Visualization of a Mathematical Model of Computation

Pradip Peter Dey, Gordon W. Romney, Mohammad Amin
pdey@nu.edu, gromney@nu.edu, mamin@nu.edu,
Alireza Farahani, Hassan Badkoobehi, and Ronald F. Gonzales
afarahan@nu.edu, hbadkoob@nu.edu, rgonzales@nu.edu

School of Engineering, Technology and Media, National University
3678 Aero Court, San Diego, CA 92123, USA

Abstract: *Mathematical models of computation such as Turing Machines, Pushdown Automata, and Finite Automata are useful in modeling real world computational problems. This paper presents dynamic visualization of some Turing Machines which clarify computational problem-solving aspects of these models. The design and implementation of the dynamic visualization are performed in an iterative process making improvements through successive iterations. The dynamic visualization is available at the following website: http://www.asethome.org/math/. An example of static visualization of one of the models is presented for consideration of the relative advantages and disadvantages of both visualization techniques.*

1. Introduction

Mathematical models including Turing Machines (TMs), two-stack Pushdown Automata (2PDA), Linear Bounded Automata (LBA), Pushdown Automata (PDA), Finite Automata (FA) and Non-deterministic FA (NFA) define the most elegant machines of computation in terms of set processors. These models of computation define computability in clear terms and provide scope and limitations of computer science in revealing ways. There are excellent textbooks on automata theory or theory of computation [1, 2, 3, 4, 5, 6, 7, 8, 9] that describe these models often with revealing illustrations. For portraying their structural relationship, the models are usually presented in the Chomsky Hierarchy shown in Table 1.

The Chomsky Hierarchy of Grammars and Languages			
Type	Language	Grammar	Acceptor
0	Recursively Enumerable	Unrestricted Phrase Structure Grammar (UG)	TMs = 2PDA = Post Machines
1	Context-Sensitive	Context-Sensitive Grammar (CSG)	LBA = Turing Machines with bounded tape.
2	Context-Free	Context-Free Grammar (CFG)	PDA = OSA
3	Regular	Regular Grammar (RG)	FA = NFA = Transition Graphs

Table 1: The Chomsky Hierarchy

TMs define the most powerful automata class for processing the most complex sets or languages, namely, recursively enumerable sets. The class of recursively enumerable sets properly includes all other computable sets as shown in Figure 1. Inspired by an address given by David Hilbert in 1900, Alan Turing developed these machines for mechanizing aspects of mathematics in the form of algorithms [10]. TMs are equivalent to 2PDA as proven by Minsky [11]. LBA are

TMs with some restrictions on the use of memory, designed for processing Context-Sensitive languages. PDA are defined to have exactly one stack and they are non-deterministic unless otherwise explicitly stated. PDA are acceptors of the class of context-free languages. They are less powerful than TMs; they cannot accept non-context-free languages. Programming languages are compiled using Context-Free Grammars (CFGs) in processing models characterized by PDA or One-Stack Automata (OSA)[12]. FA accept a proper subset of CFLs called regular languages denoted by regular expressions. FA are deterministic unless otherwise explicitly stated. Non-deterministic FA are equivalent to FA in the sense that they accept the same sets defined by regular expressions. In order to bring out the relationship among classes of grammars, languages and automata, the preceding narrative information is usually presented in the Chomsky Hierarchy [1] as shown in Table-1. The models in Chomsky Hierarchy are generally taught in an automata theory course using standard textbooks [1-9]. This study develops dynamic graphics for TMs in order to demonstrate computational aspects through animation. This approach in the teaching learning environments is based on the pioneering work of Rodger in the area of visualization of automata [13-16]. In this paper, we present visualization of TMs in order to demonstrate their behavior in actions. TMs are the most powerful machines which are capable of processing the class of recursively enumerable sets which properly includes all other computable sets [1-9] as shown in Figure 1.

