used by the student for testing and debugging their implementations.

In general, AV systems can be categorized using a taxonomy introduced by Price et al. (1998). Using this taxonomy, the above system is a specific system (used only for limited number of algorithms), and its content, which defines what aspect of an algorithm is visualized, concentrated on explaining
algorithm properties. The form, which is how a system is presented in text, graphics, and animation; and the method, which describes how to develop an explanation, are currently hand-coded from scratch. Finally, the interactions include describing invariants, answering specific questions, and implementing the algorithm.

4. Illustrative Example of Algorithm Explanation Description

In this section, an example of algorithm explanation is provided and used to demonstrate our approach.

4.1. Example: Selection Sort

This section describes the selection sort (Aho et al., 1983) example. Recall that an algorithm is explained using various levels of abstraction including root and leaf levels of abstractions; each abstraction is designed to present a single operation used in the algorithm.

4.1.1. Root Abstraction

The root abstraction provides the ADT and the representation of the algorithm using this ADT.

Abstract Data Type

The ADT consists of data and operations. The data consists of sequences of elements of type T, denoted by Seq<T>, with a linear order. In general, this order can be defined in one of three ways:

1. Type T supports the function

   int compare(const T x)

   so that y.compare(x) returns -1 if x is less than y, 0 if they are equal and +1 otherwise.

2. Type T supports the “<” operation, which can be used to compare two elements of this type.

3. There is a function (or a type)

   int comparator(const T x, const T y)
so that \texttt{comparator(x, y)} returns -1 if x is less than y, 0 if they are equal and +1 otherwise.

This paper considers the third comparing function only; the description using the other two options is almost identical. Note that the above model is \textit{generic}, in particular it does not specify a concrete ordering relation, such as \texttt{"<"} used for real numbers. Also, it does not specify any concrete data structures, for example arrays or linked lists can be used. Finally, it does not describe details of the algorithm, for example whether the algorithm works “in place” or with copying. The following six operations on an element \( t \) of the domain \texttt{Seq<T>} are available:

- \texttt{prefix(t)}, representing (a possibly empty) prefix of the sequence \( t \);
- \texttt{inc(prefix(t))}, which increments a prefix by one element;
- \texttt{suffix(t)}, where a prefix followed by the suffix is equal to the entire sequence \( t \);
- \texttt{first(suffix)}, which returns the first element of the suffix;
- \texttt{T smallest(seq<T> t, Comparator comp)}, which finds the smallest element in the sequence \( t \) (using \texttt{comp});
- \texttt{swap(T el1, T el2)}, which swaps \( el1 \) and \( el2 \).

\textbf{Algorithm Representation}

The goal of the selection sort algorithm is to sort the sequence according to a given comparator. The implementation of the selection sort using ADT described above is shown in Fig. 4.

```
void selection(Seq<T> t, Comparator comp) {
    for(prefix(t) = NULL; prefix(t) != t; inc(prefix(t)))
        swap( smallest(suffix(t), comp), first(suffix(t)));
}
```

Figure 4. The representation of the selection sort.
The student is provided with the code shown in Fig. 4, and then he or she is asked to determine the invariants of this code. There are two invariants that can be extracted from this example, including:

1. All elements in the prefix are smaller (according to the “comp” relation) than all elements in the suffix.
2. The prefix is sorted.

The first invariant is true because in each step of the “for” statement, the smallest element in the suffix is placed at the beginning of the suffix, and then the prefix is incremented, effectively placing this element at the end of the prefix. Once this invariant is established, the second invariant easily follows: appending the element from the suffix to the end of the prefix maintains the “sortedeness” of the prefix. Since the algorithm stops when the prefix contains all the elements in the sequence, the second invariant shows the correctness of this algorithm – it indeed sorts the sequence. It may be hard for the student to find both of these invariants and if they wish so, they can use the associated visualization. Fig. 5 shows all the states resulting from the execution of the code in Fig. 4 using a five sample input data. It should be noted that the visualization has been created using our visualization system, described in section 5. This visualization clearly shows the second invariant; all elements in the prefix are sorted (in Fig. 5, the prefix is represented by the dark background). In addition, this visualization can show the first invariant, by using the prefix box that is smaller than the suffix box.

