Proceedings of the 17th IASTED International Conference
MODELLING AND SIMULATION
May 24-26, 2006, Montreal, QC, Canada

CONCEPTION OF ADAPTIVE PROGRAMMING LANGUAGES

Aparecido Valdemir de Freitas[1,2] **, João José Neto** [1]
[1] Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, Nº 158. Cidade Universitária - São Paulo – Brasil
[2] Universidade Municipal de São Caetano do Sul
Instituto Municipal de Ensino Superior de São Caetano do Sul
Av. Goiás Nº 3400 – Vila Barcelona – São Caetano do Sul – CEP 09550-051 - São Paulo – Brasil
avfreitas@imes.edu.br and joao.jose@poli.usp.br

ABSTRACT

Adaptive devices show the characteristic of dynamically change themselves in response to input stimuli with no interference of external agents. Occasional changes in behaviour are immediately detected by the devices, which right away react spontaneously to them. Chronologically such devices derived from researches in the field of formal languages and automata. However, formalism spurred applications in several other fields. Based on the operation of adaptive automata, the elementary ideas generating programming adaptive languages are presented.

KEY WORDS

Adaptive devices, self-modifying devices, adaptive automata, adaptive programming language.

1. Introduction

The fundamental concept supporting adaptive technology rests on the ability of such devices to perform adaptive actions, which may be seen as internal procedures to the devices activated in response to the detection of situations requering behavioral changes. [1]

To a given current configuration and a given stimulus taken from its input, the device – through the formal machine – is moved to a new configuration. Using a finite well-defined set of rules, such devices start operation at some initial configuration. The clauses making up this set of rules test the device current configuration and lead it to a new situation.

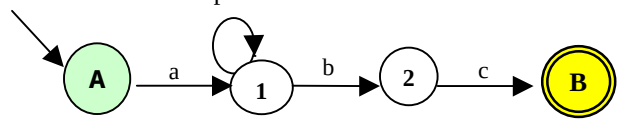
Interest in self-modifying software has increased and adaptive programming is aimed at the problem of producing software with self-modification behavior. Two segments in the computing field are driving this development. First is the emergence of ubiquitous computing, and second is the growing demand for autonomic computing. [8] [9] [10] [11] [12]

Our work differs of most of the research published in the area of self-modifying software, since it is based on devices whose behavior relies on the operation of subjacent non-adaptive devices, which incorporate a finite and well-defined group of self-modifyng rules. [13]

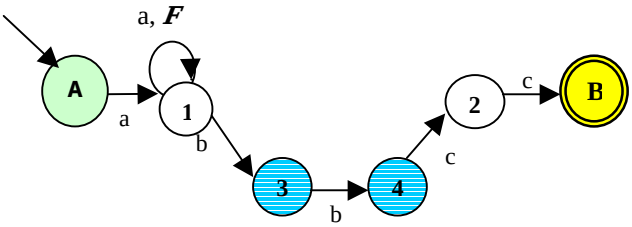
2. Adaptive Automata

Adaptive automata are special adaptive devices whose underlying formalism is the structured pushdown automaton. Their operation corresponds to a sequence of successive evolutions performed by an initial structured pushdown automaton processed by adaptive actions. To each adaptive action, a new automaton originates, keeping on the treatment in the input chain [1]

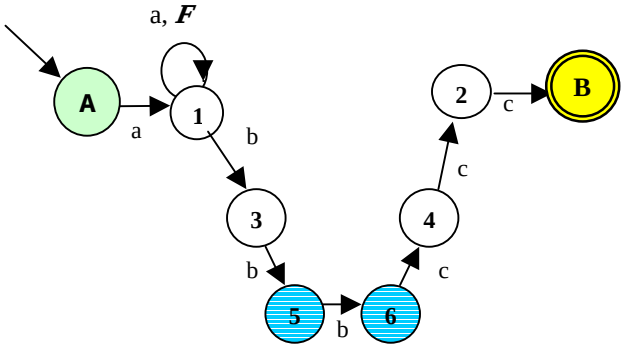
To demonstrate the operation of adaptive automata, there follows an implementation of language recognition $a^nb^nc^n$, n being > 0, which can be mapped by an automaton whose task is to recognise chains ‘abc’, ‘aabbcc’, ‘aaabbbccc’ and so forth and to reject the others. Illustration 1 shows the frame of the automaton to be modelled on the problem.



