Using Social Agents in Algorithm Animations

A Case Study with Seal

Ramin Miraftabi

14.12.2001

University of Joensuu
Department of Computer Science
Master’s thesis

Abstract

Many computer science educators note that understanding and analyzing algorithms is difficult. During the last twenty years several systems have been created that use computerized visualizations of algorithms in order to help student understanding. While some positive results can be seen, several questions remain unsolved. One of these is how to ensure that the student correctly understands the visualization that was just viewed. Using artificial intelligence in computer aided education has been presented as an option, but its traditional uses are far from usable. I argue that the use of agent-based engineering can help solve the problem of combining artificial intelligence and computer aided instruction in order to answer the question of ensuring student understanding of algorithms. As proof of the concept I present a prototypical environment for algorithm animation called Seal. It is a student-driven algorithm animation environment with an instructor agent for student guidance. As a part of Seal's development a DTD has been developed for use in representing algorithms in XML.

The ACM Computing Classification System (1998 version): K.3.2, I.2.11, H.5.1

Keywords: Software Visualizations, Algorithm Animation, Computer Aided Instruction, Intelligent Agents, Agent-Based Engineering

Preface

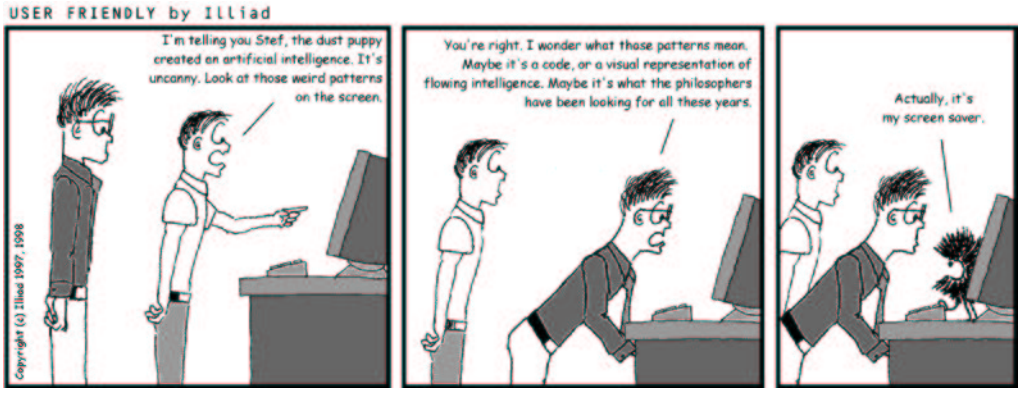

Figure 0: Copyright 1997 - 2001 Illiad (http://www.userfriendly.org). Used with permission.

I was initially guided into the world of software visualizations by Marja Kuittinen who was the advisor of my Bachelor's thesis. In it I looked at how algorithm animations can be used in computer science education. Professor Erkki Sutinen took me deeper by suggesting that I study how agents could be utilized in algorithm animations. Here we are; my sincere thanks to both of them.

The participants of the first Program Visualization Workshop provided many valuable comments on the initial design of Seal and gave me valuable ideas and resources, as can be seen in the bibliography of this thesis. Thanks to all of you.

And my family: my parents remembered to keep asking about my thesis during the years that it took to complete. Special thanks to my mother who acted as editor.

Contents

1 Introduction

As scientific disciplines go, computer science (CS) is relatively young. One of its core paradigms has its roots in mathematics (Denning et al., 1989), but other scientific disciplines — primarily engineering — had an impact on the development of computer science early in its development. CS curriculums and *Computer Science Education* (CSE) concentrate heavily on teaching algorithms. This stems from the definition that the discipline of computer science "is the systematic study of algorithmic processes that describe and transform information..." (Denning et al., 1989). As algorithms form one of the nine subareas that cover the whole discipline and lie at the core of computer science, teaching understanding and analyzing algorithms is an important part of CSE.

CSE curriculums have always used laboratory work and experimental executions of algorithms as a way of visualizing and understanding each of the steps that are taken during the execution of an algorithm. An important part of forming an understanding of an algortihm, at least in my studies, has been the step-by-step execution of an algorithm using paper and a pencil. While this approach was easy to use in the simpler algorithms in which the result of the algorithm with the given input was easily discernible, the more complicated algorithms that students face in the latter part of their studies are not quite so easily visualized.

One of the fields of research that combines computer science with other disciplines is *Computer Aided Instruction/Learning* (CAI/CAL). While CAI has been utlized in various domains and learning situations, its use in distance education offers great potential if effective tools are used (Domingue & Mulholland, 1997). Especially in CSE the use of distance education needs serious consideration as students often begin to work early on in their studies and are unable to attend all laboratory sessions and lectures.

As one solution to the difficulty of teaching algorithms several research efforts have been undertaken in which computerized visualizations are used to graphi-

cally show the steps of the algorithm and their effects on the given input data (see e.g. Sutinen, 2001 and Stasko, Domingue, Brown, & Price, 1998). Most of the visualization systems created so far have either a collection of algorithms built in or the user can input the algorithm in some programming language so that the system will then animate its execution. The amount of interaction and user control and numerous other features vary from system to system, but in essence the basic principle behind them all is the same.

From a fairly early stage of CAI research the artificial intelligence community and educationalists joined forces to create systems that combined the use of computers with current pedagogical views (Shute & Psotka, 1995). *Artificial Intelligence* (AI) is a branch of computer science that is interested in intelligent thought or rational behavior. It has a strong background in philosophy, mathematics, psychology, computer engineering and linguistics. While intelligence itself is extremely difficult to define, AI research has divided into two viewpoints: one viewpoint is concerned with modeling human thought or action whereas the other tries to create systems that either think or act rationally (Russell & Norvig, 1995). A new trend, agent-based engineering, in AI research strives to create rational software systems.

The use of AI in CAI remains relevant because the system needs to be able to understand when the user, typically a student, really understands the subject matter. Especially with visualizations the problem exists that the student does not correctly understand the visualization (Eisenstadt, Price, & Domingue, 1993), which leads to a flawed mental model of an algorithm's execution. Unfortunately the experiences of using intelligent tutoring systems in CAI has been disappointing as most intelligent tutors concentrate on toy examples (Schank, 1991) and ignore some of the fundamental ways that especially expert programmers use (Eisenstadt et al., 1993).

Unrealistic expectations are often placed on the use of CAI — and especially on intelligence in CAI. A good example can be seen in a recent issue of Newsweek (Newsweek, 2001). In the article various people from technology and education

were asked for their opinions on what a classroom would look like in 2025. Unfortunately an educator presented the view that computers will enable students to become competent writers, scientist, etc. while the teacher takes a less active role. The use of CAI in educational settings can help learning by obtaining and possibly filtering out irrelevant information, even visualizing it. However, the ability to process the information into knowledge and output it to others either by writing or by discussing is not something that a computer can teach in the foreseeable future (Schank, 1991). Various scenarios of future education and the use of computers in education can also be found in (Shute & Psotka, 1995).

While technological progress with wireless networks and ubiquitous computing will provide the means for better access to information, the ability to process the information does not come automatically with the information. While agents can help in filtering the information and creating links between relevant information and guiding a student in the right direction, they cannot, in light of present day knowledge, become educators in their own right. This point is emphasized by any who feel that intelligence, and therefore the ability to educate, requires the ability to learn (Schank, 1991). While agents can prove to be powerful tools in educational settings they have yet to prove their place in CAI, especially since an agent that truly learns is still something of the future.

Because the use of software visualizations has some benefits in an educational setting (Kehoe, Stasko, & Taylor, 1999), the effort expended to create visualization is often justified. Our discussion on the benefits of visualization techniques will concentrate on the use of algorithm animations in CSE.

While traditional artificial intelligence methodology is not viable in most modern real-world applications, the use of agents provides a way of integrating rationality into complex software systems. The rationality and emergent behavior of agents provides a new approach that can lead to beneficial uses of artificial intelligence in real-world applications.

As a testbed of the use of agents in software visualizations this thesis will concen-

trate on showing that:

> Because software visualizations are acompanied by the threat that the student does not understand the visualization that is being demonstrated, we need to find a way in which student understanding can be ensured. We will study how the use of social agents and a new approach to visualizations can provide an additional amount of guidance and visual data so that the student will notice his own misconceptions.

If we want to utilize computer aided instruction in education and especially in computer science education we need to discuss its worth as an educational tool. In this thesis I will concentrate on covering the uses of CAI software visualizations (2) in the teaching of data structures and algorithms. We will first discuss the different uses of (software) visualizations as learning tools (2.1) and take a look at some examples of systems created for animating algorithms and visualizing data (2.2). Since we are also looking into creating new systems for software visualizations we will cover some of the experiences that have already been obtained by other researchers while implementing visualization environments (2.3).

In order to circumvent some of the problems presented by intelligent tutoring systems and traditional artificial intelligence in CAI a new trend in AI research, agent-based engineering, will be examined and introduced (3). As agents are a relatively new field of research already loaded with marketing hype, we will delve deeper to uncover some of the definitions of agenthood (3.1.1). The theoretical background which agents are built upon (3.1.2) needs to be covered before we can study some of the different types of agents that can be created (3.1.3). Once we have covered the background of what comprises an agent we can begin considering different aspects of designing and implementing them (3.2).

Since our goal is to find a way into which AI can be re-introduced in CAI we will need to discuss how agents can be integrated into computer aided instruction (3.3). As a case study of how agents can be utilized in software visualizations I have

worked on developing a prototype system for agent-based algorithm animations. The system is called Seal (4).

My personal motivation to work on Seal began after a course on algorithm design and analysis. During the laboratory work we were given an assignment in which we were supposed show the paper-and-pencil execution of the Knuth-Morris-Pratt-pattern matching algorithm with the given input strings. Naturally the example input was such that the result — the given pattern was found in the main string — was obvious, but listing the states of each variable at every step of the algorithm's execution was far from simple. Even with small input strings, the KMP-algorithm required several loops that became tedious to list by hand, especially when I repeatedly had to backtrack in the execution of the algorithm when I noticed some errors in my understanding. Unfortunately, to date I still do not know if I correctly executed the algorithm since I was unable to go to the laboratory session in which the assignment was inspected. So when I was given the chance develop an application depicting the integration of CAI and agent-based engineering, I knew this was my target area.

2 Software Visualizations

Variations of animated *Software Visualizations* (SV) have been used in CSE since the beginning of the 1980s, following Baecker's classic movie *Sorting out Sorting*. Numerous applications for the use of visualizations have been created since then with work under way for many more. However, in the beginning of the 1990s only a small number of instructors used computerized visualizations in their teaching (Badre, Beranek, Morris, & Stasko, 1991).

By software visualization we mean a small subset of all forms of visualizations — a subset that is, however, very important in CSE. The term offers a visual representation of a program's execution and the state of the data that the program handles at any given time. Bergin et al. (1996) divide software visualizations into three separate subgroups that often overlap: *program visualization* or *visual debugging*, *algorithm animation*, and *data visualization*. As with all definitions, these are not set in stone and some researchers often also use program visualization to denote the whole group.

The term *program visualization* is used when we consider visualization of a programs execution. Special attention is paid to showing the data structures and variables stored in memory at runtime and their changes as execution progresses. An integral part of the visualization is showing which line of code is being executed. Many traditional debuggers give a textual representation of the data that is used to form the program visualization (Bergin et al., 1996). Another type of program visualizations includes the various models that are made of a program's execution during the design and technical documentation phases of software projects. As these are typically static representations, we will not discuss them further in this paper.

Program visualizations should not be confused with *visual programming* which includes the methods and tools for creating programs by manipulating graphical objects (FOLDOC - Computing Dictionary, 2001). Graphically manipulating ob-

jects and creating programs by ordering different functional units might be a good learning tool for novice programmers, especially children. But its value in teaching how algorithms function is questionable. A good example of a visual programming environment can be found with Lego® kits that provide a programmable interface.

When the visualization system concentrates on showing "…dynamic and interactive graphical displays of a program's fundamental operations…" (Brown, 1988) we talk about *algorithm animations*. Thus by algorithm animation environments we mean environments in which the visualization is concerned with the fundamental operations of a program instead of all of the operations of a program. This often leads to the animation of pseudo-code algorithms instead of real program code. Pseudo-code hides the low-level details that often accompany real code (Bergin et al., 1996).

Programs which concentrate on the visualization of a program's or algorithm's data belong to the third and final group: *data visualization*. Data visualization is also a fundamental part of the two aforementioned groups which is probably why programs for data visualization alone are (virtually) non-existent. Data visualizations are used in typical manual drawings in teaching situations so they do deserve their own subgroup when considering SV as a whole. Data visualization is often concerned with the visualization of all of the program's memory resident data such as memory stacks, variable values, and execution stacks in addition to showing the data in immediate use as is often the case in software visualizations and algorithm animations.

We will now go on to consider 1) when to use visualizations as a part of the learning process, 2) an overview of research on the effect of visualizations on learning, 3) a deeper investigation into different types of algorithm animation and data visualization with examples of different applications, and 4) take a look at some issues that need to be addressed when designing visualization environments especially for CSE.

2.1 Using visualizations as a learning tool

Visualizations — especially those of algorithms and data structures — have been used since the dawn of CSE. Even animated visualizations have been used for a fairly long time considering the age of the domain. Unfortunately, research on the impact that visualizations have in the learning process is still lacking even though some benefits can be discerned (Ben-Bassat Levy, Ben-Ari, & Uronen, 2001, and Kehoe et al., 1999).

In this section we will cover reasons why visualizations can be used in learning situations and some of the research that has taken place on the effects that visualizations have in learning. Because of our previous research and our work with Seal (see section 4) we will concentrate principally on the use of visualizations in learning algorithm design and analysis plus data structures at the expense of software visualizations and learning how to program in general. Data structures and algorithms have been given a vital role in what is considered the core education that all students should receive in CS (Denning et al., 1989). And we must not forget that "there is something difficult about understanding and analyzing algorithms..." (Kehoe et al., 1999).

