Adaptive Automata – A Revisited Proposal

João José Neto and César Bravo

Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto sn Travessa 3 n^o 158
CEP 05508-900 São Paulo SP BRASIL
`joao.jose@poli.usp.br`
`lupus@usp.br`

Abstract. This paper impose further discipline to the use of adaptive automata [Jos94], [Iwa00] by restricting some of their features, in order to obtain devices that are easier to create and more readable, without loosing computational power. An improved notation is proposed as a first try towards a language for adaptive paradigm programming.

Keywords: adaptive devices, rule-driven formalisms, self-modifying machines, adaptive automata, adaptive paradigm.

1 Introduction

In [Jos01], the structure and the operation of adaptive rule-based devices have been formally stated. Structured pushdown automata [Jos93] are a variant of classical pushdown automata, in which states are clustered into mutually recursive finite-state sub-machines, which restrict the usage of the control pushdown store to the handling of return states only. Adaptive automata are self-modifying rule-driven formalisms whose underlying non-adaptive devices are the structured pushdown automata. Structured pushdown automata are fully equivalent to classical pushdown automata.

However, despite these features, adaptive automata sometimes lack simplicity, turning them difficult to understand and maintain.

The proposal described in this paper imposes some restrictions to the use of the features of the model in order to obtain devices that are easier to create and understand, without loosing any of the their original computational power.

2 Adaptive Automata

In order to Adaptive automata perform self-modification, adaptive actions attached to their state-transition rules are activated whenever the transition is applied.

The Underlying Structured Pushdown Automata. A finite-state automaton is composed of a set of states, a finite non-empty alphabet, a transition function, an initial state and a set of final states. Transitions map ordered pairs specifying the current state and the current input symbol into a new state.

© Springer-Verlag Berlin Heidelberg 2003

There are two types of transitions from state A to state B:
(a) Transitions $(A, \alpha) \rightarrow B$, which consume an input symbol α; and
(b) Empty transitions $(A, \epsilon) \rightarrow B$, which do not modify the input.

A structured pushdown automaton also exhibits a set of states, a finite non-empty alphabet, an initial state, a set of final states, a pushdown alphabet and a transition function, including internal transitions, like those shown for finite-state automata, and external transitions, responsible for the calling and returning scheme. Beside the two types of internal transitions, sub-machines allow special call and return transitions:
(a) Transitions $(A, \epsilon) \rightarrow (\downarrow B, X)$ from state A, calling a sub-machine whose initial state is X. B is the return state, to which the control will be passed upon a return transition is performed by the called sub-machine. B is pushed onto the pushdown store when these transitions are executed.
(b) Transitions $(C, \epsilon) \rightarrow (\uparrow B, B)$ from some state C in the current sub-machine's set of final states. State B, which represents any state tha has been previously pushed onto the pushdown store by the sub-machine that called the current one, is popped out of the pushdown store and the caller sub-machine is then resumed at the popped state.

The Adaptive Mechanism. Adaptive actions change the behavior of an adaptive automaton by modifying the set of rules defining it. In adaptive automata, the adaptive mechanism consists of executing one adaptive action attached to the state transition rule chosen for application before the rule is performed, and a second one after applying the subjacent state transition rule.

The adaptive mechanism of adaptive automata is described in [Jos01]: it is defined by attaching a pair of (optional) adaptive actions to the subjacent non-adaptive rules defining their transitions, one for execution before the transition takes place and another for being performed after executing the transition.

At each execution step of an adaptive automaton, the device's current state, the contents of the top position in the pushdown storage and the current input symbol determine a set of feasible transitions to be applied. In deterministic cases, the set is either empty (no transition is allowed) or it contains a single transition (in this case, that transition is immediately applied). In non-deterministic cases, more than one transition are allowed to be executed in parallel. In sequential implementations, a backtracking scheme chooses to apply one among the set of allowed transitions.

Adaptive actions are formulated as calls to adaptive parametric functions. These ones describe the modifications to apply to the adaptive automaton whenever they are called. These changes are described and executed in three sequential steps: (a) An optional adaptive action may be specified for execution prior to applying the specific changes to the automaton. (b) A set of elementary adaptive actions specifies the modifications performed by the adaptive action being described. (c) Another optional adaptive action may performed after the specific modifications are applied to the automaton.

