RELATING AUTOMATA TO OTHER FIELDS*

Pradip Peter Dey, Mohammad Amin, Gordon W. Romney, Bhaskar Raj Sinha,
Ronald F. Gonzales, Alireza Farahani and Hassan Badkoobehi
National University, School of Engineering, Technology, and Media
3678 Aero Court, San Diego, CA 92123, U.S.A.
{ pdey, mamin, gromney, bsinha, rgonzales, afarahan, hbadkoob}@nu.edu

ABSTRACT

Automata classes including Turing Machines, Pushdown Automata and Finite Automata define the most elegant models of computation in terms of set processors. This study elaborates pedagogically motivated intuitive and formal relations between mathematical models and other computer science areas, so that students can relate different areas of computer science in a meaningful way. Cases that promote learning about theoretical models are presented with other related areas, such as programming languages, compilers and software design. Dynamic aspects of software can be appropriately modeled by certain automata based models. Visualizations are developed to help students in their initial stages of understanding of these relations. The visualizations demonstrate that mathematical models such as Pushdown Automata are reasonable processors of some programming language features such as balanced {'s and }'s which are evidently helpful in learning this kind of relation.

INTRODUCTION

Turing Machines (TMs), two-stack Pushdown Automata (2PDA), Linear Bounded Automata (LBA), Pushdown Automata (PDA), Finite Automata (FA) and Non-deterministic FA (NFA) are some of the most elegant mathematical models that are taught in automata theory classes. These models of computation define computability in clear terms and provide scope and limitations of computer science in revealing ways. There are some excellent textbooks on automata theory or theory of computation [4, 6, 9, 10, 11, 12, 14, 17, 19]. Following these books one can teach automata theory as a self-contained course. Some of the books, such as [9, 14], give considerable attention to

---

applications of automata. However, an instructor needs to decide about a number of related questions: (a) To what extent these automata should be related to other areas of computer science in teaching computer science classes? (b) Is there any advantage in relating automata classes to other areas? (c) What are the best strategies to relate automata theory to other areas of computer science? We struggle with questions such as these and try to find reasonable answers. Almost every book on automata theory refers to some other fields such as compilers, interpreters, programming languages, data structures and software engineering. However, most of the time automata theory students remain busy with proving theorems of equivalent classes, pumping lemmas and so on. When a course is taught, the focus of the course goes to the central ideas. Thus, when a course on programming languages is taught, usually not enough attention is given to PDA. This paper investigates the ways in which courses on different areas of computer science can be taught by relating certain aspects to benefits students. The remainder of this paper discusses a viable alternative way of teaching some related courses in computer science. It also suggests that instead of teaching a traditional course on automata theory, a new course on "Automata and Related Topics" should be taught with tools and strategies that relate the automata to other subjects.

TOOLS AND STRATEGIES FOR RELATING AUTOMATA TO OTHER FIELDS

Although applications of automata are described in various details in most textbooks [4, 6, 9, 10, 11, 12, 14, 17, 19], additional steps need to be taken to relate automata to other subjects. Some static and dynamic visualization tools can be considered for expanding the applications as relations among the subjects. A special case of the relation between PDA and programming language processing can be considered, where a balanced number of {'s and }'s needs to be processed. This case is presented with a non-regular Context-Free Language, L_m = { $\{^n c\}^n$: where n >= 0 }. It is to be noted that L_m has strings with matching number of {'s and }'s separated by a **c** . That is, strings of L_m are of the form: **c, {c}, {{c}}, {{{c}}}, {{{{c}}}},** . . . . What is interesting about L_m is that it has a string pattern that is similar to that of programming languages such as Java and C++. In fact, syntactic structures of a programming language are defined by Context-Free Grammars (CFGs) in a way that is similar to that of the CFG of L_m given below:

S ➔ {S} | c

This CFG generates or derives a balanced number of {'s and }'s. PDA are designed to accept languages with strings that have similar patterns. That is, a Pushdown Automaton will accept strings like **{c}, {{c}}, {{{c}}}** . . . . Pushdown Automata use a stack data structure for matching equal number of {'s and }'s without counting them. A stack is an interesting data-structure which allows operations such as push and pop and increases or decreases its stored contents in a Last-In-First-Out (LIFO) manner. Stacks are used in PDA for processing CFL's as described in textbooks [4, 6, 9, 10, 11, 12, 14, 17, 19]. One needs to consider multiple ways of presenting automata to students in order to highlight their formal and intuitive relations to other fields. PDA can be presented in various ways including state diagrams. In Figure 1, a PDA for L_m = { $\{^n c\}^n$: where n >= 0 } is presented visually as a finite set of states connected with transitions based on the notations given in [9] with minor adjustments that show the

stack explicitly with the bottom of the stack on the left, and define transitions with the pair: R,T/TP where R is the symbol read from the input, T preceding / is the topmost stack symbol before the transition is taken, TP following / is the sequence of topmost stack symbol(s) after the transition is taken and P is an optional symbol which appears only with "push transitions". The state diagrams for PDA given in [7] do not explicitly show the stack.