2.1.1 Why use Visualizations?

Since understanding and analyzing algorithms is difficult (Kehoe et al., 1999) we need to find ways to make the task easier. To ease the students in their learning, most instructors of computer science use visualizations of some sort in their teaching. Even if the visualizations are present on the blackboard, overhead projector or other static media an attempt exists to visualize the execution of algorithms. What is interesting to note is that in the early 1990s only a small proportion of instructors used anything other than static visualizations (Badre et al., 1991). The situation has hopefully changed as more visualization environments have become available and computers can be easily used to project dynamic and changing im-

ages to larger audiences.

Bergin et al. (1996) have collected a number of reasons why visualizations should be used as a part of CSE. They argue that describing abstract and complicated concepts without any visualizations requires a level of formalism that is often threatening to students. Visualizations provide an external image that the students can connect to the description of the concept. This image is then easier to retrieve in subsequent use (Kehoe et al., 1999). One of the reasons for the use of visualizations provided by Bergin et al. (1996) supports this thought. They feel that all students learn some concept better by visualizing them, i.e. by creating their own mental models or internal images of them. Often when visualizations and verbal descriptions are used, the mental model that is formed has connections between the visual and verbal modes of mental representations (Mayer & Sims, 1994). Considering the argument that students do not understand algorithms because they do not have an effective model of a computer to begin with (Ben-Ari, 1998), we can conclude that a visualization should help in learning the subject matter being visualized. Whether or not this is the case will be evaluated in the next subsection.

The motivational aspects of visualizations, especially when they are presented in a visually attractive way, should not be underestimated (Kehoe et al., 1999 and Lattu, Tarhio, & Meisalo, 2000). This adds weight to the claim of Bergin et al. (1996) that visualization tools used in lecturing situations can be used to focus the students' attention on what the lecturer is speaking about.

Visualization tools used in lectures also speed up the presentation of demonstrations and examples used in lectures. The use of automated tools also helps avoid errors as the inevitable errors that occur while drawing illustrations are avoided (Bergin et al., 1996). Mayer and Sims (1994) note that using verbal and visual presentation concurrently is more effective than using them sequentially. While drawing illustrations in a presentation media, lecturers rarely have the possibility of effectively explaining what the illustration means. Therefore there is some level of sequence to the presentation which would lead to less effective results. There

is still the concurrency of the illustrations still being visible in the background, but students will have tried to understand the illustration while being drawn and before it is explained.

Anecdotal evidence suggests that interactive visualizations can increase students understanding (Bergin et al., 1996). Interactivity enables feedback on algorithm correctness by enabling the use of the same "design/edit/run/debug cycle" (Eisenstadt et al., 1993) that is used by software developers. Interactivity and hands-on experience can also help alleviate the fears caused by the heavy theoretical aspects of CS (Bergin et al., 1996 and Ben-Ari, 1998). This enables the instructor to spend more time with weaker students while the more advanced students can explore the problem further. Abernethy, Gabbert, and Treu (1998) present some experiences obtained by using *inquiry-based learning* in which students take an active role of their learning process. The benefits of using interactive visualization tools in laboratory sessions has also been presented by Meisalo, Rautama, Sutinen, and Tarhio (1997).

Many examples exist of researchers expressing at least an intuitive thought that software visualizations would have a beneficial effect on learning (e.g. Eisenstadt et al., 1993, Lawrence, Badre, & Stasko, 1994, and Byrne, Catrambone, & Stasko, 1996). It is after all quite natural to suspect that a symbolic representation of abstract and non-tangible events with additional dynamic presentation that a computerized software visualization gives would have a beneficial effect on learning. In the following section we will take a look at the results and new questions that research on the topic has uncovered.

2.1.2 Visualizations and learning

The results that have been obtained on the effect of visualizations on learning are not especially encouraging. As we have already noted, many researchers and educators share the feeling that visualizations should have a positive effect on learning. Some even go so far to point out that "all students learn certain concepts

better by thinking visually about them" (Bergin et al., 1996). While visualizations often have some positive effect on the learning outcome (see e.g. the results sited in Kehoe et al., 1999 on visualization research in other domains), it is still unclear whether or not visualizations really help in learning CS concepts.

The research led by Stasko in the early 1990s is the first attempt at real evaluation of the benefits of visualizations in learning. Their results showed that the differences between an animation and non-animation group were not significant in post-tests (Byrne et al., 1996). However earlier research had shown some significant benefits between the two types of groups (Lawrence et al., 1994). The main difference between the two studies is that in the first the animation group was able to interact with the animation whereas in the second study the animation group watched pre-prepared animations. The value of interaction has also been confirmed in many other sources (e.g. Bergin et al., 1996 and Kehoe et al., 1999).

While viewing animations did not generally improve performance in a post-test, Byrne et al. (1996) noted that the animation group appeared to understand the algorithms better and were more able to use what they had learned in novel situations. Having students use the animation environments in laboratory sessions and measuring their learning based on that has also shown fairly positive benefits of using visualizations (Kehoe et al., 1999). It is important to note that in most cases students have felt that they have understood the algorithm much better once they have used visualizations, especially if they have also had the possibility to interact with the visualization and apply it to their own data sets (Stasko, 1997).

Although we can conclude that it is beneficial to use algorithm animations in learning situations we still need to figure out why students feel that they understand better once they have used visualizations. Earlier we have cited sources which discuss the relationship between visual and verbal representations and how connections can be made between them. It seems clear that enabling visual thinking does help in learning CS concepts as well as concepts in other domains (Bergin et al., 1996 and Kehoe et al., 1999).

One solution to the question of why visualizations — especially when the possibility to interact with the visualization exists — help learning is that they increase student motivation (Lattu et al., 2000). Increasing student motivation is after all an important factor although it often cannot be measured quantitatively (Kehoe et al., 1999). Interacting with the visualizations has also often caused students to spend more time with the algorithm, which also has positive benefits when considering learning (Stasko, 1997). While the results obtained so far have not been especially encouraging, most institutions that have begun using visualizations have continued their use (see e.g. Markkanen, Saariluoma, Sutinen, & Tarhio, 1998 and Kehoe et al., 1999).

2.2 Animating Algorithms and Data Visualizations

We will now take a brief look at some systems developed mainly for algorithm animations. Some of the experiences of the systems use comes from my own experience in using or instructing others in the use of the systems covered. We will mainly concentrate on systems that aim at understanding the algorithm as a recipe or the relationship of the algorithm to the code that implements it. These two types fulfill the levels I and II in the algorithm understanding scale presented in Anderson and Naps (2001). We will also classify the systems according to the instructional design scale presented in the same paper.

2.2.1 Jeliot and Jeliot 2000

Jeliot is an animation environment that has been developed by the Computer Science Department of the University of Helsinki (Haajanen et al., 1997). It takes a piece of code written in a limited subset of Java and then verifies the code and adds drawing primitives before running the code in an applet window. As such the system requirements for using Jeliot are fairly small, especially considering modern environments.

Jeliot's strength lies in the fact that it enables the users to write their own code and control which aspects of the animation are important by choosing the representation of each variable used. Unfortunately, the use of Java instead of an abstract pseudo-code limits the use of Jeliot in teaching situations to environments in which Java is used as the teaching language. A laboratory session in which several M.Sc.-level CS students where asked to modify the visualization of the provided Binary Search -algorithm to show an array pointing to the border values of the algorithm took much longer than expected since the students where unaccustomed to Java.

Jeliot's user interface, while lacking in eye-candy, is quite functional and enables many different representations of the variable values. A later research project was conducted in which Jeliot was modified to be simpler and more user friendly (Ben-Bassat Levy et al., 2001). The amount of eye-candy was increased but at the same time some of the versatility of Jeliot suffered. The screenshot in figure 1 shows the animation stage of the traditional Jeliot.

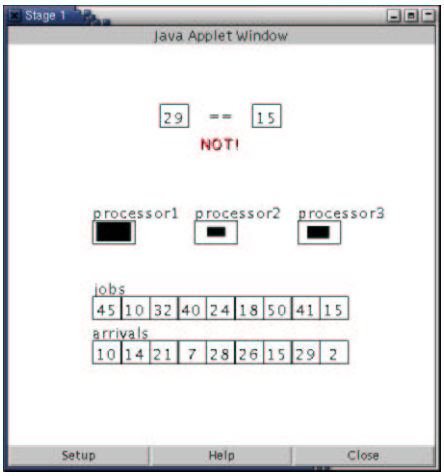

Figure 1: A view of processor loads while scheduling tasks

In actual teaching situations in which Jeliot or Jeliot 2000 have been used, significant improvements in learning have not been found — as is the case in other similar research endeavors. However, the students using Jeliot 2000 exhibited an enhanced vocabulary when describing the concepts that they had learned with the

assistance of computerized visualizations (Ben-Bassat Levy et al., 2001). Placing Jeliot on the Instructional Design Scale presented in (Anderson & Naps, 2001) proves difficult. While Jeliot provides a high level of interaction to the user by forcing the user to write the code that will be executed, no further action is required by the student and no steps are taken to ensure that the student understands the code being executed. As a system for debugging code it can serve quite well providing that the student knows what the algorithm's visualization should look like based on a paper-and-pencil implementation of the pseudo-code's execution with the same data set. Jeliot's functioning is also hampered by the fact that backtracking is not possible in any sensible fashion making it hard for students to return to a certain point in the execution of the code.

2.2.2 The GNU Data Display Debugger

The GNU Data Display Debugger (DDD) is primarily a graphical user interface for various debuggers such as the GNU Debugger (gdb). In this sense it does not differ very much from Jeliot which can also be seen as a tool for debugging code instead of an instructional visualization environment. However, DDD provides all of the power of many powerful debuggers in a fairly easily understandable graphical interface. The graphical interface means that DDD can visualize the data that a program manipulates, which leads to automatic program visualizations (Zeller, 2001).

While especially useful for students as a debugging tool for their own programming projects it can also be used by instructors as it provides unlimited undo/redo functionality and a visual representation of an algorithm without any extra work. Figure 2 shows the visualization of a simple list in DDD.

When evaluating DDD with the scales presented in (Anderson & Naps, 2001) we can easily place it in level II of the algorithm understanding scale as it provides a ready link between the algorithm and its implementation in code. A well developed library that enables the writing of an algorithm in nearly pseudo-code will

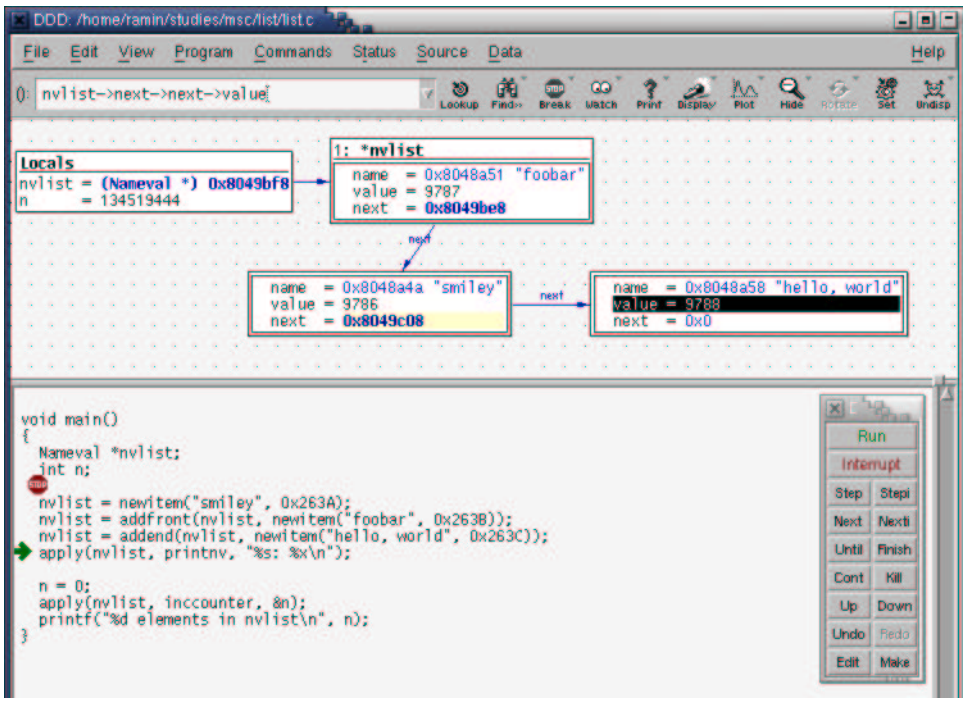

Figure 2: A list as shown in DDD

escpecially help in formulating the mapping between an algorithm in its pseudo-code representation and its implementation in actual code.

On the instructional design scale DDD can also be placed in several different levels. The first level is not fulfilled as the user must interact with the system in order to produce a visualization. The second and fourth levels are also unattainable as it does not provide any hooks for presenting additional textual material. The only material DDD uses is the program code and its possible inputs. Levels three, five, and seven are easily fulfilled as the student is often required to write the code that will be visualized, without the need to write additional visualization hooks. The student must also provide the input data for the algorithm in one way or another and has to choose which elements s/he wants to view in the visualization. The fulfillment of the eighth level depends on the course materials. In itself DDD does not provide any correspondence to any given course materials.

2.2.3 Introduction to Algorithms

Cormen, Leiserson, and Rivest (1989) is a classic of data structures and algorithms education. Its later editions come with a CD that has some visualizations of the data structures and algorithms presented in the book. While the visualizations might have been revolutionary in the early 1990s their graphic look and use of GUI components leave much to be desired. The fact that they only work in MacOS also severely limits their use in modern educational settings.

However, the CD is not wholly without merit as it is closely linked with the book itself and therefore any course that primarily relies on the book fulfills level eight of the instructional design scale presented in (Anderson & Naps, 2001). On the algorithm understanding scale it only reaches level one as all of the examples are presented in pseudo-code or formal English. Unfortunately the fulfillment of the other levels of the instructional design scale is severely lacking as the user has almost no control of the algorithm executions and, without the book, explanatory text is often unavailable. While the CD is unsuitable for use on its own, it would be interesting to know if additional value can be found of its use when using the book as primary study material.