Elementary adaptive actions specify the actual modifications to be imposed to the automaton. Changes are performed through three classes of adaptive actions, which specify a transition pattern against which the transitions in use are to be tested: (a) Inspection-type actions (introduced by a question mark in usual notation), which search the current set of transitions in the automaton for transitions whose shape match the given pattern (b) Elimination-type adaptive actions (introduced by a minus sign in usual notation), which eliminate from the current set of transitions in the automaton all transitions matching the given shape. (c) Insertion-type adaptive actions (introduced by a plus sign in usual notation), which add to the set of current transitions a new one, according to the specified shape.

The adaptive mechanism turn a usual automaton into an adaptive one by allowing its set of rules to change dynamically.

3 Improving the Formulation of Adaptive Automata

In this section we discuss some of the main drawbacks of the traditional version of the formalisms used for representing adaptive automata in previous publications.

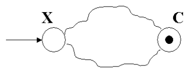
The Notation. The notation used to represent adaptive automata is the first source of drawbacks to be considered in our study, for the simplicity of the model relies on the use of notations with the adequate features: a good notation is expected to be at least compact, simple, expressive, unambiguous, readable, and easy to learn, understand and maintain.

The notations for adaptive automata and structured pushdown automata generally differ in details, but there are two main classes of notations: graphical ones, are better for human visualization, and symbolic ones, which are more compact and machine-readable.

We compared notations still in use, and chose an algebraic and a graphical one, according to their characteristics and functionality:

Transition type	Symbolic notation	Graphical notation
Transition consuming α	$(A, \alpha) \rightarrow B$	$\bigcirc^{A} \xrightarrow{\alpha} \bigcirc^{B}$
Empty transition	$(A, \epsilon) \rightarrow B$	$\bigcirc^{A} \xrightarrow{\epsilon} \bigcirc^{B}$
Initial state	Explicitly indicated	$\rightarrow \bigcirc$
Final state	Explicitly indicated	$\odot$

For structured pushdown automaton, the final choice preserves the notation established for finite-state automata to denote internal transitions in sub-machines. The symbol ϵ (the empty string) has been preserved in both finite-state and structured pushdown notations in order to maintain compatibility with traditional well-established notations. The following table adds notation for expressing (empty-transition) sub-machines calls and returns:

Transition type	Symbolic notation	Graphical notation
Call sub-machine X from state A, returning to state B	$(A, \epsilon) \rightarrow (\downarrow B, X)$	$\bigcirc^{A} \xRightarrow{X} \bigcirc^{B}$
Return to state R after executing the called sub-machine X in its final state C	$(C, \epsilon) \rightarrow (\uparrow R, R)$ for all possible R	

For adaptive automata, all transitions not calling adaptive actions are denoted as stated above. Adaptive transitions make reference up to a pair of adaptive actions, $\mathcal{B}$ (before-action) and $\mathcal{A}$ (after-action). Their notation is summarized in the table below.

A restriction on the transitions to which adaptive actions may be attached, restricts them to be attached to internal transitions only avoiding superposition of the effects of two different sources of complexity in the same transition rule.

Transition type	Symbolic notation	Graphical notation
Adaptive transition with "before" adaptive action attached	$(A, \alpha) \to B[\mathcal{B}\bullet]$	$\bigcirc^A \xrightarrow[\mathcal{B}\bullet]{\alpha} \bigcirc^B$
Adaptive transition "after" adaptive action attached	$(A, \alpha) \to B[\bullet\mathcal{A}]$	$\bigcirc^A \xrightarrow[\bullet\mathcal{A}]{\alpha} \bigcirc^B$
Adaptive transition with both adaptive action attached	$(A, \alpha) \to B[\mathcal{B} \bullet \mathcal{A}]$	$\bigcirc^A \xrightarrow[\mathcal{B}\bullet\mathcal{A}]{\alpha} \bigcirc^B$

In the general case, adaptive actions $\mathcal{B}$ and $\mathcal{A}$ are representations of parametric calls to adaptive functions, which have the general form $M\ (p_1, p_2, \ldots, p_n)$ where $p_1, p_2, \ldots, p_n$ are n arguments passed to an adaptive function named M.

Adaptive actions are symbolically declared apart from the adaptive automaton, and they comprehend a header and a body. In the header, the name and the formal parameters of the adaptive function are defined, followed by a section in which the names of all variables and generators are declared.

