Levels of Exploration

Stephan Diehl and Andreas Kerren
University of Saarland, FR 6.2 Informatik,
PO Box 15 11 50, D-66041 Saarbrücken
{diehl, kerren}@cs.uni-sb.de

Abstract

Visualization of computational models is at the heart of educational software for computer science and related fields. In this paper we look at how generation of such visualizations and the visualization of the generation process itself increase exploration. Four approaches of increased exploration in formal language theory and compiler design are introduced and for each approach we discuss an educational system which implements it.

1 Introduction

In computer science and in particular in compiler design the theory and algorithms are very abstract and usually complex. Therefore visualizations are appropriate for computer science instruction. Although compiler design is often considered a practical field within computer science, most of its techniques are based on work in theoretical computer science, e.g. formal languages, automata theory and formal semantics. In recent years we have developed several educational software systems for topics in compiler design and theoretical computer science. These systems have in common that they teach computational models by animating computations of instances of these models with example inputs. But they differ in the level of exploration.

Table 1 not only reflects the increased flexibility of the software developed, but also the chronological development of software by group, as well as the order of presentation in this paper. Higher levels of exploration demand more prerequisites and self-control by the learner.

SIGCSE 2001 2/01 Charlotte, NC, USA
 ISBN 1-58113-329-4/01/0002...$5.00

Thus, in the educational software the learner should start with static examples and as the learner advances the level of exploration should be increased. Exercises and textual hints in the educational software should guide the learner, to make sure he/she doesn't miss the important issues.

Approach	Input	Computational Model	Generator
Static	fixed	fixed	none
Interactive	user	fixed	none
First-order generative	user	user	yes
Second-order generative	user	user	yes visualized

Table 1: Levels of exploration

2 Static Approach

In the static approach the execution of an instance of a computational model is animated for a given, fixed input.

The educational software "Animation of Lexical Analysis" [1] has been developed with the authoring system Asymetrix Multimedia ToolBook 3.0 and runs on Windows 3.x/95/98/NT4. The software offers on the one hand an interactive introduction to the problems of lexical analysis, in which the most important definitions and algorithms are presented in graphically appealing form. Animations show how finite automata are created from regular expressions, as well as, how finite automata work. Currently there is only a German version of the software.

First several animations show the fundamental components of a scanner and the cooperation between parser and scanner. Then symbols and symbol classes are explained. It is shown, how input symbols, lexical symbols, symbol classes and their internal representation are connected. Next an overview about formal languages and an introduction to regular languages and regular expressions are given.