2.3 Some notes on application design

Systems for software visualizations come in many different forms, even though we only looked at a few different applications in the examples presented above. Although different programs are developed for use in different kinds of environments some guidelines for the design for software visualization applications can be found. While many of the guidelines also refer to other application domains as well, we will only cover issues that need to be kept in mind when designing software visualization applications.

Khuri's (2001) paper gives a good introduction to issues that need to be consid-

ered when creating an application for program visualization. The first step of the design process should be an analysis of the environment in which the application will be used. The needs that the application should address differ, based on the intended use. An application that is used to demonstrate an algorithm's (or program's) function in a guided laboratory session or lecture can depend on the instructor giving the explanation, whereas an application used in independent work; e.g. assignments, self-study, or distance learning; needs lots of explanatory text to insure understanding. The designers and implementors of a visualization environment must, however, remember that a given visualization will never be beneficial to all users (Khuri, 2001).

The issues that need consideration when designing a program visualization application are the same as in any other software project. Naturally the problem domain needs to be considered and studied. Luckily in most cases the designers include someone with experience both in software development and CSE. In addition to knowledge of the problem domain and an analysis of the itches that the application should scratch the designers need to consider how much interactivity and guidance the user will be given, how the user interface will be designed, and how sound will be used (Khuri, 2001).

All of the research published today on the use of program visualizations in learning environments addresses the importance of interactivity in the program visualizations (see e.g. Khuri, 2001, Kehoe et al., 1999, and Anderson & Naps, 2001). Users want interaction (Kehoe et al., 1999) and letting the users provide their own data sets for the application is often required for understanding the algorithm (Anderson & Naps, 2001). Interaction by providing supplementary text or questions at various points of an algorithm's (or program's) execution is also beneficial (Anderson & Naps, 2001). Free exploration of an algorithm's function with the provided interactivity (such as backtracking, providing one's own data sets etc.) increases user motivation and as such helps learning. A good list of all of the issues that need considering when implementing interaction is presented in (Borsook & Higginbotham-Wheat, 1991).

What is often left without consideration in program visualization applications is the benefit of the “design/edit/run/debug-cycle” (Eisenstadt et al., 1993) that programmers use continuously. In most applications the cycle cannot be implemented since algorithms come in a predefined form. However, many applications, e.g. Jeliot, let the user freely enter program code and visualize its execution. In these applications supporting and guiding the cycle is beneficial and can help prevent the fact that most visualizations present an expert’s view and do not relate to a student’s model (Kehoe et al., 1999 and Ben-Bassat Levy et al., 2001).

Designing the user interface in a way that it is easily understood and provides all of the necessary functionality is an extremely difficult task that has merited its own field of research. We will attempt to cover some of the key points that need to be considered in creating a program visualization application. The layout of the display needs to be designed so that it is divided into areas with a clear functional purpose and the screen does not become unnecessarily cluttered (Khuri, 2001). User interaction by enabling the user to choose what they feel is important for their use helps in providing an uncluttered display. It is also important to only show data that is relevant for the understanding of the algorithm in a scale that is useful (Bergin et al., 1996). The use of colors is important as they help in remembering what is shown (Khuri, 2001). When using colors, restraint should be used, as too many colors or bad color combinations will lessen user experience (Nielsen, 1999).

3 Agents

Intelligent Agents or agent-based computing is an offshoot of artificial intelligence research which some say provides a new theoretical model of computation as opposed to Turing Machines (Wegner, 1997). In many papers agent-oriented software development or agents have been touted as "the new revolution in software" (Guilfoyle & Warner, 1994) or as being "pervasive in every market by the year 2000" (Janca, 1995). Unfortunately the hype that has surrounded agents has been quite extensive and some feel that a backlash will be experienced (Jennings, Sycara, & Woolridge, 1998).

In the following we will 1) discuss the various definitions that have been provided for what an agent is, 2) cover some of the theories that govern agents, and 3) take a look at different types of classifications that have been provided. Once we have covered what an agent is, which is no simple task in itself considering the amount of disagreement that exists in the definition of agenthood, we will concentrate on paradigms and tools that will be useful when designing and implementing agents. After a brief glimpse of what kinds of uses agents have been used for, we will present some arguments for why and how agents should be used in CAI.

3.1 What are Agents?

As we have already noted, researchers do not agree on exactly what constitutes an agent (Jennings, 2000). Especially when we add the word "intelligent" to the concept, thus creating the concept of an intelligent agent. Most opponents of artificial intelligence will stop listening and others will hope for a silver bullet that will enable computers to do all the thinking, enabling humans to get on with the fun stuff. After a blatant characterization we will look at what researchers have obtained during the last ten years or so of research in the field.

3.1.1 Definitions

Before we can discuss agents any further, we must come to some sort of agreement of what we mean by an agent. Although the theoretical background of agents has its own value in defining agenthood, it will be presented in the following section. A good starting point for defining what an agent is can be by discussing two notions of agenthood. The weak notion holds that agents should be *autonomous*, *reactive*, *proactive*, and have *social ability* (Woolridge & Jennings, 1995).

By 'autonomous' we mean that the agent has its own thread of control In traditional operation systems this means that the agent has its own process. In modern concurrent talk we talk about threads which can be either native (operating system) processes or internal processes (which are given execution time by the sub environment inside a single native process). 'Proactivity' and 'reactivity' further define autonomy by requiring that the agent can react to changes in its environment (reactivity), and it can actively pursue a goal by running when needed without requiring explicit user input. Reactivity requires that the agent be *situated* in an environment that it can sense. For example, if an agent is responsible for monitoring an industrial manufacturing process, it should have a sensor that can sense the state of the process, such as temperature, chemical balances etc. Finally the requirement for social ability means that the agent has the ability to interact with other agents or the user.

Although most researchers will agree upon the weak notion of agenthood (Woolridge & Jennings, 1995), it is too weak to create anything new. For example, Unix variants come with a variety of helper applications such as `xbiff`, which notifies users of new email. The `xbiff` helper application fills the weak notion requirements: 1) it is autonomous since it runs in its own process and is given processing time by the operating system; 2) it reacts to changes in its environment by monitoring the user's incoming email file and reacting when the file is changed, i.e. new email has arrived; 3) it is proactive by actively informing the user of the changes when changes have occurred; and finally 4) it has social ability because

it is able to inform the user when new email has arrived.

Since the weak notion is not good enough a stronger notion needs to be devised so that agents can differ from widely accepted software paradigms. Woolridge and Jennings (1995) define the stronger notion to include mentalistic notions (knowledge, belief, intention, and obligation) plus emotions. The stronger notion brings more aspects that can be perceived as intelligence. No one can be fooled into thinking of `xbiff` as intelligent. Intelligence is also a requirement of agenthood (Jennings & Woolridge, 1998). Bates (1994) also includes the requirement of beliefs and intentions in order to obtain believable user agents.

The *Belief-Desire-Intention*-model is probably the best-known model of practical reasoning (Jennings et al., 1998). An agents that conforms to the BDI model has beliefs that correspond to the information it has of its environment (knowledge), desires that reflect the options that it has available to it, and intentions that reflect the possible states that the agent wishes to commit to. Once agents have committed themselves to fulfilling an intention, they will keep trying to fulfill it until it is completed or proved impossible to complete (Woolridge, 1997). Singh (1998) argues that mental agency is a mistake because it limits design autonomy and the agent is limited by how the original design is documented. According to his point of view how agents behave socially is important, because behaving by its societal standards (e.g. keeping promises), it complies to its communication.

Other researchers have defined agents in broader terms. For example Sycara, Decker, Pannu, Williamson, and Zeng (1996) define intelligent software agents as programs that act on behalf of humans by locating and accessing information, resolving inconsistencies, filtering irrelevant or unwanted information, integrating information from several sources, and adapting to a user's needs. Their definition is heavily biased toward what they call *information agents* (for a discussion on their classification of agent types see 3.1.3) which are specifically given the task of helping a user cope with the ever increasing amount of information that is available.

While the previous definitions have concentrated on defining agents based on their behavior, Genesereth and Ketchpel (1994) requires only that a software agent uses an *Agent Communication Language* (ACL — for a definition of what they are, see 3.2.2) when communicating with other agents. They do not require that an agent would be autonomous (i.e. have their own thread of control), situated or flexible. The requirement that agents are agents when they use an ACL means that single, stand-alone agents cannot be considered as such. While ACLs are often needed in *multi-agent systems* (MAS) stand-alone agents may be created and thus Geneserath & Ketchpel's (1994) definition is incomplete.

We will now return to the definition outlined by Jennings and Woolridge. In a later article they define agents as being situated, autonomous and flexible (Jennings et al., 1998). Situatedness and autonomy we have already outlined above. Flexibility requires that the agent be responsive, proactive, and social. In this case the definition has moved slightly away from mental agency toward more reflection of how the agent behaves socially as Singh (1998) argued. When agents are 'responsive' they respond to requests from other agents or users in a timely and sensible (i.e. the reply has some connection to the request) fashion. This also means that an agent needs to have some way of communicating with its social environment, be it another user or an agent. For an agent to be 'social' it therefore needs to understand and produce agent communication, or have a user interface through which it can communicate with a user. 'Proactivity' means that the agent will actively promote the completion of its goals (intentions) by taking the necessary steps. Jennings (2000) further emphasized responsiveness, proactivity, and sociability. By creating agents that fill this definition we can obtain applications that behave rationally in their own environment, but the applications will not pass the Turing test.

The requirements that agents must be strongly social implies societies of agents or multi-agent systems. As we have already noted, an agent-based system may use only a single agent, but most of the potential and research interest lies in the field of multi-agent systems (Woolridge, 1997). MAS is a part of a *distributed artifi-*

cial intelligence (DAI), which also includes *distributed problem solving* (DPS). While DPS is closely related to parallel computing with often centralized data and global control of the computation, in MAS there is no central control. Data is decentralized, computation is asynchronous, and each agent has a limited viewpoint (Jennings et al., 1998). We can now conclude that multi-agent systems are loose and dynamic groupings of agents which use each other's abilities as needed to fulfill their own goals.

3.1.2 Theories and paradigms

Agents are the fruit of the interaction between artificial intelligence, object-orientation, concurrent object-based systems, and human computer interaction (Jennings et al., 1998). While they have all had an impact on agenthood, AI is still the primary basis for agent research. That is why in the beginning of agent research the term *intelligent agent* (e.g. Sycara et al., 1996) was widely used and still is (Woolridge, 1999).

Even though AI aims at building intelligent artifacts that sense and act in an environment (Russell & Norvig, 1995), which is what agents do as well, study focused on agents did not begin in earnest until the 1980s. AI research before this — and throughout the 1980s — concentrated on the various components of what can be seen as intelligent behavior, even if no definition of has been accepted (Jennings et al., 1998). The research activity closest to agents during the 1970s was AI planning (Jennings et al., 1998), also known as *symbolic AI*.

Symbolic AI is based on a symbolic model of the world (Woolridge & Jennings, 1995). In most cases the world is shrunk to only cover a model of the agent's environment, typically in first-order predicate logic (Jennings et al., 1998). If we think of a symbolic AI system that would hold a viable model of the whole physical world, we would be faced with a theorem prover (Woolridge & Jennings, 1995). This and other issues have led to the realization that symbolic AI cannot scale up to real-world problems. Many researches have begun to question the

viability in symbolic reasoning and AI in general. Rodney A. Brooks is the best-known of these critics (Jennings et al., 1998). Brooks argues that intelligence *emerges* from the *interactions* that take place in the world (Brooks, 1991). Thus we can see that intelligence is in the eye of the beholder (Woolridge & Jennings, 1995) and therefore the Turing test can be completed.

Because symbolic AI has failed, other theories have been proposed and developed (Woolridge & Jennings, 1995). Brooks himself has developed *subsumption architecture* which employs no symbolic representations or reasoning at all. If an agent is developed using it, it will have a number of behaviors that are of the form "situation $\rightarrow$ action". Although subsumption architecture has been used successfully, some problems still remain. First of all, since the agents do not have an internal model of their environment, they must have an adequate amount of information available to them of the local environment. However, if the agent uses only local information to support its decision making, it cannot take into account non-local information and only reacts to short-term changes. This pure reactivity will not enable learning nor allow improvement of the agent's action as time passes. In addition, as the intelligence of the agent is emerges from its behavior, it is difficult to create an agent (or a collection of agents) to fulfill a specific task. (Jennings et al., 1998)

As neither purely deliberative nor reactive architectures work well in all situations, researchers have begun concentrating on *hybrid* architectures, which combine aspects of both (Jennings et al., 1998). These hybrid systems are created with several layers of functionality in which the first level to receive the perceptual input is purely reactive and often resembles Brooks' subsumption architecture. The next level deals with the knowledge level view of the agent's environment and often makes use of symbolic representations and finally the uppermost level concerns the social level. The social level is concerned with the knowledge of other agents and their goals, beliefs etc. The first two levels can be described as using often well-known methods for reacting to the sensory input and using (traditional) AI only in the knowledge representation (Woolridge & Jennings, 1995). We have

previously talked about the BDI model which is often used when talking about hybrid systems. The beliefs are represented by what the agent knows (or believes) about its environment. Desires reflect the options that are available to the agent based on its beliefs, and finally, intentions represent the goals that the agent is committed to fulfilling (Jennings et al., 1998).

Although agents have their basis in AI, they are also strongly tied to other disciplines as well. As we previously noted, object orientation, concurrent object-based systems, and human computer interaction research have also led to the adoption of agent technologies. One of the main reasons for this is that "intelligent agents are ninety-nine percent computer science and one percent AI" (Etzioni, 1996). So software engineering is very much a part of agents, a fact that most AI researchers have overlooked (Woolridge, 1997).