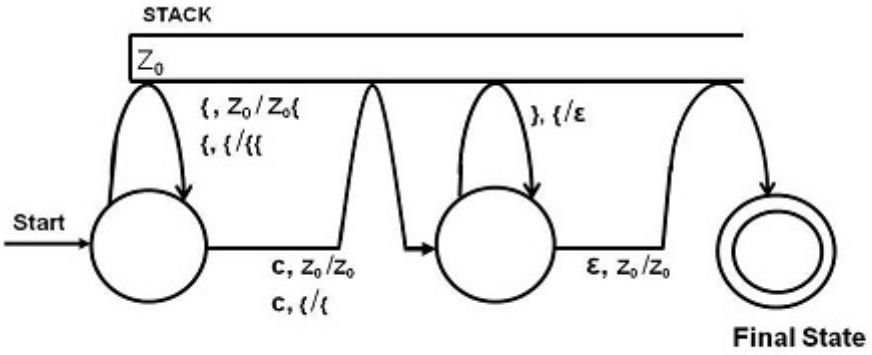

Figure 1: A Pushdown Automaton for L_m = { $\{^n c\}^n$: where n >= 0 }.

Ordinarily, static visualizations of PDA can be done with a sequence of state diagrams, such as the one given in Figure 1. One type of dynamic visualization is shown in the form of an animation on the following web site: www.asethome.org/pda. Our approach is based on the pioneering work of Rodger in the area of visualization of automata [15-16]. Some recent studies have criticized dynamic visualizations compared to that of static ones [13, 20], which does not apply to our visualizations, because these are not comparable to static ones.

Suppose a string like **{{c}}** is given as an input to the PDA of Figure 1. Then, the machine starts at the start state and scans the first { from the input and pushes a { into the stack by taking the transition marked by, **{ , Z_0 / Z_0{** . The meaning of this transition label is "when reading a { and the stack is empty (marked by Z_0) push a **{** onto the empty stack (marked by Z_0)". Then it consumes the next **{** from the input by taking the same loop with the transition marked by **{, { /{{** . Next, it consumes the symbol c by taking the transition marked by **c, { /{** which means "read a c from the input when there is a **{** on top of the stack and leave the stack unchanged". Next, it reads the fourth symbol, **}** , from the input and pops a **}** from the stack taking the transition marked by **}, { /□** . Then, it scans the next **}** by taking the same transition marked by **}, { /□** again. Then, it reaches the final state by taking the transition marked **□, Z_0 /Z_0** . At that moment the stack is empty and the entire input is consumed and therefore the input **{{c}}** is accepted by the machine. The PDA given above accepts any string with a sequence of {'s followed by a c followed a number of **}**'s that balances **{**'s. That is, strings such as **c, {c}, {{c}}, {{{c}}},** . . . . are accepted by the machine. An input is accepted by a PDA if all of the following conditions are met simultaneously: (a) the input is entirely consumed, that is, no other symbols left in the input; (b) the machine is in a final state; (c) the stack is empty. In a survey, 19 out of 26 respondents (73%) found the dynamic visualization of PDA helpful in learning the relation between PDA and programming language features.

In addition to the connection between PDA and syntactic analysis of programming languages, the relations among lexical analyses, FA and NFA need to be demonstrated. An animation of an NFA for a programming language lexical analysis [9, 18] is given on a page linked to www.asethome.org/automata. It rejects identifiers that start with a digit and accepts other well-defined identifiers. In addition, programming assignments can be developed based on FA or NFA, as shown in some examples linked to the above website. Model checking is another important area of application of automata [3].

Software development relies on modeling the software in various levels, including the design level. Design tools based on statecharts [7-8] have been very useful for modeling dynamic aspects of software. Statecharts are basically TMs presented in a notation that is appropriate for representing software features in an intuitive way. A statechart diagram representing the dynamic aspects of computing average family size of a town is given in Figure 2; an experimental implementation of the software in an applet can be found at http://www.asethome.org/automata/soft/average.html.