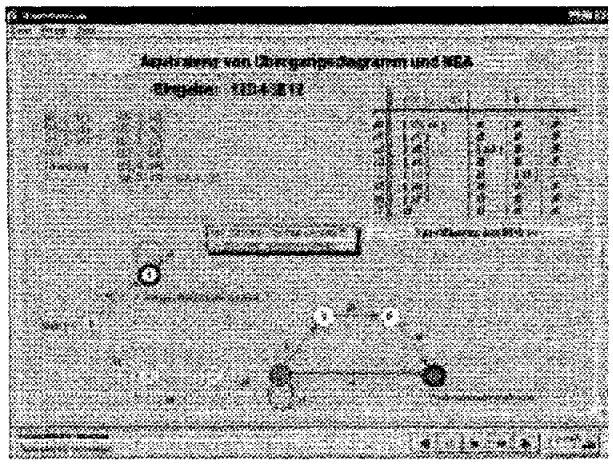

Figure 1: Equivalence of transition diagram and NFA

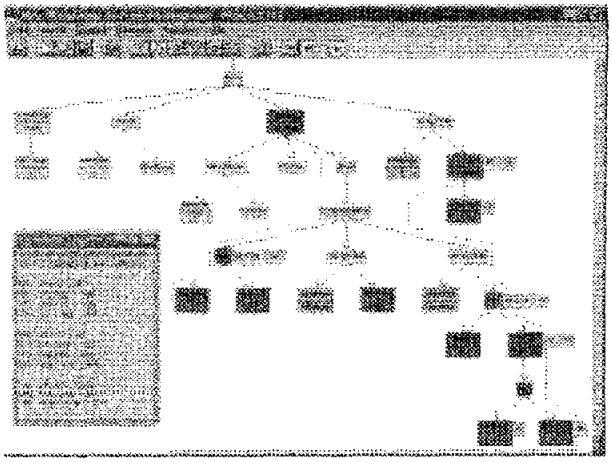

Figure 2: Visualization of the checking the context conditions

Then transition diagrams, non-deterministic (NFA) and deterministic (DFA) finite automata are described. There are animated examples for each of these that can be controlled by the user. The equivalence between regular expressions and NFA's is explained with an fixed animated example (see Figure 1). The user can follow the parallel processing of a transition diagram and an NFA with the same input string.

This software follows the static approach, because the user cannot enter own input strings. There are only animations of fixed input examples, which were designed by the developer of the educational software. The user can start animations, stop them or initiate a backtracking. But if he/she is curious to know what happens for a different input string, there is no way to find out.

3 Interactive Approach

In the interactive approach an instance of a computational model is animated for an example entered by the user/learner.

An example for this approach is our application "Animation of Semantical Analysis" [6]. It illustrates and animates the basic tasks of semantical analysis by textual and graphical examples. It covers basic knowledge, like the concepts of scoping and visibility, checking of context conditions (identification of identifiers, checking of type consistency), overloading of identifiers and polymorphism. The corresponding algorithms for analysis can be examined with own examples. As the system described in the previous section this system was implemented using Multimedia ToolBook. The dynamical component, that allows users to enter their own examples, was developed in C using the application programming interface (API) of the windows system. First our educational software presents and describes the definitions of semantical analysis step by step. Afterwards these are made clear on the basis of animated examples. Both happens completely interactive, i.e. the users can navigate through a graphical environment by mouse-click. They can select and deepen topics, which they are interested in. For these topics they can read explanatory text and look at animations. Finally the users have the possibility to enter examples, and to run the presented algorithms graphically on the dynamically drawn abstract syntax trees of these examples. Examples can be input programs, expressions or specifications for operator overloading.

The screendump in Figure 2 shows a visualization of checking the context conditions of an example program, that was entered by the user. The resulting syntax tree is automatically drawn and displayed in the application window. The user has influence on the tree layout, he/she can change the distances of sibling nodes, neighbouring nodes and parents/child nodes. Further there is the possibility to zoom and rotate the tree. These features help to place the tree in the window optimally. Thus it is possible to change the tree layout in such a way that the tree fits completely into the window. This increases the clarity with the animation. All other graph items, as for instance small information windows at the individual nodes, additional edges etc., are adapted directly to the new layout.

The abstract syntax tree is almost completely displayed. Also the type attributes of some nodes are shown. They are calculated on the basis of the types of the built-in operators, which are used in the example program and shown in an auxiliary window (bottom left). In this software the computational model is semantical analysis of a program and the instances are checking of context conditions, overloading resolution and type inference for

a language with parametric polymorphism. Although the user can enter examples he/she can only select one of the three given semantical analysis methods, which are then animated for the entered examples.

4 First-Order Generative Approach

In the first-order generative approach the user enters the specification of an instance of a given computational model. Then an interactive visualization of this instance is generated and the user can enter an example input as in the interactive approach.

As an example of the first-order generative approach we describe GANIMAM, our web-based generator for interactive animations of abstract machines [3]. Figure 3 shows a snapshot of such an animation. Abstract machines provide intermediate target languages for compilation. First the compiler generates code for the abstract machine, then this code can be interpreted or further compiled into real machine code. By dividing compilation into two stages, abstract machines increase portability and maintainability of compilers. The instructions of an abstract machine are tailored to specific operations required to implement operations of a source language or even better for languages of the same language paradigm.