Object Oriented Programming (OOP) or *Object Orientation* (OO) is a software engineering paradigm that views the world as a collection of objects, each having its own attributes and function. Objects are used to encapsulate the implementation of any given entity so that a programmer using an object does not need to know how an object function or the behavior that it exhibits is implemented. The only knowledge that programmer needs is to know what methods are available to change an object's state and therefore its behavior. (FOLDOC - Computing Dictionary, 2001)

OO programmers and researchers often fail to see anything new in an agent-based approach as both agents and objects encapsulate some state, perform actions on the state (or based upon the state), and pass messages to each other (Jennings et al., 1998). However one major difference can be seen. While an object must do everything that a message passed to it calls for, i.e. once a method is invoked the object must do what the method calls for, agents can control their behavior and decline to respond (Jennings, 2000). Thus an agent can decide when to act, which is why agents do not invoke each other's actions but request that an action be performed. To summarize (by modifying the slogan in Jennings et al., 1998), we can say that objects are like any working dog breed which will do things for

free; whereas agents embody the terrier spirit of “show me the money.” While object orientation is not suitable for agent-based software engineering in isolation, OO can nevertheless be used to develop agents, especially as most modern OO languages, such as Java, enable concurrency which can give each agent its own thread of control. This may lead to the thought that agents are active objects, but one conceptual difference remains; agents are “rational decision making system[s]” (Woolridge, 1997).

Human-Computer Interaction (HCI) is the field of research which is concerned with how people interact with computers and how such interactions can be made as facile and intuitive as possible. It has affected agents in that despite most interfaces being directly manipulated by the user more and more, we can also see programs which take the initiative (Jennings et al., 1998) . Direct manipulation means that the computer program will not do anything until the user has explicitly told it to do something. Unix has had a long history of making various little tools available that can be activated automatically to perform certain tasks (such as email filtering based on user provided rules to move messages automatically to different folders etc.). Many more “intelligent” tools are being provided. Agents used in user interfaces can be criticized for being glorified user interfaces (Singh, 1998), but the fact that “expert assistants” or “digital butlers” (Jennings et al., 1998) are made available to users make their lives much easier. Nicholas Negroponte points out that it is not an issue of IQ, but it is shared knowledge working to-wards the user’s best interests by either the user or agents themselves sending out agents to collect information on the user’s behalf (Negroponte, 1995).

Although evaluating and comparing differing agent architectures and their effectiveness in a certain environment is difficult, steps toward this have been taken. As research on different architectures is undertaken, attention must also be given to the development of programming languages and development environments that would hold a higher level of abstraction in order to be more useful for an agent programmer. (Jennings et al., 1998)

3.1.3 Classifications

The fact that we have agents that work in varying circumstances has led to researchers wanting to classify different types of agents based on the environment in which they function and their function in itself. Some of the original classifications used terms such as gophers or service providers. Later on more formal classifications appeared. One fairly descriptive classification has to do with the function of an agent. Decker, Pannu, Sycara, and Williamson (1997) present three types of agents: task, interface, and information.

Task agents are given a task that they must try to fulfill, as their name implies. The task should be clearly defined and often multiple agents are given a subtask of a task for completion as required. Task agents may have been given their task by an interface agent that interacts with other systems, often other agents or users. For the task agent to be able to fulfill its task it may need to utilize some information that is stored in an information agent or that such an agent will "research." Based on this classification a user should never interact directly with agents other than interface agents. Often, however, practicality takes precedence and the user will interact directly with both task and information agents as well.

Other sources have presented broad categories such as gophers, service performers and just predictive or proactive agents. Gophers can be thought of as information agents as they are concerned with the storage and retrieval of knowledge. Service performers are just another name for task agents. Bates (1994) calls interface agents as user agents, i.e. agents that act on behalf of the user and interact with the user. Predictive or proactive agents are a combination of information, task, and interface agents (or we can consider a MAS of such agents as a predictive agent) that are combined together e.g. for the purpose of monitoring news services for articles that the user might be interested in (Jennings & Woolridge, 1998).

3.2 Designing and Implementing Agents

Now that we have a basic knowledge of what an agent is, we can address the issues related to the design and implementation of a real live (digital) agent. Naturally, as the field of agent engineering is still relatively new, no firm or even well-tried guidelines exist, but some pitfalls have at least been found.

3.2.1 Agent Engineering

Creating complex software systems is a complicated task. In order to help software developers in the task many different paradigms have been designed to give developers tools that they can use when designing software systems. Whereas object orientation has often been considered as a solution to many of the problems, the rise of component based engineering and design patterns shows that object orientation alone does not solve many of the issues that accompany complex systems.

An agent oriented approach to software engineering differs from other engineering methodologies (e.g. procedural programming, object orientation, and design patterns) by considering the system as a collection of agents. Each of the agents has its own task that it tries to fulfill through communication with other agents (Jennings & Woolridge, 2000). Although such a hierarchical collection of interlinked components has been an established practice in software engineering for quite a while, agents enable a new level of dynamics in the dynamic organization at runtime. This leads to a system that can better cope with new unexpected situations as the developers need not predict all of the possible situations that might arise.

While agent development is still a relatively new field and no established guidelines exist, some attempts at creating such guidelines have been made. Jennings and Woolridge (2000) is a fairly recent attempt. As in all software systems, the design must begin from specifying the basic environment in which the software

system must work. In the case of agents we need to cover the beliefs that the agents have, the goals that they are trying to achieve, the actions that they have available in order to fulfill the goals, and the interactions that take place between agents over time.

Once the specification has been created it needs to be implemented. The transformation from specification can be made in two ways, but both require that the specification be created in some logical language. The first available option is to directly execute the agent specification in some environment that can execute a specification. The other available option in this case is to translate or compile the specification into a concrete computational form using automated tools. The main problem with this approach is that the agent will not be able to learn, or modify its behavior, at runtime. Since few attempts have been made to specify agents using logic (Jennings & Woolridge, 2000), we need to also consider implementing agents by using traditional engineering paradigms in the background (Jennings, 2000).

So that the development of agents need not always fall into the same pitfalls, attempts have been made to list some of the pitfalls that designers should be aware of when creating new agents. The most comprehensive list can be found in Woolridge and Jennings (1998), which we will summarize here.

Naturally, as with any new technology, the problem of over zealous marketing of the new technology exists. Agents are often sold as ultimate problem solvers for a given problem, a legacy that comes from the over optimism of AI. With the zeal also comes the enthusiasm to use the new technology everywhere, even in places in which other technologies are better suited. The same applies to agents, especially as there is not enough experience that would show in which environments agents are really useful.

Often the zeal that comes with a new technology results in statements that are difficult to understand and which cause misunderstanding of what it is good for or why it should be used. If the agents really are needed, there still exists the prob-

lem that too generic a solution is attempted when a custom solution to a particular problem would be more beneficial. Since agent systems are typically extremely complex, the created initial prototypes and industrial strength solutions are separated by a large chasm that needs to bridged.

If we consider the design and implementation phase many of the pitfalls have to do with forgetting the existence of other related technologies or other projects which have utilized agent based engineering and may have created libraries or components that might be usable in the system being designed. The designs may vary from overly intelligent (and industrially weak) agents to agents with no intelligence at all. Using ready implementations gives an added level of strength including the fact that the standards used in other systems have typically been tested and evaluated, and creating a new standard rarely offers any added value. Typically the agent system is initially created so that all of the agents, of which there may be too many or too few, run on same machine in simulated concurrency. This is good for testing a concept, but in industrial strength applications the agents also need to be tested in various networked environments which often cause new concerns. Naturally a system of multiple agents also, in most cases, needs some sort of structure so that some purpose may be obtained from the society of agents.

In addition to all of the agent specific pitfalls we need to remember that agents are also typically concurrent, distributed systems and all of the complexities related to their development also exist (Jennings et al., 1998). For example, in MAS we need to address issues such as network congestion or failure, the loss of any number of agents at any given time etc. Since these issues are not limited to agent based engineering we will not discuss them here.

3.2.2 Agent Communication

Previously we saw that one of the definitions of agenthood concluded that an agent is any piece of software that uses an agent communication language (Genesereth & Ketchpel, 1994). Finin, Labrou, and Mayfield (1994) also emphasize the

need for agent communication using a common language that all agents understand. Several attempts of creating a common language for agent communication have been undertaken with the Knowledge Query and Manipulation Language (KQML) as one example. While traditional communication between applications concentrates on communicating information, agent communication languages (at least in KQML) also communicate attitudes on the content and performatives that define the actions that agents may attempt when communicating with each other.

It is important to note that an ACL is not directly interested in the transport or higher level protocols that are used to communicate the language, although implementors of ACLs do have a strong interest in them. Some of the requirements that can be placed on ACLs are that the language needs to be defined in a way that it is easily read by humans. Since agents are often designed to work in highly heterogeneous environments, the languages they use in intercommunication need to be layered in a way fits well with other systems and is usable over a number of different transport protocols (such as HTTP, TCP/IP, etc.). Naturally considering reliability, as well as efficiency in bandwidth usage and processing time, is important when designing a language that will be implemented in a real environment. (Finin et al., 1994)

While many different languages have been created to allow intercommunication between different applications as well as agents, no ACLs have been created that have allowed agents to effectively understand each other (Jennings et al., 1998). The late 90s also saw the rise of XML (XML, 2001) which was created to enable data interaction between various systems that exist especially on the Internet. Several different technologies for sharing information across sites are also in use, e.g. Resource Definition Format that is used to "integrate[s] a variety of applications from library catalogs and world-wide directories to syndication and aggregation of news, software, and content to personal collections of music, photos, and events" (RDF, 2001).

If ACLs are to gain widespread acceptance they need to be implementable in highly performance critical environments. In such environments a generic lan-

guage is not always suitable. Implementing custom agent communication languages for specific agent environments also leads to increased security as generic protocols are often prone to misuse. Although the ability of agents to decide which requests to honor does help in reducing security risks, unexpected holes may open as well.

As agents have often been prototypically created for use as personal assistants that help in scheduling and such, it will be interesting to see whether the SyncML initiative (SyncML, 2001) will gain support from agent researches. As SyncML is a XML-based language for mainly calendar and address book synchronization, it may prove to be the starting point of the widespread acceptance of XML as the basis for ACLs. Development has also begun on a XML-based agent markup (communication) language at DARPA called DARPA Agent Markup Language (DAML, 2001) which extends the functionality of RDF to provide even more possibilities for semantic modeling.

3.2.3 Agent Applications

Many working prototypes and actual implementations of agents have been created during recent years. While agents have been applied in many industrial systems (Jennings et al., 1998), we have yet to see agents that would really enhance user experience. Although many personal assistants, such as Microsoft's "Clippy" in their Office suite, have been implemented, they are rarely used to enhance productivity. Despite this, information agents show great promise in helping users in their everyday tasks.

Klusch (2001) presents a good overview of information agent applications. Information agents are created to proactively and reactively gather information, synthesize and filter the data and present it to the user. Especially with the modern chaos of the Internet and the inability of even the best search engines to keep up with all of the changes, new systems are needed by users to provide reliable collections of information on any given topic. The intelligence required by such agents

can be significantly reduced by some of the methods of communicating semantic information on resource content that we discussed in 3.2.2.

Various task agents have been created that even handle mission critical applications, such as systems that control manufacturing processes. Other examples include air traffic control in which each airplane is represented by an agent and air traffic control is represented by other agents which try to accommodate the airplane agent's desires. (Jennings et al., 1998)

Many prototypical task agents have been created that try to automate the process of scheduling events or meetings between several different participants (Zeng & Sycara, 1995 and Kautz, Selman, & Coen, 1994). Especially with such systems, the importance of the user interface is important. An initial version developed by Kautz et al. (1994) used email forms to communicate with the users while a later version used its own GUI. Naturally users where more willing to interact with the GUI instead of an error prone email form.

Interface agents can often be used in conjunction with information agents to promote an interface agent's ability to learn by monitoring a user's actions. Such monitoring could, for example, note that a user often moves email messages to or from a certain address to a certain folder and ask the user whether they are willing to create a filter based on that action. A system that implements this sort of behavior is presented in (Maes, 1994). Many other systems have also been developed, but widespread acceptance is still lacking. Maybe the future will bring agent-based applications on to every user's desktop.

3.2.4 And the Future?

As a relatively new paradigm in software engineering, agent based engineering has its own share of challenges. Unlike new programming languages that are sponsored by companies and published with ready development environments such as Java and C#, agent engineering does not have any development environments or

programming languages that were specifically designed (Jennings et al., 1998). In order to create agent-based applications developers need to shoehorn existing programming languages and environments into a new paradigm.

Although agent based engineering is mostly about traditional software engineering, using tools developed for another paradigm causes a significant loss of productivity and effectiveness. When discussing design patterns, it is often noted that design patterns are recurring patterns that have been modeled and named. These patterns often become features of future programming languages or environments such as is the case with the Observable-pattern and its implementation in standard Java libraries. What agent based engineering needs is its own modeled and named patterns and toolkits that help developers in their designs. Naturally old designs are not made obsolete by the new as agent based engineering still faces all of the challenges that distributed and concurrent systems with flexible interactions are presented with. What we are reaching for is a new level of complexity that brings along it more power to solve even more complex real-world problems.

With agents becoming more and more pervasive at every level of computing, user trust is an issue that must be addressed (Norman, 1994). For a user to trust an agent, the users must retain the feeling of complete control (Kautz et al., 1994). Unfortunately, when we consider the applications available on a typical user's desktop, how many of the applications enable the user to retain the feeling of complete control. Currently most of the "agents" that we see on ordinary user's desktop do everything but give the user the feeling of control.

Recent developments concerning the privacy of a user when each site collects their own demographics have heightened the need for trust between the user and their agents. For a user to give an agent free reign in filtering email messages, searching for interesting news etc., the user needs to know that the information that the agent contains will not leak to any other party, and that the agent is not limited to a given set of information sources. Open standards will enable developers to create information sources and agents that utilize them without tying users to any one provider. Most people perceive this as a "good thing(tm)" while dependence on

a single source is frowned upon. The agent community needs to promote open standards that have the support of industry as well if we are to see agents on every user's desktop.