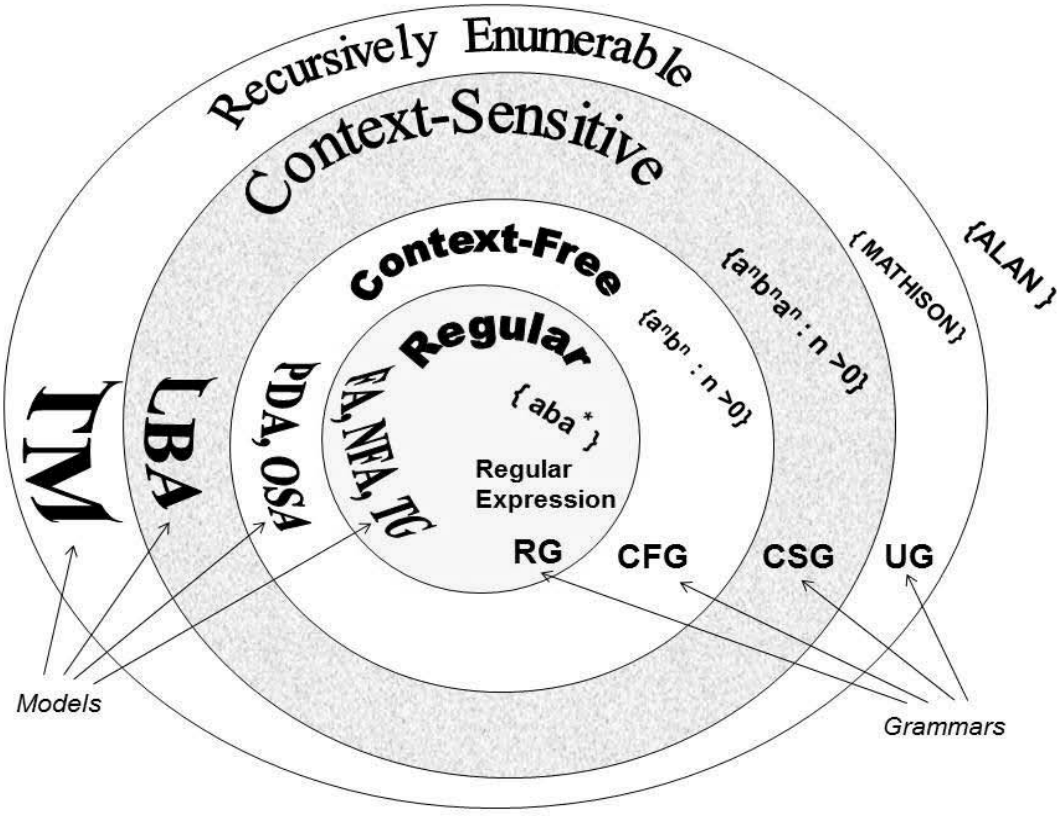

Figure 1: Relating Automata, Languages and Grammars

Please note that an example language is mentioned within { } in each class of Figure 1; each of these language classes is described in a popular textbook by Daniel Cohen [1]. Regular languages are characterized by regular expressions which are shown in the innermost circle in Figure 1. Every regular language is defined by a Regular Grammar (RG). These languages are

properly included in context-free languages or sets. Every context-free language is defined by a Context-Free Grammar (CFG). Similarly, a context-sensitive language is defined by a Context-Sensitive Grammar (CSG). A recursively enumerable set is defined by an Unrestricted Grammar (UG) or Unrestricted Phrase Structure Grammar. Please note that ALAN is a set which is proven to be not computable [1]. For demonstration purposes, we develop dynamic visualization of TMs for two languages (1) { aba* }, and (2) { $a^n b^n a^n$: n > 0 }. The first one is a regular language characterized by a regular expression. The second language is a non-context-free language with a complex matching pattern explained in the next section. The visualization examples for these two languages are portrayed on the website: http://www.asethome.org/math/.