a) Initial automaton for $a^nb^nc^n$, n being =1.



b) Automaton after the elimination of second ‘a’.



c) Automaton after the elimination of third ‘a’

Illustration 1 – Adaptive automaton for the recognition of chains expressed as $a^nb^nc^n$, n being > 0.

If the language to be recognised were defined by $a^1b^1c^1$, the finite automaton in question might be represented by states (A,1,2,B), A being the initial state and B the state of acceptance, and by transitions (A,'a',1), (1,'b',2), (2,'c',B). We will label this initial finite automaton AF_1. (Illustration 1-a).

Analogously, according to Illustration 1-b, for the language $a^2b^2c^2$, a new finite automaton (AF_2) would be needed, represented by a new set of states (A,1,2,3,4,B) and also a new set of transitions ((A,'a',1) , (1,'a',1) , (1,'b',3) , (3,'b',4) , (4,'c',2) , (2,'c',B)).

To make the recognition of language $a^nb^nc^n$ general, we can structure an adaptive automaton formed by the space of finite automata AF_1, AF_2, ... , AF_n, in such a way that – starting with initial finite automaton AF_1 and going through adaptive functions calls – the device may evolve to successive configurations AF_2, AF_3,.., AF_n , while the input chain is processed step by step.

The adaptive actions will make a dynamic rearrangement of the device, with no concurrence, therefore, of external agents. [2]

In order to form a more substantial idea of these concepts, we will include in the transition (1,'a',1) of AF_1 initial automaton an adaptive function call, identified as ***F***, to be performed before processing the corresponding transition. The transition will be identified as ((***F***) , (1,'a',1) , ( )). The blank list, specified on the right of the transition, shows that, in this instance, there will not be any adaptive function calls after the performance ot the automaton usual transition.

According to the frame shown in Illustration-1, to each atom 'a' consumed at state 1, new states at added in convenient positions, new transitions are appended and one is eliminated as the result of ***F*** adaptive function call.

This adaptive function generates at first two new states (3 and 4), another transition consuming 'b' from state 3 to state 4, still another consuming 'c' from state 4 to state 2, and eliminates the transition consuming 'b' from state 1 to state 2. (Illustration 1-b)

Therefore, at, first call, ***F*** adaptive function is made up by the following elementary actions:

+ Transition (( ) , (1, 'b', 3), ( ))
+ Transition (( ) , (3, 'b', 4), ( ))
+ Transition (( ) , (4, 'c', 2), ( ))
- Transition (( ) , (1, 'b', 2), ( ))

Once the additon of the new states 3 and 4 is provided, the position where the next two states are to be included has to be updated as well. At the first call of the adaptive function, states 1 and 2 are taken for parameters. In the next evolution of the automaton, states 3 and 4 are taken for parameters and between then two other new states have to be inserted: 5 and 6, for instance.

In general, the ***F*** adaptive function may be specified as follows: Let i and j be states of any adaptive automaton in which is present the configuration:

$$(i,'b') \rightarrow j \;\; e \;\; (j,'c') \rightarrow k$$

In the initial situation, for instance, states i and j correspond to states 1 and 2. In the following evolution, 3 and 4; in the next 5 and 6, and so forth.

To furnish the automaton with self-modifying features, the ***F*** adaptive function needs to search the adaptive automaton in order to determine states i and j, between which the new states and the new transition will be inserted and needs moreover to make up a list of elementary adaptive actions which will conduct consultations and explicit changes in the adaptive automaton's current configuration.