The user can enter a specification of an abstract machine, which is then sent to the server. A CGI script on the server generates Java code and using a Java Compiler it translates this code into class files. In combination with the GANIMAM base package classes these class files form an interactive Java applet. This applet can be loaded over the internet and the user can enter machine programs, modify the layout of the different parts of the visualized abstract machines and control the animation of the execution of his abstract machine programs. The automatic layout groups the different memories around the accumulator (the chip in the middle). Source code and stacks are placed to the right, stacks to the left, local variables above and registers below the accumulator. Associated with the accumulator is an accumulator window, which shows the expressions which are currently evaluated and the definitions of the instructions or functions which are currently executed. Double clicking with the right mouse button at an instruction in the source code window, loads its definition into the accumulator window. Double clicking with the left mouse button at an instruction sets the value of the program counter to the address of that instruction, i.e. the execution of the abstract machine program is continued at that address. Clicking at a cell of a stack, heap or register opens a window. In this window the user can change the value and type of that cell. For registers only the value can be changed.

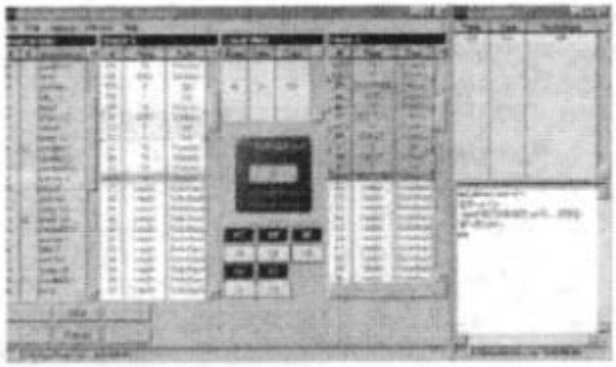

Figure 3: Screenshot of an animated abstract machine

Annotations only help to visualize principles which we know upfront. GANIMAM can also be used to detect new principles by experimenting with specifications and abstract machine programs. Such an experimental approach can be used as part of an explorative educational software. It enables students to formulate hypotheses and validate or invalidate them by changing specifications or abstract machine programs. This way he/she can learn much about the computational model, here abstract machines, but not about their generation process. The generation process is treated as a black box.

5 Second-Order Generative Approach

As in the first-order generative approach the user enters a specification of an instance of a given computational model. But in the second-order generative approach in addition to visualizing the computation also the generation process is shown as an interactive visualization.

Instead of visualizing the generation process for a certain computational model, we are currently developing a general framework to implement generators and their visualizations. Our framework combines several results of current research on algorithm animation and software visualization. As a first test case for our framework we use the implementation and visualization of a lexical analyzer generator.

Generators in compiler design usually generate tables, which control the implementation of the compiler phase together with a fixed driver. We can use this feature to generate visualizations of the generators and the compiler phases generated by them. In order to reach this goal, we develop a visualization control language GANILA, in which the generators and the drivers can be described. Then a GANILA compiler produces implementations of the generator and the driver from these specifications. In GANILA there is also the possibility to connect program points with hypermedia documents. Information, which is available at run-time at this program point, can be transferred to the document. In lit-

erate programming a connected static document is produced by the documentations of the program points. In contrast in our system the documentation of a program point can be displayed, whenever the program point is reached during the animation.