3.3 Integrating Agents into Computer Aided Instruction

The earliest examples of CAI began to show around the 1960s, and the first CAI applications containing intelligence, typically called *Intelligent Tutoring Systems* (ITS), started to appear in the 70s and flourished in the 80s. During the 1990s, however, they began to disappear from mainstream CAI publications. Initial definitions of ITS argued that they must possess knowledge of the domain (expert model), learner (student model), and teaching strategies (tutor) (Shute & Psotka, 1995). Later definitions for what the "I" in ITS means, presented in (Shute & Psotka, 1995), show how many of the ITS researchers were stuck with an obsolete view of AI. This view emphasizes modeling the world and using AI techniques based on inferences (Schank, 1991) which the AI community had already found problematic by that time (Jennings et al., 1998).

The problems found in traditional ITS systems have lead to a shift in focus from traditional AI to agent-based systems. Some of the problems that are found in traditional ITS are presented in (Eisenstadt et al., 1993) and include the following. They argue that the main problem with ITS is that they deal with toy examples and cease to be useful once the learner has left the novice stage. The same argument is emphasized by Schank (1991) who notes that one of the issues to keep in mind when developing AI systems is to use real problems instead of toy problems. The amount of user control is also one of the great debates that troubled the ITS community in the 1990s (Shute & Psotka, 1995). The problem according to Eisenstadt et al. (1993) is that traditional systems tend to force students to follow a predestined path without permitting learning through discovery.

While the use of ITS has proved to be problematic there are strong reasons why

they should be used. The worry of Eisenstadt et al. (1993) that users view visualizations without necessarily understanding them or understanding them incorrectly is a valid one. Methodologies to supervise the independent learning process need to be implemented to prevent this from happening.

What this leads to is a new level of complexity in systems design. If we are to leave traditional expert systems behind we need to find new systems that enable the modification of the knowledge base that the system will use to ensure that the student understands the visualization while still enabling the freedom of discovery. Schank (1991) defines an intelligent system as one that learns. In this case we can consider the modification of a knowledge base that an agent has as simple learning. As agent-based engineering is suggested as the new paradigm for dealing with the ever increasing complexity in software systems (Jennings, 2000), we can see the transfer from the use of AI in learning to the use of highly interactive systems often characterized by agents (Aroyo & Kommers, 1999).

Initial uses of agents in CAI can be found as early as the mid-nineties. Even the initial uses of agents offer a radically different approach from traditional ITS, as agents are used to advise only when necessary without interruptions to user tasks (Selker, 1994). However during the nineties very little was done on the uses of agents in CAI. The next time uses are seen in any abundance was in Aroyo and Kommers (1999).

While the applications of agents as new ITS presented in (Aroyo & Kommers, 1999) provide some ideas on how to use agents in CAI we are still a long way from having any useful frameworks which can be used in implementing agents in CAI. For CAI applications that use agents to become truly useful and worth the effort that is put into developing them, we need to begin building tools that are specifically designed to enable the use of agents in CAI. We are still in this situation even though Schank (1991) defined the creation of tools to help in the engineering of AI applications as one of the issues that need to be addressed. However, the transition from traditional AI views to agent-based engineering accounts for the relative standstill.

The power of agents is best utilized in an open learning environment. In such an environment the computer is used as one tool among many for information seeking, visualization, communication, scheduling etc. Agents can be used in certain well-defined domains to help the student in whatever task is at hand, but agents need not know all of the elements of the learning environment. While old ITS tried to control the whole learning environment and tried to tie the student to following a certain path, non-intrusive agents will allow the student to take their own paths. In this scenario the agents will assist the student when needed and when the agent's skill set enables assistance. The days of theorem provers and inference engines are long gone (Woolridge & Jennings, 1995 and Schank, 1991), at least in light of current knowledge of what AI is capable of.

4 A Case Study with Seal

While I do not present a new framework to help in creating agent-based CAI applications, I will present a prototype, or proof of concept, of the use of agents in software visualizations. In addition to using agents as a part of its implementation, there are some additional features that differ from existing algorithm animation environments.

I have attempted to approach computerized visualizations from a slightly different perspective by creating a desktop on which the paper and pencil are mainly controlled by the student, but the computer can automate some of the repetitive tasks that form the core of most algorithms. With the student telling the computer what each step of the algorithm does we can achieve the same benefits as using paper and pencil. Once each line in a block has been executed by the student, the animation agent will take over and continue until a new line is encountered.

If, at any time in the execution, the student sees that the animation does not comply with his mental model of the algorithm's execution, he can stop and backtrack. This will enable the student to either change their mental model by noticing that their initial guidance was incorrect or to change the guidance given to the computer so that the animation will comply with the student's mental model.

Taking over the execution of loops and such, once the student has executed each line at least once, reduces the amount of repetetive work that the student needs to do and hopefully eliminates errors caused by repetition. When the amount of repetetive work is reduced, the student can concentrate on watching the animation of the algorithm being executed and hopefully notice any errors before the algorithm's execution ends.

Once the execution of the algorithm is complete, the instructor agent will enter the stage and begin pointing out to the student what went wrong and where. In the initial prototype the instructor's ability is limited to pointing out the lines of the algorithm in which the student's guidance differed from how the algorithm is

supposed to be executed. In order to highlight some key points in the execution of the algorithm, the ability of the instructor to mark a certain line of code and show additional instructor provided text is available, but only after the algorithm's execution is complete.

In the following we will dicuss some of the design principles that were taken into account while implementing Seal[1] in addition to pointing out some issues that have arisen during the implementation. We will also present a sample of how Seal can be used in an educational setting and provide a simple evaluation of Seal's pedagogical worth.

4.1 Design and Implementation

Seal's design is based upon the following requirements, discussed in more detail below.

- Seal is intended for use in CS courses as a student aid when working on assignments that require the tracing of an algorithms state at each step of its execution.
- Seal must utilize agent technology in its implementation.
- It must also enable student debugging of the algorithm's execution.
- The instructor must be as unobtrusive as possible.
- The user interface must be as simple as possible so that the user is not tempted to consider paper and pencil as an alternative.
- The user must have control over what data is shown in the animation, e.g. a loop variable is often redundant and may be distract the user.
- Seal must work on multiple platforms and varying screen sizes.

[1] Seal's homepage is at http://cs.joensuu.fi/ seal/.

These requirements were chosen because they provide a view of the new features that Seal provides in comparison to other algorithm animation systems. The scope of the technical requirements that are listed above covers most of the general technical requirements that can be placed upon any algorithm animation environment. It must be noted that the requirements have been listed in a very free-form way when compared to the detail that is needed in actual requirement specifications, but they should serve in this context.

4.1.1 Algorithm Animation

As already mentioned, Seal is intended to replace paper and pencil based algorithm walk-throughs used as assignments in order to help students understand an algorithm's execution. While no requirements have been placed upon the user groups that can benefit from its usage, some issues need considering.

Because graphical displays are not self-explanatory (Ben-Bassat Levy et al., 2001), Seal is not intended for use at the beginning of a student's studies in computer science. It relies heavily on common notation of algorithms (pseudo-code) and data (e.g. display of arrays). Until students are somewhat familiar with pseudo-code and algorithmic presentation, the code that Seal presents is not self-evident.

The reason for the pseudo-code representation of an algorithm is based on how the algorithm is represented internally. Seal reads its algorithms from XML files which must conform to the document type definition (DTD) presented in B. The XML representation of the algorithm is transformed into HTML by the use of XSL/T transformations (XSL, 2001). The generated HTML is then used to display the algorithm. An example algorithm's (see appendix C) HTML representation can be seen in figure 5 (in section 4.2).

Because previous algorithm animation environments have used program code or custom file types as their internal representation of algorithms, sharing the algo-

rithms from one environment to another requires additional work. So that the different types of animation environments, each with their own strengths, can better benefit from the work invested in the others, a common format for algorithm representation is needed. XML has been developed to solve many of the interoperability problems common in most computer systems. While we will not concentrate on the benefits of using it in comparison to other formats, it is important to understand that XML is a human readable format with a relatively low overhead. The DTD developed as a part of Seal and presented in appendix B is the first step in this process. The current version of the DTD has limited expressiveness, but can still be used to represent simpler algorithms. Further development of the DTD is needed.

The following figure (3) illustrates the steps and documents that are needed when transforming the algorithm into its HTML representation. In our example, the algorithm file is the `sum.xml`-file and the algorithm DTD is the DTD provided with Seal, shown in appendices C and B respectively. The XSL/T rules used in the transformation are not shown but can be obtained from Seal's homepage.

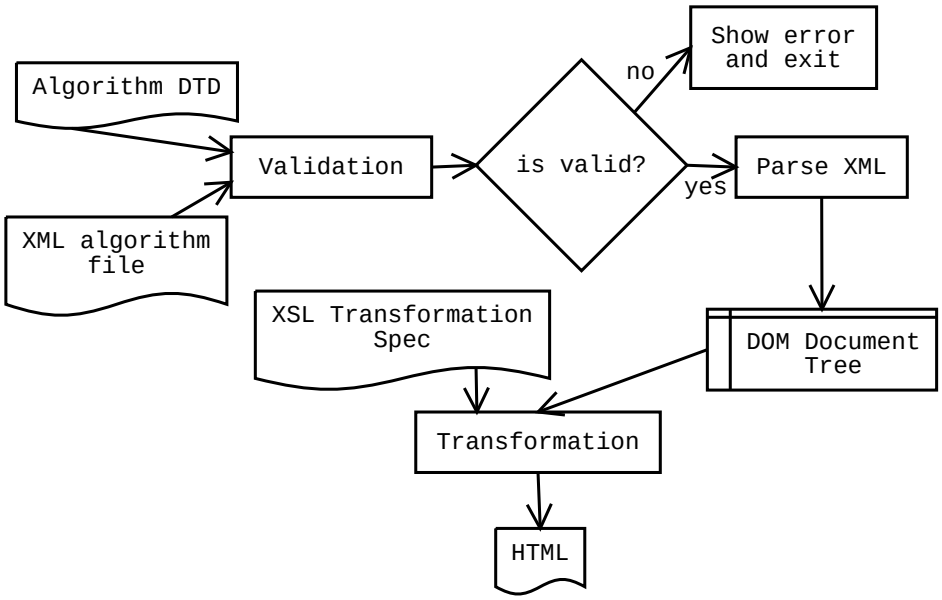

Figure 3: Diagram of the conversion process from XML to HTML.

The validation has not been implemented in the current prototype but will be implemented before instructors can be expected to create their own algorithms for use in Seal. The DOM (DOM, 2001) document tree is used by the various components to access the algorithm at the code level.

The reason for using a structured format to represent the algorithm and related data is that, in addition to easy transformation into pseudo-code, the instructor may use the internal representation to execute the algorithm with the given input data. The instructor can then use the execution stack it receives to compare with the execution stack of the user to locate the points in the execution where errors occurred.

The use of XML to represent an algorithm is a novel approach. At least searches on the world wide web for previous work in the field yielded no easily discernible results. While the definition at the moment is limited to defining a small subset of functionality that even pseudo-code algorithms generally include, it can be easily extended to fulfill most needs. Because XML is a human readable format, instructors and other interested parties can create their own algorithms for use in Seal without the need for an additional algorithm editor. Of course such an algorithm editor would be an important addition for ease of use.

4.1.2 Agent Technology

Seal is divided into three different subcomponents: animation, algorithm, and instructor. These components are tied together by the core of Seal which enables the initial communication between the components and creates the basic user interface. To fulfill the requirement that it must use agent technology in its implementation we need to see how each of the components fulfills the requirements for agent-based systems.

While it does not fulfill many of the facets that agent-based software engineering provides (mainly the lack of dynamic re-organisation) (Jennings, 2000), its sub-

components are autonomous and active. While the activity that Seal's components undertake is mainly reactive especially the instructor also has many proactive features that maintain the use of both types of activity. We will discuss each of the component's agent-like features in more detail below, but we can assert that Seal's subcomponents are infact agents.

The algorithm component is the least agent-like of all of the components, as it does not have its own thread of control but relies on requests obtained from other agents or the core system. It can be best described as an information agent because it takes some base information and transforms it into the format requested by other agents. As such it has some social ability in order to communicate with other agents. While it seems to have a user interface as well, it does not react to user actions and as such we cannot consider it to communicate with the user.

While the algorithm doesn't have its own thread, the animation agent must have its own thread in order to automatically take over algorithm execution from the user when all necessary data has been obtained. At a source code level the animator is divided into two parts: one functions in the same thread as the core system and the other is the primary animator which handles the creation of new steps by following user input automatically. The level of social interaction with the user requires fairly complicated interactions, taking the animator away from traditional algorithmic approaches and closer to agenthood (Wegner, 1997). The animator's interaction with the other agents is limited to interactions with the algorithm because the trace of the user's execution path through the algorithm is maintained by the algorithm agent based on the data it gets from the animator.

The instructor's functionality will be discussed in more detail below, but consider it in relation to the other two agents present in Seal. While it does not directly receive data from the animator, it follows the animator's execution and only begins to function in a visible way (to the user) once the animator's execution is complete or when an endless loop is encountered. Once the instructor takes control of the user session it directs the animator in order to animate the correct execution of the algorithm, if needed. The more mental features of the instructor's agenthood will

be left for discussion below.

Here we have concentrated on considering the algorithm, animation, and instructor agents' agenthood in the light of the more generic definitions of agenthood. While I do not fully subscribe to the view that agenthood requires mental agency, I view social agency as adequate (Singh, 1998 and Jennings et al., 1998). In this light the social ability present in the animation and instructor agents is sufficient. Future development can further emphasize the benefits that can be obtained from using agents in such a system.

Some future development that can be envisaged for Seal includes the use of the Internet to enable networking between various installations of Seal. If we want the instructor to be able to modify its behavior based on previous sessions with its users, a single installation's knowledge database will be woefully insufficient. Because the Internet allows easy communication around the globe, several centralized data stores could be used to store the accumulated knowledge of all of the instructors. Likewise the central storages can be used to store the algorithms created by instructor's worldwide, thus further reducing instructor load in utilizing Seal.

4.1.3 Debugging

When we discussed the design of animation environments (section 2.3) the "design/edit/run/debug"-cycle was mentioned. While Seal's nature as described above includes an inherent need for the run and debug parts of the cycle, this requirement needs additional illustration.