Considering that the evolution of the automaton will bring about new states and transitions, our elementary adaptive actions will handle several different variables, to be automatically filled, with to explicit request, and with exclusive values not yet employed in the automaton current definition. We will label these variables m* e n* and will call then *generators*.

Thus, for any i and j adaptive automaton's states, the mapping will read:

$$(i,j) : \; m^*,n^*$$

which corresponds, during the automaton's evolution to the mapping below:
(1,2): m*=3 , n*=4 ; (3,4): m*=5 , n*=6 ; . . .
In our example, ***F*** adaptive function will made up by the following elementary actions:

? [(i,'b') → j]	// consultation action delivering trasnsitions // (i, 'b', j)
? [(j,'c') → k]	// consultation action delivering trasnsitions // (j, 'c', k)
	// i and j are found...
- [(i,'b') → j]	// transition delition (i,'b',j)
+ [(i,'b') → m*]	// transition addition (i,'b', m*)
+ [(m*,'b') → n*]	// transition addition (m*,'b', n*)
+ [(n*,'c') → j]	// transition addition (n*,'c', j)

The adaptive device's initial set of productions will be represented by:

'(	(	( )	(A 'a' 1)	( )	)
	(	( )	(2 'c' B)	( )	)
	(	(***F***)	(1 'a' 1)	( )	)
	(	( )	(1 b '2')	( )	)
)					

After the device's first evolution, the following productions will come out:

'(				
(( )	(A 'a' 1)	( )	)	**kept**
(( )	(2 'c' B)	( )	)	**kept**
((***F***)	(1 'a' 1)	( )	)	**kept**
~~(( )~~	~~(1 'b' 2)~~	~~( )~~	~~)~~	~~**eliminated**~~
(( )	(1 'b' 3)	( )	)	**inserted**
(( )	(3 'b' 4)	( )	)	**inserted**
(( )	(4 'c' 2)	( )	)	**inserted**
)				

In the device's second evolution, productions will be as follows :

```
'(  (    ()      (A 'a' 1)  ()      ) kept
    (    ()      (2 'c' B)  ()      ) kept
    (    (F)     (1 'a' 1)  ()      ) kept
    (    ()      (1 'b' 3)  ()      ) kept
    (    ()      (4 'c' 2)  ()      ) kept
    (    ()      (3 'b' 4)  ()      ) eliminated
    (    ()      (3 'b' 5)  ()      ) inserted
    (    ()      (5 'b' 6)  ()      ) inserted
    (    ()      (6 'c' 4)  ()      ) inserted
)
```

A finite automaton is defined by a quintuple (Q, Σ, δ, q_o, F) [3] and may be represented in list notation as shown:

```
AUTOMATON:  (       List_of_States
                    Alphabet
                    List_of_Transition
                    Initial_State
                    List_of_Acceptance_States
            )
```

List_of_Transitions : (Transition Transition Transition . . .)

Transition: (State_Origin Atom State_Destination)

To define the adaptive automaton, transitions need to be redefined so as to support the adaptive actions. Thus, each transition of the automaton will be redefined by:

Transition: ((AA)
(State_Origin Atom State_Destination) (AP))

AA representing the previous adaptive action and AP for the following adaptive action.

These adaptive actions (both previous and following), will be defined by:

(**define** Name_Adaptive_Function ((Parameters_List) variables generators List_Elementary_Actions))

Once heed is given to these points, the adaptive automaton may be represented by means of function anbncn, which takes as parameters the entrance chain, the automaton's initial state list, the initial state, the transition's initial list, the alphabet, and the list of acceptance's final states:

```
(anbncn
       '(a a a b b b c c c );;input chain
       '( A 1 2 B ) ;;initial state list
       '( A )       ;; initial state
       '(
             ( ()  (A a 1) () )
             ( ()  (2 c B) () )
             ( (F) (1 a 1) () )
             ( ()  (1 b 2) () )
        );; transition's initial list
       '(a b c)  ;; alphabet

       '(B)   ;;   list   of   acceptance's
states
)
```

The list of parameters previously described is consistent with the framework used to represent the adaptive automata, since there is an instance of adaptive transition – represented by the list ((F) (1 a 1) ()) – in the transition's initial list.