2. Turing Machines

Alan Turing developed a class of computational models in 1936, which have come to be known as TMs [10]. In the same year, Emil Post independently introduced algorithm machines that have come to be known as Post Machines (PMs) [17]. TMs and PMs are proven to be equivalent and their theory developed in 1930s and 1940s has provided the foundation of the theory of computation. As evident from the textbooks [1-9], TMs are the most popular models for recursively enumerable sets although PMs and 2PDA are also widely used for modeling purposes. Following Cohen [1], we define TMs as follows.

Definition:
A TM is composed of six components:

1. An alphabet, Σ, which is a finite non-empty set of symbols from which input is built.
2. A READ/WRITE TAPE divided into a sequence of numbered cells; each of the cells contains one character or a blank, Δ. The input is presented to the machine one letter per cell beginning in the leftmost cell. The rest of the tape is initially filled with blanks.
3. A TAPE HEAD points to the current letter being read from the READ/WRITE TAPE. It can in one step read the contents of the READ/WRITE TAPE, write a symbol on the tape and move left or right one cell.
4. An alphabet, Γ of symbols for the READ/WRITE TAPE. This can include symbols of Σ.
5. A finite set of states including one start state from which execution of the machine begins and some (may be none) HALT states that cause execution to terminate when entered.
6. A set of transitions from state to state where each transition has three elements:
   (Read-Letter, Write-Letter, Move)
   The first element is a letter read by the TAPE HEAD of the machine from the current cell. The second element is a letter written on the tape on the same cell where the first element was read from. The third element, Move, tells the TAPE HEAD whether to move one cell right, R, or one cell left, L. The HALT state cannot have any outgoing transition. To terminate execution on certain input successfully, the machine must be led to a HALT state. The input is then said to be accepted by the machine.

The above definition of TM is based on Cohen [1]; however, similar definitions can be found in other textbooks [2-8] which substantially reflect Turing's original formulation [10]. In order to clarify important aspects of the definition, consider the example TM presented graphically in Figure 2 with the input **abaa.**

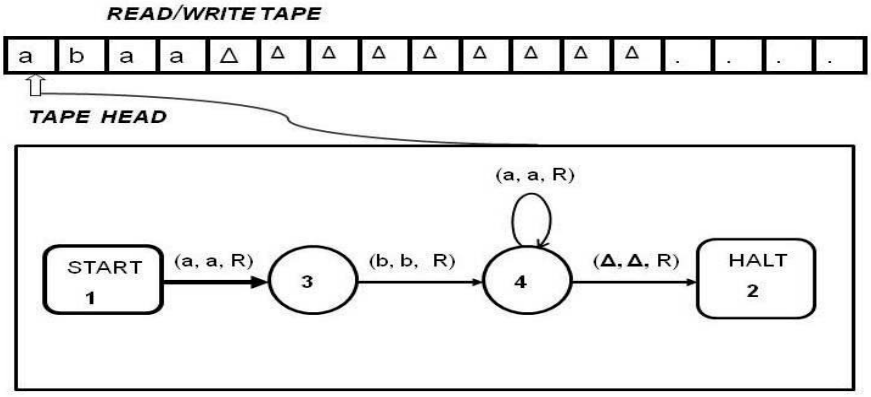

Figure 2: A Turing Machine for **aba*** with input **abaa**

The TM given in Figure 2 is designed to accept every string of the following set: { ab, aba, abaa, abaaa, abaaaa, abaaaaa, **. . .** }. Such a set is called a language. A language is a set of strings. This language is usually abbreviated as **aba*** which is a regular expression. The star in this string, after **a** is known as the Kleene star which means zero or more of **a**'s. The regular expression **aba*** means one **a** followed by one **b** followed by zero or more **a'**s. This language denoted by the regular expression **aba*** can be written as:

L_1 = aba* = { ab, aba, abaa, abaaa, abaaaa, abaaaaa, **. . .** }

The TM of Figure 2 is shown to have just begun to process the input by starting at the start state. The TM, then takes the first transition from the start state to state 3, reads the first symbol **a** from the leftmost cell of the tape, writes back **a** on the same cell and moves right. This is shown in Figure 3.