The body part is formed by an optional adaptive function call to be executed on entry, followed by a set of elementary adaptive actions, responsible by the modifications to be performed. A furhter optional call specifies another adaptive function to be executed on exit. Both calls are denoted in the usual way, as mentioned above.

There are three types of elementary adaptive actions: insertion, elimination and inspection actions. The notation chosen for the elements of the declaration of an adaptive function is shown in the table below. No graphic notation is yet suggested for adaptive functions.

Element of the declaration	Symbolic notation
Adaptive function name	M
Parameters	$(p_1, p_2, \ldots, p_n)$
Variables	$v_1, v_2, \ldots, v_n$
Generators	$g_1^*, g_2^*, \ldots, g_n^*$
Inspection actions	? [pattern]
Elimination actions	– [pattern]
Insertion actions	+ [pattern]
Pattern	Any transition

Graphical notations have been tried [Alm95] that showed to be effective in some particular cases, where the self-modifications to be performed are small and easily visible. It is difficult to represent graphically the operation of adaptive functions in their full generality. We chose to adopt symbolic descriptions for adaptive functions, even when graphical notation is used to describe the adaptive automata they refer to. In order to provide an acceptable notation for dealing with sets and predicates, we chose to adopt the usual notation of predicate calculus, including quantifiers, for expressing adaptive functions.

The Underlying Model. The underlying model for adaptive automata is the structured pushdown automaton. Sub-machines may be considered as improved finite-state automata that are allowed to recursively call each other. The best feature of this arrangement is that structured pushdown automata turn out to be easier to design and understand than pushdown automata.

Pushdown machines may be considered excessively complex for some applications, for which finite-state automata have being used successfully. In these cases, some means should be provided to avoid the presence of unnecessary features in the underlying non-adaptive automaton.

In the special cases for which a simple finite state mechanism is enough, we may suppress the pushdown storage from the notationm reducing the remaining sub-machine into a simple finite-state machine.

The device resulting from the suggested simplification becomes an adaptive finite-state automaton, and may be formally stated just as published before in section 4 of [Jos01].

The Adaptive Mechanism. The principle of this mechanism consists in modifying the set of rules of the adaptive automaton by performing two adaptive actions, one before and another one after executing the underlying state-transition rule. However, one may ask whether a pair of adaptive actions is really a need. A element that substantially contributes to harden understanding adaptive devices is the structure of the adaptive functions themselves.

Adaptive functions are allowed to perform a pair of adaptive actions, one before and another after the modifications the adaptive function is expected to perform.

The set of parameters allowed in adaptive functions is another feature that is questionable when we search for simplicity: is it really needed to allow an arbitrary number of parameters? Should it be better to limit the number of parameters to a minimum? What should this minimum be? Should adaptive functions have no parameters at all? In the case of allowing parametric adaptive functions, should parameter types be controlled instead of arbitrarily chosen? Should adaptive functions be allowed as parameters?

Elementary adaptive actions are another source of complexity, since no restrictions are imposed to their use. Some questions may be posed concerning these elements of the formulation of adaptive automata: Should multiple variables be allowed in inspecting and eliminating elementary adaptive actions? Should looping be allowed within elementary adaptive actions?

In the following text we propose answers to several questions posed here, with the intent of achieving for adaptive automata a formulation according to our simplicity goals.

Adaptive Actions. In adaptive automata, this adaptive mechanism consists of executing the pair of adaptive actions attached to a rule at the time it is chosen for application. The first adaptive action in the pair is executed before the rule is performed, while the second one is executed after applying the subjacent state-transition rule. We limit to one the number of adaptive actions attached to the corresponding subjacent rule. Indeed, [Iwa00] shows a proof for the following theorem, stating that there is no need of attaching a pair of adaptive actions per rule, but the use of a single one is enough.