The functional program may be represented by function anbncn, which will take as parameters the elements bound to make up our finite adaptive automaton:

```
(define
       (anbncn
              input_chain
              initial_state_list
              initial_state
              transition's_initial list
              alphabet
              list_of_acceptance's_states
       )
)
```

Function anbncn starts operation first as if the usual finite automaton were being processed .

The program first positions the machine at initial state and, from then on, processes the entrance chain.

Once the atom is read and the current state defined, there starts a search of transitions defined as argument. If the transition answering to the conditions of the moment are non-adaptive, the automaton will evolve naturally, just like any common finite automaton.

However, if the transition attending to the search condition is adaptive, the program carries out adaptive function calls, which may take place before or after the execution of the non-adaptive transition. In our example, the adaptive function is called before the execution of adaptive transition and will be responsible for the automaton's new configuration.

It is possible to present the procedure for this function in the following way:

;; place the machine at starting state
;; reading of entrance chain
;; search of transitions from current state and read atom
;; search, in case adaptive calls are found in the list of transitions
;; if there are adaptive calls, carry them out.
;; after the adaptive call, keep the automaton's usual evolution
;; if there are adaptive calls after the transition, carry them out
;; after the processing of entrance chain
;; check is final state is one of acceptance
;; if so, then the chain has been recognised ...

For the adaptive treatment to be materialised in our example, there must be defined the adaptive function required to allow the device the self-modifying property.

Considering that the adaptive function will always be made up of elementary actions of consultation, removal or addition of transitions, it is expedient to have at disposal a set of routines, which may be conveniently assembled to implement the adaptive functions. Such set of routines defines an "adaptive layer" responsible for encapsulating the elementary adaptive actions.

3. The adaptive actions

As aforementioned, they are the adaptive function calls (pointed out by adaptive actions) that will make the self-modifying features of our device possible. These functions will be made up by elementary function calls – available on the adaptive layer -, which by means of the basic operations of consultation, inclusion, and deletion of productions will afford the adaptiveness to the device.

We will denote theses elementary actions ?**adapt**, **+adapt** e **–adapt**, responsible respectively for the actions of consultation, addition, and elimination of transitions from the list of current transitions. [4]

Naturally, the elementary actions will be extracted from the adaptive layer for each particular problem in the convenient amount and sequence for the device adhesive to the problem.

The consultation actions must likewise be tied to the specific device, since they depend on each problem to be dealt with. In other words, for each problem a different or more convenient consultation must be specified in the list of productions.

To generalise such consultation function, queries modelled on pattern-matching may be specified so as to be taken back to the list of productions that attends to the consultation.

Thus to transfer the list of transitions from the adaptive device along with the pattern ((*) (8,*,*) (*)) to function **?adapt** on the adaptive layer, we will get as return all transitions (adaptive or not) leaving from state 8.

Analogously, on passing the list of transitions of our adaptive device, along with the pattern ((F) (*,'a',*) (*)) to the **?adapt** function of the adaptive layer, we will get in return a list of all adaptive transition which call to F adaptive function before conducting the non-adaptive transition, when an 'a' atom is consumed.

If we transfer the list of transitions of our adaptive device along with the pattern ((*) (*,*,*) (**G**)) to the **?adapt** function of the adaptive layer, we'll get in return a list of all adaptive transitions that call **G** adaptive function after conducting any transition whatever.

The pattern ((*) (*, *, *) (*)) transferred with the list of transitions of the device to the **?adapt** function of the adaptive layer, will return the list of all current productions of the adaptive device.

As a rule, **?adapt** function will take for parameter a query pattern definition and the list of transitions of the device, returning a list of transitions that attends to the previous pattern, or else an empty-list, in case there has been no matching..