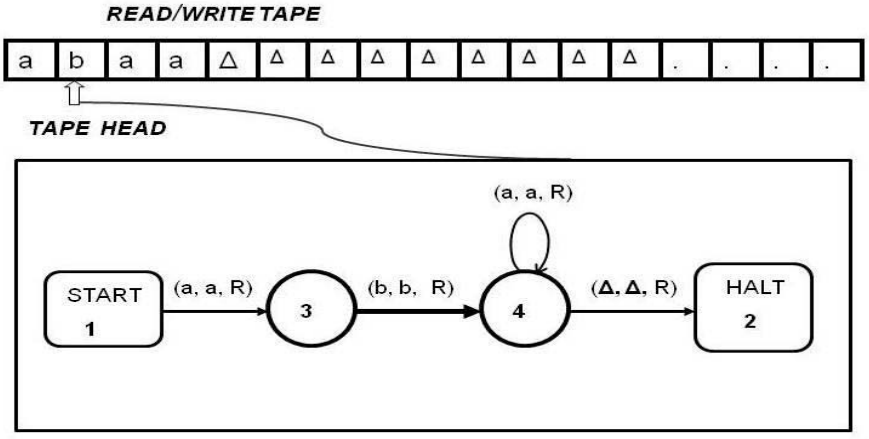

Figure 3: A Turing Machine for aba* which has just scanned the first symbol

Next, the machine, from state 3, taking the transition marked by (b, b, R) reads the symbol **b** from the second cell of the tape and writes back **b** on the same cell and moves right on the tape. Taking this transition the machine reaches state 4. The resulting machine configuration is shown in Figure 4.

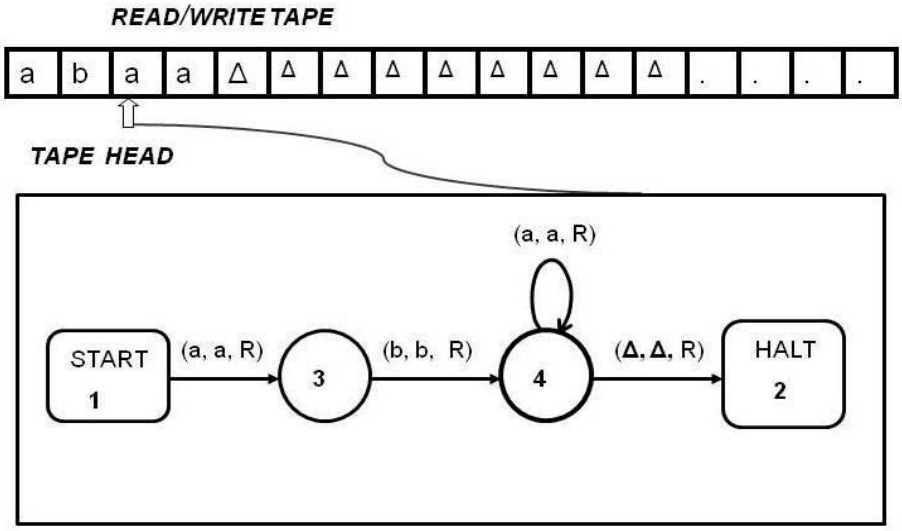

Figure 4: A Turing Machine for aba* which has just scanned the 2nd symbol from the input

Next, the machine, from state 4, taking the transition marked by (a, a, R) reads the symbol **a** from the third cell of the tape and writes back **a** on the same cell and moves right on the tape. Taking this transition the machine comes back to state 4. The machine configuration is shown in Figure 5.

