Diss. ETH No. 15654

Exorciser: Automatic Generation and Interactive Grading of Structured Exercises in the Theory of Computation

Dissertation

submitted to the
Swiss Federal Institute of Technology
Zurich

for the degree of
Doctor of Technical Sciences

presented by

Vincent Tscherter

Dipl. Informatik-Ing. ETH
born on June 10, 1974
citizen of Neuchâtel NE

Accepted on the recommendation of
Prof. Dr. Jürg Nievergelt, examiner
Prof. Dr. Werner Hartmann, coexaminer
Prof. Dr. Juraj Hromkovič, coexaminer

2004

Acknowledgements

I would like to express my gratitude to the following people for their support and assistance to my thesis project:

First of all, I would like to thank my supervisor *Jürg Nievergelt* for making this work possible and the opportunities he gave me to gain pragmatic insights into the field of effective computer supported instruction. Thanks go also to *Werner Hartmann* as my co-advisor for his constructive comments, his suggestions, and especially for his guidance and support during the research. I was glad that *Juraj Hromkovič* agreed to be my second coexaminer on such short notice. I would further like to thank *Reto Lamprecht*, *Christian Häfeli*, *Matthias Dreier*, *Nicolas Lefebvre*, *Alexander Imfeld*, *Pascal Erni*, *Daniel Müller*, *Thomas Briner*, *Oliver Biderbost*, and *Philipp Schlegel* who contributed to the EXORCISER System.

I am grateful to *all my friends* at ETH Zürich. Special thanks go to *Pamela Ravasio* for being a friend to share many of my interests and for her tireless moral support. I would also like to thank my family, especially my parents *Françoise* and *Ernest Tscherter* for their support and encouragement I received along my academic and non-academic life. Finally, I would like to thank my wife *Elisabeth Enggist*. I feel immense gratitude for her constant support and love throughout this time.

Contents

Abstract

For almost two centuries educators and technicians have been designing and evaluating teaching devices having the capability to coach, support learning processes and to asses the student's mastery of new skills by means of tests and exercises. Without a personal tutor, "paper and pencil" exercises follow a presentation-activation-feedback pattern that takes days before the student receives feedback on her performance. The concept of an automated tutor that conveys exercises (presentation), demands a response (activation), and then informs the student of the correctness of her response (feedback), aims to replace the linear activation-feedback phase by an interactive process that allows the student to make intelligent adjustments based on available feedback.

In a suitable domain of knowledge the techniques developed in this thesis lead to an effective automated tutor, able to reliably and quickly check details, and to provide feedback just in time—freeing the human tutor to address more demanding and thought-stimulating activities. The key idea is to design both a class of exercises and a solution space within which the student's solution is confined. This is possible in selected mathematically structured domains of knowledge where solutions and their processes can be verified algorithmically.

The validity of this approach is demonstrated in our system Exorciser, which contains 25 examples of such interactive exercises and which has been in use for several years. We have tested the usability and verified the effectiveness of Exorciser in actual instructional settings by means of a comparative study.

The opportunity to receive immediate individual feedback, together with the possibility of generating an unlimited number of exercise instances of various levels of difficulty, combined with the possibility to look at the solution of an exercise on request, makes Exorciser's teaching capabilities truly interactive and attractive. Further reasons for the student's acceptance of the Exorciser environment is its intuitive user interface and the fact that the student always maintains control of the system.

Kurzfassung

Seit Jahrhunderten beschäftigen sich Pädagogen und Techniker mit der Entwicklung technischer Systeme, welche der Vermittlung von Lerninhalten und der Unterstützung von Lernprozessen dienen. Zur stärkeren Individualisierung des Unterricht sind Übungssysteme, welche den Lernenden ermöglichen mit der Materie zu arbeiten und dabei ihren Wissensstand zu erfahren, von grosser Bedeutung.

Während es im klassischen universitären Übungsbetrieb mangels einer persönlichen Betreuung oft Tage bis Wochen dauert, bis die Studierenden die Korrekturen zu ihren Lösungen erhalten, bieten hier moderne Informations- und Kommunikationstechnologien unter gewissen Bedienungen Alternativen. In streng mathematisch strukturierten Themenbereichen lassen sich Aufgabenklassen und zugehörige Lösungsräume entwerfen, die eine automatische, schnelle und zuverlässige Korrektur der Lösungen ermöglichen. Die Lehrpersonen werden dabei von eintönigen und repetitiven Korrekturarbeiten entlastet.

Das entwickelte System Exorciser beinhaltet eine Sammlung von 25 interaktiven, selbstkorrigierenden Übungen zu Themenbereichen der Theoretischen Informatik. Studierende im zweiten Jahr testen mit Exorciser ihr Grundverständnis der aus Lehrbüchern erworbenen und in der Vorlesung vermittelten Inhalte. Exorciser wurde an der ETH Zürich über mehrere Jahre in einer Vorlesung mit durchschnittlich 200 Studierenden erfolgreich eingesetzt und evaluiert.

In Exorciser können die Studierenden Aufgaben in vielen Varianten durchspielen und sich so vertieft mit den behandelten Problemen auseinandersetzen. Die Tatsache, dass die Studierenden beim Benutzen von Exorciser ständig die Kontrolle über das System behalten, sowie die Möglichkeit, unmittelbar individuelle Korrekturen und Kommentare zu jedem Lösungsschritt zu erhalten und jederzeit die Lösung der generierten Aufgaben einsehen zu können, tragen wesentlich zur Akzeptanz des Übungssystem bei.

Chapter 1

Computers and Education: Theories, Systems, Guidelines

> "One of the most important tasks of the teacher is to help his students. This task is not quite easy; it demands time, practice, devotion, and sound principles."
>
> Polya (1945)

The quality and capabilities of information and communication technologies (ICT) continue to improve at a steady pace, making more and more powerful tools available to the dedicated user who is willing to learn how to use them. In the last half century much research (also in education) was driven by technological advancement rather than by needs. Whilst developers of teaching devices have created some of the most provocative and stimulating ideas in the history of education, "the purpose of computing is insight, not numbers." Hamming's wise quote reminds us that technical advances are a means to an end, not an end in itself.

For almost two centuries educators and technicians have been designing and evaluating devices with the ability to teach and support learning. Barbey (1971) refers to patents dating back to 1809. Thorndike (1912, p. 165) wrote: "If, by a miracle of mechanical ingenuity, a book could be so arranged that only to him who had done what was directed on page one would page two become visible, and so on, much that now requires personal instruction could be managed by print." Having Thorndike's vision in mind, Sidney Pressey (1926) built mechanical teaching devices for automatic testing and scoring. Figure 1.1 shows one of these. Pressey believed that teachers are burdened by routine of drill and practice. He stated (p. 374) that mechanical teaching devices could: "Lift from her [the teacher's] shoulders as much as possible of this burden and make her free for those inspirational and thought-stimulating activities which are, presumably, the real function of the teacher."

Personal teaching devices allow students to move at their own pace and receive individual feedback—in contrast to mass instruction where admittedly a teacher is barely able to simultaneously supervise adequately more than a few students. Whereas personal interaction may be the best way to motivate a student, to convey the main ideas, and to correct conceptual

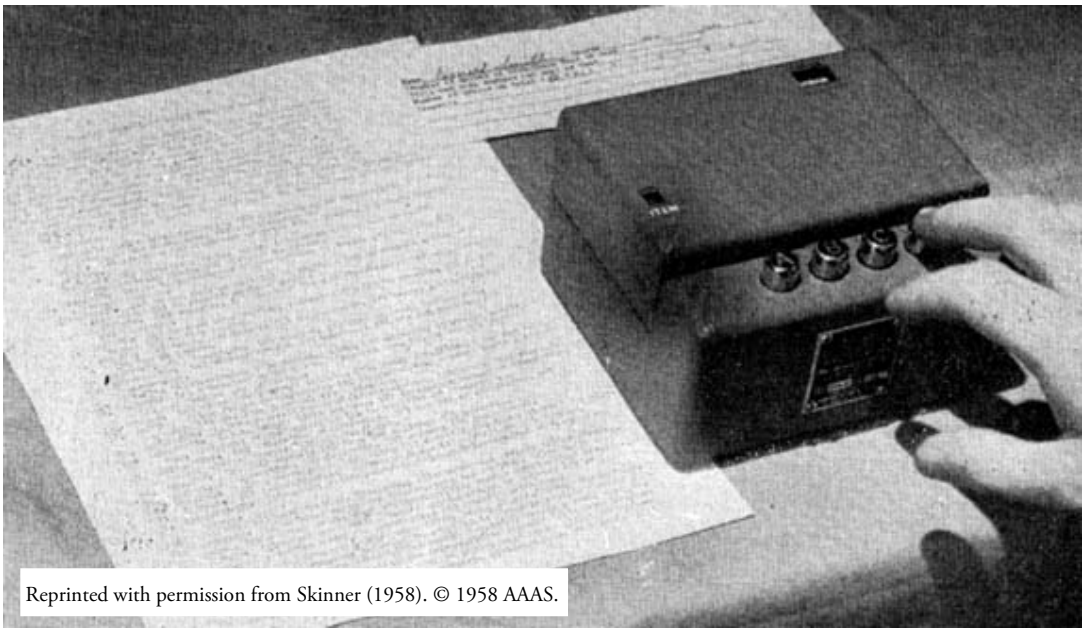


Figure 1.1: Pressey's self-testing machine. The device directs the student to particular item in a multiple-choice test. The student presses the key corresponding to his choice of answer. If his choice is correct, the device advances to the next item. Errors are totaled. (Skinner, 1958, p. 970)

misunderstanding, it is a waste of human potential for teachers to check the details of exercises of dozens or hundreds of students. Computers can do this job far more reliably and quicker. Feedback delivered within a lesson is much more effective than deferred feedback, e.g. in form of revised exercises (van Houten, 1980).

We can take advantage of the computer's power to set up new instruments for extending our senses and intuition on a subject matter.

1.1 Educational Theories and CAI

Psychologists have different ideas on what teaching devices work best to aid student learning. With respect to historical development, at least three fundamental learning theories applicable to computer aided instruction (CAI) can be identified. CAI will be interpreted in a broad sense: in principle, any way in which a computer can assist instruction will be considered to belong to this field. In practice, we will limit our discussion to the interactive use of computers, where the device conveys exercises (presentation), demands a response (activation), and then informs the student of the correctness of her response (feedback).

Behaviorism and Programmed Instruction One of the oldest educational theories of animal and human learning is the behavioristic conception of learning by reinforcement (e.g. Skinner, 1954). This theory relies on objectively observable behaviors only. Behavior theorists define learning as nothing more than the acquisition of new behavior.

The first generation of CAI systems in the mid sixties was in direct continuation of Pressey's and later Skinner's (1958) mechanical teaching devices. This technology finally provided the flexibility whose absence had stood in the way of successful mechanical realization. Programmed Instruction is a technique for presenting a domain of knowledge to students who can work through it at their own learning speed. It consists of a network of statements—*frames*, or, in today's terminology, *learning objects*—and tests, which direct the student to the next level of difficulty depending on the individual's performance. Clearly this is a behaviorist approach to learning. The reinforcement mechanism—right or wrong feedback—is easy to implement, but the restriction to a single teaching strategy can hardly be justified. "Programmed instruction, with its rigid control of the dialog by the program, should yield to (or at least not exclude) modes where the user controls the dialog, such as exploration and simulation" (Nievergelt, 1975).

Cognitivism and Intelligent Tutoring Systems In sharp contrast to the passive view of learning adopted by the behaviorist school, a different view emerged: cognitivism. In the opinion of cognitivists, restrictions to purely observable phenomena limited the applicability of behaviorist learning theories. "Cognitivist learning theories moved away from stimulus/response experiments, exploring mental processes and developing approaches to model acts of thinking and learning" (Dara-Abrams, 2002, p. 15).

In the 1970s, researchers were looking for new educational paradigms to take advantage of breakthroughs in computer technology. The origin of Intelligent Tutoring System (ITS) was an attempt by Carbonell (1970) and others to combine Artificial Intelligence and CAI. Carbonell's system SCHOLAR (for teaching the geography of South America) introduced the typical architecture of ITS which consist of at least four subsystems: (1) an expert, (2) a tutor, (3) the user interface, and (4) a model that represents the student in the system. By comparing the student's answer to an exercise with the expert's optimal solution, the artificial tutor is in a position to provide meaningful feedback and to guide the student on the shortest track to the desired solution. This cognitivistic approach,

although promising, works only as long as the tutor recognizes the deviation of the student's response to the expert's position correctly and a corresponding feedback has been foreseen by the system's designer.

Constructivism The third educational theory—the constructivist theory of learning—emphasizes a problem-centered, explorative and informal approach to learning. The purpose of the teacher or the teaching device is not to cover material but to help the student *uncover* the facts and ideas in a domain of knowledge.

Rather than focusing on a purposive guidance, constructivistic approaches grant full control to the student. Guidance along a specific task is of course provided on request. The most widely known representative of constructivism is the LOGO project, started in the early seventies by Seymour Papert (1970). He developed the programming language LOGO—from which emerged the well known Turtle Geometry—to encourage rigorous thinking about mathematics. Papert wanted it to be accessible to children and to be able to express easily procedures for simple tasks familiar to children. LOGO soon became the language of the elementary school computer literacy movement.

1.2 Pioneering Systems

Papert's LOGO and Carbonell's SCHOLAR cited above are only two example on how to enhance classroom instruction by adding interactive content and make use of "new" technology. In this section we give a brief summary of those pioneering systems that have inspired our work.

The Early Pioneers The PLATO project (started in 1959 at the University of Illinois at Urbana-Champaign) and TICCIT (started in 1967) were the first two large-scale projects for the use of computers in education. PLATO was developed as a technological solution for delivering individualized instruction that was designed to give students control, i.e. to allow them to navigate through the courseware in any directions, to skip ahead, or to ask the system for more detailed explanation. The PLATO system with several thousand terminals served undergraduate studies as well as elementary education, at schools and universities across the US.

It is worth remembering that with the exception of the Web, the technology in use today was introduced more than thirty years ago. Xerox PARC was the birthplace of many aspects of modern computing, including direct manipulation interaction techniques such as how objects and text

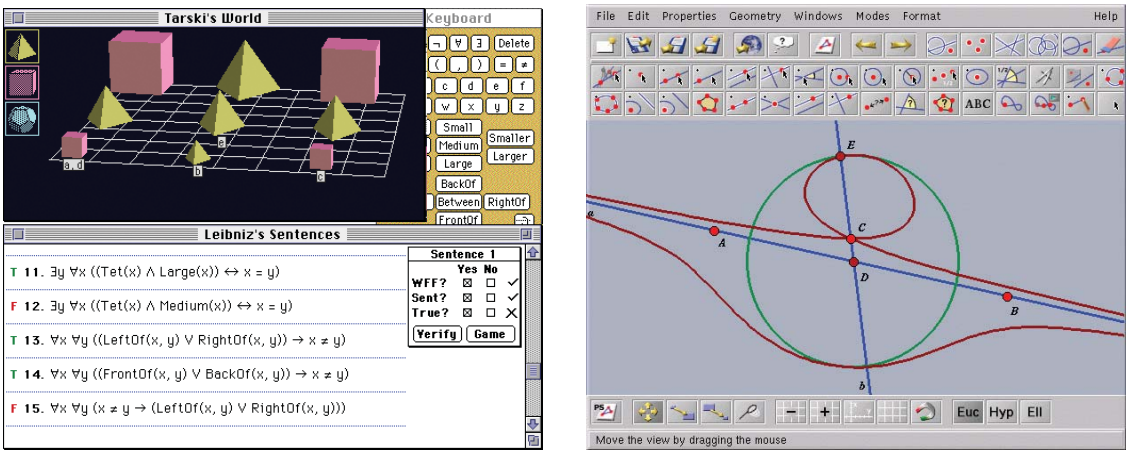


Figure 1.2: Recent developments: TARSKI's World and CINDERELLA

are selected, opened and copied, overlapping windows, WYSIWYG text editors, the laser printer, the desktop computer, and last but not least with the SMALLTALK language, object oriented programming.

Most of these innovations can now be found in classrooms albeit in new clothes: the object oriented programming language SMALLTALK has become SQUEAK. The success of LOGO Turtle was renewed with KAREL (Pattis, 1995), KARA (Reichert, 2003) and other of the its successors, and the functional range of PLATO and TICCIT could still compete with modern learning management systems.

Recent Developments TARSKI's World (Barwise and Etchemendy, 1997) and CINDERELLA (Richter-Gebert and Kortenkamp, 1999) are two recent developments that have significantly influenced our own investigations in the field of CAI. Two characteristics of both of these systems are (1) that students really feel involved with the subject matter due to a high level of interaction, and (2) the possibility to solve exercises with a high level of cognitive abstraction similar to "paper and pencil" exercises. These two systems allow a student—in absence of a personal tutor—to test her mastery of basic concepts while the system's fine-grained feedback pinpoints the source of any misunderstanding.

The goal of TARSKI's World is to introduce students to the theory of first-order logic. Using this program students quickly master the meaning of operators and quantifiers. TARSKI's World allows students to build three-dimensional worlds, and to describe them in first-order logic. They can evaluate the sentences in the constructed world and if their evaluation is incorrect, the program provides them with a kind of game that leads them to understand where they went wrong.

CINDERELLA is a well known software for doing geometry on the computer. This software includes a theorem prover and offers interactive exercises, where the correctness of a solution can be checked automatically, and independently of the student's construction. Teachers can create their own exercises with CINDERELLA, and publish them on the web immediately.

The effectivity of teaching devices that help students understand a subject matter depends on a multitude of factors. We focus in the next section on five guidelines of "axiomatic" character for the design, and implementation of interactive teaching devices.

1.3 Guidelines

1. **Convey challenging exercises that demands creativity** The exercises conveyed by the teaching device have to be relevant and also somehow demanding. "In order to retain the student's attention for long periods, the dialog must be interesting and pleasant; to assure that he learns, the dialog must be somewhat demanding" (Nievergelt, 1975). Likewise, challenging exercises have more than one correct response, and solutions are systematically constructed rather than selected from a set of alternatives. Of course, it is much more difficult to build a machine able to evaluate a composed response than to score multiple-choice answers.

2. **Provide feedback and guidance on request** Similar to a skillful human tutor, the automated tutor helps the student to find the right answer. This can be achieved through well thought-out guidance along the way and by hinting, prompting or making suggestions based on an analysis of the student's response. While guidance tells the student the most likely ways to achieve her goal; feedback tells her whether she is on track or off track in a way that enables her to self-adjust. Feedback and guidance are provided on request, which has advantage that the student controls the dialog and not vice-versa.

3. **Put the student in control** Unlike lectures and textbooks, teaching devices can induce continuous interaction between the student and the device's representation of subject matter. However control over the dialog should remain in the hands of the student.

4. **Provide the solution on request** It is important that students experience success at solving exercises. Automated error recovery

and the system's ability to refine a student's solution step-by-step is therefore important as it reduces the number of exercises a student leaves unfinished, and thereby increases the chance to succeed.

5. **Provide a highly usable system** In educational software *ease of use* is of great importance. It cannot be assumed that the student is a trained operator. She wants to learn the subject matter of interest, and not an interface. It is therefore worth the designer's time and effort to ensure teaching devices are as intuitive to use as possible, and to check its usability in real, large-scale educational settings.

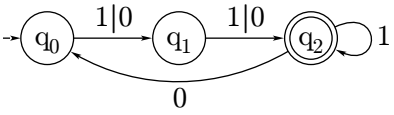
1.4 Domain of Knowledge

"The success of a teaching device depends on the material used in it. Machine material must be self-contained and wholly adequate" (Skinner, 1958, p. 971). Designing and implementing educational software systems is a challenging task. Automatic grading of exercises whose solution is not restricted to a small set of options can only work in a highly structured, formalized, mathematically tractable domain of knowledge, e.g. formal domains of knowledge like algebra, geometry, or logic, and domains of knowledge whose terminology is strongly formalized such as chemistry.

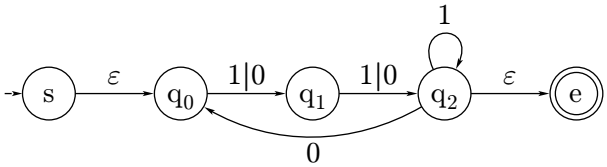
In the following we focus on selected topics in the *Theory of Computation*. Theory of Computation is a fundamental and persistent part of the computer science curriculum, and it meets the requirements set.

Example: Translating Finite Automata to Regular Expressions Given the state diagram of an arbitrary finite automaton M a student has to construct a regular expression R that describes the language $L(M)$ accepted by M. The algorithm consists in converting M into a *generalized nondeterministic finite automaton* (GNFA), then to apply step-by-step appropriate rules to convert the GNFA into a regular expression (e.g. Sipser, 1997). Example:

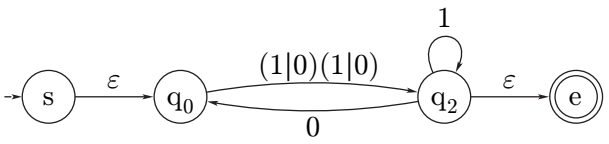
the finite automaton M



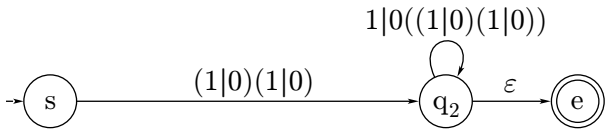
an equivalent GNFA to M



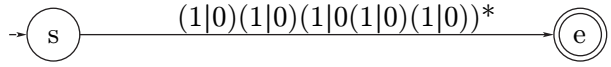
step 1



step 2



step 3



The resulting expression depends on the order in which states are removed and how the GNFA is transformed in each step. Another correct solution is $(0|1)((0|1)1^*0(1|0))^*(0|1)1^*$. The fact that there is more than one correct solution results in a more attractive exercise to students.

Grading exercises with the possibility of several different correct outcomes is a tedious and error-prone activity for humans, but error-free for computers.

1.5 The Exorciser Project

EXORCISER is an interactive learning environment for exercises in a standard introduction to the Theory of Computation (e.g. Sipser, 1997; Taylor, 1998; Hopcroft and Ullman, 1979; Wegener, 1993). The aim of the project has been to explore all avenues that promise effective automatic support for large-enrollment courses in a formalized domain like Theory of Computation. The term *automatic support* in this context means interactive automated exercises designed to substitute exercises previously done with paper and pencil. Our approach involves the automatic generation of exercises and of pertinent comments to every step during the student's solution process.

Context and Motivation The convergence of several trends caused us to start the project to automate exercises in a compulsory introductory course on the Theory of Computation in the computer science curriculum of the Swiss Federal Institute of Technology, ETH Zurich. Firstly, the number of students entering our computer science curriculum rose from 126 in 1995 to 340 in 2001. This contrasted with only a modest growth in faculty and teaching assistants. Secondly, a renewed surge of interest in computer-based educational technology encouraged the use of technology

to alleviate a lack of resources. Thirdly, information and communication technologies had become ubiquitous. Nowadays, almost all computer science students have their own notebook.

Scope and Contributions The aim of this thesis is the development of an automated tutor for the support of teaching some fundamental concepts of computer science that are relevant to the second year of university studies. Therefore we have developed concepts and techniques for generating exercises and interactively checking and commenting every step of the solution process. The key idea is to design both a *class of exercises* and a *solution space* within which the student's solution is confined. This enables the automated tutor to check each step of the solution process, to comment on it meaningfully, and to correct it if the student wishes so. The feedback provided to the student ranges from indicating the incorrectness of the proposed solution to a full correction of the solution.

The validity of this approach is demonstrated in our system Exorciser, which contains 25 examples of such interactive exercises and has been used for several years. The developed system is the first high quality learning system for the considered topic that gives the student the possibility of controlling the actual degree of interactivity and the hardness degree of generated exercises with respect to the given teaching aim. We have tested the usability and verified the effectiveness of Exorciser in actual instructional settings by means of a comparative study.

The opportunity to receive immediate individual feedback, together with the possibility of generating an unlimited number of exercise instances of various levels of difficulty and the possibility to look at the solution of an exercise on request, makes Exorciser's collection of exercises truly attractive. Further reasons for the student's acceptance of the Exorciser environment is its intuitive user interface and the fact that the student always keeps control over the system.

Chapter 2

Interactive Exercises: Examples

"I hear and I forget; I see and I remember; I do and I understand"—this old and wise proverb reminds us that learning is more than a matter of acquisition of information; it is the mastering of new skills. Interactive exercises allow a student—in the absence of a personal tutor—to test her mastery of basic concepts while the system's fine-grained feedback pinpoints the source of any misunderstanding. The effectiveness of such systems depends on a multitude of factors. We focus on three essential ones: (1) the intensity of interaction, (2) the exercise's level of cognitive abstraction, and (3) the flexibility in formulating answers.

An intensive interaction with the system's representation of subject matter is an essential factor in the mastering process. Interaction intensity (Schulmeister, 2002b) ranges from passively *listening and viewing* to actively *constructing objects and receiving meaningful feedback*. The student's motivation increases with higher interactivity levels—so Schulmeister's hypothesis. Laurel (1993) defines this type of interaction as follows: "You either feel involved in the computer representation or you do not. The crucial point is the ability to interact with the representation, and not how often the software feigns communication with you." Highly interactive software is rare in higher education. The reason for this is due to the difficulty of developing such software. We believe the effort is justifiable, if the software covers fundamental and persistent parts of the curriculum.

The second aspect is the exercise's level of cognitive abstraction. Bloom (1956) created a taxonomy for categorizing levels of cognitive abstraction of questions that commonly occur in educational settings. The taxonomy ranges from *recall of information* (knowledge) to *make judgments about the value of ideas or materials* (evaluation). Higher levels of cognitive abstraction are certainly a lot more demanding for the student and the exercise programmer, but they are also more interesting.

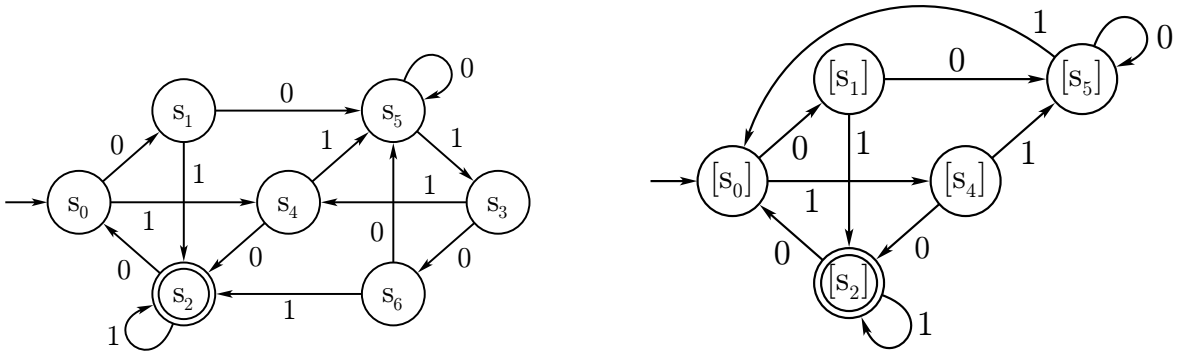
Finally, it is a challenge to design exercises that give a student a fair amount of flexibility in formulating her answer, but remain algorithmically tractable. There are fundamental differences in the kind of exercises human and automatic tutors grade well. Computers can hardly grade creative exercises. On the other hand, computers often do a better job when

the grading process is intricate, time-consuming, and thus error-prone.

The potential of an automatic tutor to guide the student is related to all three aspects mentioned. Let us present two sample exercises, in order to illustrate this relation. The first exercise *State Minimization* deals with working out a finite automaton's minimal equivalent; the second exercise presented in this chapter is about *Markov algorithms*. Markov algorithms are a deterministic model of computation where an input string is transformed into an output string according to substitution formulas that are scanned sequentially.

2.1 State Minimization

Finite Automata are an important model of computation in computer science. Given the state diagram of a deterministic finite automaton (DFA) A the student has to construct the state diagram of a DFA M with the smallest number of states that accepts the same regular language as A. It turns out that there is only one—up to isomorphism—of minimal size. See Figure 2.1 for an example.



(a) DFA A (b) Minimal DFA M

Figure 2.1: Example of state minimization

An algorithm for state minimization consists in finding sets of equivalent states, i.e. states that are indistinguishable by any word fed to the automaton. Hopcroft (1971) presents an efficient $O(n \log n)$ algorithm, where n is the number of states of the given DFA.

To prove two states equivalent, it suffices to test all inputs of length shorter or equal to the number of states in the automaton. The simplicity of this approach was the reason for us to teach our students a $O(n^3)$ dynamic programming algorithm rather than Hopcroft's efficient one. The algorithm gradually identifies distinguishable state pairs and records for

those in a state pair table a distinguishing shortest word, called a *witness*. Dynamic programming has several advantages: (1) as a runtime data structure we only need a state pair table, (2) the rules to fill in the table are easy to understand, and (3) it is possible to grade the partially filled table automatically in order to provide meaningful feedback at any time the student request it.

Algorithm: State Minimizations

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the unique start state, and $F \subseteq Q$ is the set of accepting states.
In a table of all state pairs: mark all pairs that can be distinguished by the empty word ε, i.e. one state is accepting, the other is not accepting. This step gives us the initial partition π_0 of the equivalence relation.
repeat
For each unmarked pair (q, r) $q, r \in Q$, $q \neq r$ check for all $a \in \Sigma$, if the pair $(\delta(q, a), \delta(r, a))$ has been marked distinguishable. If so, mark (q, r) distinguishable with shortest witness $w = aw'$, where w' is inherited from $(\delta(q, a), \delta(r, a))$.
until no more pair can be marked in one pass

The algorithm runs in time $O(|Q|^3|\Sigma|)$ and gradually identifies distinguishable state pairs. Thereby the algorithm records for distinguishable state pairs a distinguishing *shortest witness* as exemplified in Figure 2.2.

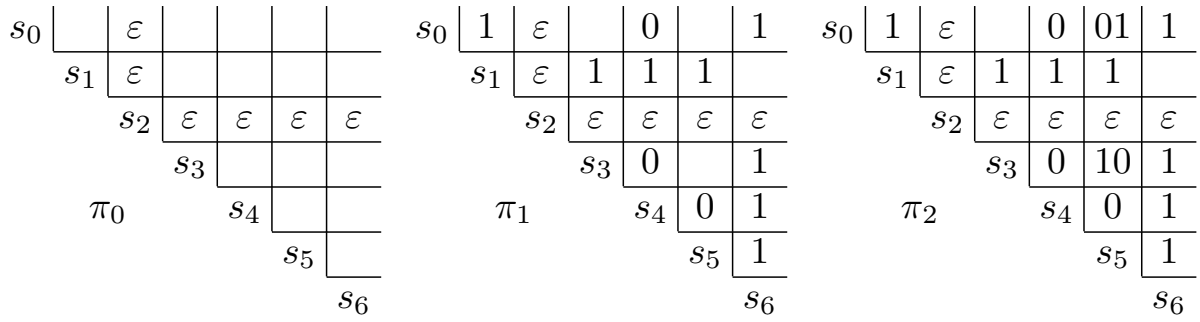


Figure 2.2: Step-by-step state minimization

The initial ε-induced partition π_0 is $\{\{s_0, s_1, s_3, s_4, s_5, s_6\}, \{s_2\}\}$. As the result of the first iteration, we get $\pi_1 = \{\{s_0, s_3, s_5\}, \{s_1, s_6\}, \{s_3\}\}$. In the second iteration, when computing the entry for (s_3, s_5) notice that $\delta(s_3, 1) = s_4$, $\delta(s_5, 1) = s_3$. Since s_3, s_4 have already been proven distinguishable by $w' = 0$, the pair (s_3, s_5) is distinguishable by $w = 10$. Checking the last unmarked pair (s_0, s_3) yields to no new equivalent states as

$\delta(s_0, 1) = \delta(s_3, 1) = s_4$ and $\delta(s_0, 0) = s_1$, $\delta(s_3, 0) = s_6$, but (s_1, s_6) have not yet been proven distinguishable. This terminates the process with the partition $\pi_2 = \{\{s_0, s_3\}, \{s_1, s_6\}, \{s_2\}, \{s_4\}, \{s_5\}\}$. In other words s_0, s_3, respectively s_1, s_6 are equivalent.

State pairs for which no witness was recorded at the end of the algorithm are equivalent and can be merged. Recording witnesses is not a necessity for finding equivalent states. A simple checkmark '✓' marking already identified distinguishable pairs would do the job. The reason is a pedagogical one: It is much easier to argue with the student on a wrong-recorded witness rather than on a misplaced '✓'.

Interactive Exercises for State Minimization We have designed the learning environment for state minimization (Figure 2.3) according to the algorithm presented above. After starting the exercise the student sees a randomly generated DFA and the state pair table needed for the dynamic programming algorithm. Four levels of difficulty are available: (1) *Beginners*, (2) *Advanced*, (3) *Expert*, and (4) *Custom*. The first three levels differ in the automaton's total number of states. All generated automata contain at least two equivalent states. In the custom level, the user can enter manually any DFA.

User Guidance and Grading Facilities The system plays a role as advisor, grader, or specialist, depending on the situation. It is not easy to find the right degree of guidance. A student may ask for strong guidance at every step of the process, or merely a verification of the final result, depending on his confidence and degree of understanding. We leave it therefore to the student to request feedback. The grading systems checks for the following three flaws:

1. Wrong witnesses, i.e. any wrong entry in the table. An example of an error notification generated on request is given in Figure 2.3.

2. Witnesses not of minimal length: in our example '1' would distinguish s_0 and s_2 as $\delta(s_0, 1) = s_4$ and $\delta(s_2, 1) = s_2$; s_2 and s_4 are distinguishable by ε. But ε is a shorter witness to distinguish s_0 and s_2.

3. Missing table entries: We do not consider missing entries as a student's error. The student may have simply requested feedback for a partially solved exercise.

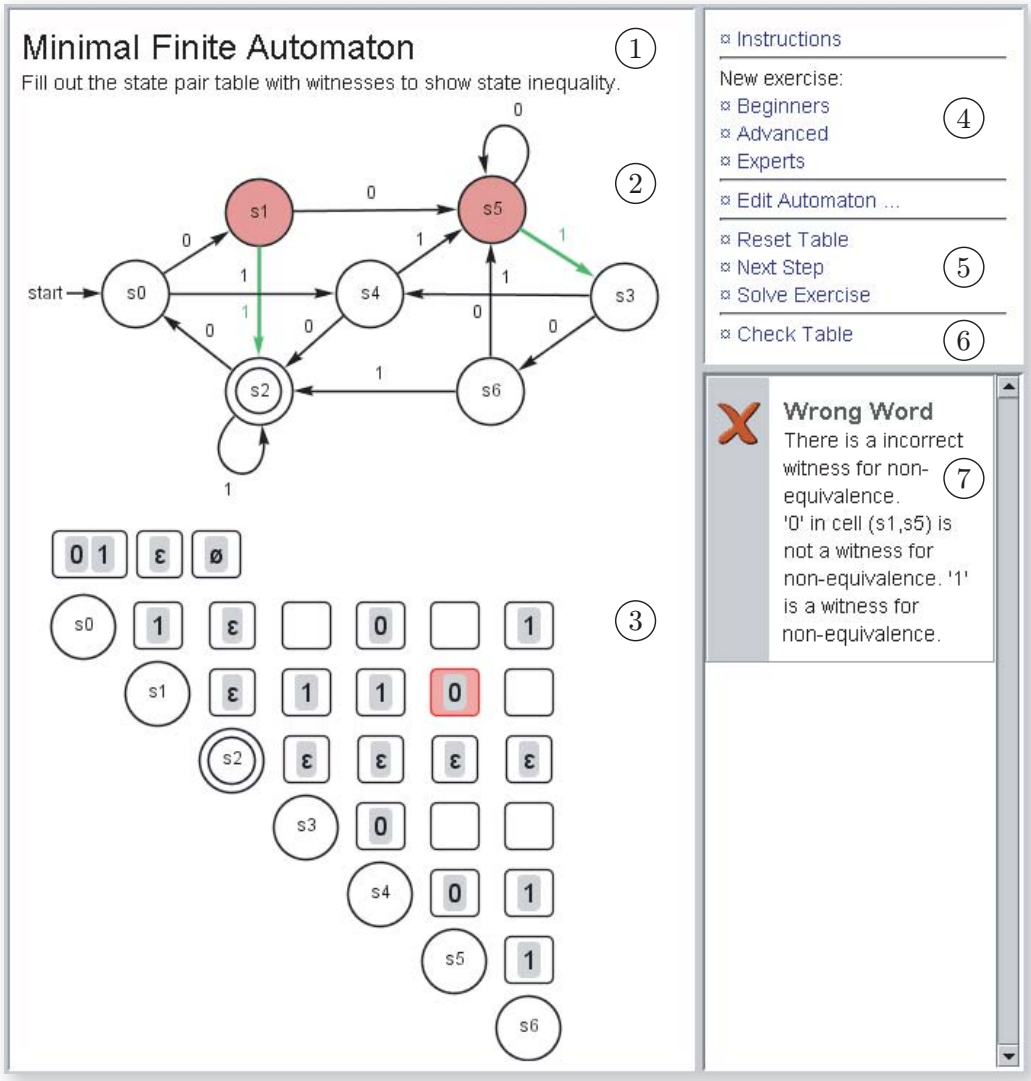


Figure 2.3: The interactive state minimization exercise (1. exercise description, 2. DFA, 3. state pair table, 4. exercise generator 5. solution generator, 6. grader, 7. error notification)

On each grading cycle the system selects the "worst" error and comments on it meaningfully, i.e. the program highlights the errors in the table and proposes to the student a correct solution. Apart from grading on request, the program informs the student immediately if one of the following special events occurs:

1. Circular references: A distinguishing witness w for a state pair (q, r) is added to the table simply by dragging the witness's first letter a into the table while the rest of the witness is inherited from $(\delta(q, a), \delta(r, a))$. If the inherited part of the witness can not be resolved due to circular dependencies within the cells, then the witness is interpreted as a regular set and the student is immediately notified (Figure 2.4).

2. Equivalent states: State pairs for which at the end of the algorithm no witness was recorded are equivalent and can be merged. This is denoted by the empty set symbol '$\varnothing$' with the meaning that no witness distinguishing this state pair exists. The program merges $\varnothing$-marked state pairs automatically and displays them in a *reduced finite automaton* to the right of the original automaton. An already merged state pair that commonly points to another equivalent, but not yet merged state pair, will causes the reduced automaton to become temporarily non-deterministic (Figure 2.5). At the end, the minimal DFA remains.

The Solution Generator Finally, a solution generator completes the functional range of the state minimization exercise: at the user's request the solution generator either hands over a solution for the current exercise, or performs the next step in the algorithm. We take advantage of the iterative nature of dynamic programming algorithms that allows the generator to resume the algorithm starting from any partial solution and to run it to completion. If the table contains wrong entries, or entries of non-minimal length, they are previously removed.

The solution generator, together with the possibility of generating an unlimited number of exercise instances of various levels of difficulty and the opportunity to receive immediate individual feedback, makes this interactive exercise attractive.

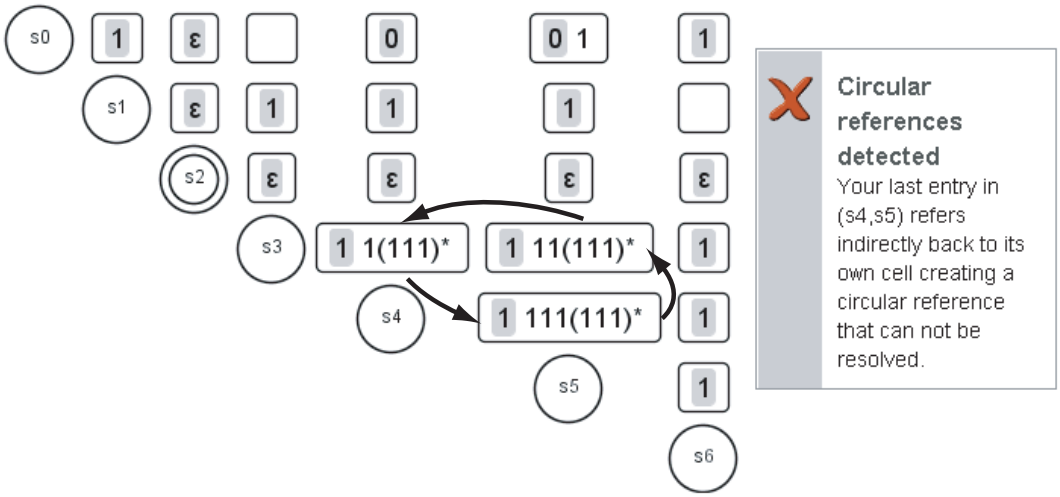


Figure 2.4: Circular references detected in the table.

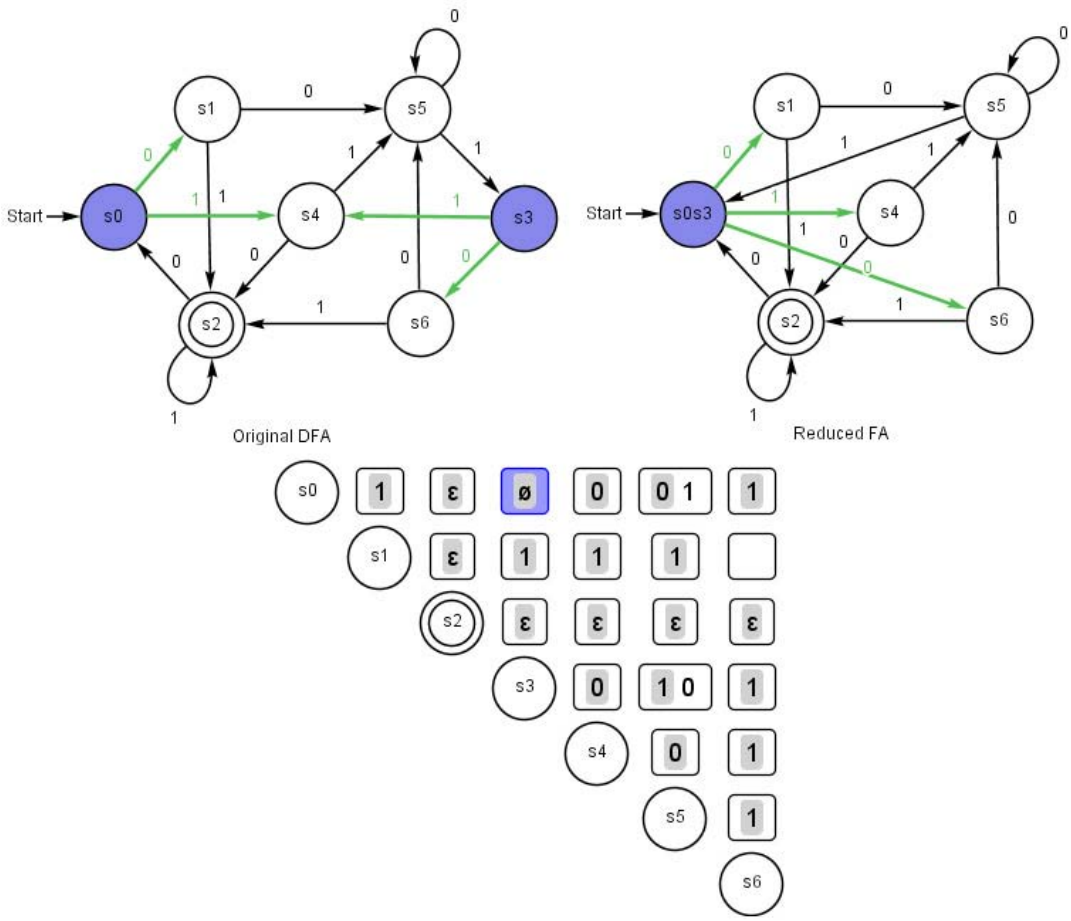


Figure 2.5: Visualization of identified equivalent states. s_0 and s_3 have been marked compatible. A close look at the automaton to the right shows that states s_1 and s_6 are also equivalent as $\delta(\{s_0, s_3\}, 0) = \{s_1, s_6\}$.

2.2 Markov Algorithms

Markov algorithms (Markov, 1962) are a computer architecture for memory with sequential access only, where computation is modeled as a deterministic transformation of an input string into an output string according to substitution formulas that are scanned sequentially. This model is equivalent to universal models of computation such as *Turing Machines*, or *Lambda Calculus*.

Programming string manipulation algorithms is helpful for students to gain insight into the basic theory and constructions of models of computation. Let us first introduce EXORCISER's interactive learning environment for Markov algorithms by means of the *String Reverse* problem.

The String Reverse Problem Construct an algorithm R over an alphabet $A = \{0, 1\}$, changing every word in A^* into this word's reversal. Example:

$$R(0010) = 0100.$$

An algorithm R can be written as follow, where $\alpha \notin A$ is an auxiliary letter introduced by the programmer. We call this letter a *marker*.

$$\begin{array}{lrl}
1. & \alpha\alpha\alpha & \rightarrow \alpha\alpha \\
2. & \alpha\alpha 0 & \rightarrow 0\alpha\alpha \\
3. & \alpha\alpha 1 & \rightarrow 1\alpha\alpha \\
4. & \alpha\alpha & \rightarrow \cdot \\
5. & \alpha 00 & \rightarrow 0\alpha 0 \\
6. & \alpha 01 & \rightarrow 1\alpha 0 \\
7. & \alpha 10 & \rightarrow 0\alpha 1 \\
8. & \alpha 11 & \rightarrow 1\alpha 1 \\
9. & & \rightarrow \alpha
\end{array}$$

The algorithm's execution (adapted from Markov and Nagorny, 1988, p. 132) is illustrated in Figure 2.6 on the input $w = 0010$. At first (Rule 9) the marker α appears in front of the input w. It "attaches itself" to the word's first letter and "drags" it to the right, across all letters of the alphabet A (Rules 5 to 8). Then in front of the word obtained in such manner, there again appears α (Step 4 in Figure 2.6). If the word w has not yet been used up, then α "attaches itself" to its second letter and carries it across all letters of A, but stops short of the nearest marker α. This cycle is repeated until w is entirely exhausted. By this time (Step 9), the word w has already been reversed, but appears interspersed with α's. In the following clean-up phase (Steps 10 to 18), these markers α are

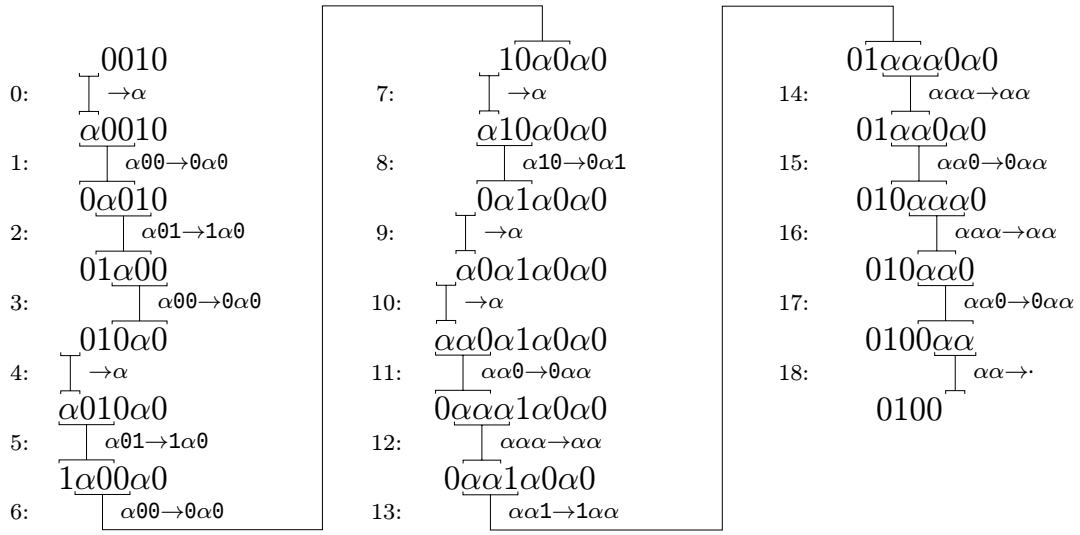


Figure 2.6: Execution of the string reversal algorithm R on input $w = 0010$

erased. Erasing all remaining occurrences of α begins with the appearance of α (Step 10), once more in front of the word being transformed. It turns out that there are now two of them next to each other. This pseudo marker $\alpha\alpha$ "runs" through the word from left to right, jumping past letters of A (Rules 2 and 3) and erasing each marker α (Rule 1). Having reached the end of the word, $\alpha\alpha$ disappears (Rule 4, a terminating rule as indicated by $\rightarrow\cdot$), and with this R's execution ends (Step 18).

With this example of string reverse, we now introduce the interactive exercise, which offers two powerful tools that help the student constructing algorithms: (1) an *execution tracer*, and (2) an *exercise grader*.

The Interactive String Reverse Exercise After starting the exercise, the student sees the problem description and an empty algorithm scheme. The algorithm editor (Figure 2.7, 2.) is *syntax-driven* and therefore intuitive to use. The student needs to supply an algorithm but may be unfamiliar with the syntax to use. The mouse-controlled editor allows the student to enter an algorithm in the correct syntax only. Note, the syntax we use in our algorithm editor differs slightly from Markov's original syntax: (1) ε denotes the empty word, and (2) '$\dashv$' stands for '$\rightarrow\cdot$' (terminal substitution).

Error Detection and Recovery It is well known that, in general, the problem of deciding whether an algorithm is correct is undecidable. In consequence, how do we know if the student's algorithm performs in the manner specified? First, we encourage our students to use the execu-

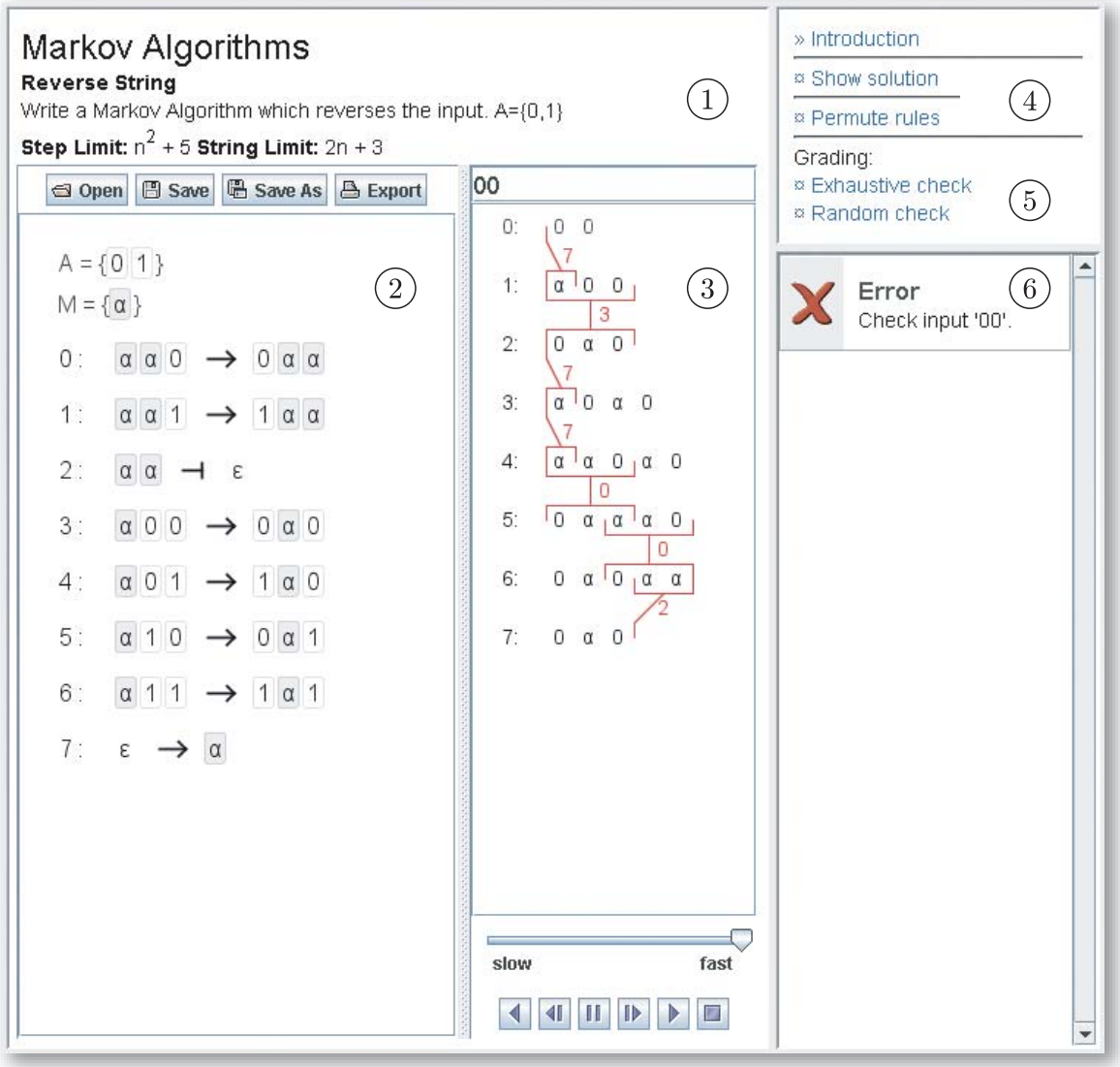


Figure 2.7: The string reverse exercise (1. exercise description 2. syntax driven algorithm editor 3. execution tracer 4. solution & permutation keys 5. grader 6. error notification)

tion tracer (Figure 2.7, 3.). The tracer illustrates the execution of the algorithm on the user's specified input. The execution controller allows variable speed, step-by-step and reverse execution. The second debugging facility consists of using an automatic algorithm checker. The automatic checking of algorithms has a long tradition (e.g. Hollingsworth, 1960; Forsythe and Wirth, 1965) culminating in Manuel Blum's Turing awarded work on program checking (e.g. Blum and Wasserman, 1994). As the correctness of algorithms is undecidable, the systems checks the student's answer by sampling. For a set of inputs, the system checks if the output produced by the student's algorithm is identical with that produced by the algorithm proposed by the exercise programmer. The set of inputs is either defined for this purpose by the exercise programmer, or generated randomly. Inputs leading to discrepancy between the student's solution and the master's solution serve to provide guidance messages to the student. EXORCISER's automatic grader for Markov Algorithms compares exhaustively or randomly the output generated by the student's algorithm with the expected output and produces an appropriate feedback as exemplified in Figure 2.7. We have set runtime constraints for each exercise in order to prevent infinit loops. Checking by sampling is of course neither a formal proof of correctness nor does it allow us to "localize" and to "identify" the error in the student's algorithm. Nevertheless, it is fast and therefore it is highly probable that the grader will find an input—if one exists—for which the student's algorithm produces a false output.

2.3 Conclusions

We have presented two interactive exercises. They have in common that each of them has more than one possible solution, and there are even several ways to construct these solutions. An interactive grader checks the proposed solution at any time, and the number of exercises is not limited to a few predefined exercise instances. This makes the exercises truly interactive and attractive.

The State Minimization example shows us that a fixed structure such as the state pair table seems to be necessary for an accurate localization of errors in order to adaptively provide elaborated feedback. The String Reverse example on the other hand is less structured, therefore it offers more flexibility but the system's feedback possibilities are limited.

User guidance is not *an all or nothing* proposition. EXORCISER offers students both: weak and strong guidance. As week and strong guidance are complementary to one another, we leave it to the student to choose the degree of guidance that meets her needs.

Chapter 3

Guidance: What is the Right Degree of Control?

Luehrman (1972) questioned provocatively in an early paper: "Should the computer teach the student, or vice-versa?" Giving the student control is like putting someone in the driver's seat of a car. To put a student in control means that the programmer assumes the student knows what she wants to do. To put software in control means that the software assumes it knows the best way for the student to perform, and will guide her along the task. A Student rarely masters a challenging exercise the first time (Wiggins, 1993). For a student to succeed, she depends upon feedback and guidance—not on initial teaching only. The aim of a teaching device should therefore be to enable a student to self-assess and self-adjust with maximal efficiency. Mastery involves the student's ability to make intelligent adjustments based on available feedback; therefore it is natural to provide feedback and guidance until students perform correctly.

Guidance and Feedback Wiggins (p. 184) describes feedback and guidance as follow: "Guidance gives directions; feedback tells me whether I am on course. Guidance tells me the most likely ways to achieve my goal; feedback tells me whether I am on track or off track in a way that enables me to self-adjust." The aim of educative assessment is to improve students' performance, and not to measure it. Educative assessment (Gilbert, 1978) has at least three key components: (1) the standards against which performance will be assessed; (2) the indication whether the standard has been met or not, *feedback*; and (3) procedures of correcting sub-standards, *guidance*. Feedback and guidance are effective only if there is prior clarity about goals to achieve.

What can students expect from an automatic assessment system? Several authors (e.g. Brusilovsky, 1996) claim that adaptive devices lead to better student performance. An adaptive device (Langley, 1999, p. 358) "is a software artifact that improves its ability to interact with a user by constructing a user model based on partial experience with that user." An "intelligent" tutor tries to fit feedback and guidance to the student's needs, i.e. by varying the feedback's specificity, e.g. omitting positive reinforcement, or deferring feedback according to the student's advance in

mastery. But, adaptability has two serious drawbacks. Firstly, when a system adapts to an evolving student model, each time a student re-performs a certain exercise, the system may behave differently. Unstable behavior may confuse the student. Secondly, we need a suitable student model. Unfortunately, attempts to model the student never got very far (Schulmeister, 2002a, pp. 218). Human nature may just be too complex to model in any meaningful way.

An alternative to adaptability based on a student model involves letting the student select the level of difficulty among a small set of options, e.g. *Beginner*, *Advanced* and *Expert*. This technique of self-assessment as used it in EXORCISER is common in games and other forms of immersive entertainment. Doing so, we are able to offer some amount of adaptability, trying to reflect the student's performance back to them in a way that enables them to assess her performance rather than trying to model the student.

3.1 The Solution Space

Providing meaningful feedback is a challenging task. The exercise designer has to anticipate appropriate evaluative statements for each possible scenario. The *solution space* for a given class of exercises is a structure that captures all consistent scenarios, i.e. all sequences of operations for solving problems of that type by following appropriate algorithms. The number of scenarios for a single instance of a problem explodes quickly. To avoid generating complete sets of scenarios, a solution space can be described as a structure which contains branch points, and so represents compactly within itself all the scenarios without repetition. Such a structure is a graph (Figure 3.1), and scenarios are paths in this graph that traces refinement processes a student may go through when developing solutions to an exercise. The graph is constructed on the fly in the vicinity of the student's current position. It serves to guide the student from his current position to the nearest target node in the solution space.

The central idea is to use the concept of the solution space as a basis for specifying an interactive exercise, where feedback and guidance are explicitly defined. A formal description of an exercise's solution space makes it easier to design a consistent interaction model. Maintaining a consistent representation of the object of study is important as it ensures the predictability of system response to student inputs and actions.

Example: State Minimization Recall the state minimization exercise from Section 2.1. For the DFA A (Figure 3.2), the solution space shows

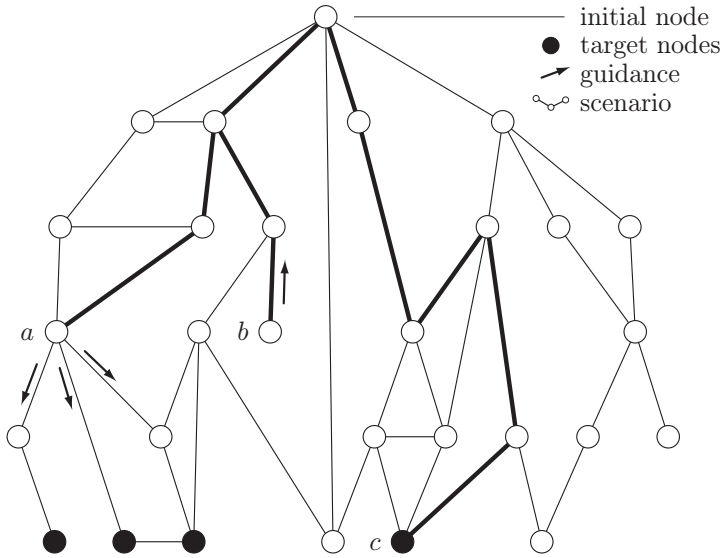


Figure 3.1: Solution Space

sample scenarios computing the automaton's set of equivalent states. Each edge in the solution space corresponds to a student's action that consists in filling an entry in the state pair table. The complete solution space contains a total of 60.466.176 nodes, two of which (1.10 and 5.10) are target nodes. The number of nodes in the solution space grows exponentially with the number of states in the automaton. The exercise starts on the empty table, represented by the initial node (1.0). At least ten steps are necessary to reach one of the two target nodes. Other nodes such as (2.10) could be considered as valid target nodes too. Nevertheless, these are suboptimal as some of the entries in these tables are substitutable with shorter witnesses. The solution space (Figure 3.2) shows also five tables (3.8–10 and 4.9–10) with incorrect entries, where e.g. 'c' is a word that fails to distinguish states q_2 and q_4.

In the following, we distinguish and characterize two exercise types. The first type is oriented towards step-by-step guided performance. The second type, weak guided exercises, is oriented towards the exercise's output rather on the dynamics of performance. We illustrate these two types with examples from EXORCISER's exercise collection.

3.2 Step-by-Step Guidance

Step-by-Step guidance is particularly advisable when an exercise requires the student to perform a long and error prone sequence of operations. Along this refinement process the student has to perform a sequence of operations where, in each step, she must choose among a limited set of

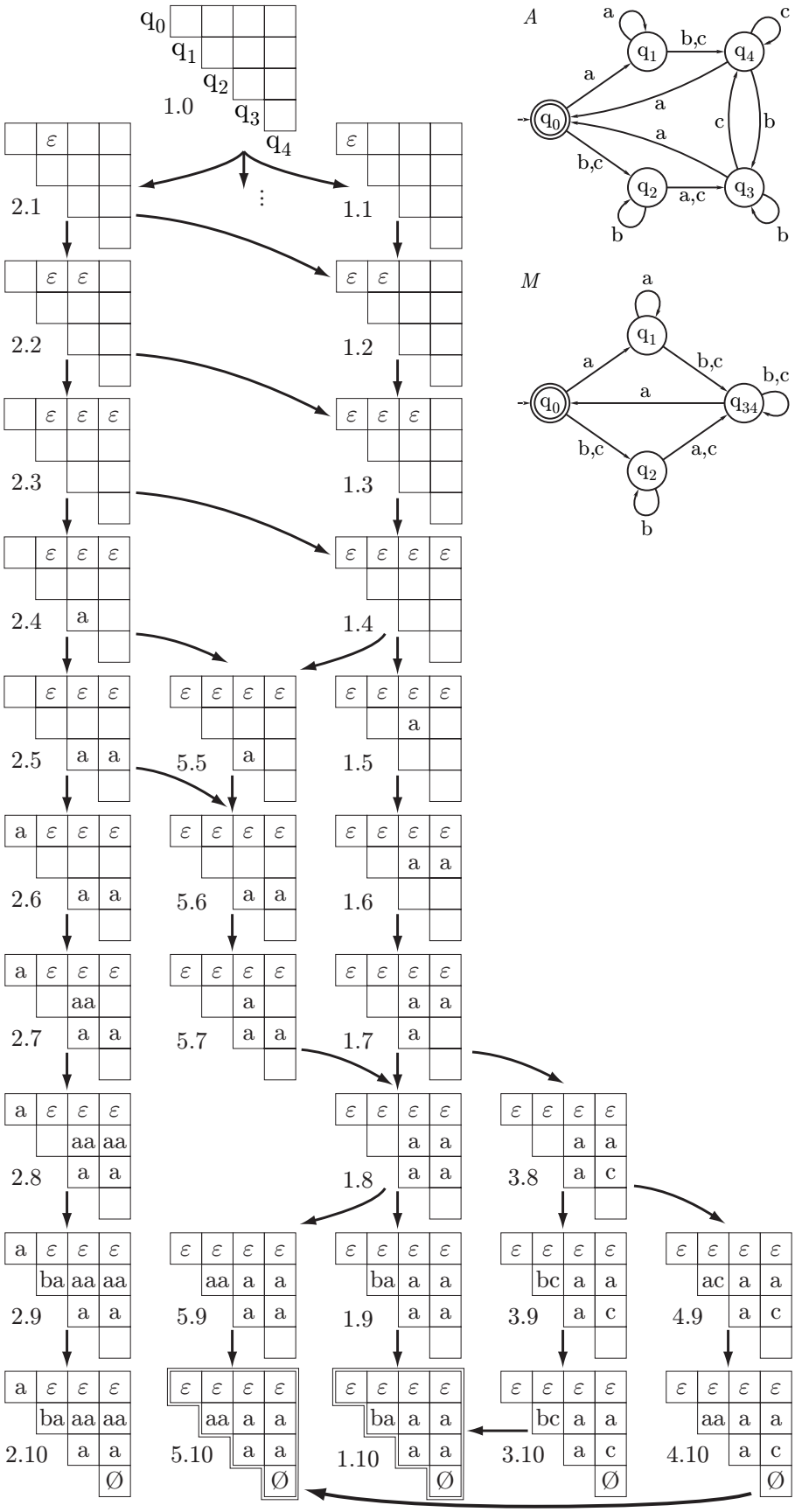


Figure 3.2: A part of the solution space of the state minimization exercise

actions one that will bring her closer to a solution. This approach implies the existence of an algorithm of the form

$$\textbf{while } \neg T \textbf{ do } \{H\} \text{ “select and perform” } \textbf{end while } \{T \wedge H\},$$

where T is a termination condition of the algorithm and H is the loop invariant. The loop invariant is a property of the variables values that is true prior and after each iteration of the loop. While the solution space captures all consistent scenarios, such an algorithm may capture a subset of the solution space only. In fact, it has to be a real subset as the student must be allowed to commit errors. However, the system remains in a consistent state in terms of its ability to provide feedback and guidance.

Example: Translating Finite Automata to Regular Expressions Given the state diagram of an arbitrary finite automaton M, construct a regular expression R for the language accepted by M. In general, converting finite automata to regular expressions can lead in the worst case to expressions of size exponential in the size of their describing automaton (Ehrenfeucht and Zeiger, 1974). Sipser (1997) describes a simple and intuitive algorithm to translate *generalized nondeterministic finite automata* (GNFA) to regular expressions.

Algorithm: Translating Finite Automat M to Regular Expression

1: $M' \leftarrow \text{GNFA}(M)$
2: $s, e \leftarrow$ unique start state of M', unique accepting state of M'
3: **while** $2 < |Q_{M'}|$ **do** $\{M'$ has ≥ 2 states $\wedge$ M' is equivalent to $M\}$
4: $\quad q_{\text{rip}} \leftarrow \text{RND}(Q_{M'} \setminus \{s, e\})$
5: $\quad$ **for all** $(q_i, q_j) \in (Q_{M'} \setminus \{e, q_{\text{rip}}\}) \times (Q_{M'} \setminus \{s, q_{\text{rip}}\})$ **do**
6: $\quad\quad \delta_{M'}(q_i, q_j) \leftarrow \delta_{M'}(q_i, q_{\text{rip}})(\delta_{M'}(q_{\text{rip}}, q_{\text{rip}}))^* \delta_{M'}(q_{\text{rip}}, q_j) | \delta_{M'}(q_i, q_j)$
7: $\quad$ **end for** $\{M'$ is equivalent to $M\}$
8: $\quad Q_{M'} \leftarrow Q_{M'} \setminus \{q_{\text{rip}}\}$
9: **end while** $\{M'$ has 2 states $\wedge$ M' is equivalent to $M\}$
10: $R \leftarrow \delta_{M'}(s, e)$

First, the finite automaton is translated into an equivalent GNFA that is described as a generalized transition diagram having a single initial state with no incoming transitions and a single accepting state with no outgoing transitions. In the second phase of the translation the GNFA is then converted to a regular expression by reducing step-by-step the number of states until only the initial state and the accepting state remain.

The translation algorithm above is not deterministic. The size of the resulting regular expression depends on the sequence by which states are

removed. Note also, that the loop invariant $\{M'$ is equivalent to $M\}$ also holds for the inner loop of the algorithm. The structure of this non-deterministic translation algorithm leads directly to the solution space shown in Figure 3.3. A single step consists first, in selecting (Line 4) among the states the candidate q_{rip} that will be ripped out (Line 8) and then to adapt the corresponding transitions according to the Rule (Line 6). The student is free to choose in which order to perform and the system accepts any expression as long as the loop invariant holds. The state q_{rip} disappears as soon as the student has adapted the transitions as requested. Figure 3.3 shows possible scenarios starting with the finite automation in the top left corner:

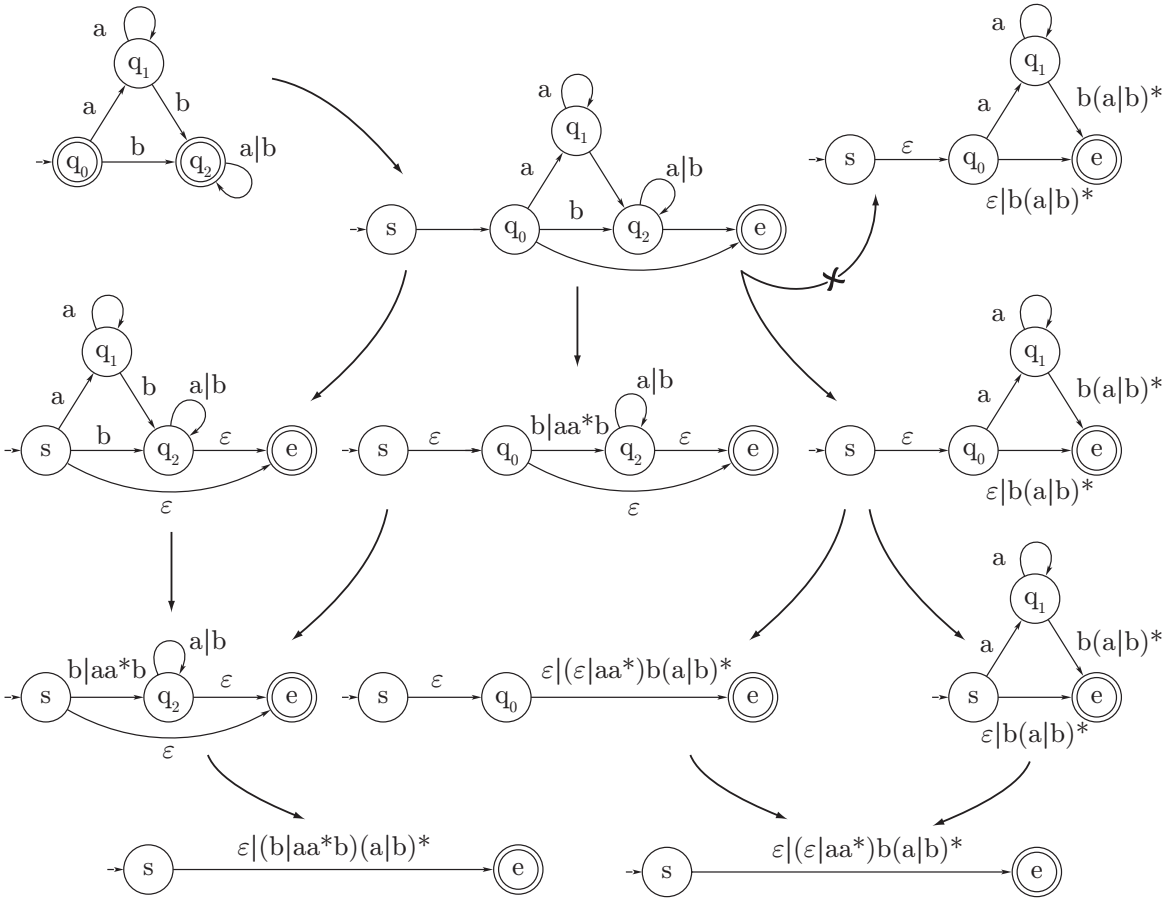


Figure 3.3: Part of solution space of the *finite automata translation to regular expressions* exercise

In the corresponding interactive exercise (Figure 3.4), the student initiates an iteration of the algorithm by selecting the state q_{rip} to be removed. The system then insists that she relabels all transitions correctly before she selects another state. Such a rigid from of control is useful to novices, but might be constrictive to an advanced student who may

want to perform multiple iterations in a lump or simply wants to check a proposed solution. To overcome this lack of liberty it is possible to jump over all intermediate steps and to propose a final or instant solution. The interactive exercise for translating automata to expression allows both, the novice and the experienced student, to perform in a way appropriate to their needs. The system offers the student two choices: (1) a GNFA ready for systematic reduction, and (2) the possibility to submit any regular expression in order to get it graded. Mixed mode is possible too, e.g. a student may first perform a few reductions, and as soon as she "sees" the solution proceed directly to the advanced mode.

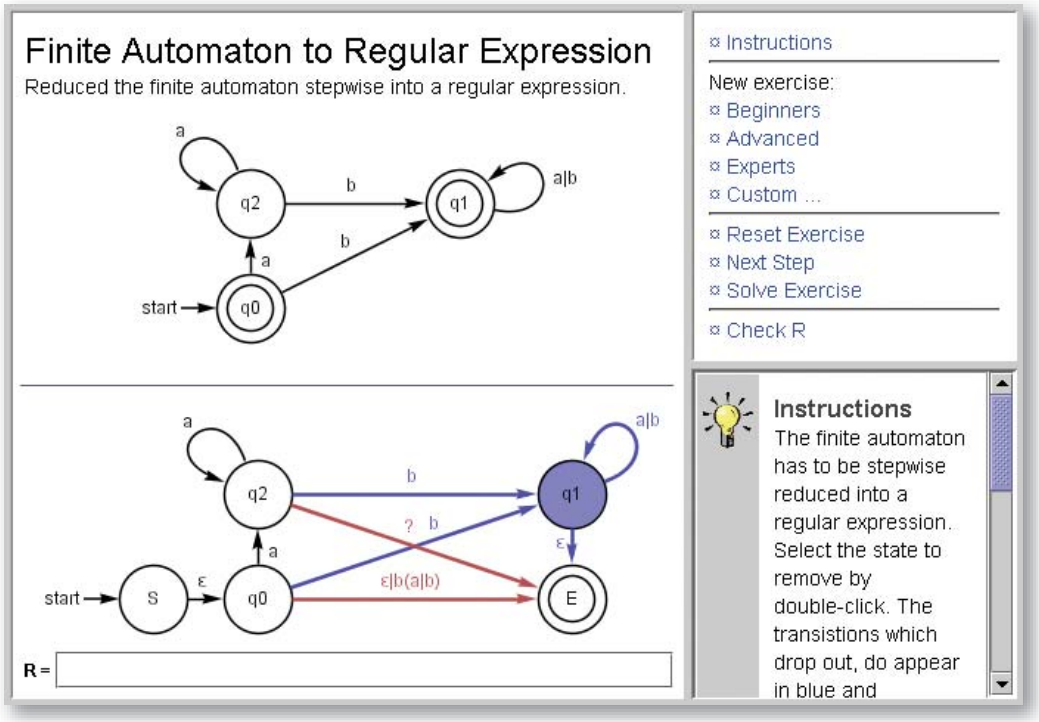


Figure 3.4: Exorciser's interactive exercise for translating finite automata to regular expressions

The Form Metaphor Filling in a form, such as the state pair table in the minimization exercise is a useful metaphor for constructing step-by-step guided exercises. Dynamic programming is especially appropriate for this approach. Another example is the Cocke-Younger-Kasami parsing algorithm for context free grammars (Kasami, 1963; Younger, 1967) also based on dynamic programming. This algorithm requires the student to fill in the entries above the diagonal of an $n \times n$ table, where n is the length of the word to be parsed. Different ways of filling in the entries

as well as different sequences may all lead to a correct solution, but there are dependencies among the entries that must be observed.

In the exercise shown in Figure 3.5, the student drags and drops productions—rewrite rules—from the grammar on the left hand side into the table. At each step, the program checks the consistency of the solution process and provides feedback and guidance on request.

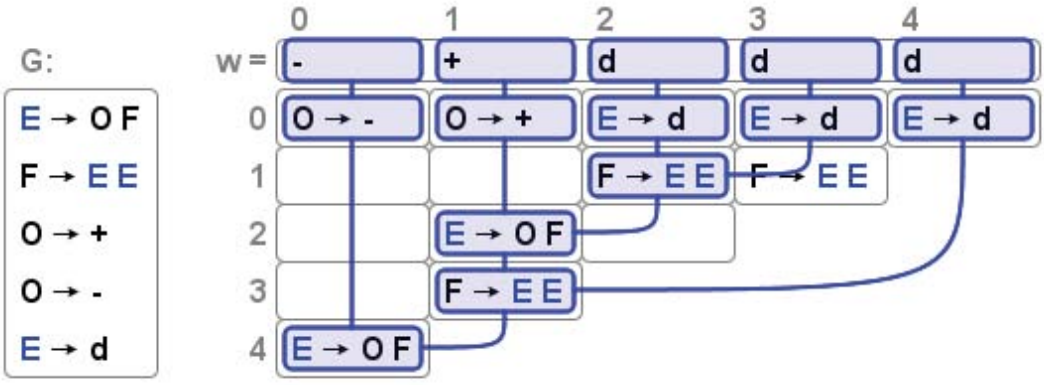


Figure 3.5: Application of the CYK top-down parsing algorithm on the word `-+ddd` for the grammar on the left for arithmetic expressions in prefix notation.

3.3 Weak Guidance

In the previous section, we have discussed a number of concepts where the student has to follow rigidly the approach selected by the exercise programmer. Limiting the student to a single approach is useful only to a beginner, as it allows the teaching device an accurate localization of errors in order to adaptively provide elaborate feedback. Nevertheless, other approaches than the one chosen by the exercise programmer to solve an exercise might be valid too. To design a solution space that captures multiple approaches to the same problem is difficult, and a student may always come up with a new, unforeseen idea. In this section, we discuss solution spaces that capture only the structure of a submitted solution, rather than the dynamics of constructing the solution as discussed in the previous section.

In human computer interaction a number of concepts have been designed to tackle the problem of editing structured data, e.g. structure-oriented editors for writing program source code (e.g. Minör, 1992) or the direct manipulation concept for editing graphically represented objects (Shneiderman, 1983). Structured solutions are required, as the exercise

grader will process them automatically. It is a challenging task to design editors that concisely reflect the underlying exercise's solution space. The student may be familiar with the solution of an exercise but may not know the exact syntax the system requires. Using an unfamiliar structure leads to syntax errors and the solution cannot be validated by the grader.

Direct Manipulation EXORCISER's graphical editor for finite automata (Figure 3.6) and the structure-oriented editor for Markov algorithms (Section 2.2) are two examples of editors where a student can only select and manipulate complete syntactic units.

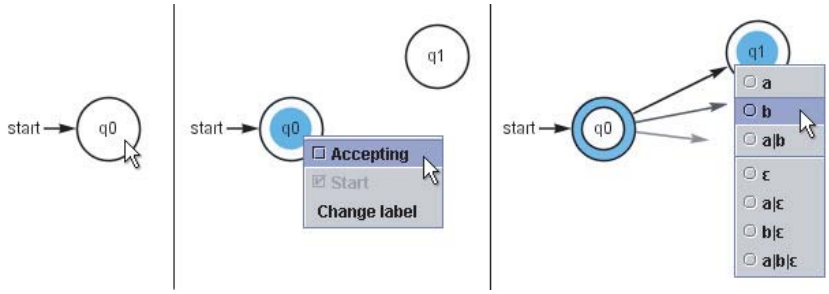


Figure 3.6: Direct manipulation on finite automata

The chances of errors are reduced if the student is either given complete knowledge of the syntax to be used or, in case of direct manipulation, by enforcing syntactic consistency. Of course, enforced syntactic consistency limits the student's freedom to pass through inconsistent intermediate states. On the other hand, enforced consistency reduces the number of errors and increases satisfaction even if the execution time may go down.

One sample exercise that makes use of the EXORCISER's editor for finite automata is the conversion of NFAs to DFAs (Figure 3.7).

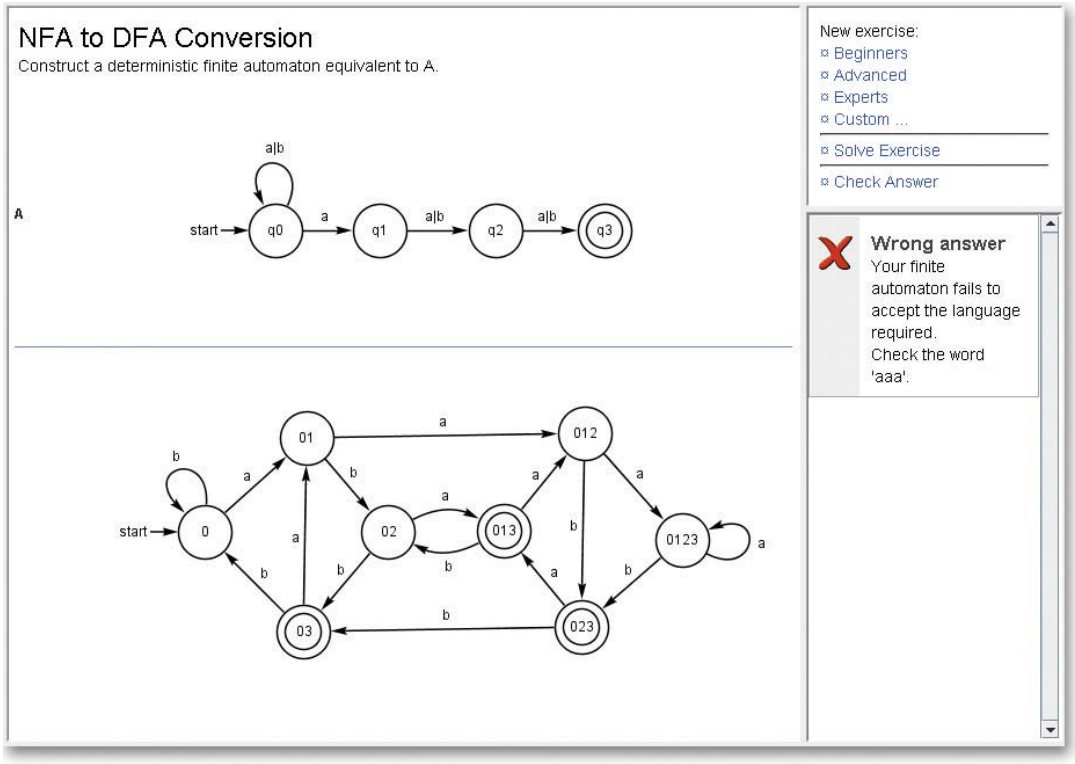


Figure 3.7: Converting NFAs to DFAs using Exorciser's editor for finite automata

Chapter 4

Errors: Detection, Management and Recovery

> "A good error detection and recovery scheme should maximize the number of errors detected but minimize the number of times it reports an error when there is none."
>
> Graham and Rhodes (1973, p. 52)

Errors may be unpleasant, but they can also be a source of insight when commented on expressively. To comment on errors in a meaningful way—rather than simply to present the solution on a silver platter—is a challenging task. No matter what errors a student has made and how uninterpretable her position in the solution space is, the system has to be able to offer a way to recover. Given this context what feedback is appropriate when a student "gets stuck" while solving an exercise?

4.1 Error Detection

Instead of building an "intelligent tutor" able to catch a limited set of scenarios foreseen by the exercise programmer, the central idea in EXORCISER is to design together with each individual solution space an individual grading mechanism that is able to catch all possibly occurring errors and to provide meaningful feedback. In a next step, the most significant among the detected errors is selected and commented on in a way that allows the student an efficient recovery. As a consequence, the obtrusion induced by the provided feedback is minimized according to the above citation, which interestingly does not stems from research on CAI but from research on compilers.

Example: Commenting on Errors in the CYK Parsing Table The Cocke-Younger-Kasami (CYK) parsing algorithm (Kasami, 1963) for context free grammars is based on dynamic programming and requires the student to derive a parse tree by filling in the entries above the diagonal of an $n \times n$ table, where n is the length of the word to be parsed; see Section 3.2. In contrast to the bottom-up logic of the CYK algorithm, we

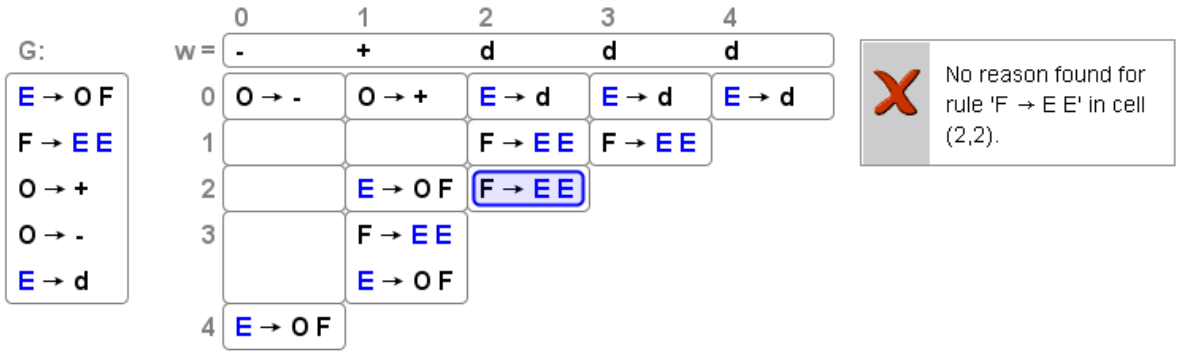
let a student fill in any entry of the matrix with productions of her choice, even if previous (in the algorithm's logic) entries are still missing. We call such an entry *unsuppored* as exemplified in Figure 4.1 (a). On request the system checks the consistency of the table entries and comments on the most critical error. Unsupported entries in the table are more critical than missing entries; unsupported entries from which a student has already derived other entries are by our definition the most critical errors. The error notification in Figure 4.1 (a) is an example of a critical error: while the entry '$E \rightarrow EE$' in cell $(2, 2)$ is unsupported, the entry '$E \rightarrow OF$' in cell $(1, 3)$ is unsupported too, as it depends from the entry in cell $(2, 2)$. As we do not want to obtrude the student too many error notifications only the unsupported entry from cell $(3, 1)$ is commented.

4.2 Counter-Examples

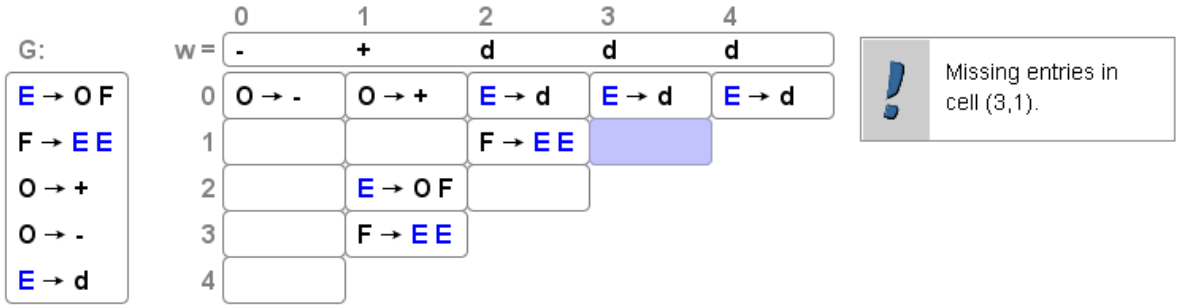
As soon as exercises have more than a single solution, it is particularly difficult to precisely locate errors and to provide fine-grained feedback. In case of mistakes it is useful to provide the student with a witness that proves the incorrectness of her answer. Witnesses are an important issue in educational software as they can give the student the decisive hint for improving his or her answer, without the need to present the entire solution. A witness contains neither explicit information on the precise location where the proposed solution needs improvement, nor does it provide explicit guidance.

Example 1: Witnesses proving the Inequality of Regular Languages Any regular expression R can be converted into an equivalent finite automaton M. In this exercise the student draws a state diagram using EXORCISER's structure-oriented editor for finite automata. This happens in a fashion much akin to paper and pencil. On request, the teaching device checks the equality of the proposed finite automaton with the regular expression. If found not equal, this is commented with a witness disproving the validity of the student's answer. Figure 4.2 gives an example. The word 'a' is accepted by the automaton, where a is not in $L(R)$.

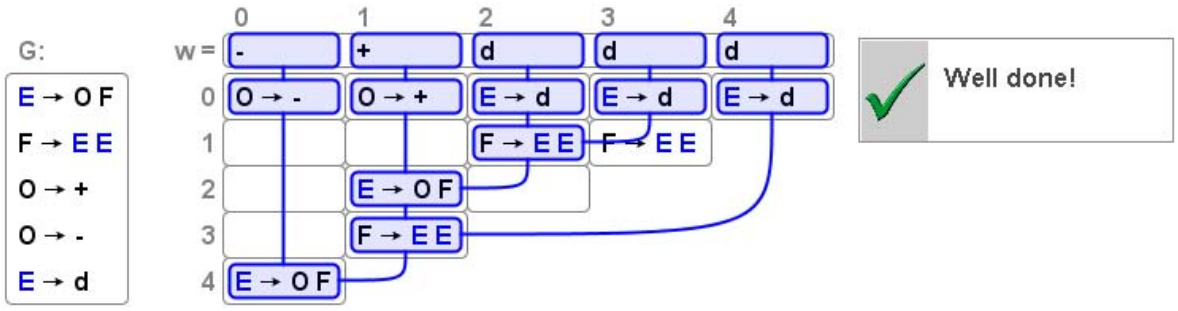
Example 2: Witnesses proving the incorrectness of Algorithms In Section 2.2 we already have presented a method for testifying errors in algorithms. While checking the correctness of the proposed Markov algorithm is undecidable, automatic checking by testing is effective for algorithms with short runtime.



(a) Unsupported table entry: Either there is no equivalent entry in the master table or some entries are missing in the rows above



(b) Incomplete table: Even if the entry '$F \rightarrow EE$' in cell $(3, 1)$ is of no use later for the construction of the parse tree, following the CYK parsing algorithm systematically requires inserting '$F \rightarrow EE$' in cell $(3, 1)$



(c) No Errors: The parse tree shows the correctness of the table entries

Figure 4.1: Tree levels of error detection in the CYK parsing algorithm exercise

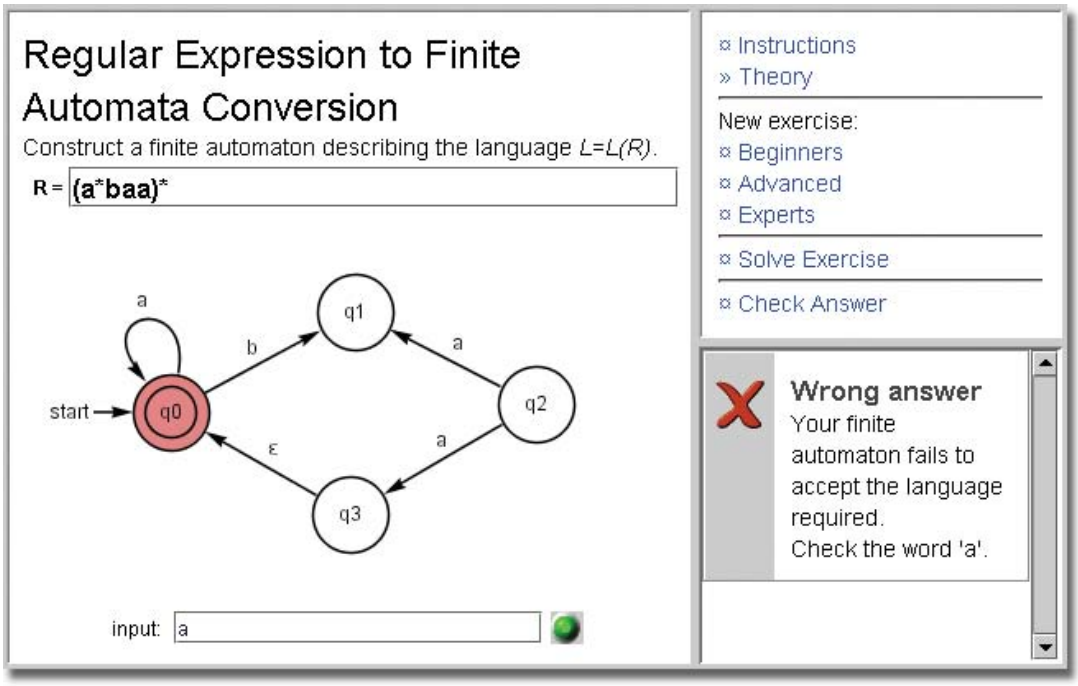


Figure 4.2: Counter-Example: The word 'a' is a witness proving the inequality of proposed finite automaton with the given regular expression

4.3 Automated Error Recovery

It is important that a student experiences success at solving exercises. Success increases the student's self-efficacy, i.e. it builds a robust belief in one's personal efficacy that is an important factor in education (Bandura, 1994). Error recovery and the system's ability to refine a student's solution step-by-step is therefore important as it reduces the number of exercises a student leaves behind unfinished and increases the chance to succeed.

Example: Refining the CYK Parse Table Rather than first removing all unsupported entries and in a second phase to add missing entries, the refinement function proceeds top down—row by row—in order to minimize the number of changes necessary to fix the parse table (Figure 4.3). Note for instance that the unsupported entry '$E \to OF$' in cell $(0, 4)$ stays in the table being part of the master solution.

Step-by-step refinement implies the possibility of *rollback*, i.e. to "undo" some of the operations a student has performed to construct her erroneous answers. Rollback brings back the answer to a previous less awkward stage. The simplest attempt to perform rollbacks consists in either using checkpoint techniques or the editor's multiple-step-back "undo" facility. Nevertheless a rollback facility is only useful to exercises where a student is

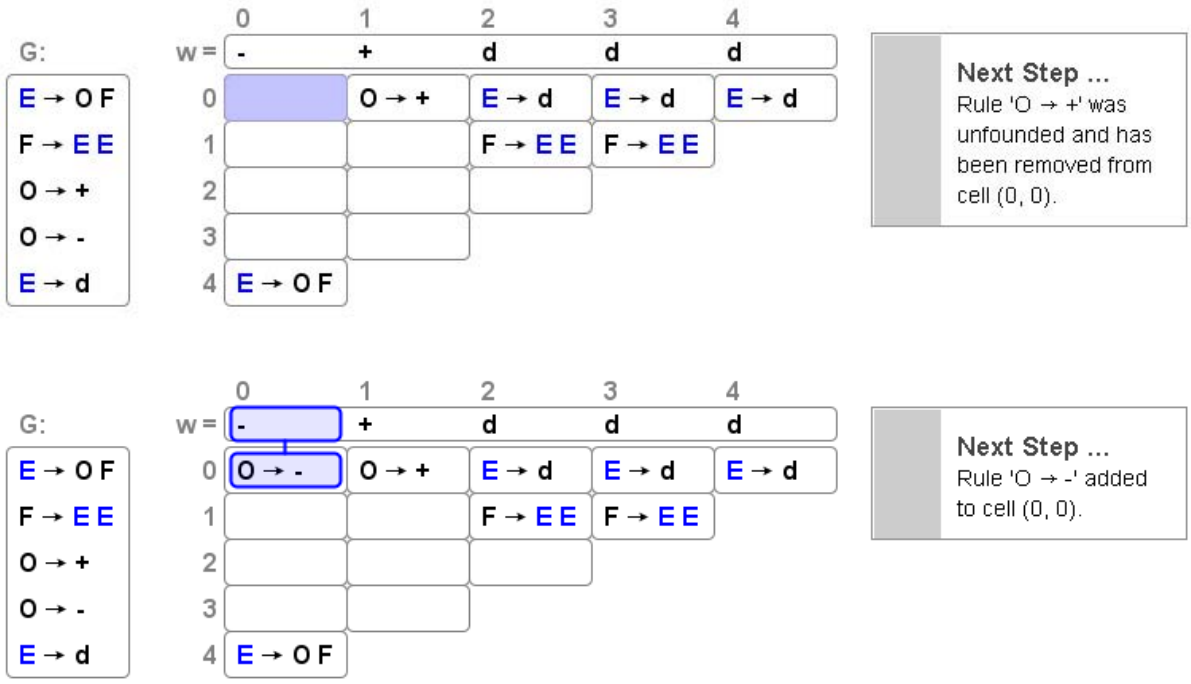


Figure 4.3: Two steps refining the CYK parse table

constrained to perform strictly along a given path. In a from-driven exercise such the CYK algorithm the undo facility would indifferently remove both, correct and incorrect entries.

Chapter 5

Generating Exercise Instances

In this chapter we introduce techniques for the automated generation of concrete *exercise instances*. In order to encourage a student to solve exercises systematically, exercises have to be (1) relevant, and (2) somewhat demanding. Both criteria are essential for the student's motivation. No matter whether an exercise instance is man made or generated by the teaching device, it is important to avoid trivial exercises and guessable solutions.

5.1 Generation on the Fly

Creating exercise instances on the fly implies that a large number of similar problems can be generated quickly. A program that can produce a practically unlimited number of exercise instances that share a common structure but differ in detail is called an *exercise generator*.

Example: Reverse Generation of Non-Minimal DFAs *Reverse generation* is a useful approach to generate exercises, where first a solution of the desired complexity is generated, followed by the generation of a concrete exercise instance derived from this solution. Non-minimal DFAs serve in EXORCISER for instance as input to the state minimization exercise (Section 2.1). For constructing a non-minimal DFA the generator proceeds as follow: first it constructs a minimal DFA; a minimal DFA can easily be derived from a random graph. Second, based on the minimal DFA, the generator derives a non-minimal DFA by successively cloning states. Figure 5.1 gives an example. The size of the initial minimal DFA depends on the level of difficulty the student has selected. Applying the cloning operation successively several times, and taking care not to disconnect the state diagram, the generator creates non-minimal DFA with 4–9 states and 1–3 equivalent state pairs.

Comments Some people may argue that machine generated exercise instances are of little interest to students, and that a few particularly instructive exercise instance are more effective than dozens of instances

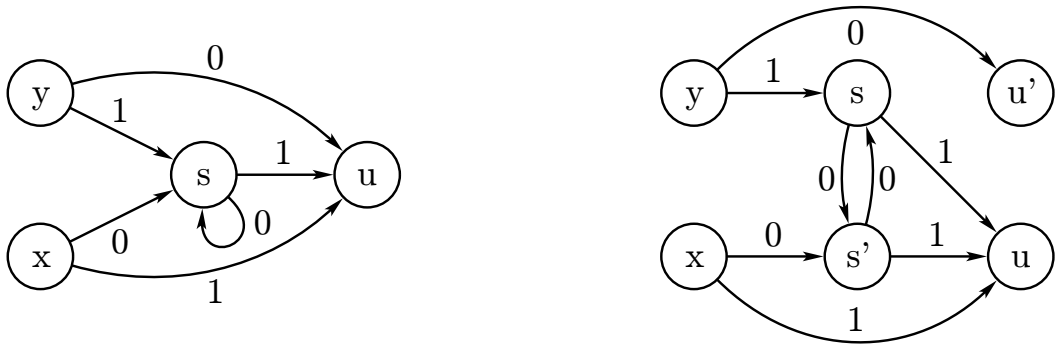


Figure 5.1: Cloning states of a minimal DFA.

for the same exercise class. Nevertheless, a student will appreciate to have the opportunity to use this kind of unfailing and tireless tools.

5.2 Generated Collections

Exorciser contains both, handcrafted collection of exercises such as on Markov algorithms, and generated collections. Generating an exercise collection rather than generating exercises on the fly is a valuable alternative, whenever creating exercises on the fly sensibly affects the system's responsiveness.

Example: Creating Concise Regular Expressions Regular expressions are widely used in Exorciser, e.g. students train to convert them to equivalent finite automata and vice-versa. Creating concise regular expressions based on randomly generated expression fails, as nowadays no efficient simplification algorithm is available to convert a syntactically correct, but awkward expression such as $\varepsilon|(b|aa^*b)a|b)^*$ to its concise equivalent $(a^*ba^*)^*$.

In a system for educational use, we can take advantage of the fact that almost all regular expressions of interest are short, and collect all these in a dictionary of regular expressions. The access key of a regular expression R in this dictionary is a unique identifier derived from the minimal DFA accepting $L(R)$. Exorciser's regular expression dictionary is an exhaustive list of distinct minimal sized regular expression up to length 14 over the alphabet $\Sigma = \{a, b\}$. The lengths of the expressions are measured by counting all nodes and leaves in the expressions syntax tree. Our dictionary is a union of disjoint sub-dictionaries D_i containing expressions of the same length. The size of the sub-dictionaries is given in

Table 5.1, while Table 5.2 gives the distribution of the size of the associated minimal DFA.

$$
\begin{aligned}
D_1 &= \{\oslash, \varepsilon, a, b\} \\
D_2 &= \{a^*, b^*\} \\
D_3 &= \{\varepsilon|a, \varepsilon|b, ab, ba, aa, bb, a|b\} \\
D_4 &= \{(ab)^*, (ba)^*, (aa)^*, (bb)^*, (a|b)^*, a|b^*, b|a^*, a^*b, b^*a, aa^*, ab^*, ba^*, bb^*\} \\
D_5 &= \{(ab^*)^*, (a^*b)^*, \varepsilon|ab, \varepsilon|bb, a(\varepsilon|b), aaa, (a|b)a, aba, \ldots \\
&\vdots
\end{aligned}
$$

l	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\lvert D_l \rvert$	4	2	7	13	32	90	189	580	1,347	3,978	10,334	29,137	81,329	226,063

Table 5.1: The size of the sub-dictionaries D_l for $1 \leq l \leq 14$

$n \backslash l$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Σ	$f_2(n)$
1	1			1											2	2
2	1	2			2	2		4	6	2		3		2	24	24
3	2		3	10	5	24	14	24	20	28	44	46	87	77	328	1,028
4			4	2	17	40	71	198	218	426	478	584	599	869	3,506	56,014
5					8	24	86	200	586	1,370	2,358	4,770	6,710	9,898	25,810	3,705,306
6							18	144	428	1,268	4,127	9,440	21,358	43,970	80,753	286,717,796
7								10	85	776	2,505	8,734	27,528	68,383	108,021	
8									4	94	648	4,422	16,403	59,977	81,548	
9										14	146	914	5,987	29,542	36,603	
10											20	188	1,834	9,362	11,404	
11											6	34	599	2,898	6,537	
12											2		160	797	959	
13													50	178	228	
14												2	4	40	46	
15														24	24	
16														14	14	
17														20	20	
18													8	10	18	
19													2		2	
20																
21																
22														2	2	

Table 5.2: The distribution of the size of the associated minimal DFA in the dictionary of regular expressions, where l denotes the length of regular expressions, n the number of states of a corresponding minimal DFA

The entry in row n and column l in the table above gives the number of regular expressions (dictionary entries) of size l accepted by some minimal DFA with n states. E.g. the sub-dictionary D_{14} contains two regular expressions of length 14 that are accepted by a minimal DFA with 2

states. The last column provides a comparison between the total number of pairwise non-isomorphic minimal DFAs with n states over a 2-letter input alphabet computed by Domaratzki et al. (2002).

Implementation Issues and Comments The dictionary contains a total of 356,029 distinct expressions over the alphabet $\Sigma = \{a, b\}$. Two expressions R_1 and R_2 in the dictionary are called (a, b)-isomorphic, iff they are transformed into each other by replacing all occurrences of a with b, and vice-versa. E.g. $aaab^*$ and $bbba^*$ are (a, b)-isomorphic. By removing (a, b)-isomorphic expressions the size of the dictionary can be cut in half to 176,637, leading to a space requirement of 2MB.

The use of the dictionary is not limited to the generation, or selection of expression. We use the list of access keys as a source for "generating" minimal DFAs too.

The concept of generated collections such as the regular expression dictionary can be transferred to generating minimal NFAs. The number of distinct languages accepted by NFAs with 3 resp. 4 states over a 2-letter alphabet is 221,184, resp. 4,020,240,384 (Domaratzki et al., 2002).

5.3 Exercise Chains

To build an *exercise chain* implies to define appropriate sequences of exercises and to guide the student along them. Based on the student's answer to a specific exercise the system offers the student a list of exercises to solve next. Rather than creating new instances, the student's answer serves as input for the next exercise in the chain. Using the technique of exercise chains we hope to give to the student a glimpse of the big picture in which exercises are embedded.

Example EXORCISER contains a collection of exercises about regular languages. While each exercise treats a single problem, e.g. the conversion of an NFA to an equivalent DFA, connecting them together in an exercise chain allows us to illustrate a fundamental fact, the equivalence of regular expressions and finite automata (Figures 5.2 and 5.3).

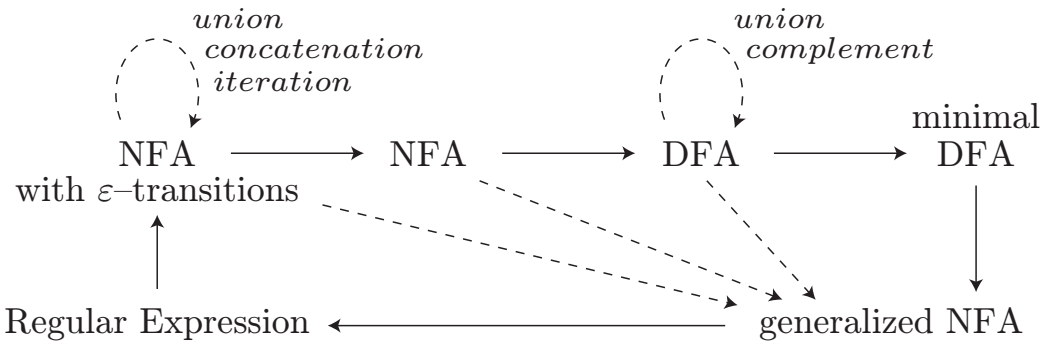


Figure 5.2: Exercise chain, the transformation of regular expressions to finite automata and vice-versa.

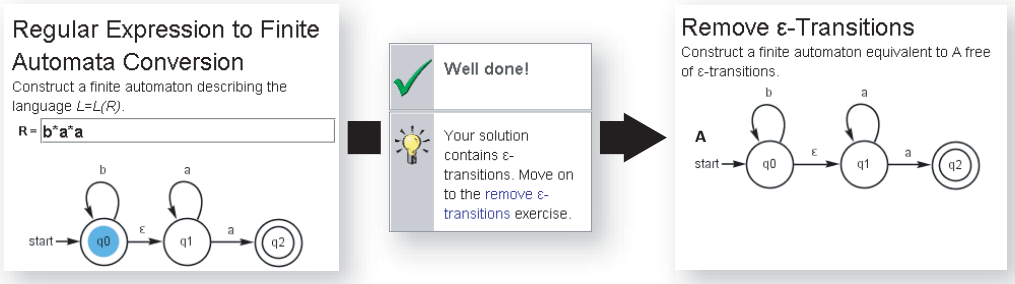


Figure 5.3: Implementation of an exercise chain.

Chapter 6

The Exorciser System: Architecture, Design and Implementation

The *Exorciser System* is an application framework that supports the implementation and distribution of interactive self-grading exercises. Moreover it is a visual framework with guidelines that offer a consistent interface, no matter what specific exercise the programmer is creating. The three main goals of this system are:

1. The exercise programmer gets an *extensible* framework facilitating the development of interactive courseware.

2. The teacher gets a *customizable* learning environment capable of integrating her own lecture notes and link them to the interactive material.

3. The student gets an *attractive* collection of exercises related to the topic of Theory of Computation.

Our solution consists of a highly scalable framework of Java components and XML resources. The pivot element of this framework is the *Exercise Browser*. Comparable to a web browser the exercise browser manages access to exercises and the navigation between them.

In an earlier work with Häfeli and Lamprecht (2001) we made initial investigations in the field of interactive exercises. During this work we developed the basic techniques of *generating and grading* and implemented and tested a first set of exercise prototypes. This collection of prototypes has constantly evolved since 2001 (e.g. Suter, 2001; Müller, 2002; Erni, 2002; Imfeld, 2002; Dreier and Lefebvre, 2002; Biderbost and Schlegel, 2003) resulting in what we call the *Exorciser Standard Distribution* presented in this thesis. The challenge was to blend together a growingly heterogeneous collection of contributions in order to make them appear as *one*, deliberately designed: easy to use and navigate.

In the following we focus first on the system's overall architecture, including selected implementation issues, and afterwards on the visual framework.

6.1 The Overall System Architecture

The *Exorciser System* consists of three macro-structural layers. The *layers' architectural pattern* (Buschmann et al., 1996) helps to structure applications and enhances the maintainability and the extensibility of the system. The three layers (Figure 6.1) are organized as follows:

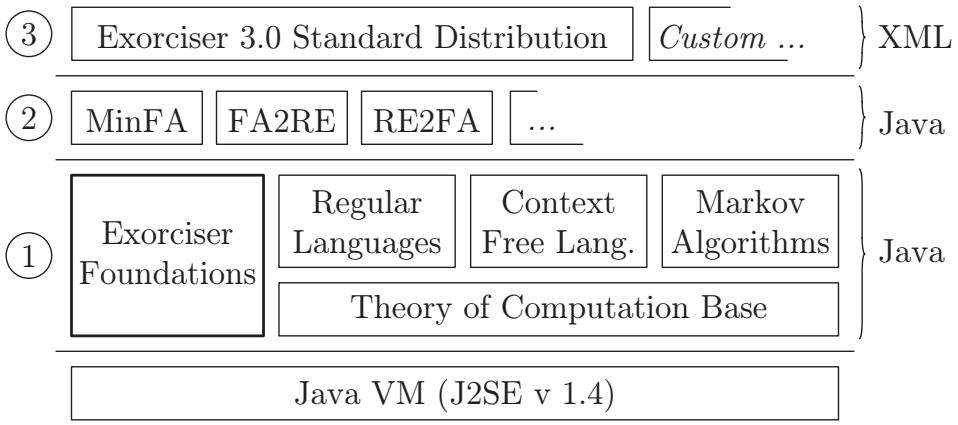


Figure 6.1: EXORCISER's overall system architecture

1. The base layer of our system consists of two independent parts: (1) the *Exorciser Foundations* containing the exercise browser that is completely independent of the subject matter for which it is used; and (2) a domain specific code base, in our case for Theory of Computation (See Section 6.2).

2. The middle layer consists of a collection of configurable *exercise templates*. Typically, configurable characteristics of an exercise are its dialog patterns, the range of available commands, and the exercise's grading scheme.

3. The distribution layer: The purpose of a distribution is to bundle a set of preconfigured exercises for its use in some context such as a lecture. The *Standard Distribution* consists of all currently available interactive exercises, but does not contain any lecture notes.

6.2 The Exorciser Foundations

The *Exorciser Foundations* provides the classes and interfaces of the system's core navigation, notification, and grading facilities. The key elements of this package include the *Exercise Browser*, *Notifications*, and the *Grader*.

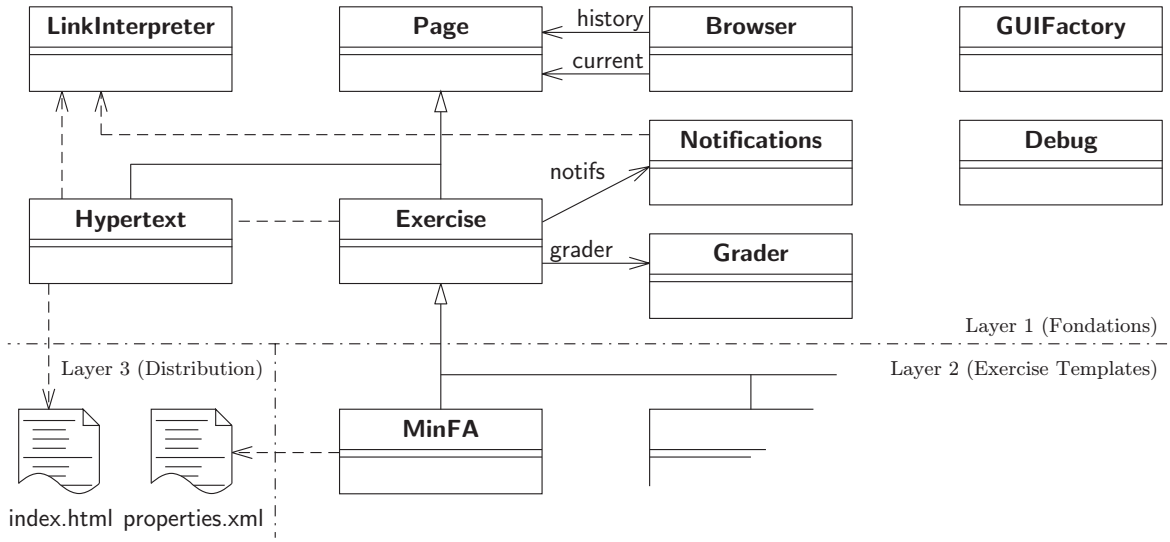


Figure 6.2: The *Exorciser Foundations'* class diagram (UML)

The Exercise Browser The browser is the system's pivot element. It enables the student to access and display *pages* provided in a distribution. Pages are either interactive exercises or hypertext documents (Figure 6.2). In each distribution, there is a designated *index page*, i.e. the place where exercises are started or hypertext documents are referred to. The browser offers further a navigable history of visited pages.

Notifications The dialog between student and system is an important issue in educational software. By adopting the same notification mechanism in all exercises the student experiences consistent interaction within the system. The *Notification Platform* provides an application programming interface (API) for presenting notifications to students. The primary goal of the notification platform is to improve the exercise's accessibility. Notifications do not require an explicit acknowledgement from the student—they do not distract the student as standard dialog-boxes do. Notification typically contains (Figure 6.3) a heading, body text, and an icon. Optionally, notification also includes commands (links).

Graders and the Grading Language Each exercise has its own individual grading scheme that defines under which conditions the student will receive particular types of feedback. A grading scheme is a list of *if*...*then*-rules that are scanned sequentially either on the student's request or after specific events the exercise's programmer has foreseen. While in all exercises the same syntax (the grading language) is used,

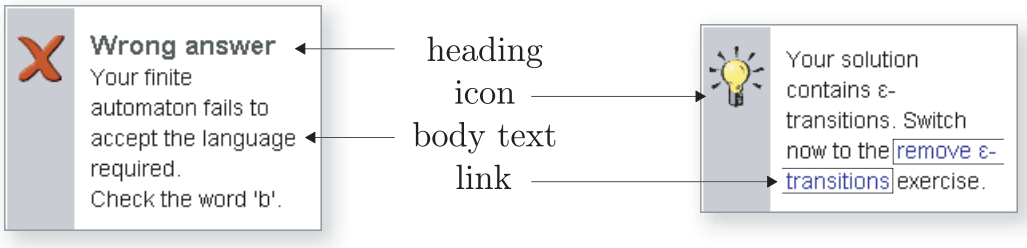


Figure 6.3: Two sample notifications created using EXORCISER's notification platform API

each exercise comes with its own set of keywords denoting events and raise-conditions. Example:

```
 NO_STATES >
    WARNING:"Empty Answer":"To create states, ... " STOP ;
!EQUIVALENT >
    ERROR:"Wrong answer":"Your finite automaton fails to accept
    the language required.<br>Check the word ‘$WITNESS’." ;
!EQUIVALENT & !HAS_ACCEPTING_STATE >
    HINT::"None of your states is accepting. ..." STOP ;
...
```

The grading scheme above belongs to an exercise where the student has given a regular expression R to construct a finite automaton M such that R and M become equivalent descriptors. If the student has not yet specified an automaton (`NO_STATES`), we assume that the student does not know how to use the finite state editor. In this case, we provide the student with a short manual (`To create states, ...`). Then the grading process stops as indicated by the keyword `STOP`. In the case where the student has already submitted an automaton then we check if the answer M is equivalent (`EQUIVALENT`) to R. If not (`!`), then we provide the student with a witness (`$WITNESS` serves as placeholder in the dialog pattern) proving the inequivalence of R and M.

The fact that we use grading schemes rather than hard-coded checking mechanisms has several advantages. A teacher may want to adapt the grading scheme of an exercise to her special needs; she can do that without any Java programming skills and without need to recompile the source. Furthermore EXORCISER is ready for internationalization—all grading schemes and text pattern used in the program are saved in property-files and not in the code. Currently versions in English, German and Slovenian are available.

6.3 The Visual Framework

EXORCISER is an educational software, and thus its user interface has to be designed to be as easy-to-use as possible. A student is typically willing to invest time and energy to learn interfaces to perform the tasks necessary for their daily activities. Unfortunately solving exercises is not a regular activity. Typically a student spends less than two hours solving exercises of the same type. It is therefore our duty to design exercises in a way to make the student's experience transferable from one exercise class to the other.

The challenge is to blend together a heterogeneous collection of interactive exercises that all belong to the same domain of knowledge in order to make them look like *one*, deliberately designed: easy to use and navigate. When a user interface uses consistent color, font, and layout, and when titles and navigational aids—signposts—are in the same place every time, users know where to find things (Shneiderman, 1997). They do not have to figure out a new layout each time they switch context from one exercise to another. The solution to this problem is known as *Visual Framework* (Tidwell, 2002) and consists in drawing up an overall look-and-feel for all exercises in our collection (Figure 6.4).

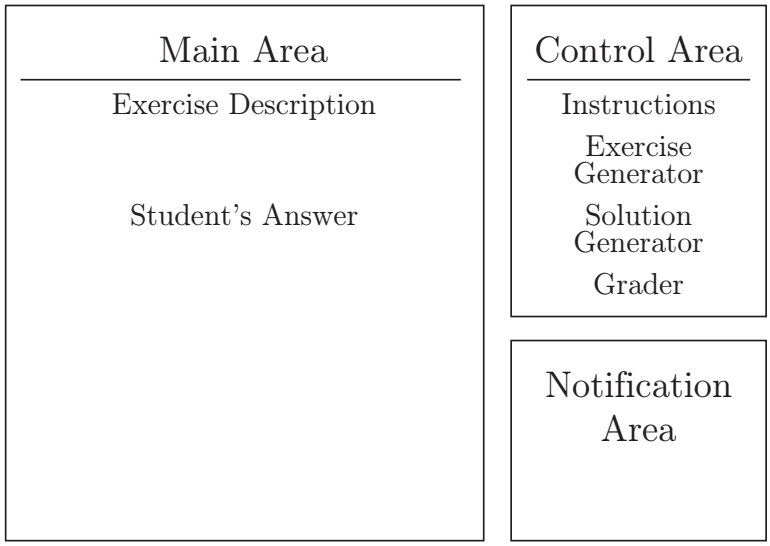


Figure 6.4: The visual framework

1. The *main area* to the left contains all the student needs to solve the exercise: (1) the exercise's description on the top, and (2) below the exercise's solution editor. All editors are syntax-driven, as only structured solutions that are restricted to a known set of options can be processed automatically.

An exception to this guideline are regular expressions where we use standard text editors. In such a case, it is useful to provide a self-explanatory *input hint* (Tidwell, 2002), i.e. to propose a short example or explanatory sentence, and put it below the text field.

2. The *control area* in the top right corner is the place where we locate commands such as for grading the exercise or generating new exercise instances and solutions.

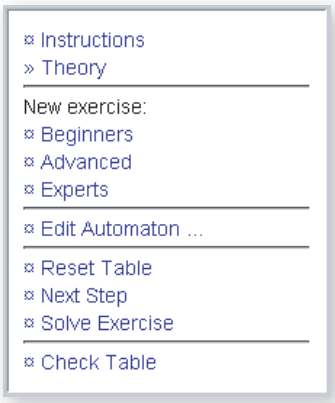
   The list of commands is specified in HTML and commands are encoded as hyperlinks (`<a href="...">...</a>`). Hyperlinks in Exorciser are interpreted in an extended way: either (1) as *common link* pointing to an HTML document, (2) as a *method link* pointing to a public method of the object that represents the actual exercise, or (3) as *new instance link* that will create a new exercise page. E.g. the hyperlink

   ```
   <a href="random?level=beginner">...</a>
   ```

   is not a reference to a HTML document, but encodes the invocation of the method random(String query). We use Java's default reflection package to obtain the reflective information about classes and objects such as methods and constructors.

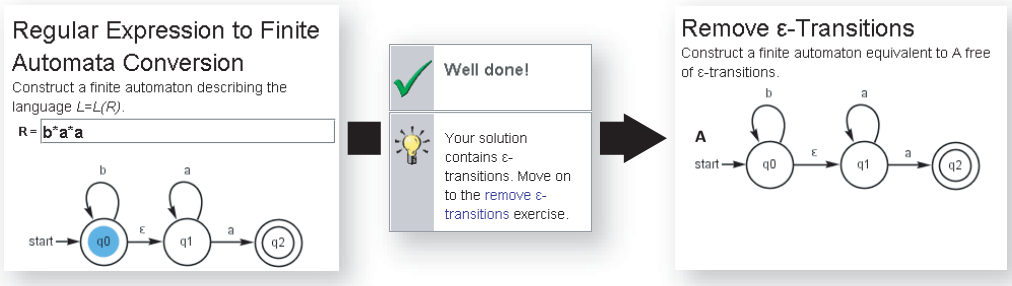
3. The *notification area* in the bottom right corner of the screen is the location where notifications are placed. It is possible to include hyperlinks into the notification's text-body using the same extended hyperlink mechanism as described above. This allows us to chain exercises (Section 5.3), i.e. to use the solution of an exercise as starting point of another exercise (Figure 6.6).

The Index Page The index page, i.e. the place from where a student starts exercises, is laid out differently than the exercises themselves, but it still shares certain characteristics, e.g. the title of the topic (home) and the font and color schemes.




```
  <a href="instructions">&#164; Instructions</a><br>
  <a href="resources/minfa_theory.html">&#187; Theory</a>
<hr>
  New exercise:<br>
  <a href="random?level=beginner">&#164; Beginners</a><br>
  <a href="random?level=advanced">&#164; Advanced</a><br>
  <a href="random?level=expert">&#164; Experts</a>
<hr>
  <a href="edit">&#164; Edit Automaton ...</a>
<hr>
  <a href="reset">&#164; Reset Table</a><br>
  <a href="next">&#164; Next Step</a><br>
  <a href="solve">&#164; Solve Exercise</a>
<hr>
  <a href="check">&#164; Check Table</a>
```

Figure 6.5: Sample of a control area and its source code (state minimization exercise)




```
Your solution contains $EPSILON-transitions.
Move on to the <a href="ch.ethz.exorciser.rl.Generic
  ?type=remove_epsilon&fa=$ANSWER">remove $EPSILON-
transitions</a> exercise.
```

Figure 6.6: Chained exercises

Chapter 7

Experience and Evaluation

EXORCISER has been used since the spring of 2001 in an introductory course on the Theory of Computation at ETH Zurich. Students must attend this course in the fourth semester of their computer science studies. The course covers topics such as models of computation, automata, formal languages, computability and complexity. Since the first use of EXORCISER, the feedback we have received from these students has been highly positive.

An additional piece of educational software we use in this course is KARA (Reichert, 2003). Our students use KARA for programming and exploring Turing machines. The two systems KARA and EXORCISER together have supplanted approximately one fourth of the course's exercises that had previously been done with paper and pencil.

7.1 Early Experience

In the summer semester 2001, 130 students used an early version of EXORCISER containing exercises related to finite automata and context free grammars. Overall acceptance was high. 80% of the students judged the system as *helpful*. The difficulty of the generated tasks was rated from easy (54%) to difficult (42%).

This led us to add the option of selecting the degree of difficulty of exercises. The main complaint was the lack of an instant solution key. In the present version, EXORCISER offers the option of looking at the solution of an exercise at any time.

7.2 Usability

At the end of summer term 2002, we conducted a survey with 131 students who had attended the course. The survey addressed two questions:

1. What are the students' opinions on the benefits of using EXORCISER?

2. How do our students judge the quality of EXORCISER's user interface?

The results of the survey show that practically all the students thought that Exorciser was useful with respect to their learning about the Theory of Computation. 90% of the students selected the highest or second-highest possible ranking on a 5 point scale (Figure 7.1). A number of students explicitly said that Exorciser's just-in-time elaborated feedback helped them considerably to understand the subject matter.

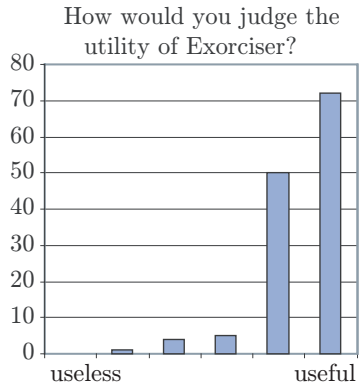


Figure 7.1: Students' assessment of their learning benefit with Exorciser.

To answer the second question, we investigated the idea of *joy of use* presented by Hassenzahl et al. (2001). This article introduces questionnaires to determine the user's perception of the system's usability. The basic idea is to use so-called *semantic differential questions*. For example, is the user interface *innovative* or is it *conservative*? Is it *outstanding* or *second-rate*? 55% of the students gave for Exorciser's usability on a 5-point scale the highest or second-highest possible ranking (Figure 7.2).

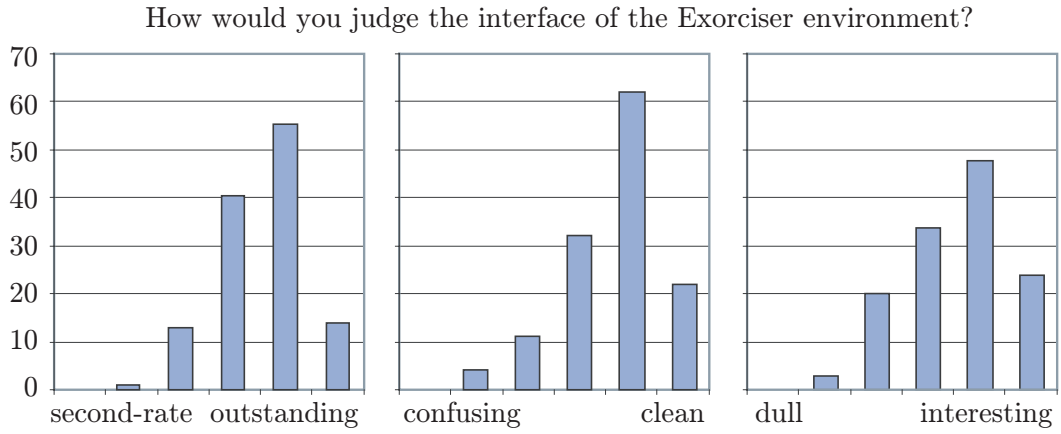


Figure 7.2: Students' assessment of Exorciser's user interface (2002).

7.3 Exorciser vs. "Paper and Pencil"

At midterm of summer 2003, we conducted a comparative study with 195 students who had attended the course. The aim was to record the student's ability to solve exercises—with paper and pencil—after they had performed a supervised training session of one hour with EXORCISER. A control group also performed in the training session, but participants were given no additional help other than the sample solutions provided with the training exercises. The training and test exercises consisted of two types of problem: the conversion of regular expression to finite automata, and the minimization of DFAs.

1) Regular Expressions In the first part of the test, the students were given a regular expression and had to construct a finite automaton. Two expressions had to be converted: $(\varepsilon|a)b^*a^*$ and $(a^*(b|aa)a)^*$. Overall, two-thirds of the students succeeded in constructing a correct finite automaton for the first expression. We did not find a significant difference between the two groups (Table 7.1, a). The second expression that contains nested Kleene star operation was more demanding. Only 43 of 98 students in the experimental group and 30 of 97 in the control group managed to construct a correct finite automaton (b).

2) Minimization In the second part of the test the students were given a DFA and had first to determine equivalent state pairs using a state pair table (c), and then to draw the equivalent minimal DFA (d). The corresponding exercise in EXORCISER is especially designed to train the student's ability to fill in the state pair table correctly. In consequence, it is not surprising that those who had the training session with EXORCISER performed better than the control group.

(a) RE to FA, beginners ($\alpha < 12\%$)

group	n	#correct	rel.
experimental	98	66	68 %
control	97	61	62 %

(b) RE to FA, experts ($\alpha < 1\%$)

group	n	#correct	rel.
experimental	98	42	43 %
control	97	30	31 %

(c) state minimization, table ($\alpha < 6\%$)

group	n	#correct	rel.
experimental	98	66	68 %
control	97	59	60 %

(d) state min., automaton ($\alpha < 1\%$)

group	n	#correct	rel.
experimental	98	80	83 %
control	97	88	90 %

Table 7.1: Comparative Study, Results

Recapitulation At the 5% level of confidence the overall performance of the experimental group was significantly better than the control group who worked with paper and pencil only. These exercises are a solid alternative to paper and pencil. Nevertheless, EXORCISER's purpose is not to substitute the human tutor. We simply encourage our students to take advantage of EXORCISER's capability to check details, and to provide feedback just in time—making the human tutor free for more demanding and thought-stimulating activities.

7.4 Conclusions

Students use EXORCISER purposefully and effectively to solve exercises and receive feedback.

A primary reason for the student's acceptance of EXORCISER might be the availability of the solution generator, together with the possibly of generating an unlimited number of exercise instances of various levels of difficulty and the opportunity to receive immediate individual feedback.

A second reason for the student's acceptance of the EXORCISER environment is probably its intuitive user interface. Simplicity is important as students use EXORCISER for a few hours only, of which the time spent learning to operate the user interface must be a negligible part.

Thirdly, we have shown that it is important to avoid trivial exercises where the student can guess the solution. The exercises have to be relevant and challenging in order to encourage the student to learn to solve them systematically.

The final reason might be due to the fact that the students were always in control of the system. Too much guidance can be worse than no guidance at all.

EXORCISER does not intend to replace teachers. On the contrary, we think it is a useful equipment to be used by teachers to save time and labor and to be used by students as a substitute for a personal tutor.

Appendix A

Exercises in Exorciser

Regular Languages

From Regular Expressions to Finite Automata Given a regular expression R construct a finite automaton M such that $L(M) = L(R)$.

From Finite Automata to Regular Expressions Given the state diagram of a finite automaton M construct a regular expression R describing the language $L(R) = L(M)$.

ε-Transitions Given the state diagram of a finite NFA N containing spontaneous ε-transitions construct a finite automaton N' equivalent to N free of ε-transitions.

NFA to DFA Given the state diagram of a finite NFA N free of spontaneous ε-transitions construct a DFA M equivalent to N.

Minimal DFA Given the state diagram of a DFA M construct the class of equivalent state pairs in Q_M using a state pair table.

Kleene Star Given the state diagram of a NFA N construct a NFA N' such that $L(N') = L(N)^*$

Union Given the state diagrams of two NFA N_1 and N_2 construct a NFA N such that $L(N) = L(N_1) \cup L(N_2)$

Concatenation Given two NFA N_1 and N_2 construct a NFA N such that $L(N) = L(N_1) \circ L(N_2)$

Context Free Languages

CYK Parsing Algorithm Given a context free grammar G in Chomsky normal form and a word $w \in L(G)$ construct a parse tree for w.

NPDA Browser Given a nondeterministic pushdown automaton M and a word w navigate along the tree of all possible configurations of M.

Markov Algorithms

Append Prefix Construct an algorithm P over an alphabet A appending the prefix $p \in A^*$ to every word in A^*.

$$P(w) = pw$$

Constraints: 1 for time, $n + |p|$ for space. Solution scheme:

$$\{ \quad \to \cdot\, p$$

Append Suffix Construct an algorithm S over an alphabet A appending the suffix $s \in A^*$ to every word in A^*.

$$S(w) = ws$$

Constraints: $n + 2$ for time, $n + |s|$ for space. Solution scheme:

$$\begin{cases} \alpha\xi \to \xi\alpha \\ \alpha \to \cdot\, s \\ \to \alpha, \end{cases}$$

where ξ runs through the alphabet A, and α is an auxiliary letter not in A.

Erase First Construct an algorithm F over an alphabet A erasing the first letter of every word in A^*. Example:

$$F(101) = 01$$

Constraints: 2 for time, $n+1$ for space. Solution scheme:

$$\begin{cases} \alpha\xi & \to\cdot \\ \alpha & \to\cdot \\ & \to \alpha, \end{cases}$$

where ξ runs through the alphabet A, and α is an auxiliary letter not in A.

Erase Last Construct an algorithm L over an alphabet A erasing the last letter of every word in A^*. Example:

$$L(10001) = 1000$$

Constraints: $n+2$ for time, $n+1$ for space. Solution scheme:

$$\begin{cases} \alpha\xi & \to \xi\alpha \\ \xi\alpha & \to\cdot \\ \alpha & \to\cdot \\ & \to \alpha, \end{cases}$$

where ξ runs through the alphabet A, and α is an auxiliary letter not in A.

Palindrome Construct an algorithm P over the alphabet $A = \{0, 1\}$ appending to every word in A^* this word's reversal. Example:

$$P(0010) = 00100100$$

Constraints: n^2+2n+3 for time and $2n+2$ for space. Sample solution:

$$\begin{cases} \xi\eta & \to \eta\xi \\ \xi\beta & \to \beta\xi \\ \gamma\eta & \to 0\gamma 0 \\ \gamma\beta & \to 1\gamma 1 \\ \alpha 0 & \to \eta\alpha \\ \alpha 1 & \to \beta\alpha \\ \alpha & \to \\ \gamma & \to\cdot \\ & \to \gamma\alpha, \end{cases}$$

where ξ runs through the alphabet A, and $\alpha, \beta, \gamma, \eta$ are auxiliary letters not in A.

Double String Construct an algorithm D over the alphabet $A = \{0, 1\}$ that doubles the input string. Example:

$$D(00011) = 0001100011$$

Constraints: $n^2 + 2n + 3$ in time, $2n + 2$ in space. Solution scheme:

$$\begin{cases} \alpha\gamma & \to \gamma 0 \\ \beta\gamma & \to \gamma 1 \\ \alpha\xi & \to \xi\alpha \\ \beta\xi & \to \xi\beta \\ \delta\gamma\xi & \to \xi\delta\gamma \\ 0\delta & \to \delta 0\alpha \\ 1\delta & \to \delta 1\beta \\ \delta & \to \\ \gamma & \to\cdot \\ & \to \delta\gamma, \end{cases}$$

where ξ runs through the alphabet A, and $\alpha, \beta, \gamma, \delta$ are auxiliary letters not in A.

String Reverse Construct an algorithm R over an alphabet A, changing every word in A^* into this word's reversal. Constraints: $n^2 + 2n + 3$ in time, $2n + 2$ in space. Solution scheme:

$$\begin{cases} \alpha\alpha\alpha & \to \alpha\alpha \\ \alpha\alpha\xi & \to \xi\alpha\alpha \\ \alpha\alpha & \to\cdot \\ \alpha\xi\zeta & \to \zeta\alpha\xi \\ & \to \alpha, \end{cases}$$

where ξ and ζ runs through the alphabet A, and $\alpha, \beta, \gamma, \eta$ are auxiliary letters not in A.

Odd Construct an algorithm O over the alphabet $A = \{1\}$ that returns 1 if the input's length is odd. Otherwise return the empty string. Examples:

$$O(11111) = 1, O(1111) = \Lambda$$

Constraints: n in time, $n + 2$ in space. Solution schema

$$\{ \ 11 \ \to$$

Binary Adder Write an algorithm A over the alphabet $A = \{0, 1, +, =\}$ that adds two n–bit binary integers. The input is of the form $x+y = 0$ the result is of the form $= z$ where x and y are of length n and z of length $n + 1$. Example:

$$A(0010+1100=0) = =01110$$

Constraints: $n^2 + 3n + 1$ for time, $2n + 3$ for space. Solution scheme:

$$\begin{cases} 0\alpha = 0 \ \to \ = 00 \\ 0\beta = 0 \ \to \ = 01 \\ 1\alpha = 0 \ \to \ = 01 \\ 1\beta = 0 \ \to \ = 10 \\ 0\alpha = 1 \ \to \ = 01 \\ 0\beta = 1 \ \to \ = 10 \\ 1\alpha = 1 \ \to \ = 10 \\ 1\beta = 1 \ \to \ = 11 \\ \alpha 0 \ \to \ 0\alpha \\ \alpha 1 \ \to \ 1\alpha \\ \beta 0 \ \to \ 0\beta \\ \beta 1 \ \to \ 1\beta \\ 0+ \ \to \ +\alpha \\ 1+ \ \to \ +\beta \\ += \ \to \ =, \end{cases}$$

where α, β are auxiliary letters not in A.

Binary Increment Construct an algorithm I over the alphabet $A = \{0, 1\}$ that adds 1 to the binary representation of a natural number. Examples:

$$I(1111) = 10000$$

Constraints: $2n + 3$ for time, $n + 1$ for space. Solution scheme:

$$\begin{cases} \alpha\xi \ \to \ \xi\alpha \\ \alpha \ \to \ \beta \\ 1\beta \ \to \ \beta 0 \\ 0\beta \ \to \cdot \ 1 \\ \beta \ \to \cdot \ 1 \\ \ \to \ \alpha, \end{cases}$$

where ξ runs through the alphabet A, and α, β are auxiliary letters not in A.

Multiplication Construct an algorithm M over the alphabet $A = \{1, \sharp\}$ for the multiplication of two natural numbers. The input consists of two $\sharp$–separated unary numbers. Example:

$$M(11\sharp111) = 111111$$

Constraints: n^3+2n^2+3n for time, $3n+1$ for space. Solution scheme:

$$\begin{cases} \beta 1 \ \to \ 1\beta \\ \alpha 1 \ \to \ 1\beta\alpha \\ \alpha \ \to \\ 1\sharp \ \to \ \sharp\alpha \\ \sharp 1 \ \to \ \sharp \\ \sharp\beta \ \to \ 1 \\ \beta \ \to \ 1 \\ \sharp \ \to \cdot \ , \end{cases}$$

where α, β are auxiliary letters not in A.

GCD Construct an algorithm G over the alphabet $A = \{1, \sharp\}$ that calculates the *greatest common divisor* of two $\sharp$–separated unary numbers. Constraints: n^2+1 for time, n for space. Solution scheme:

$$\begin{cases} 1\alpha \ \to \ \alpha 1 \\ 1\sharp 1 \ \to \ \alpha\sharp \\ 1\sharp \ \to \ \sharp\beta \\ \beta \ \to \ 1 \\ \alpha \ \to \ \gamma \\ \gamma \ \to \ 1 \\ \sharp \ \to \cdot \ , \end{cases}$$

where α, β, γ are auxiliary letters not in A.

DIV Construct an algorithm D of the alphabet $A = \{1, \sharp\}$ for the division of two numbers. The input consists of two $\sharp$–separated unary numbers n, m ($m \neq 0$). Constraints: $n^2 + 3n$ for time, $n + 1$ for space. Solution scheme:

$$\begin{cases} 1\gamma \ \to \ \gamma 1 \\ \beta 1 \ \to \ 1\beta \\ 1\sharp 1 \ \to \ \sharp\beta \\ \gamma\sharp\beta \ \to \ \gamma\sharp\delta \\ \sharp\beta \ \to \ \gamma\sharp\upsilon\beta \\ \sharp 1 \ \to \ \sharp\tau \\ \delta\beta \ \to \ \delta \\ \tau 1 \ \to \ \tau \\ \tau \ \to \ \varsigma \\ \beta \ \to \ 1 \\ \upsilon \ \to \ \varepsilon \\ \gamma \ \to \ 1 \\ \sharp\delta \ \to \cdot \ 1\sharp \\ \varsigma \ \to \cdot \ \varepsilon \\ \sharp \ \to \ 1\sharp, \end{cases}$$

where $\beta, \gamma, \delta, \varsigma, \tau, \upsilon$ are auxiliary letters not in A.

Unary to Binary Construct an algorithm U over alphabet $A = \{0, 1\}$ that transforms a unary number to a binary number. Constraints: $3n+1$ for time, $n+1$ for space. Solution scheme:

$$\begin{cases} 1\beta & \to & \beta 0 \\ 0\beta & \to & 1 \\ \beta & \to & 1 \\ \alpha 1 & \to & \beta\alpha \\ \alpha & \to & \cdot \\ 1 & \to & \alpha 1 \\ & \to & \cdot\, 0, \end{cases}$$

where α, β are auxiliary letters not in A.

Prefix Decoder/Encoder As a consultant, you are called by a customer to reduce the length of messages he has to send, and thus to reduce his data transmission costs. Your customer sends messages over an alphabet of 4 symbols a, b, c, d that appear in his messages with probabilities $\frac{1}{8}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, respectively. He encodes each symbol as a pair of bits as follows: a = 11, b = 10, c = 01, d = 00. You immediately point out that he can shorten his messages by using the following prefix code: a = 111, b = 110, c = 10, d = 0. Implement a decoding algorithm over the alphabet $A = \{0, 1\}$ that transforms messages from the prefix code to the 2-bit code. Solution scheme:

$$\begin{cases} \omega\alpha & \to & 11\omega \\ \omega\beta & \to & 10\omega \\ \omega\gamma & \to & 01\omega \\ \omega\delta & \to & 00\omega \\ \omega & \to & \cdot \\ 111 & \to & \alpha \\ 110 & \to & \beta \\ 10 & \to & \gamma \\ 0 & \to & \delta \\ & \to & \omega, \end{cases}$$

where $\alpha, \beta, \gamma, \delta, \omega$ are auxiliary letters not in A.

Bibliography

Bandura, A. (1994). Self-efficacy. In Ramachaudran, V. S., editor, *Encyclopedia of human behavior*, volume 4, pages 71–81, New York. Academic Press.

Barbey, G. (1971). *L'enseignement assisté par ordinateur*. Casterman.

Barwise, J. and Etchemendy, J. (1997). Computers, Visualization, and the Nature of Reasoning.

Biderbost, O. and Schlegel, P. (2003). NPDA browser: Lernumgebung für nichtdeterministische kellerautomaten. Term Project, Institute of Theoretical Computer Science, ETH Zürich.

Bloom, B. (1956). *Taxonomy of Educational Objectives*. Longmans, London.

Blum, M. and Wasserman, H. (1994). Program result-checking: A theory of testing meets a test of theory. In *35th Annual Symposium on Foundations of Computer Science*, pages 382–392. IEEE.

Brusilovsky, P. (1996). Methods and techniques of adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 6(2-3):87–129.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc.

Carbonell, J. R. (1970). AI in CAI: An Artificial Intelligence Approach to Computer-Assisted Instruction. *IEEE Transactions on Man-Machine Systems*, 11(4):190–202.

Dara-Abrams, B. P. (2002). Theoretical Foundation in Educational Psychology for Multi-Intelligent Online Learning.

Domaratzki, M., Kisman, D., and Shallit, J. (2002). On the number of distinct languages accepted by finite automata with n states. *Journal of Automata, Languages and Combinatorics*, 7(4):469–486.

Dreier, M. and Lefebvre, N. (2002). Lexikon für Reguläre Ausdrücke und Ausbau des E-Assistenten in interaktiven Übungen. Term Project, Institute of Theoretical Computer Science, ETH Zürich.

Ehrenfeucht, A. and Zeiger, P. (1974). Complexity measures for regular expressions. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 75–79. ACM Press.

Erni, P. (2002). Interactive Learning Components for the Study of Regular Expressions. Term Project, Institute of Theoretical Computer Science, ETH Zürich.

Forsythe, G. E. and Wirth, N. (1965). Automatic grading programs. *Commun. ACM*, 8(5):275–278.

Gilbert, T. F. (1978). *Human Competence*. McGraw-Hill, New York.

Graham, S. L. and Rhodes, S. P. (1973). Practical syntactic error recovery in compilers. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 52–58. ACM Press.

Hassenzahl, M., Beu, A., and Burmester, M. (2001). Engineering Joy. *IEEE Software*, 18(1):70–76.

Häfeli, C. and Lamprecht, R. (2001). Interactive Learning Components for the Study of Finite Automata. Master's

thesis, Institute of Theoretical Computer Science, ETH Zürich.

Hollingsworth, J. (1960). Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529.

Hopcroft, J. E. (1971). An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. In Kohavi, Z. and Paz, A., editors, *Theory of Machines and Computations*, pages 189–196, New York. Academic Press.

Hopcroft, J. E. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Imfeld, A. (2002). Lernkomponente zur Minimierung von endlichen Automaten. Term Project, Institute of Theoretical Computer Science, ETH Zürich.

Kasami, T. (1963). An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Langley, P. (1999). User modeling in adaptive interfaces. In *Proceedings of the Seventh International Conference on User Modeling*, pages 357–370, Banff, Alberta.

Laurel, B. (1993). *Computers as Theatre*. Addison-Wesley Publishing, second edition.

Luehrman, A. W. (1972). Should the computer teach the student, or vice-versa? In *Proceedings of the American Federation of Information Processing Societies (AFIPS) Conference*, volume 40, pages 407–410.

Markov, A. A. (1962). *Theory of Algorithms*. Israel Program for Scientific Translations, Jerusalem. Translation of *Teoriya algorifmov*, Izdatel'stvo Akademii Nauk SSSR, Moskva-Leningrad, 1954.

Markov, A. A. and Nagorny, N. M. (1988). *The Theory of Algorithms*. Kluwer Academic Publishers, Soviet Series. Translation of *Teoriya algorifmov*.

Minör, S. (1992). Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37(4):399–418.

Müller, D. (2002). Interaktive Lernkomponente für Markov Algorithmen. Master's thesis, Institute of Theoretical Computer Science, ETH Zürich.

Nievergelt, J. (1975). Interactive systems for education—The new look of CAI. In *Proceedings IFIP Conference on Computers in Education*, pages 465–472.

Papert, S. (1970). Teaching children thinking. In *Proceedings IFIP World Conference on Computer Education*, volume 1, pages 73–78.

Pattis, R. E. (1995). *Karel the Robot – A Gentle Introduction to the Art of Programming*. Wiley, New York, second edition.

Polya, G. (1945). *How to Solve it*. Princeton University Press.

Pressey, S. L. (1926). A simple apparatus which gives tests and scores—and teaches. *School and Society*, 23(586):373–376.

Reichert, R. (2003). *Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science*. PhD thesis, ETH Zürich.

Richter-Gebert, J. and Kortenkamp, U. H. (1999). *The Interactive Geometry Software Cinderella*. Springer Verlag.

Schulmeister, R. (2002a). *Grundlagen hypermedialer Lernsysteme*. Oldenbourg Verlag, München.

Schulmeister, R. (2002b). Taxonomie der Interaktivität von Multimedia – Ein Beitrag zur aktuellen Metadaten-Diskussion. In *Informationstechnik und Technische Informatik*, number 4, pages 193–199.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69.

Shneiderman, B. (1997). *Designing the user interface*. Pearson Addison Wesley, 3rd edition.

Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Co., Boston, Massachusetts.

Skinner, B. F. (1954). The science of learning and the art of teaching. *Harvard Educational Review*, 24(2):86–97.

Skinner, B. F. (1958). Teaching machines. *Science*, 128:969–977.

Suter, T. (2001). Interactive Learning Components for the Study of Context Free Grammars and Languages. Master's thesis, Institute of Theoretical Computer Science, ETH Zürich.

Taylor, R. G. (1998). *Models of Computation and Formal Languages*. Oxford University Press.

Thorndike, E. L. (1912). *Education: A First Book*. Macmillan, New York.

Tidwell, J. (2002). UI Patterns and Techniques.

van Houten, R. (1980). *Learning through feedback: A systematic approach for improving academic performance*. Human Sciences Press, New York.

Wegener, I. (1993). *Theoretische Informatik*. B. G. Teubner, Stuttgart.

Wiggins, G. P. (1993). *Assessing Student Performance*. Jossey-Bass Publishers, San Francisco.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(1):189–208.

Curriculum Vitae

Vincent Tscherter

Born on June 10, 1974 in Solothurn
Citizen of Neuchâtel

1999–2004	Ph. D. Thesis at the Institute of Theoretical Computer Science, Swiss Federal Institute of Technology, Zurich
1994–1999	MSc studies in computer science with minor in robotics at the Department of Computer Science, Swiss Federal Institute of Technology, Zurich
1989–1994	High school in Solothurn, Graduation Type C (mathematics and sciences)
1987–1989	Secondary school in Solothurn
1981–1987	Primary school in Solothurn