In our example, the rules making up our adaptive function may be implemented starting at the adaptive layer, as follows:

Rule: Mapping with the Adaptive Layer

```
? [(i,'b') → j ]
(set 'L1  (?adapt ( ( (*),(*,'b',*),(*) ) , List_TR ) ) )
// returns L1 with transitions (i, 'b', j) – bound to 'b' atom
? [ (j'c') → k ]
(set 'L2  (?adapt ( ( (*),(j,'c',*),(*) ) , L1 ) ) )
// returns L2 with transition (j, 'c' , k)
// setting of  states (i , j)
- [(i,'b') → j ]
(set 'Lista_TR  (-adapt ( ( ( ),(i,'b',j),( ) ) , List_TR ) ) )
// returns List of Transitions deleting transition
// ( ( ),(i,'b',j),( ) )
+ [(i,'b') → m* ]
(set 'Lista_TR  (+adapt ( ( ( ),(i,'b', m* ),( ) ) , List_TR ) ) )
// returns List of Transitions including transition
// ( ( ),(i,'b', m* ),( ) )
+ [(m*,'b') → n*]
(set 'Lista_TR  (+adapt ( ( ( ),( m* ,'b', n* ),( ) ) ,List_TR ) ) )
// returns List of Transitions including transition
// (( ),( m* ,'b', n* ),( ))
+ [(n*,'c') → j ]
(set 'Lista_TR  (+adapt ( ( ( ),( n* ,'c', j ),( ) ) , List_TR ) ) )
// returns List of Transitions including transition
// ( ( ),( n* ,'c', j ),( ) )
```

Considering that ***F*** adaptive function will operate ont the automaton's transition list, it is only natural for this function that the list of transitions comprising the respective changes determined by the elementary actions of insertion or exclusion be transferred as a parameter.

F adaptive function may be defined in functional notation as shown below:

```
;;-----------------------------------------------
;; Adaptive Function F
;;-----------------------------------------------
(define  (  F    List_TR  )
(set 'L1   (?adapt ( ( (*),(*,'b',*),(*) ) , List_TR ) ) )
(set 'L2  (?adapt ( ( (*),(j,'c',*),(*) ) , L1 ) ) )
//(i,j) is returned
(set 'Lista_TR  (-adapt ( ( ( ),(i,'b',j),( ) ) , List_TR ) ) )
(set 'Lista_TR  (+adapt ( ( ( ),(i,'b', m* ),( ) ) , List_TR ) ) )
(set 'Lista_TR  (+adapt ( ( ( ),( m* ,'b', n* ),( ) ) , List_TR ) ) )
(set 'Lista_TR (+adapt ( ( ( ),( n* ,'c', j ),( ) ) ,List_TR ) ) ) )
```

It is quite clear that, having at our disposal an adaptive layer with a set of elementary functions capable of materialising the elementary actions, the definition of adaptive functions will be easily specified, so much so that the rules governing them are directly coded by the respective calls of the adaptive layer.

Our adaptive layer must be broad enough to be used in any other problem applying adaptive technology.

4. The Conception of Adaptive Languages

Based on the concepts of adaptiveness applied on implementing language recognition $a^n b^n c^n$, being $n > 0$, we will outline our adaptive language by idealising, in a

way similar to the evolution of adaptive automata, a programming language whose functions and code present self-modifying behaviour.

We will use a functional language as a basis for the incorporation of mechanisms of the adaptive formalism. [6]

Considering lambda calculus to be the basis for the functional paradigm of the programming, a language supporting lambda calculus will be taken as the core of the proposed language. [5] [6]

The reason for such a choice rests on this language's better adherence to the concepts of adaptive technology, sence this pattern favours the building of expressions dealing with dynamic codes, as pointed out on the example below:

```
> (set 'exp '(+ 3 4 ) )
> (eval exp)
> 7
```

After these considerations, we will set off from a functional nucleus based on the lambda calculus, which will serve as an interpreter of the language coded by the user.

On this basic functional nucleus, an adaptive layer evaluating adaptive action calls will be projected.

