Animation of the Generation and Computation of Finite Automata for Learning Software

Beatrix Braune, Stephan Diehl, Andreas Kerren, and Reinhard Wilhelm

University of Saarland, PO Box 15 11 50, D-66041 Saarbrücken, Germany
{braune, diehl, kerren, wilhelm}@cs.uni-sb.de

Abstract. In computer science methods to aid learning are very important, because abstract models are used frequently. For this conventional teaching methods do not suffice. We have developed a learning software, that helps the learner to better understand principles of compiler construction, in particular lexical analysis. 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. We discuss principles used throughout the design of the software and give some preliminary results of evaluations of the software and discuss related work.

1 Introduction

The daily task of a computer science lecturer/teacher is to teach abstract knowledge and to promote the correct and lasting understanding of this knowledge by the listeners. For example, assume that a lecturer wants to describe the functionality of a pushdown automaton. In the most cases a large board and a sufficient number of colored chalk are available. Now he has the challenge to explain the functionality of the automaton on the basis of a small example input, a finite number of states and a stack picture. After three or four steps he begins to erase states in the stack picture, to add new states etc. The listener will have to spend more energy to reconstruct the complicated operational sequence of wiping and writing and to discover a sense in the disorder than to understand the functionality of a pushdown automaton. Thus the demonstration of such an automaton is difficult to reproduce by the learner. Visualization and graphic processing of the pushdown automaton are a possible solution for this dilemma. Because of dynamic processing animations are first choice for such technical problems. It is important to edit the information in such a way that cognitive and affective data processing of humans are addressed. The first is sequential and logical reasoning based on rules and regularities. The second thinks in pictures, uses analogies, ignores rules, reacts spontaneously and creatively. When we look at a suitable picture for an abstract term, we use both "information channels" and enable the connection of the actual term with a graphical imagination. This is also known as "integration learning" (see in addition [1]). Who understands to visualize information well, can increase the knowledge and understanding of learners with this method.

2 Animation of Lexical Analysis

The learning software "Animation of Lexical Analysis" has been developed with the authoring system Multimedia ToolBook 3.0 of the company Asymetrix and requires the free runtime version of ToolBook and Windows 3.x/95/98/NT4. The learning software covers the description of regular languages by regular expressions, theory of finite automata and the generation of a minimal deterministic finite automaton from any regular expression as described in [23]. Currently there is only a German version of the software.

As an introduction to lexical analysis, 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.

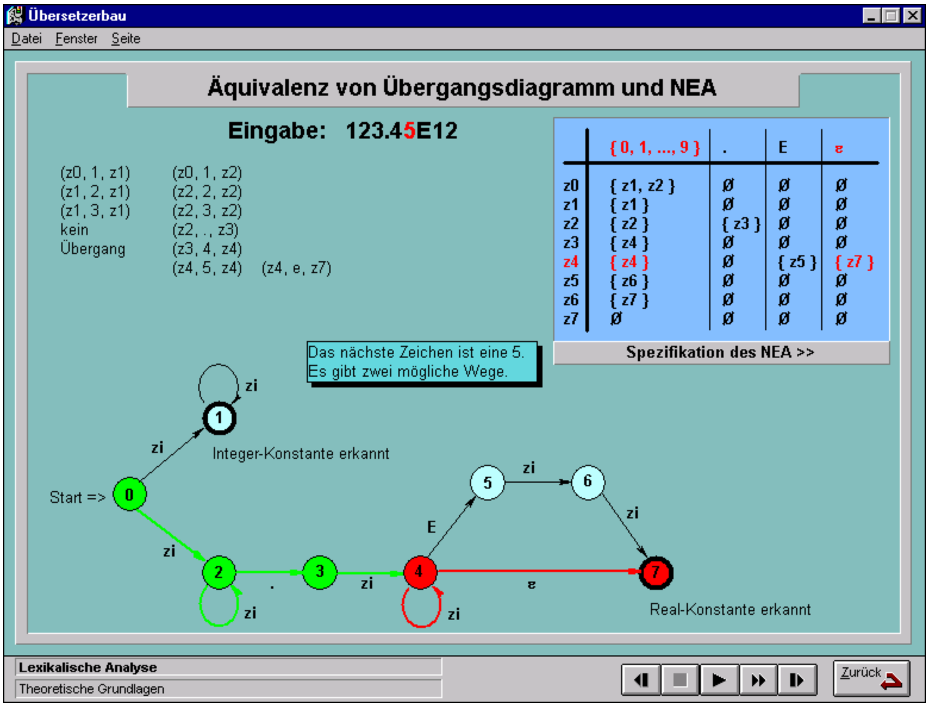
Next an overview about formal languages and an introduction to regular languages and regular expressions are given.