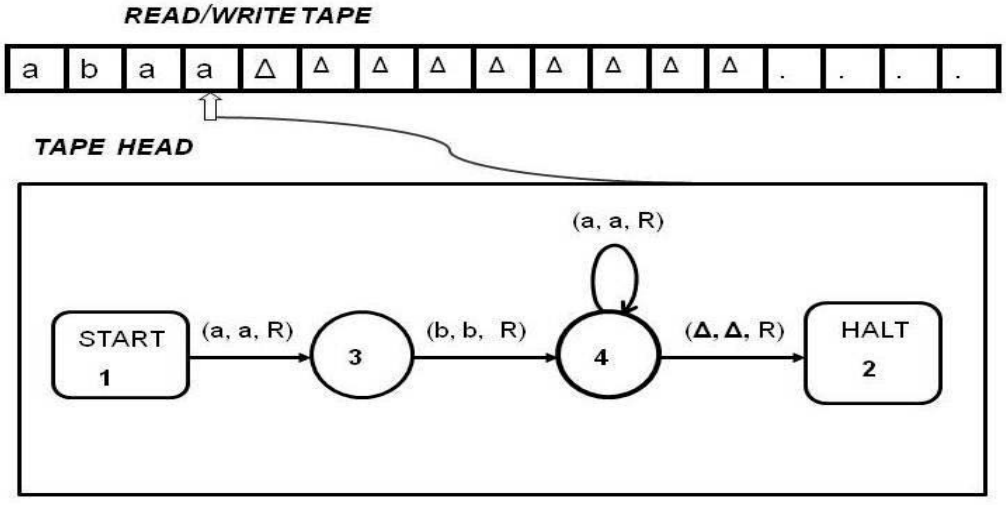

Figure 5. A Turing Machine for aba* which has just scanned the 3rd symbol from the input

Next, the machine, from state 4, taking the transition marked by (a, a, R) reads the symbol **a** from the fourth cell of the tape and writes back **a** on the same cell and moves right on the tape. Taking this transition the machine comes back to state 4. The machine configuration is shown in Figure 6. The TM has finished reading the input, the TAPE HEAD now points to blank symbol immediately following the input.

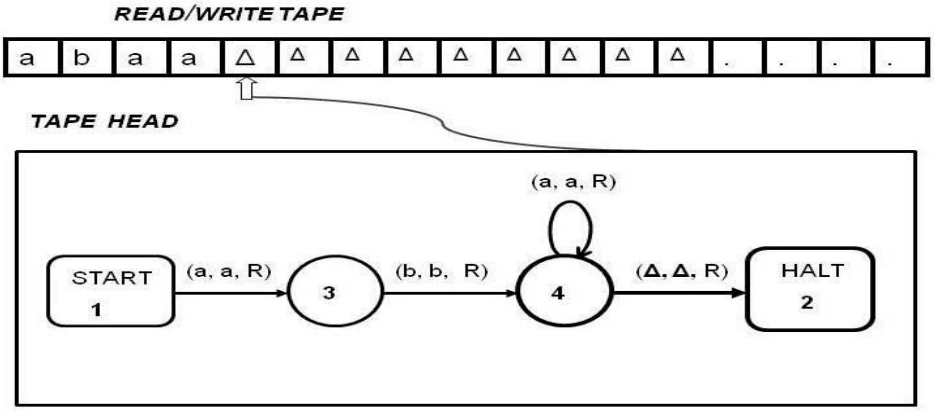

Figure 6. A Turing Machine for aba* which is scanning a null symbol from the tape

In the next step, taking the transition marked by (Δ, Δ, R) from state 4, the machine reads Δ from the fifth cell of the tape, writes back Δ on the same cell and moves right on the tape. Taking this transition the machine reaches the HALT state. The machine accepts the input, **abaa**, since it reaches the HALT state after traversal of transitions from the start state. The machine configuration is shown in Figure 7. From the TM instances of this section, it is obvious that the machine is designed to accept every string of the set, { ab, aba, abaa, abaaa, abaaaa, abaaaaa, abaaaaaa, **. . .** } = **aba*** which is a regular language.