An inherent part of being able to debug a program or algorithm's execution is the ability to provide the input data on which the program will operate. Seal allows the student to enter their own input data (ideally is formed so that the student will know what the final solution of the algorithm's execution will be). If this is the case, the user should be able to see when the execution of the algorithm, based on

their guidance is incorrect, if the desired result cannot be obtained.

This enables the use of the debugging part of the cycle as the student may backtrack to the point in which s/he thinks that the execution went on the wrong track and try to correct it. The ability to backtrack through an algorithm's execution will also let the student view the algorithm's execution several times, thus enabling better recollection.

4.1.4 The instructor

The instructor forms the core of the agent-based implementation of Seal. After all, the definition of the algorithm and animation components as agents is an exercise in definitions. The instructor is required to be as unobtrusive as possible. This means that it must not force the student to follow a predetermined path but must allow free investigation as well as trial and error.

To fulfill this requirement the instructor has been designed so that it does not require the student's attention until the student's execution of the algorithm is completed. Once the student's execution is complete, the instructor will point out the first error, if any, of the student. It will also show the correct execution of the given statement. After this the student is expected to continue the execution. Again, once the execution is complete the instructor will take over and repeat the previous if necessary.

Once the algorithm's execution has been successfully completed, the instructor can give the student a walk-through of the algorithm's correct executions with additional textual material that is provided with the algorithm. This additional textual material is not shown to the student during the initial phase as it will distract the student and add to the student's cognitive load. It is shown at the end to better ensure student understanding.

While the instructor does have some beliefs, it is still lacking in its mental attributes. Further development needs to concentrate on better models of typical

problems with basic programming constructs that can be used to guide the student towards the correct execution of algorithms.

User authentication, in addition to the modeling of students, will enable the instructor and other agents to add intelligence to their own behavior once a student is known to understand certain constructs. For example, in CS1 algorithms the student will be required by the animator to show how a loop variable is used as an array index whereas in CS3 the student should already understand the relation. User modeling and authentication can help by allowing the animator to automate the use of loop variables as array indexes if the student is known to understand the connection.

Enabling the instructor's learning based on the student errors and solution paths needs to be researched as it can help in developing the beliefs of the instructor. While the creation of a complete student model is not even attempted, some modeling will be useful. In this case, using centralized storage areas for such information will provide more data for each instructor to use so that emergent behavior can be achieved.

4.1.5 User Interface

Since we are designing an application that will be rarely used and as a small tool, the user interface must designed so that the cognitive load that it presents is as small as possible. Unfortunately user interface design is not an easy or quick task and the current prototype is not self-evident or easy to use. Further development must concentrate on some of the following issues to make Seal more user friendly.

The overall view of the software needs some fine tuning. Currently Seal opens up a huge window (using all available window space) and several smaller windows inside the parent window. This is a common approach on Windows platforms, but Macintosh and X-Windows users are more used to the approach of several independent windows. The use of independent windows gives the user a better

choice of what is visible and where to place each window.

The animation engine needs a severe rewrite so that the animations are smoother and value changes are illustrated instead of their instantaneous appearance. The animation of the values will also better illustrate how different variables in the algorithm modify each others value's during execution.

The use of sound has not been studied very much. Many interface design guidelines suggest the use of sounds to give auditory cues in additional to visual cues but the use of sound in program visualizations has not been extensively researched. In normal use Seal does not necessarily need to have sounds enabled, but future research on the use of sounds with visualizations may provide reasons to include the use of sounds in Seal.

4.1.6 Data Visualization

The need for the user to select what data is shown was discussed in connection with some of the issues that need consideration when designing program visualizations. Most researchers agree that users must have the ability to limit the amount of data that is shown so that the screen does not become cluttered. Seal offers the user the ability to hide or show variables as needed. The ability to position the variables in a pleasing and sensible order is important, since the animation engine cannot infer sensible locations for the variables by itself.

Future development needs to investigate ways of deciding how visible data should be placed so that the animation appears smooth and uncluttered. Currently none of the available program visualization applications provide any kind of automation that places variables in a meaningful way. This may be another application area in which agents can be beneficial.

4.1.7 Platform independence

Because an instructional tool needs to be used in widely differing settings, platform independence was placed as one of the requirements for Seal. In the current situation Java is the only language that offers a sensible approach to developing platform independent software that requires a graphical user interface.

Although Java has been used to create Seal it is not being deployed as an applet. The primary reason for the application approach of Seal is that supporting all of the various browsers and running environments is quite a task and downloading an applet for each use is a waste of resources.

The choice of Java as the development language was strengthened by the easy networking opportunities it provides. As some of the options for future development have shown, support for various networking scenarios is needed.

In addition to the selection of a portable programming environment, platform independence has placed its own requirements on interface design. Seal does not use any hard-coded values for screen sizes, colors, or fonts as they might not be available on all platforms.

While Java does offer significant benefits when designing platform independent software, it has some unfortunate limitations as well. One of the main limitations is that in addition to Seal the Java Runtime Environment adds to the disk space required. Unfortunately Java is also quite slow especially on older computers. Both of these factors limit its usability on older computers on which native code would be fast enough. Luckily even on most slower computers Seal's functioning, once started, is fast enough.

4.1.8 Implementation experiences

Because the development time of the current version of Seal was limited, the lack of suitable agent frameworks or even design patterns had an impact on the de-

velopment. Building the whole architecture with custom components, excluding the standard Java libraries and XML and XSL/T libraries, has taken an undue amount of time. This time has reduced the time that could have been used to further develop the functionality of the instructor towards a more rational entity. The instructor needs to be developed more before concentrating on other issues.

Since most of the formal specification models for agents (Jennings & Woolridge, 2000) were too heavy for use in the specification of Seal, a traditional software engineering approach was taken in which the agents are custom built. Unfortunately, agent development, as all software development has its own set of pitfalls (Woolridge & Jennings, 1998) that we saw in our own development process more than once.

My original design had four different agents that used an ACL to intercommunicate. This plan was already partly implemented into code before its feasibility was questioned and the code needed to be dumped. The current version, as already noted, has only three agents and all communication takes place through normal method calls.

I have tried to avoid creating a system that is a good prototype but far from an industrial strength solution. Of course the fact that we are talking about educational software that is not intended to be used alone means that the requirements are not as high as in air traffic control environments. Nonetheless bad choices during the design would have lead to a system that is unusable in any real learning environment.

As already noted, I initially designed the application with too many agents but at some point the whole agent approach needed reconsidering as well. Hopefully the current design provides a good balance between the number of agents and their intelligence. Future development needs to enhance the intelligence of the instructor but that does not concern the other parts of the application directly.

Because 99% of agent development is still traditional software engineering (Wool-

ridge, 1997) — and Seal is closer to 99.9% — most of the problems that I faced during the development were caused by the challenges that most software developers face every day. Some of the problems had to do with the lack of proper documentation for new technologies while others had to do with problems related to the use of external libraries.

One of the main problems in developing agent based applications in Java is that agents need a high level of concurrency which is often problematic in Java. While Java does support threads and concurrency, some of its undocumented and sometimes buggy "features" can easily give the developer more than a few gray hairs. Luckily in another project I was required to write even more concurrent code that needed industrial level reliability. This helped prevent several problems that would have certainly arisen otherwise.

One of the biggest hurdles in Seal's development was trying to modify the highlighting behavior of Java's text-component that allows showing HTML code. Only after several hours of workaround development and browsing of obscure documents was the solution to my problem found — a solution that would have been easy to document in the developer documentation of Java's class libraries.

4.2 Example of use

We will now walk-through a student's example session with Seal. Its initial view, after startup, can be seen in figure 4. The sum-algorithm is loaded on startup by default and we will consider it in this example. The sum-algorithm's original XML-representation can be seen in appendix C and the converted HTML, the student's view of the algorithm, in figure 5.

Figure 4 shows the various windows that Seal opens at startup. The 'Animation controls' window should be self-evident, it is also the window needed to start the execution of the algorithm. The Value changer allows the student to enter a new value that is used to modify a variables value. The modification of the variable's

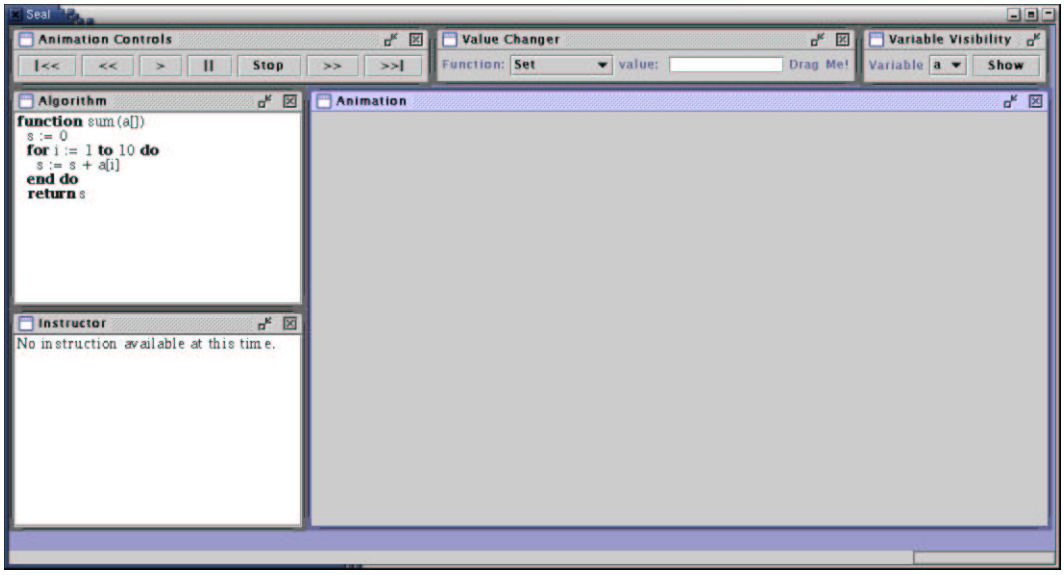

Figure 4: Initial view of Seal.

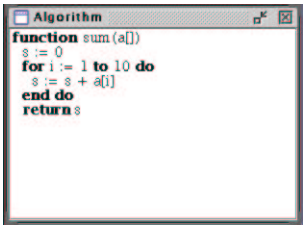

Figure 5: Seal's view of the sum-algorithm.

value is based on the function that is selected with set as the default.

Once the student has selected which variables they want visible — all in this case — it is time to start the execution of the algorithm by clicking the play button in the animation controls. As the function in question has a parameter, the user is prompted to set the initial values of the parameter as seen in figure 6. With an array all of the values must be set, but this should not be an obstacle as in most examples the size of the arrays should be fairly small so that setting all the values does not take an undue amount of effort. If the student notices an error in the value of the parameter that they set, they may change it without any effect on the algorithm execution before the first line's functionality is embarked upon.

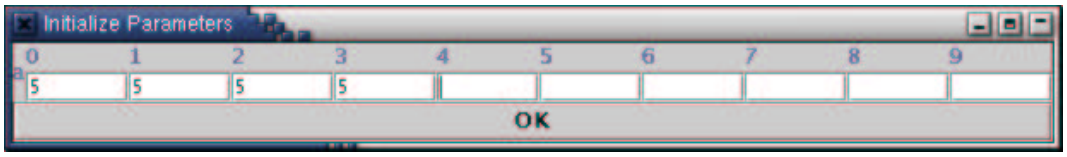

Figure 6: Setting the function's parameters.

Once the parameters have been set, the execution of the algorithm will begin in earnest. In this case the first line of the execution requires the initialization of the variable `s` which holds the sum. To initialize the variable, the student must enter the value 0 into the 'Value changer' window, set the function to Set and drag the value of the correct variable. The state of the execution after the parameter has been set and the first line has been executed can be seen in figure 7.

The next step has to do with initializing the control variable for the for-loop. For readers familiar with algorithmic representations the meaning of the statement is quite clear, but let us assume that our example student has just begun his studies in CS and even a simple loop is a difficult concept. The first time the line is encountered its function is apparent to our student, the variable `i` gets the value 1 in the same way as described above when initializing `s`.

On to the next line of the execution — so far so good. In this line the value of `s` should grow with the value found in array `a`'s `i`th slot. So, our student selects

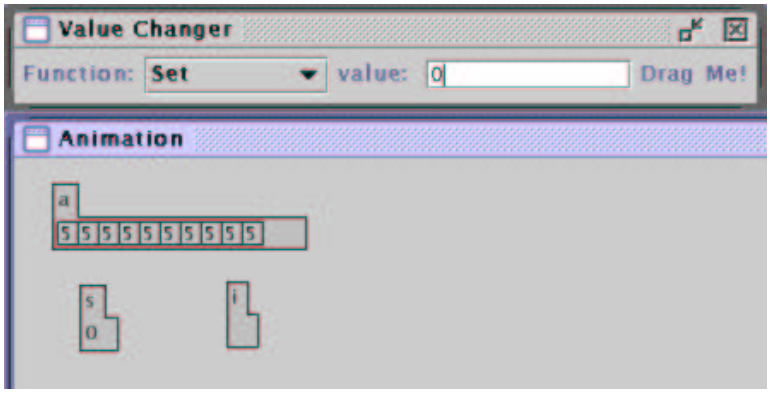

Figure 7: The state of the animation after the first line of the algorithm has been executed.

Sum as the function in the 'Value Changer' window and then correctly drags the first value of the array to `s`. After this line's execution the value of `s` is 5 as seen in figure 8.