Then transition diagrams (TD), 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. Currently, we see a snapshot of the NFA in state $z4$. The next character to be consumed is the character *5*. Now the NFA can read the character *5* or it can do a transition via ϵ. The animation shows both possibilities. Analogously the actual path is highlighted in the TD. The two edges from node 4 to node 7 and from node 4 back to node 4 are marked red. The shadowed box in the center of the window briefly describes what the NFA and TD actually do. In a next step, the animation will color the edge labeled *E*, update the description box, mark the state $z4$ as actual state and dismiss the second transition $(z4, \epsilon, z7)$ of the NFA, because the next character to be consumed is *E*.

Three algorithms are explained with controllable animations: the transformation of a regular expression to an NFA, the transformation of an NFA to a DFA and the transformation of a DFA to a minimal DFA.

Figure 2 shows the rules of the algorithm *regular expression to NFA* that transforms a regular expression into an NFA. In an animated example it is shown how the algorithm works. It begins with a graph consisting of two nodes and one edge that is labelled with the regular expression. Step by step the suitable rule is applied (alternative, concatenation, Kleene star or parentheses) and the graph is expanded to the resulting NFA.

In the algorithm *NFA to DFA*, the original NFA and the text of the algorithm are initially shown. With each step the corresponding line of the algorithm is



Äquivalenz von Übergangsdiagramm und NEA

Eingabe: 123.45E12

(z0, 1, z1)
(z1, 2, z1)
(z1, 3, z1)
kein
Übergang

(z0, 1, z2)
(z2, 2, z2)
(z2, 3, z2)
(z2, ., z3)
(z3, 4, z4)
(z4, 5, z4) (z4, e, z7)

Das nächste Zeichen ist eine 5.
Es gibt zwei mögliche Wege.

	{ 0, 1, ..., 9 }	.	E	ε
z0	{ z1, z2 }	Ø	Ø	Ø
z1	{ z1 }	Ø	Ø	Ø
z2	{ z2 }	{ z3 }	Ø	Ø
z3	{ z4 }	Ø	Ø	Ø
z4	{ z4 }	Ø	{ z5 }	{ z7 }
z5	{ z6 }	Ø	Ø	Ø
z6	{ z7 }	Ø	Ø	Ø
z7	Ø	Ø	Ø	Ø

Spezifikation des NEA >>

Lexikalische Analyse
Theoretische Grundlagen

Fig. 1. Equivalence of Transition Diagram and NFA

highlighted and the actual nodes and edges in the graph are colored. It can be seen which nodes from the NFA build a new state in the DFA. Simultaneously to the processing of the algorithm, the new DFA is created.

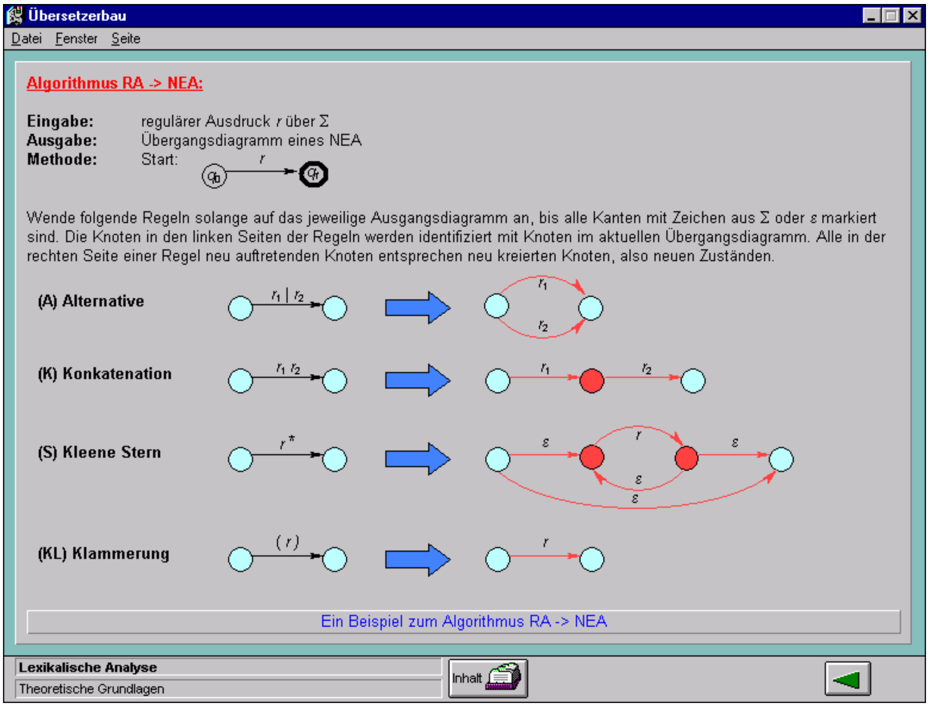
Similarly the algorithm *DFA to minimal DFA* shows the original DFA and the algorithm text. The partition classes are shown in the original graph (through coloring) and the minimal DFA is created simultaneously.