Fig. 5 also shows how our visualization system works. The input data is entered into the input text box by the student who then presses the ‘Start’ button to begin the visualization. The ‘Step Forward’ and ‘Step Backward’ buttons allow the student to move forward and backward through the algorithm visualization one step at a time. The ‘Change View’ button allows the student to switch between displaying the complete history of steps, only the current and previous steps, or the current step.
4.1.2. Leaf Abstraction

Leaf abstractions represent various operations from the ADT described in section 4.1.1. For this ADT, there may be as many as six abstractions that represent the children of the root abstraction; each abstraction focuses on a single operation from the ADT. Most of these operations however are intuitively obvious. Therefore, this AAM will have only one child of the root abstraction associated with the operation `smallest()`. Other ADT operations are considered primitive. This abstraction explains the function `smallest()`.

Abstract Data Type

The ADT consists of data described in section 4.1.1, and the following operations:

- `first(t)`, returns the first element of the sequence `t`;
- `next(current, t)`, returns the element of the sequence `t` following `current`, or `NULL` if there is no such element.
Algorithm Representation

Fig. 6 shows the implementation of `smallest()` using the ADT described above.

```c
T smallest(Seq<T> t, Comparator comp) {
    small = current = first(t);
    while((current = next(current, t)) != NULL)
        if(comp(small, current) < 0)
            small = current;
    return small;
}
```

Figure 6: The representation of the function `smallest`.

Fig. 7 shows all the states resulting from the execution of the code in Fig. 6 using a five sample input data. The student is able to step through the operation of the `smallest()` function until the data is completely sorted, or the student returns to the higher level visualization by pressing either the ‘Step Forward’ or ‘Step Backward’ button.

Figure 7: Visualization of `smallest()` at the leaf abstraction.

4.1.3. The Abstract Iterator Implementation Model

This section describes a specific AIM, called the Abstract Iterator Implementation Model (AIIM). Recall that the goal of this model is to provide the intermediate representation of all primitive operations from AAM. For the purpose of our example that uses the selection sort, this model has to provide representations for five ADT operations from the root abstraction and two operations from the child
abstraction. AIIM assumes that there is an Iterator type. This Iterator helps to avoid exposing the underlying representation of the object to the client and provides a sequential access to a data object it points to. Iteration is performed over a range, which is a half-closed interval \([a, b)\). Here, \(b\) can only be used for comparison purposes and may not be accessed. Iterator type supports the following operations:

- two iterators can be compared for equality and inequality;
- an iterator can be moved forward by executing the operation \(\text{inc}()\);
- an iterator can be dereferenced to access the value it points to.

In addition, the AIIM makes several assumptions about the ADT described in section 4.1.1; specifically:

- the following type is available: \(\text{Seq}<T>::\text{Iterator}\)
- if \(t\) is a sequence in \(\text{Seq}<T>\), then the following three operations are defined:

  \[
  \begin{align*}
  \text{Iterator} & \ t.\text{begin()} & \text{points to the beginning of } t \\
  \text{Iterator} & \ t.\text{end()} & \text{points beyond } t \\
  \text{swap}(i1, i2) & \text{swap two iterators}
  \end{align*}
  \]

Table 1 shows how the seven primitive operations are represented in this model. In this table, \(\text{eop}\) and \(\text{current}\) stand for variables of type \(\text{Seq}<T>::\text{Iterator}\).

<table>
<thead>
<tr>
<th>Abstract Operation</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix(t)</td>
<td>([t.\text{begin()}, \text{eop}])</td>
</tr>
<tr>
<td>inc(prefix(t))</td>
<td>inc(\text{eop})</td>
</tr>
<tr>
<td>suffix(t)</td>
<td>([\text{eop}, t.\text{end()}])</td>
</tr>
<tr>
<td>First(suffix)</td>
<td>Value of (\text{eop})</td>
</tr>
<tr>
<td>swap(T el1, T el2)</td>
<td>Swap values of iterators respectively pointing to (el1) and (el2)</td>
</tr>
<tr>
<td>first(t)</td>
<td>Value of (t.\text{begin()})</td>
</tr>
<tr>
<td>next(current, t)</td>
<td>inc(\text{current})</td>
</tr>
</tbody>
</table>

Table 1: The representation of the primitive operations.