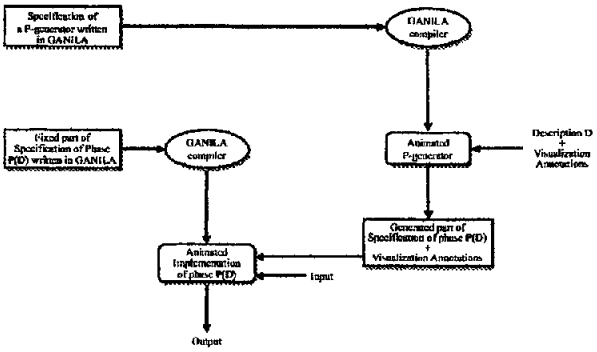

Figure 4: Generation of animated generators and compiler phases

From the extended specifications the GANILA compiler generates animations of the generator and of the generated compiler phase, see Figure 4. In addition to annotating the specification of the compiler phase, as described in the first approach, we annotate the generator and driver programs by marking program points with 'interesting events' and we define views on their data structures, i.e. among other things the generated table. For each view we determine, how it handles each event.

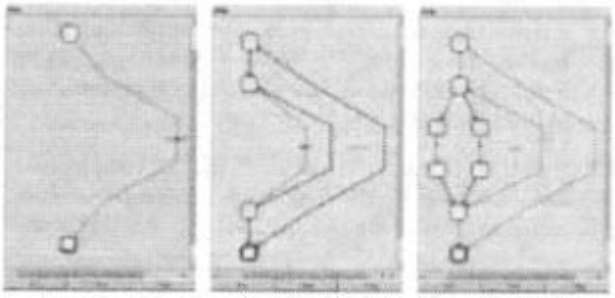

Figure 5: Intermediate and final NFAs for the RE $(a|b)^*$.

The screendump in Figure 5 shows how the generation process of an lexical analyzer is visualized. In this example, it shows how the conversion of a regular expression $(a|b)^*$ into an appropriate nondeterministic finite state automaton ($RE \rightarrow NFA$) is animated.

The generator has been integrated in an applet for visualizing generation and computation of finite automata, which is used in our electronic textbook on the theory of finite automata (see Figure 6). The GaniFA applet visualizes and animates the following algorithms:

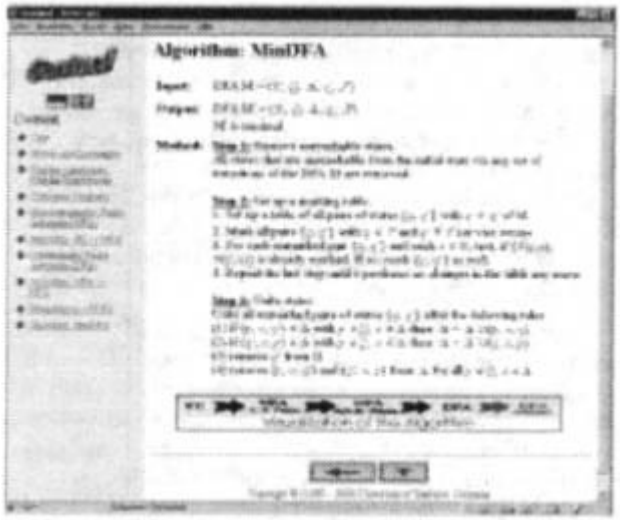

Figure 6: Screendump of the Electronic Textbook.

- Generation of a non-deterministic finite automaton (NFA) from a regular expression (RE) [11].
- Removal of ε-transitions of a NFA [8, 11].
- Transformation of a deterministic finite automaton (DFA) from a NFA without ε-transitions [8, 11].
- Minimization of a deterministic finite automaton (minDFA) [5].
- For each of the above automata generated above, the applet can visualize the computation of the automaton on an input word.

GaniFA is customizable through a large set of parameters. In particular, it is possible to visualize only some of the algorithms and to pass a finite automaton or a regular expression as well as an input word to the applet.