3 Design Principles

A prerequisite of implementing a good learning software is the application of good design principles. These principles were developed before the implementation of our learning software and revised during the implementation process. Some of these principles result from the research on Human-Computer-Interaction (HCI), see [17]. We propose the following guidelines:

3.1 Text

– *Font size:* If the font chosen is too small, then the user will have problems to read the text correctly, in particular sub- or superscripted text. If the font



Algorithmus RA -> NEA:

Eingabe: regulärer Ausdruck r über Σ
Ausgabe: Übergangsdiagramm eines NEA
Methode: Start:

Wende folgende Regeln solange auf das jeweilige Ausgangsdiagramm an, bis alle Kanten mit Zeichen aus Σ oder ε markiert sind. Die Knoten in den linken Seiten der Regeln werden identifiziert mit Knoten im aktuellen Übergangsdiagramm. Alle in der rechten Seite einer Regel neu auftretenden Knoten entsprechen neu kreierten Knoten, also neuen Zuständen.

(A) Alternative

(K) Konkatenation

(S) Kleene Stern

(KL) Klammerung

Ein Beispiel zum Algorithmus RA -> NEA

Lexikalische Analyse
Theoretische Grundlagen

Fig. 2. From Regular Expression to NFA

is too big, then the designer of the learning system has to solve the problem of placing enough information on the page.

- *Alignment:* Justification is not suitable for small text widths. In this case we prefer left alignment.
- *No serifs if the font is small:* Small fonts with serifs are difficult to read, because monitor resolution is not compareable with printer resolution. In computer science formulas with superscripted or subscripted letters are used frequently. This letters are very bad to read if we use a font with serifs.

3.2 Colors

- *Use few colors only:* Too many colors can irritate the user of the learning software. But colors help to direct the user's attention. Therefore colors should be used for things, to which the user's attention should be drawn.
- *Colors should harmonize:* The use of a light background color doesn't allow light font colors. The contrast between the background and the objects located on it should be high enough.
- *One fixed color for one fixed meaning:* Colors for certain links or buttons should not be changed or merged, e.g. the color blue is used for "hotword" links in our software.

3.3 Screen Arrangement

- *Main activity in one window only:* The attention of the user should be directed to one goal only. Too many windows on the screen can promote disorientation.
- *One lesson on one page if possible:* In order to avoid disorientation each basic lesson is arranged on a single page. Additional information is reached by using "hotwords" or buttons.
- *Consistent design:* Certain window areas should always be on the same place, e.g. the control buttons of the animations are consistently located in a special bar below the main window. All links must have one fixed color in case of "hotwords" or one fixed symbol in case of links to animations, definitions, etc.
- *No overloading of windows:* If a window contains very much text and many animations, then the user has difficulties to understand the important informations.

3.4 Definitions

- *Accumulate definitions:* All definitions relevant to lexical analysis are accumulated in an independent window. A first advantage is the space reduced in the main window, which is important, if the definition is very long. Furthermore the user has an overview of all definitions and can look up definitions. They can be sorted in alphabetical order or in succession of their occurence in the explanatory text.

3.5 Orientation and Navigation

- *Easy navigation and good orientation:* The actual chapter and section of the lessons are shown in a state bar located below the main window. The user can navigate to the index and from this point to another page by clicking an *Index*-button in the state bar.

3.6 Animations

- *Flexible control:* Animations should be adjustably in speed. It should be possible to execute them step by step, to stop and to reset them at each point in time. A reversed execution of animations is not in all cases meaningful and also it is frequently technically difficult to realize. However, as an alternative an Undo-operation is appropriate, that allows for a finite number of backtrack steps. Control buttons should have a well-known look, perhaps like the control buttons of a cassette recorder.
- *Clearly defined object movements:* Movements of objects should be made as directly as possible to their target, but not move over too many other objects. Several objects, that are not logical coherent, should not be moved at the same time and the movement should not be too complex and jerky.