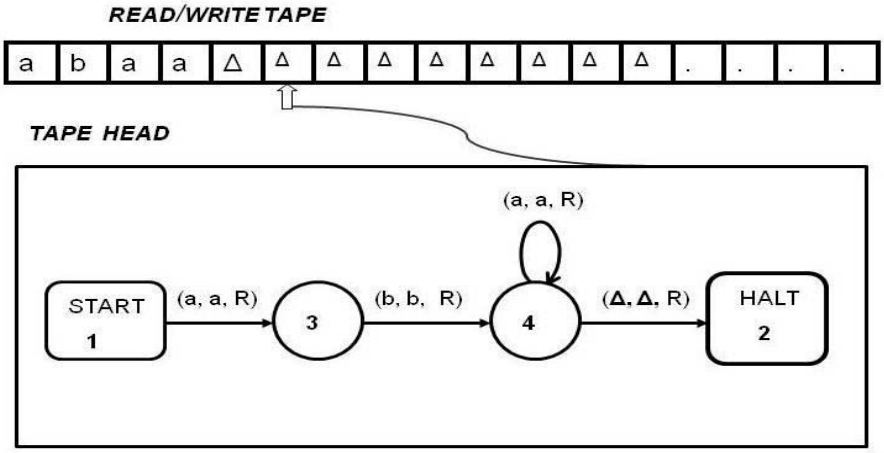

Figure 7. A Turing Machine for **aba*** which has reached the Halt state and accepted the input

The sequence of TM instances of this section constitutes an example of static visualization for processing input by a TM. Ordinarily, static visualizations of TMs can be done with a sequence of state diagrams. One type of dynamic visualization is shown in the form of an animation on the following web site: www.asethome.org/math. Our approach to dynamic visualization is based on the pioneering work of Rodger in the area of visualization of automata [13-16]. Some recent studies have criticized dynamic visualizations compared to that of static ones [18-22], which does not apply to our visualizations, because these are not comparable to static ones. We have provided more features in the dynamic visualization compared to the static one.

3. Languages and their Processing Models

In the previous section, we have run a TM on an input string of L_1 = aba* = { ab, aba, abaa, abaaa, abaaaa, abaaaaa, . . . } which is a regular language. Since regular languages are relatively simple, an NFA, such as the one given in Figure 8 could also process the same language (see [1] for notations).

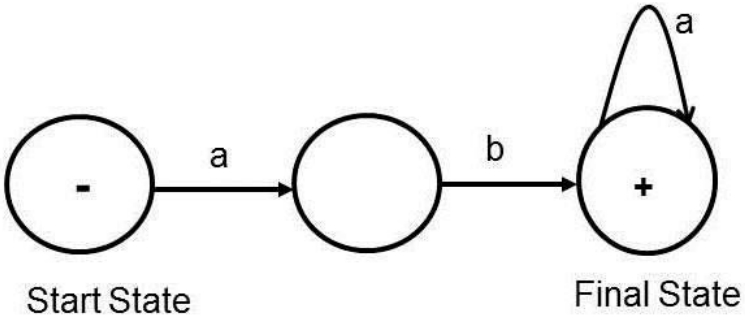

Figure 8. An NFA for **aba***

For understanding of TMs, visualization of two TMs for two very different languages is designed and implemented. The languages are: (1) L_1 = { aba* } = { ab, aba, abaa, abaaa, abaaaa, abaaaaa, . . . }, and (2) L_2 = { $a^n b^n a^n$: where n > 0 } = {aba, aabbaa, aaabbbaaa, aaaabbbbaaaa, . . . }. L_2 is a non-context-free language with a complex pattern for which no NFA or PDA can be given. Usually, TMs are initially introduced with a relatively simple language such as L_1. A screen shot of the initial screen of the dynamic visualization of a TM for L_1 is given in Figure 9. It is designed to help the learner with adequate clarifications of the basic ideas behind TMs. The motivation is to keep everything simple so that the user can concentrate on processing aspects of the TM. This visualization is implemented using HTML and JavaScript Technologies. It basically plays a sequence of picture frames in a loop after the user presses the **START_ANIMATION** button. The animation can be stopped or paused by pressing on the **STOP/PAUSE** button. The comments on the side are intended to help the user to get started and get an understanding of the actions of the TM. A moving arrow shows the execution path of the machine. The reported animations are designed and implemented in an iterative process in the agile software development model [23-25]. The iterative process allows the developer to make changes or improvements in successive iterations [24].