6 Exploration and Learner Control

Many authors argue that learning software, in which the computer appears as corrector, is discouraging and not very successful. The existing studies for this are however partially contradictory [9]. From an educational perspective there are a large number of theoretical models, empirical studies and instructional projects, which come to contradictory conclusions. On the one hand Paris and Newman argue in [7], that "in traditional instruction, the teacher is predominantly active and the students are passive. This imbalance should be reversed. Self-generated, self-organized, self-controlled and self-evaluated learning (in contrast to learning that is directed by others and controlled by the teacher) is perceived as an important, if not the essential, prerequisite for understanding, insight and discovery". On the

other hand Brown and Van Lehn [2] maintain the statement, that "self-organized learning and forms of low teacher-controlled instruction may lead to substantial conceptual deficits in students' knowledge". An answer to the question, what the better instruction model is, was given by Weinert and Helmke [10]: "An old piece of educational wisdom is that no single method of instruction is the best for all students and for all learning goals, and that even very effective instructional procedures can have deficits with respect of single criteria".

Our approach offers a way for explorative, self-controlled learning. The learner can focus on certain aspects in the generated, interactive animation and see what effects small modifications in the specification have. With the help of such observations he formulates hypotheses and checks these empirically. In the interactive approach such checkable hypotheses are restricted to the instance of the computational model. In the first-order generative approach also hypotheses about the computational model and in the second-order generative approach about the generation process itself can be checked.

More precisely, in our learning software the learner describes processes by specifications as exercise. In conventional learning software such responses, i.e. answers of an exercise, are checked for correctness, if this is possible at all. Possible errors are indicated to the learner, and he/she is requested to revise his response. In computer science, many properties of computational models can not be checked because of the halting problem. As a consequence we need alternative ways to provide feedback for the learner.

In our approach an interactive animation is produced from the response (specification) of the learner. Then the learner can test it on the basis of his/her own examples. In this way he can detect errors. There is no anonymous, all-knowing authority, which shows his errors.

Such a visual experimental approach is not meant to replace, but to enhance classical teaching of theoretical contents. The acceptance and effectiveness of such explorative learning software must be proven in practice, i.e. in instruction. In cooperation with cognitive psychologists we have done some and are currently developing new experiments for such evaluations.

7 Conclusion

In this paper we discussed how generation of visualizations can be used in educational software for computer science and related fields. For each approach we presented an implementation of an educational software system. All software can be downloaded or tested on our project homepage [4]. We have finished the development of the first three systems and these systems are publically available. The fourth system is under development, but there is a functional prototype implementation on our web page. It has been implemented using a powerful framework for interactive, web-based algorithm animations. We plan to use it not only to teach finite automata theory but also more complex generation processes in compiler design.

References

[1] Braune, B., Diehl, S., Kerren, A., and Wilhelm, R. Animation of the Generation and Computation of Finite Automata for Learning Software. In *Proceedings of Workshop on Implementing Automata* (Potsdam, 1999).

[2] Brown, J. S., and Lehn, K. V. Towards a generative theory of "bugs". In *Addition and subtraction: Developmental perspectives*, T. Romberg, T. Carpenter, and J. Moses, Eds. Hillsdale, NJ: Lawrence Erlbaum Asssociates Inc., 1982.

[3] Diehl, S., and Kunze, T. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems 16*, 7 (2000).

[4] Ganimal. Project homepage. http://www.cs.uni-sb.de/GANIMAL, 2000.

[5] Hopcroft, J., and Ullman, J. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[6] Kerren, A. Animation of the semantical analysis. In *Proceedings of 8. GI-Fachtagung Informatik und Schule INFOS99 (in German)* (1999), Informatik aktuell, Springer.

[7] Paris, S. G., and Newman, R. S. Developmental aspects of self-regulated learning. *Educational Psychologist 25* (1990).

[8] Rabin, M., and Scott, D. Finite automata and their decision problems. *IBM J. Res. Dev 3/2* (1959).

[9] Schulmeister, R. *Foundations of Hypermedial Learning Systems (in German).* Addison Wesley, Bonn, 1996.

[10] Weinert, F. E., and Helmke, A. Learning from wise mother nature or big brother instructor: The wrong choice as seen from an educational perspective. *Educational Psychologist 30* (1995).

[11] Wilhelm, R., and Maurer, D. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.