– *Direct feedback of user actions:* Particularly in context with animations an optical feedback of the software is important, if users interact with the system. If an animation is stopped, then this stop should take place immediately and the animation should not continue a undefined amount of time.
– *Minimal memorization requirements for users:* Animations and appropriate assertions should run within a spatial and logical framework. With dynamic, automatically generated animations at runtime, this principle often conflicts with high requirement for space on the screen.
– *Spatial requirements of an animation:* More complex animations should take place on a page its own. Smaller animations should be arranged near their textual explanation.

The most challenging task when developing the learning software was to try to satisfy as much as possible of the above partially conflicting requirements.

4 Evaluation

Our target groups are students, who take a computer science course at high school, as well as students of computer science. In a pre-evaluation we left the students alone with the system. They should move independently through the graphical environment of the system and discover the learning contents on their own. With this approach we made good experiences, whereby we presupposed that the students have already been familiar with the operating system Microsoft Windows. The students got along well with the learning system, since they met a well known graphical user interface. If previous knowledge in the compiler design was available, then we noticed a better acceptance of the system, as with students, who knew still nothing about the construction of compilers. The students moved playfully through the visualizations and animations and were also able to connect these correctly with the theoretical background (definitions, algorithms, ...). Surprisingly this method worked so well that the students referred us to inconsistencies and typing errors in definitions. Students liked the optical organization of the user interface and animations.

Further presentations of our system at teachers advanced training (International Conference and Research Center for Computer Science, Dagstuhl Castle) and at the booth of the University of Saarland on the computer fair CeBIT98 and CeBIT99 in Hannover (Germany) provided positive feedback. However these measures are not yet sufficient for a serious evaluation. For this reason we co-operate at present with cognitive psychologists and develop an experiment for schools and universities, in order to receive statistically significant data of our software. For this certain aspects and characteristics of our work, e.g. the page layout or the animation control are regarded separately, while all other variable system properties remain unchanged. The use of visualization and animation is confronted to the use of the doctrine of a teacher. We still are in the preparation, so there are no results yet.

We have done the pre-evaluation with 8 highschool students (16-18 years old, Oberstufe Gymnasium) from a computer science course and got some preliminary results: 6 of them would use the system at home, 3 of them had problems with the control of the animation, none had problems with "hotwords" and only one partially disliked the design of the pages and examples.

5 Related Work

In recent years at the University of Saarland also other visualizations in the context of compiler design have been developed, including visualizations of abstract machines for imperative, logical and functional programming languages ([12], [21] and [22]). These visualizations were implemented under X-Windows (UNIX). They show the effects of the execution of machine instructions on the run time stack and heap, howewer they contain few animations. Furthermore a tool was developed for the visualization of graphs from the area of compiler design, called VCG ("Visualization of Compiler Graphs"). The VCG tool exists for several computer systems, including the Microsoft Windows system. See for this [13], [14], [15] and [16].

Another learning system developed at the University of Saarland is the "Animation of Semantical Analysis" [10], [11]. This application 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 many examples. The user can even enter his own example programs and specifications. From these inputs animations and visualizations are generated.

Also there exist a huge number of algorithm animations today, there is only a small number of fundamental work in the field. Marc H. Brown developed several algorithm animation systems, like BALSA, ZEUS, CAT, etc. These systems are frameworks, in which algorithms can be animated by annotations ("interesting events") and by definitions of graphical views ([2], [3], [4], [5] and [6]). John T. Stasko conceived the path transition paradigm and implemented it in the systems TANGO, XTANGO, SAMBA, etc., see [18], [19] and [20]. Also these systems use the concept of "interesting events". All newer versions of the above systems are complete environments, which offer some editors for the creation of views, in which the algorithms are animated. The WEB-based animation system CAT (or the newer JCAT, which is implemented in the programming language *Java*) is a complete development environment for the creation of algorithm animations in the WWW. This system offers more possibilities than ad-hoc programmed Java applets. It is possible to create algorithm animations, which a teacher can demonstrate to his students online. The interaction of the students is limited thus, but the system represents a step towards the so-called "electronic classroom". An animation can be configured in such a way that the students have the possibility to intervene interactively. They can control the animation and select other views on the algorithm. The paper [9] gives a good outline of most of the systems for algorithm animation mentioned above.

6 Conclusion

We have developed a learning software "Animation of Lexical Analysis" that helps the learner to better understand principles of compiler construction, in particular lexical analysis. 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.