Theorem 1. *The result of the execution of any adaptive action is equivalent to a non-empty sequence of elementary adaptive actions.*

Proof. This theorem is fully demonstrated in [Iwa00] by simulating the device with the specified restrictions. □

Further simplifications may be achieved by using the following theorem:

Theorem 2. *For each rule defining an adaptive automaton, there is an equivalent set of rules, all of which have at most one attached (after-) adaptive action.*

Proof. This theorem may be proved by using the previous one and by showing that each adaptive transition having an attached before-adaptive action may be decomposed into a sequence of two simpler ones: the first one is an empty transition having as its attached after-adaptive action the before-adaptive action in the original transition; the second one is a copy of the original transition, from which the before-adaptive action has been removed. □

Theorem 3. *For each adaptive function that calls a pair of attached adaptive actions there is an equivalent set of simpler adaptive functions, all of which have at most one attached (after-) adaptive action.*

Proof. The proof of this theorem follows from the result of the previous one, and is based on showing that each adaptive function F that calls a before-adaptive action B may be decomposed into a sequence of two simpler ones in the following way: F becomes a copy of B's body, followed by a call to an auxiliary function F_1, where F_1 is a copy of F from which the call to B has been removed. □

4 Improving the Formulation of the Underlying Model

Structured pushdown automata use their pushdown store in two extremely limited situations, in which no symbol is consumed from the input string: (a) when a sub-machine is called, the return state is pushed onto the pushdown store before control is passed to the starting state of the sub-machine being called, and (b) when a sub-machine finishes its activity, a return state is popped from the pushdown store, and a return is made to that state in the calling sub-machine.

In this section we propose some changes to the underlying structured pushdown automata. These suggestions rely on the following theorem.

Theorem 4. *Adaptive (structured pushdown) automata are equivalent to adaptive finite-state automata.*

Proof. Proving that an adaptive structured pushdown automaton may simulate an adaptive finite-state automaton is straightforward. The opposite clause of the theorem is proved by simulating the pushdown store with states and transitions of adaptive finite-state automata. The resulting model allows performing the same work without using explicit memory. Such simulation may be sketched in three steps:

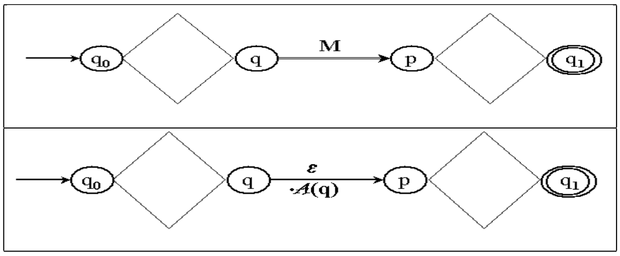
- We can suppose, without lost of generality, that all sub-machines in the adaptive automaton have single initial and final states.
- For sub-machines that are not self-embedded, substituting all sub-machine calls by an equivalent empty adaptive transition is enough. Indeed, if a calling sub-machine N invokes submachine M, we replace the following adaptive transition in N:

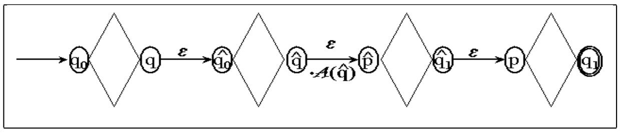
$$\bigcirc^{q} \xrightarrow[\bullet\mathcal{A}(q)]{\alpha} \bigcirc^{p}$$

  for the original sub-machine call:

$$\bigcirc^{q} \xRightarrow{M} \bigcirc^{p}$$

  where ϵ is the empty word and $\bullet\mathcal{A}(q)$ is an adaptive function which performs a macro expansion of the call to sub-machine M by replicating its topology in the exact place where the former call was, in such a way that the initial state of submachine M is reached, by a unique empty transtition from state q and the state p is reached by a unique empty transition from the unique final state of M. Note that the adaptive function $\bullet\mathcal{A}$ is executed only once. Such replacements finishes in a finite number of steps since, by hyphotesis, there are no self-embedding in M.
- If any sub-machine in the adaptive automaton is self-embedded, then it suffices to substitute every self-sub-machine call (top of the following table) by the adaptive transition (bottom of the table). Rhombuses represent the body of sub-machine M:





Where, again, the ϵ is the empty word but now $\bullet\mathcal{A}$ is an adaptive function with the following effect shown in the last box.