Context-free languages are more complex than regular languages. These cannot be processed by NFA. Programming language features such as balanced braces or balanced brackets cannot be defined by NFA, because these are context-free. An example of a more complex language than context-free languages is L_2 = { $a^n b^n a^n$: where n > 0} = { aba, aabbaa, aaabbbaaa, aaabbbaaa, aaaabbbbaaaa, . . . }. In this language there is a number of initial **a's** followed by the same number of **b's** followed by the same number of **a's**. That is, this language requires three way matching. It is expected that this example gives a deeper understanding of TMs than L_1.

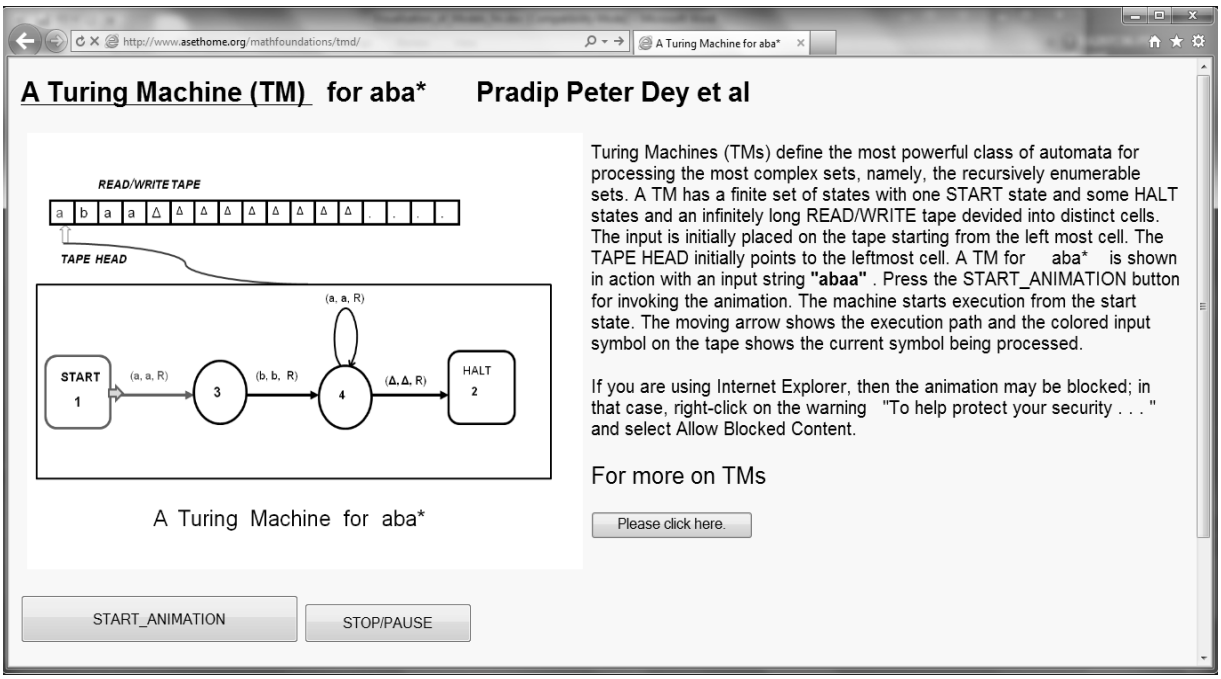

Figure 9. Visualization of a TM for **aba***

4. Concluding Remarks

TMs are useful for solving computational problems. Software development relies on modeling the software in various levels, including the design level. Design tools based on statecharts [26-27] 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. It is reasonable to assume that the visualization of TMs described here would be helpful in the modeling and problem solving process of real world problems.