In our current evaluation we would like to find out whether the presentation of the learning content through the learning software has pedagogical advantages and where the software indicates weaknesses. Questions to be answered are for example, whether animations can be controlled intuitively, where the animation controls should be placed etc. From a technical point of view the use of the authoring system MTB 3.0 is questionable. It has large restrictions and the runtime system takes up much storage space. For these reasons usually important sections of the software must be implemented in another programming language, like *C*, when using authoring systems. The advantage of the system is its simplicity of operation and programming.

A new generative approach to learning software is pursued in our current project GANIMAL, that is funded by the "Deutsche Forschungsgemeinschaft – DFG". The goal of the project is to create an explorative learning software for compiler design, in which for each compiler phase the implementation **and** the appropriate visualization or animation are generated from specifications automatically. To achieve platform independence we use the programming language *Java*. Experience with designing the learning software presented here as well as its evaluations will serve as a basis for the GANIMAL project (see also [7], [8]).

The experience gained is not only applicable to the technical area of computer science, but can be transferred also to other areas, in which processes are to be visualized, for instance the medicine, electro-technology, etc. The reader finds further information about the current level of development, as well as the newest versions of the software in the WWW [24].

References

1. A. Alteneder. *Visualize with the Help of Computers: Computergraphics and Computer Animations.* Siemens, VCH (in German), 1993.
2. M. H. Brown, M. A. Najork. *Collaborative Active Textbooks: A Web-based Algorithm Animation System for an Electronic Classroom.* SRC Research Report 142, DEC, 1996.
3. M. H. Brown. *Algorithm Animation.* MIT Press, 1987.
4. M. H. Brown. *Zeus: A System for Algorithm Animation and Multi-View Editing.* SRC Research Report 75, DEC, 1992.
5. M. H. Brown. *The 1992 SRC Algorithm Animation Festival.* In IEEE Symp. on Visual Languages, pp. 116-123, 1993.
6. M. H. Brown, R. Sedgewick. *A System for Algorithm Animation.* In SIGGRAPH '84, Computer Graphics 18(3), pp. 177-186, 1984.

7. S. Diehl, T. Kunze, A. Placzek. *GANIMAM: Generation of Interactive Animations of Abstract Maschines.* In Proceedings of "Smalltalk und Java in Industrie und Ausbildung STJA'97" (in German), pp. 185-190, Erfurt (Germany), 1997.
8. S. Diehl, T. Kunze. *Visualizing Principles of Abstract Machines by Generating Interactive Animations.* In Proceedings of Workshop on Principles of Abstract Machines, Pisa (Italy), 1998.
9. A. Hausner, D. P. Dobkin. *Making Geometry Visible: an Introduction to the Animation of Geometric Algorithms.* Computer Science Department, Princeton University, 1996.
10. A. Kerren. *Animation of the Semantical Analysis.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1997.
11. A. Kerren. *Animation of the Semantical Analysis.* In Proceedings of "8. GI-Fachtagung Informatik und Schule INFOS99" (in German), pp. 108-120, Informatik aktuell, Springer, 1999.
12. G. Kohlmann. *Visualization of the abstract P-Machine.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1995.
13. I. Lemke. *Development and Implementation of a Visualization Toolkit for Applications in Compiler Construction.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1994.
14. I. Lemke, G. Sander. *Visualization of Compiler Graphs.* User Documentation, 1994.
15. G. Sander. *Visualization of Compiler Graphs.* Technical Report, University of Saarland, Saarbrücken (Germany), 1995.
16. G. Sander. *Visualization Techniques for Compiler Construction.* Dissertation (in German), University of Saarland, Saarbrücken (Germany), 1996.
17. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 3rd Edition, Addison-Wesley, 1997.
18. J. T. Stasko. *A Framework and System for Algorithm Animation.* Computer, 18(2), pp. 258-264, 1990.
19. J. T. Stasko. *The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces.* Journal of Visual Languages and Computing (1), pp. 213-236, 1990.
20. J. T. Stasko. *Using Student-Built Algorithm Animations as Learning Aids.* Technical Report GIT-GVU-96-19, Georgia Institute of Technology, Atlanta, 1996.
21. B. Steiner. *Visualization of the abstract Machine MaMa.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1992.
22. S. Wirtz. *Visualization of the abstract Machine WiM.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1995.
23. R. Wilhelm, D. Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.
24. "http://www.cs.uni-sb.de/RW/anim/animcomp_e.html" and "http://www.cs.uni-sb.de/GANIMAL"