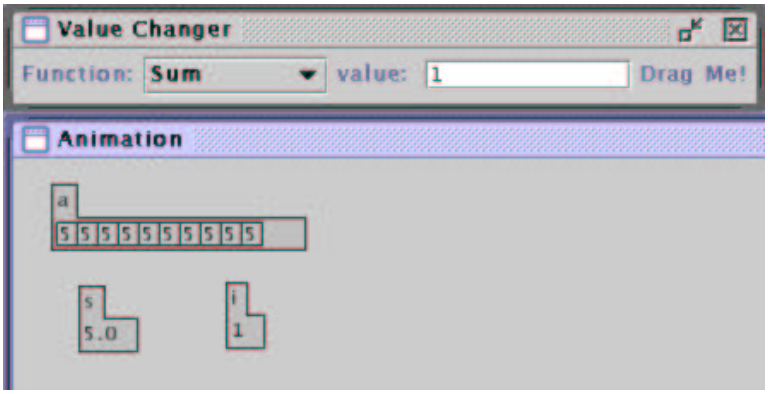

Figure 8: View of the animation after the first round of the for-loop.

What our student does not understand is that when the for-loop is executed again, `i` should be incremented by 1. Since he does not know why the execution stopped at the for-loop's line, he clicks 'play' to make the animation continue. Since the animator has received guidance on what to do with the body of the for-loop when the value of `i` is 1, it will begin looping automatically, always incrementing `s` with the first value of the array `a`.

This mode of execution will create an endless loop since the end condition of the for-loop, $i = 10$, is never reached. Once the for-loop has been executed eleven times, the instructor will interrupt the execution and guide the student back on track by informing the student of the for-loop's end condition. The execution will then be rolled back to the state visible in figure 8, the instructor's guidance can be seen in figure 9.

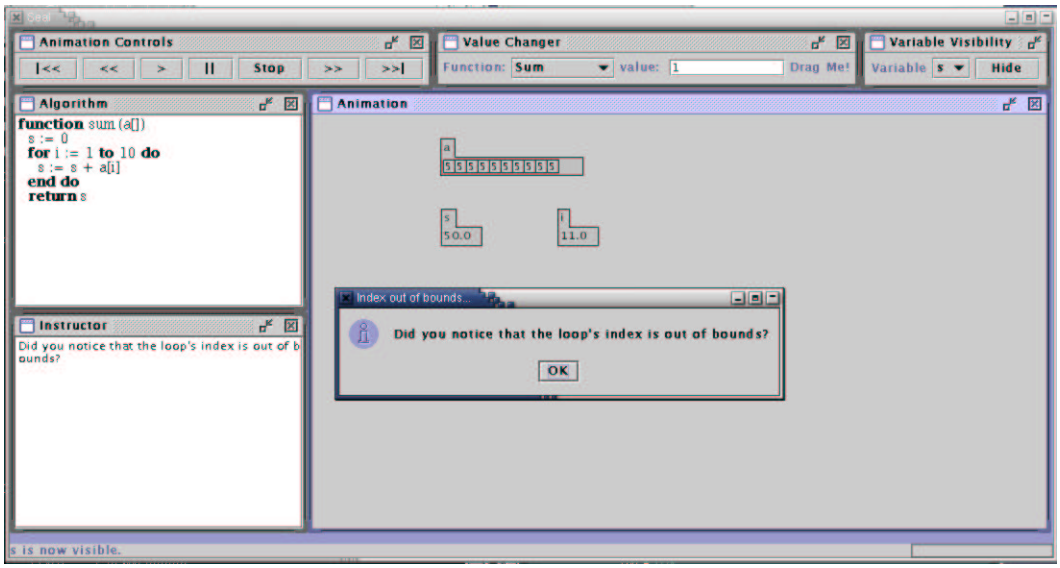

Figure 9: Instructor's guidance when the for-loop has been executed too many times.

At this time some bells begin ringing in our student's head. He remembers the last lecture in which the lecturer described how for-loops function and remembers to increment `i`'s value with 1 now instead of just continuing. Since the animator does not know what to do with the body of the for-loop it will pause to ask for guidance from the student. This time around the student checks that the function in the 'Value changer' window is Sum and drags the second value of the array `a` to `s`. After this the animator infers that each time the beginning of the for-loop in reached `i` is incremented by one and `s` is incremented by the `i`th element of the array `a`.

Once the execution of the for-loop is complete, the last line of the algorithm holds

the return value of the function. Once this has been reached, the instructor will grab the value that would have been returned and ask the student whether the value is correct (see figure 10). If the student answers correctly a final acknowledgment is given; otherwise the student is prompted to consider whether his preconception of the result is correct.

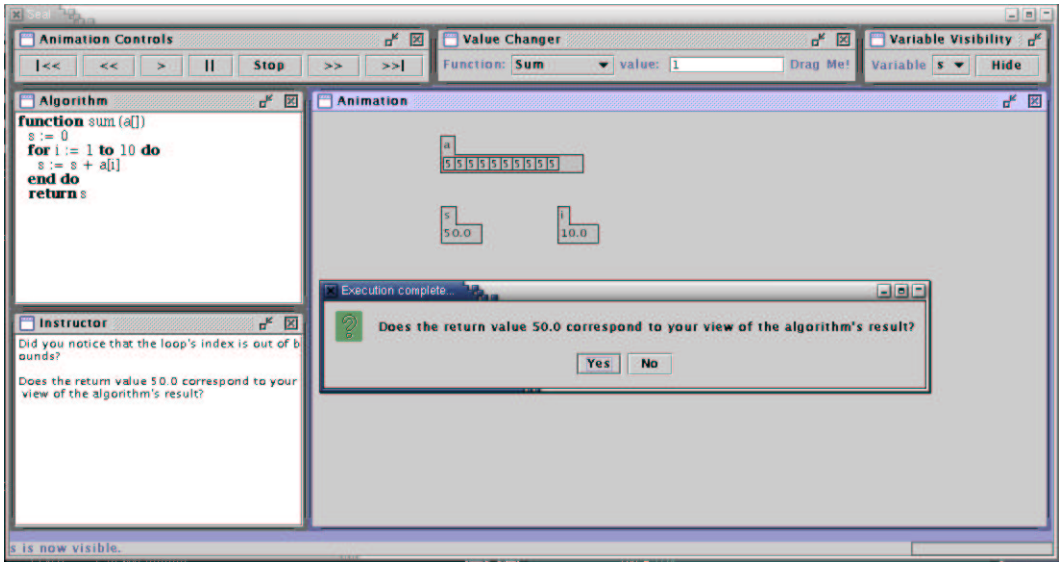

Figure 10: The end of the algorithm has been reached. Is the result correct or not?

4.3 Evaluation

Before we use any generic evaluation checklists we will look at how Seal fits into the level classification of Anderson and Naps (2001) that we used earlier (see section 2.2). While Seal is clearly geared towards understanding the algorithm as a recipe (level I) with its pseudo-code representations, it can still be considered as belonging to the second level. While Seal does not show actual implementation code to the student it requires that the student understands each step of the algorithm's execution which should be fairly easy to transcribe into program code by a student familiar with the programming environment. It is possible that Seal can be used to help in efficiency analysis and correctness proofs but these issues have

not been a deciding factor during its design.

On the instructional design scale Seal falls into many of the levels quite easily. It does not belong to the first level in which the animation can be viewed passively as it requires active student participation. While the current version does not support showing textual materials alongside the animation, the requirement for level II, the development roadmap has this feature marked up as high priority and the architecture fully supports this functionality. I must thank Naps for his input on this matter at the First Program Visualization Workshop, without it this functionality would have been overlooked.

Level III, user designed input data, is clearly implemented by Seal as seen in figure 6. While the student is not asked to predict what will take place in the algorithm's execution at certain breakpoints (level IV), the student must guide the algorithm's execution at each new statement. While level IV is Seal's placement on the scale, level V is not. Although the user can decide which variables he wants to view during the execution, Seal is not designed to cope with large amounts of input data so that the amount of information viewed would really need to be adjusted (level V).

Levels VI and VII are also a part of Seal's repertoire as the user has the ability to — and might be required to — backtrack in the algorithm's execution to correct it (level VI), and the user actively works on the design of the visualization (level VII). The final level on the instructional scale has to do with the linkage between the algorithm animation and other course materials. Naturally we cannot evaluate Seal's placement in this level as it depends on the design of the course materials and instructor more than anything else. Seal is not specifically designed to be a part of a certain course.

To evaluate Seal, I have chosen the checklist presented in (Delta Checklist, 1989). It has been chosen over the extended Ravden & Johnson checklist (Tukiainen & Lempinen, 1994) because of its focus on cognitive issues instead of technical and user interface issues. Once Seal is completed and has left the prototype stage

the evaluation should include the use of a technical checklist to evaluate the user interface.

The following table shows the numeric grades of Seal in the different evaluation categories of the Delta-project. A grade of NA means that the category is not applicable, otherwise 0 is the worst and 3 is the best. The potential column is used to describe how the planned development of Seal should change its grade in the category as we are still evaluating an extremely raw prototype. We will take a closer look at Seal's evaluation in the categories below with the rationale for the grade.

Evaluation Category	**Grade (0–3)**	**Potential**
Engagement	3	
Interactivity	3	
Tailorability	1	++
Appropriateness of multimedia mix	NA	
Mode and style of interaction	NA	
Quality of interaction	NA	
Quality of end-user interfaces	0	++
Learning Styles	1	
Monitoring and assessment techniques	1	++
Built-in intelligence	0	+
Adequacy of ancillary learning support tools	NA	
Suitability for single user/group/distributed use	3/0/NA	
Availability in terms of cost and delivery platforms	3	
Outstanding strengths and attractive features	NA	
Outstanding limitations and weaknesses	NA	

I cannot be an objective evaluator of Seal. This evaluation is intended to gather and underline the educational aspects that have guided its design and implementation. Engagement is given the best possible grade since what CS student would rather use paper and pencil instead of a computer, especially when the computer

is clearly supposed to reduce the amount of work. In reality however, only experiences of using it in actual learning situations will show what the grade will be.

I have left some of the interactivity subgroups without a grade as the general level of interactivity in Seal is important. In an environment designed for algorithm animation, the use of animation is automatically appropriate. Because Seal is an inherently interactive system — after all, the student guides the whole animation process — the quality, mode and style of the interaction cannot really be given a grade. Overall its interactivity does rate an excellent grade, again due to its nature more than anything else.

Most of the categories marked with a potential of development are those that had to be given less attention in the creation of the prototype. Its tailorability has to do more with how instructors use it as a part of their courses. Seal is designed to work in guided laboratory sessions or independent work but needs to be integrated into the course context for it to be useful.

The Delta-project's definition for built-in intelligence is described as how well the program remembers what the student has already learned. While my view of built-in intelligence is somewhat different, Seal can be implemented with the ability to remember which algorithms the student has covered and completed. In Seal's case this should not be an important attribute.

Monitoring and assessment techniques received a fairly low grade as currently the amount of assessment is only adequate. Future development should provide more abilities for the instructor to assess the student's actions and modify the guidance accordingly. Seal's support for different learning styles is limited; after all it is a tool for automating assignments. Because Seal is primarily designed for use by a single student, its suitability for group use is poor. While problem solving in a group may be beneficial, in this case the important factor is ensuring student understanding which requires that each student's understanding can be monitored.

Because Seal is published under the GNU Public License (GPL, 2001) and is developed with Java it should rank fairly high when considering its availability. Because of the last two years spent designing and implementing Seal, I feel that objectivity in evaluating its strengths, weaknesses, attractive features, and limitations is impossible. Therefore I will leave them without an evaluation and let others evaluate Seal further when the development has progressed.

5 Discussion

Computer based educational environments have come to the point in which traditional uses of artificial intelligence are inadequate and no clear replacements have surfaced. Agents — in all of their diversity and ambiguity — have been proposed as an alternative to traditional AI systems. Because intelligence on the computer side is often needed in learning environments the use of agents in CAI needs to be studied and evaluated. Several prototypical systems have been created that utilize agent-based technology in educational environments, but the silver bullet is yet to be found.

As we have already discussed agent-based computing is not that much about creating AI systems that learn rather than creating systems that help users manage complexity by providing networks of interactions that might have been unforeseen by the original designers of the system.

While teaching understanding algorithms can be seen as something helped by intelligent agents, many other topics are harder to reduce into an algorithmic or interactive set of rules. As we saw in the description of Seal (4) it is easy for a computer to know when the execution of an algorithm by the student differs from how it should be executed. Unfortunately very few topic matters have such clear-cut boundaries between the correct and incorrect answer. Of course, the proponents of agent-based engineering often argue that an algorithmic view is less powerful than interaction (Wegner, 1997). This enables better possibilities for using intelligence in the computer-aided instruction of subjects other than algorithms as well.

Seal, although still a barely usable prototype, shows that social agents and interaction can be used in algorithm animations. The interaction that takes place in Seal between the student and the animator or instructor-agents helps ensure that the student really understands the connection between the algorithm and the visualization.

The additional guidance that Seal is able to provide for the user in comparison

to many other algorithm animation environments comes from the use of social agents. This guidance takes the form of pointing out the steps in which the algorithm's execution as defined by the student differs from the correct execution. Thus, when faced with the deviance from the correct execution, the student has the opportunity to reconsider his own view of the algorithm's function or result.

The development of Seal has shown that agent-based engineering offers new opportunities. Unfortunately, as agent-based engineering is a relatively new branch of computer science, the lack of suitable development environments and common standards means that developing agent-based applications requires quite extensive resources. Hopefully as agent-based engineering matures design guidelines, development environments, frameworks, and libraries will be created that will help in the creation of new agent applications.

My work with Seal has brought forth the need for a common way of representing algorithms and related textual information for use in algorithm animations and other uses of algorithms. The work begun on the XML DTD presented in appendix B is the first step towards such a common format.