Once the adaptive actions are processed, a new instance of the program is reached and the execution once again switched to functional nucleus, which will proceed with the execution.

The adaptive language, consequently, will be formed by the space of codes LF_1, LF_2, ..., LF_n, in such away that – starting with initial language LF_1 and through the adaptive function calls – language may evolve to the successive configurations LF_2, LF_3,.., LF_n while the execution is being processed. [4]

The proposed adaptive mechanism keeps a close structural analogy with the adaptive device used as the example in item 2.

Adaptive formalisms materialised as programming languages (functional languages in our instance) will present as first a code block to be directly processed by the basic functional nucleus interpreter until the execution of some adaptive action specified int the pattern takes place.

In order to process our adaptive language, a processing environment has to be created, made up of a functional nucleus and a control module, to be represented by the adaptive machine the duty of which is to evaluate the adaptive calls.

A full functional program limits itself to the call of a single function, made up of the composition of several other functions.

To help visualise our functional program in the light of the adaptive paradigm, some mechanism must be resorted to in order to address each function making up our program univocally. Such procedure must equally be applied to every native function present in the language.

In our model, the functions making up our code will be indexed by means of labels defined by identifiers.

As the functional program may be represented by a tree-like structure, each label will stand for a node of this tree.

As long as the procedure enabling the binding of labels to each function of the program suitable to take part in the adaptive calls is defined, it is possible to determine the adaptive functions to account for the adaptiveness of the language code.

Since our functional basic nucleus is an interpreter of lambda calculus, any function defined in the program may be represented by a call in lambda calculus. For example:

```
> (define ( func x )  (+ x x ) )
(lambda  (x)  ( + x x ) )
> (func 3)
6
```

Any label can be attributed to the function by means of:

```
> (define (label  rotulo x) (set rotulo x)  x )
(lambda (rotulo x ) (set rotulo x)  x)
```

In this way, it is possible to associate any label with function "func", e.g.,

```
> (label 'rotulo1 func)
(lambda  ( ( + x x ) ) ) )
```

Function "func" then will be associated with label 'rotulo1'.

```
> rotulo1
(lambda  ( ( + x x ) ) ) )
> (rotulo1    3)
6
> ( (lambda ( ( + x x ) ) ) 3)
6
```

Another example shows a function that returns the sum of squares.

```
> (define (soma_quad x y ) (+ ( * x x ) (* y y ) ) )
(lambda  (x y )  (+ (* x x )  (* y y ) ) )
> (soma_quad 2 4 )
20
```

For any given adaptive function to change the function code of the example, we have to address the function undergoing self-modification in a univocal way, which can be done by the call of label function of the adaptive layer.

```
>  (define (soma_quad  x  y )  ( +  ( *  x  x )
            ( label 'rot1 '(* y y )    ) ) )
(lambda (x y) (+ (* x x ) (label 'rot1 (* y y ) ) ) )
>  ( (lambda (x y) (+ (* x x ) (label 'rot1 (* y y ) ) ) ) 2 4 )
20
```

In this example, label 'rot1' has associated with function (* y y). It is worth remarking that, although the application of the label function of the adaptive layer has changed the lambda expression, it has not borne the same evaluation result, which proves that the labels associated with the functions do not alter the semantic meaning of the expression.

Still keeping the data from the example above, let a given adaptive function access label 'rot1' and change the function from (* y y) to (+ y y).

For this adaptive function to accomplish this alteration, it is enough to consider the program lambda expression as a list and to apply to it the elementary actions specified in the adaptive function. In our instance, label 'rot1' must be searched in the lambda expression and replaced by the new expression.