References

[1] Cohen, D. (1997). *Introduction to Computer Theory*, (2nd ed.), John Wiley & Sons.
[2] Goddard, W. (2008). *Introducing the Theory of Computation*, Jones & Bartlett.
[3] Hopcroft, E., Motwani, R., Ullman, D. (2007). *Introduction to Automata Theory, Languages and Computation*, Pearson Education.
[4] Kinber, E., Smith, C. (2006). *Theory of Computing: A Gentle Introduction*, , Prentice Hall.
[5] Kozen, D. (2006). *Theory of Computation*, Springer.
[6] Lewis, H., Papadimitriou, C. (1998). *Elements of the Theory of Computation*, Prentice Hall.
[7] Rich, E. (2007). *Automata, Computability and Complexity: Theory and Applications*, Prentice Hall.
[8] Sakarovitch, J. (2009). *Elements of Automata Theory*, Cambridge University Press.
[9] Sipser, M., (2006). *Introduction to the Theory of Computation*, PWS Publishing.
[10] Turing, A.M. (1936). On Computable Numbers, with an Application to the Entscheidungs problem, *Proceedings of the London Mathematical Society*, 2. 42: 230–65.
[11] Minsky, M. L. (1961). Recursive insolvability of Post's problem of 'Tag' and other topics in Theory of Turing Machines, *Annals of Mathematics*, 437-55, 1961.

[12] Dey, P., Amin, M., Sinha, B. R. & Farahani, A. (2010) One-Stack Automata as Acceptors of Context-Free Languages, The Journal of Computing Sciences in Colleges, Vol. 26(1), pages 118-123.
[13] Rodger, S.H. (2006). Learning automata and formal languages interactively with JFLAP. ITiCSE 2006: 360.
[14] Rodger, S. H. & Finley, T. W. (2006). *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett Publishers.
[15] Rodger, S.H., Bressler, B., Finley, T. & Reading, S. (2006). Turning automata theory into a hands-on course. SIGCSE 2006: 379-383.
[16] Rodger, S. H., Wiebe, E., Lee, K. M., Morgan, M., Omar, K. & Su, J. (2009). Increasing engagement in automata theory with JFLAP. SIGCSE 2009: 403-407.
[17] Post, **A. (**1936). Finite Combinatory Processes - Formulation 1, *Journal of Symbolic Logic*, 1: 103-105.
[18] Tversky, B., Morrison, J. B., & Betrancourt, M. (2002). Animation: can it facilitate?. *International Journal of Human–Computer Studies*, 57, 247–262.
[19] Lowe, R. K. (1999). Extracting information from an animation during complex visual learning. *European Journal of Psychology of Education*, 14, 225–24.
[20] Lowe, R. K. & Schnotz, W. (Eds) (2007). *Learning with animation*, New York: Cambridge University Press.
[21] Wong, A., Marcus, N., Ayres, P., Smith, L., Cooper, G., Paas, F. & Sweller, J. (2009). Instructional animations can be superior to statics when learning human motor skills, *Computers in Human Behavior*, Volume 25, Issue 2, March 2009, Pages 339-34.
[22] Hegarty, M. (2005). Dynamic visualizations and learning: getting to the difficult questions. *Learning and Instruction*. v14. 343-351.
[23] Braude, E. & Bernstein, M. (2011). *Software Engineering: Modern Approaches*, (2nd Edition), John Wiley & Sons.
[24] Pressman, R. (2010). *Software Engineering: A Practitioner's Approach.* (7th ed.), McGraw-Hill.
[25] Sommerville, I. (2010). *Software Engineering*, 9th Ed., New York, Addison-Wesley
[26] Harel, D. (1987). Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, v.8 n.3, p.231-274.
[27] Harel, D. & Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill.