Proceedings of the 4th WSEAS Int. Conf. on Information Security, Communications and Computers, Tenerife, Spain, December 16-18, 2005 (pp423-428)

Adaptive device with underlying mechanism defined by a programming language

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, N^o 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 N^o 3400 – Vila Barcelona – São Caetano do Sul – CEP 09550-051
São Paulo – Brasil
avfreitas@imes.edu.br and joao.jose@poli.usp.br

Abstract: - An adaptive device is made up of an underlying mechanism, for instance, an automaton, a grammar, a decision tree, etc., to which is added an adaptive mechanism, responsible for allowing a dynamic modification in the structure of the underlying mechanism. This article aims to investigate if a programming language can be used as an underlying mechanism of an adaptive device, resulting in an adaptive language.

Key-Words: - adaptive devices, self-modifying machines, adaptive automaton, functional adaptive programming language, algorithms and computation theory.

1 Introduction

Adaptive devices were first studied and applied in the field of compiler implementation in order to get purely syntactic mechanisms intended to enlarge pushdown automata's expressions capacity [6]. The theory of adaptive devices started with simple-to-use formalisms, such as finite state automata, which enabled them - by adding a set of rules – to represent more complex problems, as those concerning non-regular languages and even context-depedent [11].

Adaptive technology involves techniques and methods associated with the practical applications of adaptive devices. Chronologically, these devices originated from researches into formal languages and automata. Formalism, however, stimulated applications in several different fields [7].

An adaptive device is made up of an underlying mechanism (eg an automaton, a grammar, a decision tree, etc.), to which what we call *adaptive mechanism* is conjoined, allowing the structure of the underlying mechanism to be dynamically modified. If, for instance, an adaptive mechanism is added to a finite state automaton, it is possible to increment or remove transitions and/or states during the input stream process [8] and to increase its expression power.

Adaptive technology, therefore, is basically characterized by re-using consolidated formalisms and increasing their expression power at the cost of a small increment in formal complexity [11].

An adaptive device should always be associated with a fixed and finite set of rules, which account for the reconfiguration of the device, occasionally considering an input stimulus and generating some exit symbol.

When guided by rules, a device starts operation in a given configuration and proceeds with the application of some rule from its set up to when there are no more input stimuli or when a configuration is reached to which application of rule is not possible. At this point, it is possible to determine, based on the configuration reached, whether the device accepts the complete sequence of input stimuli or rejects it.

In an adaptive device, a set of rules can be modified by another one, called adaptive actions, working on the original set of rules. An adaptive device, then, is made up of an underlying layer – a non-adaptive device guided by rules – and an adaptive layer defined by a set of adaptive actions.

Starting from adaptive automata and following the evolution of adaptive devices, new adaptive devices were conceived and associated with several areas, with different underlying mechanisms, such as statecharts[12], Markov nets [13], computational learning [18], natural languages processing [14], grammars [15], multilanguage environments [16], robotics [17], decision trees [11], etc.

2 Objective

This paper intends to investigate and assure that programming language can be used as an underlying mechanism of an adaptive device. Just like an adaptive finite automaton incorporates a finite automaton as an underlying mechanism, an adaptive programming language must use some programming language as an underlying mechanism. We are going to investigate, then, if the implementation of adaptive language is feasible, therefore applicable to problems recommending adaptive technology.

The adaptive mechanism found in the adaptive language will make it play a self-modifying role, performed by dynamic codes. Self-modifying codes employed in machine languages are often hard to write and to keep. Our proposal, however, dodges the difficulties common in machine languages because the adaptive technology considers the use of adaptive functions, specified by fixed well-defined rules, which secure reliability to the proposed solution.

Another objective is to show the technical viability of designing programming languages that modify themselves to tackle the problem to solve. A program is often divided into parts, in a top-down methodology. With adaptive languages, the division can be made by the bottom-up approach, the language modifying itself towards the problem [3].

To reach this objective a functional programming language will be used, with extensible features for the incorporation of the mechanisms of adaptive formalism [9]. The choice of functional paradigm doesn't restrict the conception of adaptive languages exclusively to this paradigm. As soon as the feasibility of the adaptive languages is assured, further research in other programming paradigms can be undertaken.

3 Adaptive Devices

An adaptive automaton is the result of a sequence of evolution of a structured pushdown automaton processed by adaptive actions. To each adaptive action a new automaton is implemented for the continuity of the input stream treatment.

To illustrate the applicability of adaptive devices, there follows a practical example quite useful for the exclusively syntactic resolution of problems often found in the recognition of context-dependent languages. It is a collector of names (identifiers) which, setting out from an initial situation in the input stream and classifies then either as found previously or not. By a change in the structure of the device that implements it, later recognitions of newly-collected identifiers will be dealt with as already found before.

Based on the technique shown in the example, it is possible to treat – without recourse to tables – the names of the variables of a language with a single scope. So, the mechanism implemented in this example can inspire substitutes in the recogniser project that replace the symbol tables with a purely syntactic mechanism of name collecting [6].

At first the adaptive automaton recognises, through its initial machine of states, any valid identifier, and classifies it as an ignored identifier. By a structured self-modifying in the device, the same identifier – if found on a later occasion – will not be classified as ignored again.

A group of transitions is then created so that the new identifier is associated with a path by the automaton states and those take it to a final state, which accepts the identifier and characterises it as a known identifier.

As soon as the identifier is characterised as known, the old path accepting such identifier must be eliminated from the automaton. Likewise, the adaptive call functions responsible for these operations must also be rearranged so that new enlargements can take place as a consequence of recognitions of identifiers sharing common prefixes to previously found identifiers.

To each new identifier found, a set of transitions is inserted into the automaton enabling it to recognise the new symbol (taking into account the other identifiers, previously incorporated) and the transitions that implement the path leading to the recognition of this identifier in the original automaton is eliminated.

4 Definition of the Underlying Mechanism

Our investigation probes into the viability of using a functional programming language as an underlying mechanism of the device, in a way similar to that of an adaptive automaton using a structured pushdown automaton as an underlying mechanism for self-modification. This investigation will likely enhance the development of a functional programming language with the adaptive characteristics.

Taken lambda calculus as the basis for functional paradigm of programming, we will adopt a

language supporting lambda calculus as an underlying mechanism [2] [4].
It is the better adherence to the concepts of adaptive technology that validates this choice because the model allows the construction of expressions with the treatment of dynamic codes (usually implemented with constructions of *eval* type in *Lisp*, *Scheme* or their dialects).
That duly considered, the underlying mechanism will be made up of a functional nucleus based on lambda calculus to operate as an interpreter of a language codified by the user.
Having this basic functional nucleus extensive characteristics, we will design an extensible adaptive layer to evaluate calls of adaptive actions, which will modify the host language, making it adaptive.
With the processing of adaptive actions new code instances are obtained and the execution of language is processed by successive context switch between the adaptive layer and the underlying mechanism responsible for the interpretation of each code instance.
Adaptive programming language will present, at an early stages, a code block liable to direct processing by the underlying mechanism until the execution of some adaptive action specified in the model takes place.
Adaptive language, thus, will be made up of a space of codes LF_1, LF_2,.., LF_n , so that, starting from initial language LF_1 and proceeding through calls of adaptive functions, language will evolve into successive configurations LF_1, LF_2, .., LF_n while execution is under process. The proposed adaptive mechanism holds a close structural analogy with the adaptive automaton dealt with in item 3.
To demonstrate the feasibility of our proposal, we will put to use a simplified language based on expressions. As a rule, an expression is a list in which the first element denotes a function and the following others denote the arguments. Each expression has to return a value. The language presents some native functions to treat arithmetic expressions and logics with, but the user may also define new functions to be combined with those already defined in the language.

5 Adaptive Language Operation

For our adaptive language to be processed, a processing environment must be created, made up of the functional nucleus and a control module, represented by the adaptive machine whose responsibility will be to evaluate adaptive calls.
Fig.1 shows what happens along the processing of adaptive language.

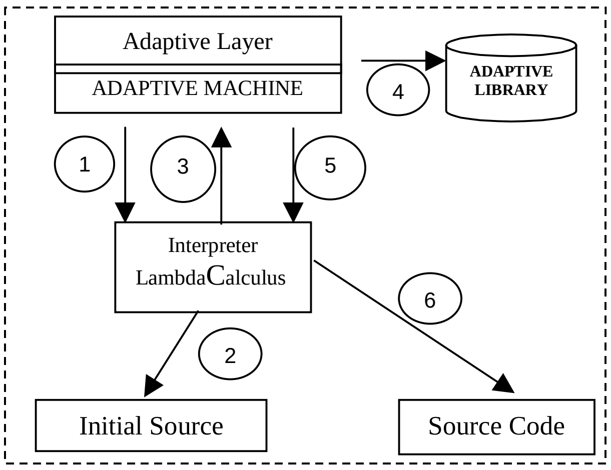

Fig.1 - Adaptive Language – Processing Environment

1. The adaptive machine makes the functional interpreter's call based on lambda calculus and passes to it the initial source code codified by the user;
2. The functional interpreter will start the evaluation processing of functions in the usual way until some call of adaptive function takes place. If there aren't adaptive calls, the interpreter will behave just as it were in a non-adaptive functional environment;
3. The control returns to the adaptive machine, which will provide the adaptive call handling;
4. Being adaptive actions made up of elementary actions, the adaptive machine may use adaptive layer functions;
5. As a result of the execution of adaptive actions, a new source code is generated;
6. This new source code is handed down to the functional interpreter, which will take upon itself the continuity of the execution of the adaptive functions.

6 Adaptive Layer Project

In a manner similar to that along the evolution of an adaptive automaton, when inclusions and/or exclusions of transitions and/or states take place – derived from the calls of adaptive actions – our adaptive language will also develop during execution through the inclusion or exclusion of functions, thus characterising the dynamics of adaptive code.
The calls of adaptive functions (denoted by adaptive actions) will then favour the features of

self-modification in the device. These functions will be constituted by calls of elementary functions, available in the adaptive layer, which through basic operations of query, inclusion, and removal of language functions will provide the device with dynamic modification.

Our adaptive layer, thus, will be made up of elementary actions *?adapt*, *+adapt*, and *–adapt*, responsible respectively for the query, addition, and eliminations functions of the underlying language. This set of elementary actions will correspond to the rules modifying in consequence the execution of adaptive language.

Naturally, for each individual problem, elementary actions will be extracted from the adaptive layer in amount and sequence convenient for the problem at issue.

To generalise consult functions, queries modelled according to pattern matching can be specified so as to return the functions list that fulfills the query.

For the functions of adaptive layer to perform their tasks, the functions making up our source program must be somehow addressed, in a similar way that an adaptive action – when eliminating or inserting a transition in an adaptive automaton – needs to refer to each of the original automaton's states or transitions.

The language used will be simple and based on expressions, so that a program in this underlying language may be reduced to the call of a single function, resulting from the composition of several other functions.

A program P then can be represented under the configuration if a list. For instance:

```
( a     ( b     ( c )    ( d )   ( e f g ) )   (h  i ) )
```

To make the idea of our functional program easier to envisage to the light of adaptive paradigm, the opening of parentheses will be indexed with the use of labels represented by whole numbers. These labels will start with the value 1 in the definition of the first function and will gradually increase as the new definition or function calls turn up along the text of the program.

Since a functional program can be represented by a tree-type structure, each label should be seen as a node from such a tree. So, in the previous example, our functional program P will be represented as follows:

```
1        2        3        4        5          6
( a      ( b      ( c )    ( d )    ( e f g ) ) (h  i ) )
                    3        4             5 2      6 1
```

For an adaptive treatment to be operated in a functional program, it takes first of all a translation framework allowing form a representation of the program closer to the adaptive formalism.

After the mapping is completed, the program has to be converted in to a new representation P':

```
          1   2    3    4    5         6
(  (F)   ( a ( b  ( c )  (d)  (e f g ) ) (h  i) )   (G) )
                    3    4         5 2     6 1
```

F and ***G*** corresponding respectively to the adaptive functions previous and subsequent to the evaluation of the function representing program P'.

These functions will be responsible for the self-modification of the device and will operate on the labels attributed to each function making up the initial program P'. For instance: if it becomes necessary for the adaptive function ***F*** to eliminate function *d* in program P', label *4* is passed as parameter and as a result of the evaluation of adaptive function ***F***, node 4 corresponding to function *d* will be eliminated from the tree.

This being the procedure, the functional program P is then represented by a new P' function, in which all component functions are mapped form the adaptive functional notation. If there aren't adaptive functions in the device, the representation P' will naturally be the same as the initial P, which, in this instance, doesn't characterize the existence of self-modifications.

Through the identification of each function in the program, each corresponding node in the tree of expressions can be referred. In the previous example, the tree would be shown like this:

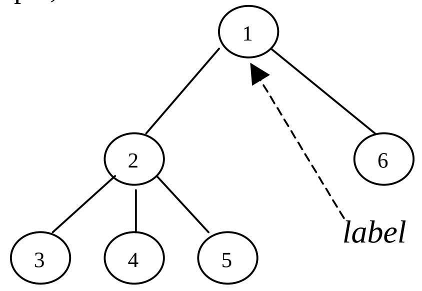

Fig.2 - Tree representing a program P

With the use of this address outline, exclusive for each component of the program, it is possible to specify adaptive functions that, through calls of the adaptive layer, will increase, remove, or alter the nodes of the tree, thus generating an adaptive code.

For our adaptive functions to accomplish the edition of the tree, labels should be included in the adaptive language, which allow the addressing of the several functions that make up the program. The linking of labels to the functions of the

program should naturally be optional and turned to only when the function at issue undergoes self-modification.
To implement this function of adaptive layer, a mechanism similar to the one employed in *Common Lisp* language [10] can be used by means of the "*go*" and "*tagbody*" native functions. In case these constructs are not present in the functional nucleus of the adaptive environment, it is possible to emulate such functions through definitions starting from other functions present in the language [1].

7 Conclusions

This paper considers the use of the programming language an underlying device in the adaptive device. The language thus obtained allows the introduction of dynamic characteristics into the initial code of language. We have shown the steps leading to this objective.
With the concepts and procedures presented along this paper, we have shown that the implementation of the adaptive language is feasible and can be consequently applied to problems to which adaptive technology is recommended.
The next steps will involve strict specification of the syntax and semantics of the language proposed along with the implementation of the adaptive module responsible for the control of the adaptive environment. The definition of the functions of label treatment and elementary actions of the adaptive layer is to be attended to, as well as the implementation of routines to control module-switching (adaptive x functional) and the procedures to treat environment variables.
After these specifications, it will be possible to implement an environment allowing the execution of the adaptive language. Some experiments with the project of adaptive language have already been made, the NewLisp environment [5] being used in the implementations.

8 References

[1] Baker, Henry G. - Metacircular Semantics for Common Lisp Special Forms. Nimble Computer Corporation. ACM Lisp Pointers V, 4 (Oct/Dec1992), 11-20.

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

[3] Knott, Gary - LispBook – Civilized Software, Inc. – 1997.

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

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

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

[7] 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.

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

[9] Rocha, Ricardo Luis de Azevedo da e Neto, João José - Uma proposta de linguagem de programação funcional com características adaptativas. IX Congreso Argentino de Ciencias de la Computación, La Plata, Argentina, 6-10 de Outubro, 2003.

[10] Steele, Guy L. - Common Lisp, The Language; 2nd Ed. Digital Press, Bedfor, MA, 1990.

[11] Pistori, Hemerson – Tecnologia em Engenharia da Computação: Estado da Arte e Aplicações. Tese de Doutorado – Escola Politécnica da Universidade de São Paulo. 2003

[12] Santos, J.M.N. - Um formalismo adaptativo com mecanismo de sincronização para aplicações concorrentes. Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo, Brasil, 1997.

[13] Basseto, B. A., Neto, J.J. - A stochastic musical composer based on adaptative algorithms. In: Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação. SBC-99. PUC-Rio, Rio de Janeiro, Brasil – 1999.

[14] Menezes, C.E.D. – Um método para a construção de Analizadores Morfológicos, Aplicados à Lingua Portuguesa, Baseado em Autômatos Adaptativos. Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo, São Paulo – Brasil, Julho 2000.

[15] Iwai, M. K. – Um formalismo gramatical adaptativo para Linguagens dependentes de Contexto. Tese (Doutorado) – Escola Politécnica da Universidade de São Paulo, São Paulo, Brasil, 2000.

[16] Freitas, A. V.; Neto, J.J. - Uma ferramenta para construção de aplicações multilinguagens de programação. In: CACIC 2001 – Congreso Argentino de Ciencias de la Computacion. El Calafate, Argentina, 2001.

[17] Souza, M. A. A., Hirakawa, A. R., Neto, J. J. - Adaptive Automata for Mobile Robotic Mapping. Proceedings of VIII Brazilian Symposium on Neural Networks - SBRN'04. São Luís/MA - Brazil. September 29 - October 1, 2004.

[18] Rocha, R. L. A.; Neto, J.J. - Uma proposta de método adaptativo para a seleção automática de soluções. In: Proceedings of ICIE Y2K – International Congress on Informatics Engineering. Buenos Aires, Argentina. 2000.