Figures 8 and 9 show the abstract, AIIM implementation of the function \(\text{smallest}()\) and the selection sorting algorithm respectively.
The following three sections show how the abstract AIIM representation can be mapped to three different programming languages (C, C++ and Java).

4.1.3.1. Implementation in C

An iterator in C language can be represented by a pointer. For the code presented in Figures 8 and 9, a complete C implementation is shown in Fig. 10.

```c
T smallest(Seq<T> t, Comparator comp) {
    Iterator small = value of t.begin();
    Iterator current;
    for(current = t.begin(); current != t.end(); inc(current))
        if(comp(value of current, value of small) < 0)
            small = current;
    return value of small;
}
```

```c
void selection(Seq<T> t, Comparator comp) {
    Seq<T>::Iterator eop;
    for(eop = t.begin(); eop != t.end(); inc(eop))
        swap(smallest(eop, t.end()), comp), eop);
}
```

```c
static T* smallest(T* const first, T* const last, int comp(const T, const T)) {
    T* small = first;
    T* current;
    for(current = first; current != last; ++current)
        if(comp(*current, *small) < 0) s = current;
    return small;
}

static void swap(T* const a, T* const b) {
    T x = *a;
    *a = *b;
    *b = x;
}

void selection(T* x, T* const end, int comp(const T, const T)) {
    T* eop;
    for(eop = x; eop != end; ++eop)
        swap(smallest(eop, end), eop);
}
```

Figure 10: C implementation of selection sort.
4.1.3.2. Implementation in C++

Iterators in C++ can be easily represented using STL (Musser, 1996). Fig. 11 shows a complete C++ implementation of the code presented in Fig. 9. The function smallest has an immediate implementation using the STL function `min_element`.

```cpp
template <typename Iterator, typename Predicate>
void selection(Iterator first, Iterator last, Predicate comp) {
    Iterator eop; // end of prefix
    for (eop = first; eop != last; ++eop)
        swap(*min_element(eop, last, comp), *eop);
}
```

Figure 11: C++ implementation of selection sort.

4.1.3.3. Implementation in Java

An iterator in Java can be represented using the `Iterator` class from the Java Collection classes. Fig. 12 shows a complete Java implementation of the code presented in Figures 8 and 9.

```java
public static void selection(List aList, Comparator aComp) {
    for (int i = 0; i < aList.size(); i++)
        swap(smallest(i, aList, aComp), i, aList);
}

private static int smallest(int from, List aList, Comparator aComp) {
    int minPos = from;
    int count = from;
    for (ListIterator i = aList.listIterator(from); i.hasNext(); ++count)
        if (aComp.compare(i.next(), aList.get(minPos)) < 0)
            minPos = count;
    return minPos;
}

private static void swap(int i, int j, List aList) {
    Object item = aList.get(i);
    Object jtemp = aList.get(j);
    aList.set(i, jtemp);
    aList.set(j, item);
}
```

Figure 15: Java implementation of selection sort.
4.1.4 Post Test

A list of questions specific for the selection sort algorithm may look as follows:

1. What is the number of comparisons and swaps performed when selection sort is executed for:
   a. sorted sequence;
   b. sequence sorted in reverse.
2. What is the time complexity of the function \texttt{isSorted(t)}, which checks if \( t \) is a sorted sequence?
3. Hand-execute the algorithm for a sample set of input data of size 4.
4. Hand-execute the next step of the algorithm for various states.
5. What’s the last step of the algorithm?
6. There are two invariants of this algorithm; explain which one is essential for the correctness of \texttt{swap(smallest(), eop)}, and explain why.
7. Use “do it yourself” mode.

5. Visualization System Implementation

The visualization system presented in this work is implemented using Macromedia Flash MX (Macromedia, 2003). The individual graphical elements used by the visualization system are graphic symbols defined in Flash MX. One or more instance of a symbol may be displayed and manipulated using Flash MX’s scripting language, ActionScript, which is used by the visualization system to control the visualizations of the various abstract operations that are described by the AAM.

Each non-primitive abstraction in the AAM has an associated visualization function which controls the visualization for that particular abstraction. For example, the root abstraction for the insertion sort algorithm contains a function using graphical symbols to visualize insertion sort operating at its highest level on a given set of data. Insertion sort’s only non-primitive child abstraction, the insert operation, also has an associated visualization function using similar graphical symbols to visualize the insert operation at its various stages, providing a lower level visualization of the insertion sort algorithm.
We provided two additional visualizations, one for visualizing a sequence of data and one for visualizing an iterator. These two visualizations serve as components for the visualization of the abstract operations of the AAM. For example, the high level visualization for insertion sort uses the visualization for the sequence of data to display the different states of the sequence of data as the insertion sort progresses. Similarly, the visualization for the insert operation uses both the visualization for the sequence of data and the visualization for the iterator to display the various states during the operation’s execution.

Three possible views are provided for the visualization of each non-primitive abstraction in the AAM. The first shows only the current state of the abstract operation. The second shows both the current and the previous states of the operation. The third shows the complete history of states of the operation up to and including the current state. These three views may be cycled at any time as the user steps through the algorithm visualization.

Rather than hard-coding the visualization for a specific algorithm into the visualization system, the system uses a hierarchy of classes for visualizing specific algorithms. At the top of the hierarchy there is a \texttt{Sort} class which is treated as an abstract class containing only an array of data and the declaration of the methods \texttt{display()} and \texttt{step()}. Below this abstract class in the hierarchy are the concrete classes, including \texttt{IterativeSort} and \texttt{RecursiveSort}. (Current visualizations do not use this class and it will be used for future extensions, see section 6.) Each of these classes implements the \texttt{display()} method, providing a default visualization for all iterative and recursive sorting algorithms in the system. Specific sorting algorithms are implemented as subclasses of either the \texttt{IterativeSort} or the \texttt{RecursiveSort} class. These classes provide the implementation of their respective algorithms by implementing the \texttt{step()} method which performs one step of the algorithm execution and, if needed, overriding the default visualization function provided by its respective super class. Additionally, these classes provide the implementation and the visualization for any non-primitive abstractions defined in the AAM for their respective algorithm.
Using this hierarchical design, the visualization system only accesses data and methods defined in the `Sort` class. This allows the system to visualize a specific sorting algorithm without knowing any of the specific details of that algorithm.

6. Conclusions and Future Work

Many algorithm visualization techniques have overlooked or misused multimedia, such as text, graphics and animation to reify the execution of an algorithm. To facilitate the understanding of algorithms, this paper proposed a new approach to learning algorithms, in which an algorithm is explained using a hierarchy of abstractions. Each abstraction explains a single operation from the algorithm, using ADT operations from this abstraction. Some ADT operations have implementations at the lower levels of the hierarchy, while others are left as primitives. These primitive operations are represented using an abstract implementation, which can be easily mapped to various programming languages. All operations are shown in a textual form associated with visual description, used to help the student understand basic properties of the algorithm. Therefore, an explanation of a single operation uses only two media, namely text and an associated visualization. The visualization system, based on Macromedia Flash FX, is a generic system that can be used to create visualizations of any iterative sorting algorithm.

In our future research, the visualization of recursive algorithms will be the main focus. Visualization of recursive algorithms is much harder than visualization of iterative algorithms. A standard approach is to show a call tree, with a root representing the first call, and leaves representing stop conditions in the recursion, for example the quick sort visualization in Goodrich and Tamassia (2001), pp. 469-471). Another standard approach is to show the runtime stack resulting from the recursive calls as described in Stern and Naish (2002a; 2002b). We believe that neither of these two approaches is helpful, because to they do not focus on recursive properties.

To understand, and possibly to prove recursive definitions, one uses a mathematical induction, which involves the first step, and the recursive step. Therefore, we started designing a new visualization of recursive calls, using a modified UML transition diagrams (Fowler, 2000).
References


Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience*, MIT Press, pp. 7-27.


http://www.cs.duke.edu/courses/spring01/cps049s/students/ack11/jawaasorting.html