Note that this adaptive function duplicates the sub-machine topological structure and the states q_0, q_1, q and p. It must be clear, also, that this process can be a non-stopping one, but anyway it is a safe way to simulate the adaptive (structured pushdown) automata with an adaptive finite-state automata. □

Improving the Formulation of the Adaptive Mechanism. The adaptive mechanism is indeed an important source of complexity in the formulation of adaptive automata which may be simplified in several aspects, some of which are the following: (a) by limiting to one the number of adaptive actions attached to each rule and/or called inside adaptive functions (b) by restricting the nature and number of parameters allowed for adaptive functions (c) by avoiding multiple variables to be inspected at the same time in a single inspection (d) by avoiding loops within elementary adaptive actions (e) by avoiding adaptive functions passed as parameters. Unfortunately, if we impose too much simplifications to the formulation, it becomes less expressive, requiring more clauses to perform the desired effect. However, by restricting the formulation, simpler facts are expressed by each adaptive action, rendering the formulation easier to understand. The hints above surely help searching for a cleaner and more effective formulation. This is a challenge yet to be overcome.

5 Illustrating Example

In this section, we chose a Non-deterministic Adaptive Finite Automaton, and used it to solve the well-known string-matching problem of determining whether a given string is a sub-string of some text. One classical solution for this problem is as follows: (a) Create a non-deterministic finite-state automaton that solves the problem. Constructing such automaton is straightforward: at its initial state, a loop consumes any symbol in its alphabet; next, a simple path consuming the sequence of symbols in the string we are looking for, and, at the end of this path, a unique final state consumes any further alphabet symbols. The explicit non-determinism in this automaton is located at the beginning of the path that accepts the required pattern. (b) Use a standard method to eliminate the non-deterministic transitions in the automaton. (c) Use a standard algorithm to minimize the resulting deterministic automaton.

In [Hol00] an algorithm is presented that constructs directly the desired deterministic finite-state automaton; it has the advantage of eliminating the need to eliminate non-deterministic transitions, since the complexity of this process is exponential. Additionally, in this method the full automaton must be constructed and minimized a priori.

Our approach avoids the exponential transformation, and does not require any unnecessary *a priori* work: we start from an initial adaptive non-deterministic finite-state automaton and let it process a text sample; whenever a text being analyzed activates a non-deterministic transition, the execution of the corresponding adaptive function performs the required topological transformations in order to render it deterministic. By doing so, our approach uses an incremental strategy to force them to perform in a deterministic way all transitions they execute, without changing the remaining non-deterministic transitions present in their formulation.

5.1 An Exact String Matching Non-deterministic FSA

Here we follow [Hol00]; the straightforward non-deterministic finite-state automaton is constructed for accepting the pattern **aba**, over the alphabet $\Sigma = \{a, b\}$:

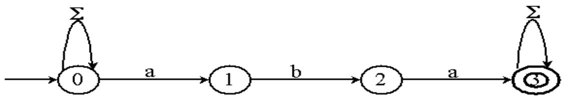


Fig. 1. A non-deterministic finite-state automaton

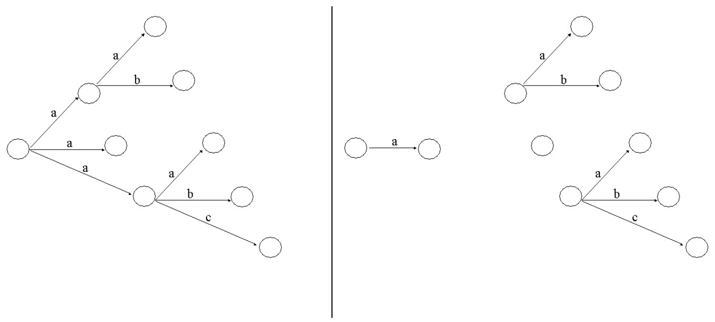


Fig. 2. Eliminating non-deterministic transitions.

5.2 An Equivalent Adaptive FSA for Exact String Matching

Now, let us turn the attention to our adaptive approach. In figure 2 (left) a non-determinism is present. In order to remove such non-determinism, we introduce a new state (fig. 2, right).

Now, in order to make it reachable, from the newly created state, all states that were reachable through all transitions departing from the conflicting states, we add further transitions leaving the new state and arriving to the target states, consuming the appropriate symbols, as shown in fig. 3. These operations may be sketched as an adaptive function B, in fig. 4.

This adaptive function receives a state and a token as parameters. In this formulation, it declares three variables and one generator; in line 3 quantifiers are used to allow testing whether there is more than one transition departing from

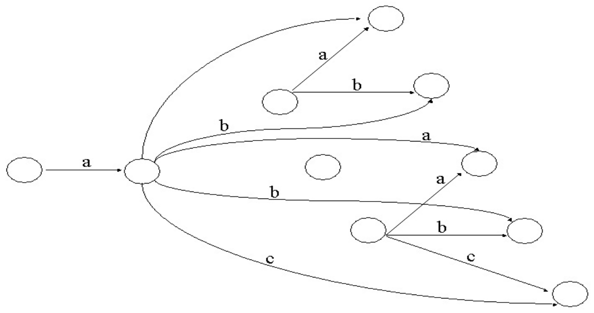


Fig. 3. Inserting transitions connecting the new state to the automaton

1	$B(q,\sigma) = \{$	declaring name and parameters of B
2	$p, n*, r, a;$	declaring variables (n is a generator)
3	$(!p)(?[(q,\sigma) \to p])\{$	only if there are more than one rule with this shape,
4	$+[(q,\sigma) \to n]$	add a new transition with his shape with destination n
5	$(\forall p)(?[(q,\sigma) \to p])\{$	for all transitions emerging from state q
6	$(\forall a)(\forall r)(?[(p,\alpha) \to r])\{+[(n,\alpha) \to r]\}$	insert corresponding transitions departing from n
7	$-[(q,s) \to p]\}\}\}$	after all insertions, remove the original transition

Fig. 4. Adaptive function B that dynamically eliminates non-deterministic transitions from our non-deterministic adaptive finite-state automaton

the state received as the first parameter, and consuming the symbol received as the second parameter; if the answer is negative, then the query (the clause introduced by a question mark in this notation) will produce an empty result, and in this case the clause in braces (comprehending lines 4, 5, 6, 7) will not be executed. Otherwise, in line 4 a new transition is created from the current state to a new one, by means of generator n. Line 5 states that for each output transition, a proper output transition is generated and the original transition is deleted. The resulting adaptive non-deterministic finite-state automaton is shown in figure 5.

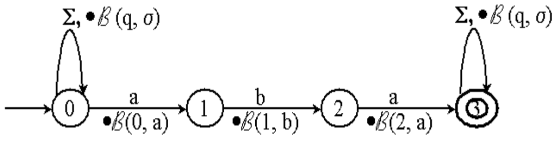


Fig. 5. Adaptive non-deterministic finite-state automaton

6 Conclusions

The proposed simplification to the formulation of adaptive automata seems to be reasonably expressive, compact and readable, allowing them to be stated in a rather intuitive form. Most features and development practices already established are preserved and respected to a large extent.

The proposed formulation caused almost no impacts to the power of adaptive automata, so the net result of its use will be a significant increase in the readability and soundness of the formulation without loss of the devices computational power.

From the programming point of view, our proposal has also advantages over the earlier notations, allowing programmers to build and debug adaptive automata in a more expedite way, resulting better products and a far better documentation.

References

[Alm95] Almeida Junior, J.R. STAD - Uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos. São Paulo 1995, 202p. Doctoral Thesis. Escola Politécnica, Universidade de São Paulo.[In Portuguese]

[Hol00] Holub, Jan. Simulation of Non-deterministic Finite Automata in Pattern Matching. PhD Dissertation Thesis, Faculty of Electrical Engineering, Czech Technical University, February 2000, Prague.

[Iwa00] Iwai, M. K. Um formalismo gramatical adaptativo para linguagens dependentes de contexto. São Paulo 2000, 191p. Doctoral Thesis. Escola Politécnica, Universidade de São Paulo.

[Jos93] José Neto, J. Contribuição à metodologia de construção de compiladores. São Paulo, 1993, 272p. Thesis (Livre-Docência) Escola Politécnica, Universidade de São Paulo.[In Portuguese]

[Jos94] José Neto, J. Adaptive automata for context-dependent languages. ACM SIGPLAN Notices, v.29, n.9, p.115–24, 1994.

[Jos01] José Neto, J. Adaptive automata for syntax learning. XXIV Conferencia Latinoamericana de Informática CLEI'98, Quito - Ecuador, Centro Latinoamericano de Estudios em Informatica, Pontificia Universidad Católica Del Ecuador, tomo 1, pp.135–146. 19 a 23 de Outubro de 1998.