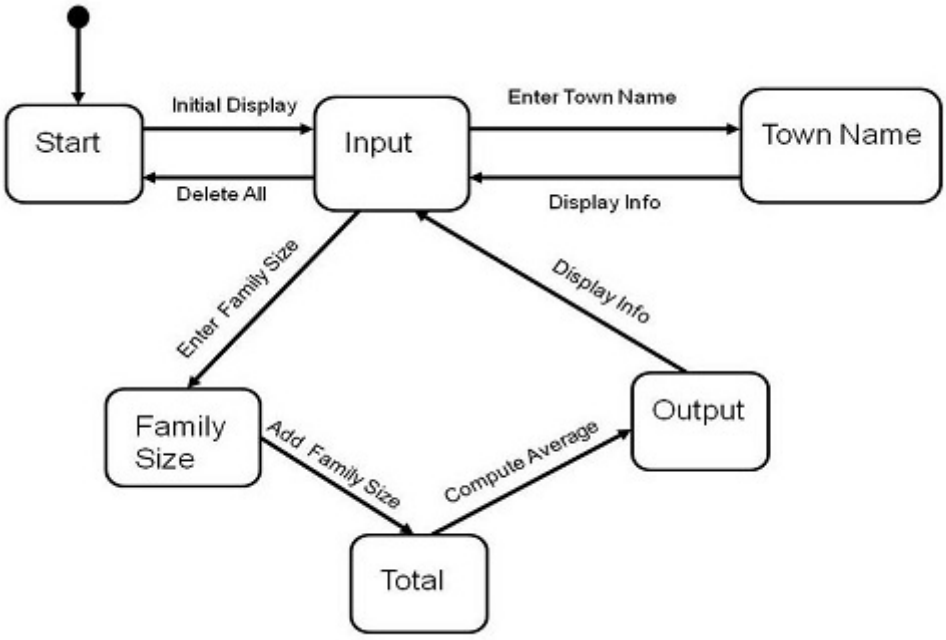

Figure 2: A Statechart Diagram for computing average family size

The relationships among automata, software engineering, compilers, model checking, data structures and programming languages can be best understood in a course on "Automata and Related Topics" at the undergraduate level. A description for this course is not given here, because the description should be based on the needs of the program in a college. Either [9] or [14] can be used as the main text for the course along with supplemental materials such as [15]. Examples of applications described in [9] and [14] are very helpful for teaching purposes. The course may use agile problem driven teaching [5] or problem-based learning [1-2] or game-based learning [21] for effective teaching [22]. Agile Problem Driven Teaching is closely related to problem-based learning [1-2]; however, it combines problem-based free inquiry with direct instructions in an agile process in order to achieve the course learning outcomes. Visualization

clarifies concepts and supports problem solving. The dynamic visualization of PDA, mentioned earlier, was introduced to the students first in the context of problem solving, although it was always available at the web site.

CONCLUDING REMARKS

There is no doubt that several courses, including compilers, programming languages, data structures and software engineering can be related to automata theory. These courses should be taught using the tools and strategies that relate them well. A new course on "Automata and Related Topics" can be taught using agile problem driven teaching strategies along with static and dynamic visualizations. There are advantages in relating topics of different courses together in order to teach them with reasonable interpretations and applications. Future research may concentrate on collecting and analyzing data for measuring teaching effectiveness and students' satisfaction in this area.

REFERENCES

[1] Barell, J., *Problem-based learning: An Inquiry Approach*, Corwin Press, 2nd Edition, 2006.

[2] Barrows, H. S., *How to design a problem-based curriculum for the preclinical years*. New York: Springer, 1985.

[3] Clarke, E., Grumberg, O., Peled, D., *Model Checking*, MIT Press, Cambridge, 1999.

[4] Cohen, D., *Introduction to Computer Theory*, (2nd ed.), John Wiley & Sons, 1997.

[5] Dey, P., Gatton, T., Amin, M., Wyne, M., Romney, G., Farahani, A., Datta, A., Badkoobehi, H., Belcher, R., Tigli, O., Cruz, A., Agile Problem Driven Teaching in Engineering, Science and Technology. In *the Proceedings of the American Society for Engineering Education-Pacific Southwest 2009 conference* ASEE-PSW 2009, San Diego, California, U.S.A., March 19-20, 2009. http://www.asethome.org/asee/ASEE_PSW_2009_Proceedings.pdf

[6] Goddard, W., *Introducing the Theory of Computation*, Jones & Bartlett, 2008.

[7] Harel, D., Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, v.8 n.3, p.231-274, 1987

[8] Harel, D., Politi, M., *Modeling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill, 1998.

[9] Hopcroft, E., Motwani, R., Ullman, D., *Introduction to Automata Theory, Languages, and Computation*, Pearson Education, 2007.

[10] Kinber, E., Smith, C., *Theory of Computing: A Gentle Introduction*, Prentice Hall, 2001.

[11] Kozen, D., *Theory of Computation*, Springer, 2006.

[12] Lewis, H., Papadimitriou, C., *Elements of the Theory of Computation*, Prentice Hall, 1998.

[13] Lowe, R., Schnotz, W., *Learning with Animation: Research Implications for Design*, Cambridge University Press, 2007.

[14] Rich, E., *Automata, Computability and Complexity: Theory and Applications*, Prentice Hall, 2007.

[15] Rodger, S. H., Finley, T. W., *JFLAP: An Interactive Formal Languages and Automata Package,* Jones & Bartlett Publishers, 2006.

[16] Rodger, S.H., Bressler, B., Finley, T., Reading, S., Turning automata theory into a hands-on course. SIGCSE 2006: 379-383., 2006.

[17] Sakarovitch, J., *Elements of Automata Theory*, Cambridge University Press, 2009.

[18] Sebesta, R., *Concepts of Programming Languages*, 9th Ed., Addison-Wesley, 2009.

[19] Sipser, M., *Introduction to the Theory of Computation*, PWS Publishing, 2006.

[20] Tversky, B., Morrison, J., Betrancourt, M., Animation: Can It Facilitate? *International Journal of Human Computer Studies,* Volume 57, 247-262, 2002.

[21] Van, R. E., Game-based learning. *EDUCAUSE Review*, 41(2), 16-30, 2008.

[22] Wallis, C., How to make great teachers? *Time*, 171 (8), 2008.