The resulting lambda expression will then read:

```
> (define (soma_quad  x  y )  ( +  ( *  x  x )
(   label 'rot1 '(+ y y )   ) ) )
(lambda (x y) (+ (* x x ) (label 'rot1 (+ y y )  ) )  )
>  ( (lambda (x y) (+ (* x x ) (label 'rot1 (+ y y ) ) ) ) 2 4 )
12
```

In our model of adaptive language, therefore, the adaptive functions will start a string processing in the lambda expression corresponding to the program, and will generate a new string adhering to the labels defined by the adaptive actions. With this new instance in the program, the control will be once more switched to the underlying lambda calculus interpreter, which will carry on the program execution.

At this point, it is required the implementation of a control to ensure that the program's new instance will resume from the point of interruption and not from the start of the program's new instance. Moreover, the adaptive device is expected to return to the new instance all space of variables previously allocated by the preceding instance.

5. Conclusion

It has been shown that adaptive devices are Turing-powerful. The resulting generality of the model, its learning capability due to its self-modification feature, as well as the strength of its expressiveness make adaptive devices very attractive and suitable to handle difficult situations arisen when searching for computational solutions for complex problems. [14]

This article exemplifies, by solving a particular problem ($a^n b^n c^n$), the use of adaptive techniques in a programming functional language. The language thus obtained favours the introduction of dynamic features in to language initial code.

The problem brought to this article could have been solved by resorting to the usual functional language, but the example shown here was intended to make the application of the adaptive technology in a programming language easier to understand.

We believe to have demonstrated through the concepts and procedures expounded in this article that implementing the adaptive language is feasible and that it may consequently be applied to more complex problems to which adaptive technology is recommended.

Some experiments with the adaptive language project have already been developed and for these implementations the NewLisp environment is being adopted. [7]

Acknowledgement: Our thanks for the reviewers' valuable comments that significantly improved this paper.

References

[1] Neto, João José - Adaptive Rule-Driven Devices - General Formulation and Case Study**.** Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol.2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.

[2] Neto, João José - Adaptive Automata for Context - Sensitive Languages. SIGPLAN NOTICES, Vol. 29, n. 9, pp. 115-124, September, 1994.

[3] Harry R. Lewis e Christos H. Papadimitriou - Elements of the Theory of Computation. Second Edition. Prentice-Hall Inc. 1998.

[4] Freitas, A. V. - Neto, João José – Adaptive Device with underlying mechanism defined by a programming language. - 4th WSEAS International Conference on Information Security, Communications and Computers (ISCOCO 2005) – Special Session Artificial Intelligence and Soft Computing.

[5] Barendregt, H.P. - The Lambda Calculus: its syntax and semantics – (2nd ed.), North-Holland, 1984.

[6] McCarthy, J. - Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part-I. CACM 3,4 (1960), 184-195.

[7] Mueller, Lutz - NewLisp User's Manual and Reference V. 7.50 – 2004 – www.newlisp.org.

[8] Philip K. McKinley, Seyed Masoud, Sadjadi, Eric P. Kasten, Betty H. C. Cheng – Composing Adaptive Software - Michigan State University - IEEE Computer Society - 2004.

[9] Peter Norvig and David Cohn – Adaptive Software - PCAI Magazine. Jan, 1997.

[10] Richard Taylor, Chris Tofts – Self Managed Systems – A Control theory perspective - Trusted Systems Laboratory - HP Laboratories Bristol - HPL-2004-49 - March 25, 2004.

[11] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, Mary Lou Soffa - Instrumentation in Software Dynamic Translators for Self-managed Systems - University of Pittsburgh and University of Virginia - WOSS'04, Oct 31-Nov - Nov 1, 2004. - Newport Beach CA – USA

[12] Jackson, Quinn Tyler. Adaptive Predicates in Natural Language Parsing Perfection, n. 4, 2000. - http://members.shaw.ca/qtj/

[13] www.pcs.usp.br/~lta – Home Page of the Lab. of Language and Adaptive Technologies – University of São Paulo.

[14] Rocha, R. L. A. e Neto, J. J. Autômato adaptativo, limites e complexidade em comparação com máquina de Turing. In: Proceedings of the Second Congress of Logic Applied to Technology - LAPTEC'2000. SP (in Portuguese). São Paulo, 2000.