A List of Abbreviations

Abbreviation	Description	Defined in
ACL	Agent Communication Language	3.2.2
AI	Artificial Intelligence	1
BDI	Belief-Desire-Intention	3.1.1
CAI	Computer Aided/Assisted Instruction	1
CAL	Computer Aided/Assisted Learning	1
CS	Computer Science	1
CSE	Computer Science Education	1
HCI	Human-Computer Interaction/Interfaces	3.1.2
ITS	Intelligent Tutoring Systems	3.3
MAS	Multi-Agent System	3.1.1
OO	Object Orientation	3.1.2
SV	Software Visualization(s)	2

B The DTD used to describe algorithms in Seal

```
<?xml version="1.0"?>
<!--======================================================================

The algorithm representation DTD for use in Seal
(c) 2001 Ramin Miraftabi <ramin@cs.joensuu.fi>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

======================================================================-->

<!ELEMENT algorithm (description, (function)+)>
<!-- algorithm attributes, for descriptions on what the level means
     see http://www.acm.org/sigcse/cc2001/steelman/cs-courses.html
-->
<!ATTLIST algorithm name    CDATA    #REQUIRED
                    level   (CS1|CS2|CS3) "CS1">

<!-- Description and related fields, basically very simplified XHTML 1.0...
-->
<!ELEMENT description (CDATA|(p|ul|ol)+)>

<!ELEMENT p (CDATA|em|strong|tt)+>

<!ELEMENT ul (li)+>
<!ELEMENT ol (li)+>
<!ELEMENT li (CDATA|em|strong|tt)+>

<!ELEMENT em CDATA>
<!ELEMENT strong CDATA>
<!ELEMENT tt CDATA>

<!-- The function and related fields -->
```

```
<!ELEMENT function (param*, body>
<!ATTLIST function name CDATA #REQUIRED>

<!ELEMENT param (var)>

<!ELEMENT body (line)+>

<!ELEMENT line ((comment)?, (assignement|for|while|do|if|return)>

<!ELEMENT assignement (var, value)>
<!ELEMENT for (begin, end, body)>
<!ELEMENT while (clause, body)>
<!ELEMENT do (body, clause)>
<!ELEMENT if (clause, body, (else)?)>
<!ELEMENT else (if|body)>
<!ELEMENT return (value)>

<!ELEMENT begin (assignement)>
<!ELEMENT end (value)>
<!ELEMENT clause (value)>

<!ELEMENT value (var|literal|expression)>

<!ELEMENT expression (value, operator, value)>

<!ELEMENT operator EMPTY>
<!ATTLIST operator type (sum, sub, mult, div) #REQUIRED>

<!ELEMENT var (index)?>
<!ATTLIST var name CDATA #REQUIRED
              type (char, int, double, boolean) "int"
              size CDATA "1">

<!ELEMENT index EMPTY>
<!ATTLIST index name    CDATA #IMPLIED
                literal CDATA #IMPLIED>

<!ELEMENT literal (#PCDATA)>

<!ELEMENT comment (CDATA|(p|ul|ol)+)>
```

C Listing of the sum.xml used in the examples of Seal

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl" href="xsl/algorithm2html.xsl"?>
<?cocoon-process type="xslt"?>
<algorithm name="sum">
  <level>
    <cs1/>
  </level>

  <description>
    This algorithm calculates the sum of the values in the parameter array.
    For correct execution, all of the array's values should be numbers,
    characters and strings will cause errors.
  </description>

  <function name="sum">
    <param>
      <var name="a" type="array" size="10"></var>
    </param>
    <body>
      <line>
        <assignement>
          <var name="s"></var>
          <value><literal>0</literal></value>
        </assignement>
      </line>
      <line>
        <for>
          <begin><assignement><var name="i"></var>
              <value><literal>1</literal></value></assignement></begin>
          <end><value><literal>10</literal></value></end>
          <body>
            <line>
              <assignement>
                <var name="s"></var>
                <value>
                  <expression>
                    <value>
                      <var name="s"></var>
                    </value>
                    <operator type="sum"/>
                    <value>
```

```
                    <var name="a"><index name="i"/></var>
                  </value>
                </expression>
              </value>
            </assignement>
          </line>
        </body>
      </for>
    </line>
    <line>
      <return>
        <var name="s"></var>
      </return>
    </line>
  </body>
 </function>
</algorithm>
```

References

Abernethy, K., Gabbert, P., & Treu, K. (1998). Inquiry-based computer science instruction: Some initial experiences. In G. Davies & M. ÓhÉigeartaigh (Eds.), *Proceedings of the 3rd annual conference on integrating technology into computer science education* - ITiCSE *'98* (pp. 14–17). Dublin, Ireland.

Anderson, J. M., & Naps, T. L. (2001). A context for the assessment of algorithm visualization systems as pedagogical tools. In E. Sutinen (Ed.), *Proceedings of the first program visualization workshop* (pp. 121–130). Porvoo, Finland.

Aroyo, L., & Kommers, P. (Eds.). (1999). *Journal of interactive learning research: Special issue on intelligent agents for educational computer-aided systems.* Association for the Advancement of Computing in Education.

Badre, A., Beranek, M., Morris, J. M., & Stasko, J. (1991). *Assessing program visualization systems as instructional aids* (Tech. Rep. No. GIT-GVU-91-23). Georgia Institute of Technology.

Bates, J. (1994). The role of emotion in believable agents. *Communications of the ACM, 37*(7), 123–125.

Ben-Ari, M. (1998). Constructivism in computer science education. *SIGCSE Bulletin, 30*(1), 257–261.

Ben-Bassat Levy, R., Ben-Ari, M., & Uronen, P. A. (2001). An extended experiment with Jeliot 2000. In E. Sutinen (Ed.), *Proceedings of the first program visualization workshop* (pp. 131–140). Porvoo, Finland.

Bergin, J., Brodlie, K., Goldweber, M., Jiménez-Peris, R., Khuri, S., Martínez, M. Patiño, McNally, M., Naps, T., Rodger, J. S., & Wilson, J. (1996). An overview of visualization: Its use and design. *SIGCSE Bulletin, 28*, 192–200.

Borsook, T. K., & Higginbotham-Wheat, N. (1991). Interactivity: What is it and what it can do for computer-based instruction. *Educational Technology, 31*(10), 11-17.

Brooks, R. A. (1991). Intelligence without reason. In *Proceedings of the twelfth international joint conference on artificial intelligence (*IJCAI-*91)* (pp. 569–595). Sydney, Australia.

Brown, M. (1988). *Algorithm animations.* Cambridge, MA: MIT Press.

Byrne, M. D., Catrambone, R., & Stasko, J. T. (1996). *Assessing program visualization systems as instructional aids* (Tech. Rep. No. GIT-GVU-96-18). Georgia Institute of Technology.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1989). *Introduction to algorithms.* The MIT Press and McGraw-Hill Book Company.

DARPA *agent markup language (*DAML*).* (2001). http://www.daml.org/. (14.12.2001)

Decker, K., Pannu, A., Sycara, K., & Williamson, M. (1997). Designing behaviours for information agents. In *Proceedings of the first international conference on autonomous agents (*AGENTS-*97).*

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. *Communications of the ACM*, *32*(1), 9–23.

*Document object model (*DOM*).* (2001). http://www.w3.org/dom/. (14.12.2001)

Domingue, J. B., & Mulholland, P. (1997). Teaching programming at a distance: the internet software visualization library. *Journal of Interactive Media in Education*. (http://www-jime.open.ac.uk/jime/02/jime-02.html (14.12.2001))

EC *delta-project* ILDIC*: Integrating learning design in interactive compact disc.* (1989). http://www.edu.fi/oppimateriaalit/lumaromput/deltasap.html. (14.12.2001)

Eisenstadt, M., Price, B. A., & Domingue, J. (1993). Software visualization as a pedagogical tool: redressing some ITS fallacies. *Instructional Science, 21*.

Etzioni, O. (1996). Moving up the information food chain: Deploying softbots on the world-wide web. In *Proceedings of the thirteenth national conference on artificial intelligence (*AAAI*-96).* Portland, OR.

*Extensible markup language (*XML*).* (2001). http://www.w3.org/xml/. (14.12.2001)

*The extensible stylesheet language (*XSL*).* (2001). http://www.w3.org/style/xsl/. (14.12.2001)

Finin, T., Labrou, Y., & Mayfield, J. (1994). KQML *as an agent communication language* (Tech. Rep.). Gaithersburg, Maryland.

FOLDOC - *computing dictionary.* (2001). http://wombat.doc.ic.ac.uk/foldoc/index.html. (14.12.2001)

Genesereth, M. R., & Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, *37*(7), 48–53.

The GNU *public license (*GPL*).* (2001). http://www.gnu.org/licenses/gpl.html. (14.12.2001)

Guilfoyle, C., & Warner, E. (1994). *Intelligent agents: The new revolution in software.* Ovum Report.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., & Vanninen, P. (1997). Animation of user algorithm on the web. In IEEE *symposium on visual languages* (p. 360-367). Los Alamitos: IEEE Comuter Society Press.

Janca, P. C. (1995). *Pragmatic application of information agents.* BIS Strategic Report.

Jennings, N., & Woolridge, M. (1998). Applying agent technology. In N. Jennings & M. Woolridge (Eds.), *Agent technology: Foundations, applications, and markets* (pp. 3–28). Berling, Germany: Springer-Verlag.

Jennings, N., & Woolridge, M. (2000). Agent-oriented software engineering. In J. Bradshaw (Ed.), *Handbook of agent technology.* AAAI/MIT Press.

Jennings, N. R. (2000). On agent-based engineering. *Artificial Intelligence, 117*(2), 277–296.

Jennings, N. R., Sycara, K., & Woolridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems, 1*, 7–38.

Kautz, H. A., Selman, B., & Coen, M. (1994). Bottom-up design of software agents. *Communications of the ACM, 37*(7), 143–147.

Kehoe, C., Stasko, J., & Taylor, A. (1999). *Rethinking the evaluation of algorithm animations as learning aids: An observational study* (Tech. Rep. No. GIT-GVU-99-10). Georgia Institute of Technology.

Khuri, S. (2001). Designing effective algorithm visualizations. In E. Sutinen (Ed.), *Proceedings of the first program visualization workshop* (pp. 1–12). Porvoo, Finland.

Klusch, M. (2001). Information agent technology for the internet: A survey. *Journal on Data and Knowledge Engineering, Special Issue on Intelligent Information Integration, 36*(3).

Lattu, M., Tarhio, J., & Meisalo, V. (2000). How a visualization tool can be used: Evaluating a tool in a research and development project. In *12th workshop of the psychology of programming interest group* (p. 19-32). Corenza, Italy.

Lawrence, A. W., Badre, A. N., & Stasko, J. T. (1994). *Do algorithm animations aid learning?* (Tech. Rep. No. GIT-GVU-94-07). Georgia Institute of Technology.

Maes, P. (1994). Agents that reduce work and information overload. *Communications of the ACM, 37*(7), 31–40.

Markkanen, J., Saariluoma, P., Sutinen, E., & Tarhio, J. (1998). Visualization and imagery in teaching programming. In J. Domnigue & P. Mulholland (Eds.), *Proc. 10th annual meeting of the psychology of programming interest group* (pp. 70–73). Milton-Keynes, England.

Mayer, R. E., & Sims, V. K. (1994). For whom is a picture worth a thousand words? extensions of a dual-coded theory of multimedia learning. *Journal of Educational Psychology, 86*(3), 389–401.

Meisalo, V., Rautama, E., Sutinen, E., & Tarhio, J. (1997). Teaching algorithms with animation — a case study using Eliot. In O. Hatakka (Ed.), *Proc. LeTTET '96, learning technology and telematics in education and training* (pp. 79–84).

Negroponte, N. (1995). *Being digital.* Hodder and Stoughton.

Newsweek. (2001). The classroom of the future. *Newsweek*(Oct. 29 Issue). (http://www.msnbc.com/news/645566.asp?cp1=1 (14.12.2001))

Nielsen, J. (1999). *Designing web usability.* New Riders.

Norman, D. A. (1994). How might people interact with agents? *Communications of the ACM, 37*(7), 68–71.

Resource definition framework (RDF). (2001). http://www.w3.org/rdf/. (14.12.2001)

Russell, S., & Norvig, P. (1995). *Artificial intelligence: A modern approach.* Prentice-Hall.

Schank, R. (1991). Where is the AI? AI *Magazine*, *12*(4), 38–49.

Selker, T. (1994). A teaching agent that learns. *Communications of the ACM*, *37*(7), 92–99.

Shute, V. J., & Psotka, J. (1995). Intelligent tutoring systems: Past, present, and future. In D. Jonassen (Ed.), *Handbook of research on educational communications and technology.* Scholastic Publications.

Singh, M. (1998). Agent communication languages: Rethinking the principles. *IEEE Computer*, *31*(12), 40–47.

Stasko, J. T. (1997). Using student-built animations as learning aids. In *Proceedings of the acm technical symposium on computer science education* (pp. 25–29).

Stasko, J. T., Domingue, J. B., Brown, M. H., & Price, B. A. (Eds.). (1998). *Software visualization.* MIT Press.

Sutinen, E. (Ed.). (2001). *Proceedings of the 1st program visualization workshop.* Porvoo, Finland.

Sycara, K., Decker, K., Pannu, A., Williamson, M., & Zeng, D. (1996). Distributed intelligent agents. *IEEE Expert*.

The SyncML *initiative.* (2001). http://www.syncml.org/. (14.12.2001)

Tukiainen, M., & Lempinen, R. (1994). *Muokattu* R*avdenin ja* J*ohnsonin menetelmä ohjelmiston käytettävyyden arviointiin* (Tech. Rep. No. B-1994-1). Department of Computer Science, University of Joensuu.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, *40*(5), 80–91.

Woolridge, M. (1997). Agent-based software engineering. In IEE *proceedings on software engineering* (Vol. 144, pp. 26–37).

Woolridge, M. (1999). Intelligent agents. In G. Weiss (Ed.), *Multiagent systems.* The MIT Press.

Woolridge, M., & Jennings, N. R. (1995). Intelligent agents: theory and practice. *Knowledge Engineering Review, 10*(2), 115–152.

Woolridge, M., & Jennings, N. R. (1998). Pitfalls of agent-oriented development. In *Proceedings of the second international conference on autonomous agents (agents '98)* (pp. 385–391).

Zeller, A. (2001). Animating data structures in DDD. In E. Sutinen (Ed.), *Proceedings of the first program visualization workshop* (pp. 69–77). Porvoo, Finland.

Zeng, D., & Sycara, K. (1995). *Cooperative intelligent software agents* (Tech. Rep. No. CMU-RI-TR-95-14). The Robotics Institute, Carnegie-Mellon University.