Using Adaptive Automata in a Multi-paradigm Programming Environment

João José Neto and Aparecido Valdemir de Freitas
Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, 158 – Cidade Universitária
S. Paulo – Brasil
joao.jose@poli.usp.br and avfreitas@imes.com.br

ABSTRACT

In this paper the architecture of an experimental multi-paradigmatic programming environment is sketched, showing how its parts combine together with application modules in order to perform the integration of program modules written in different programming languages and paradigms. Adaptive automata are special self-modifying formal state machines used as a design and implementation tool in the representation of complex systems. Adaptive automata have been proven to have the same formal power as Turing Machines. Therefore, at least in theory, arbitrarily complex systems may be modeled with adaptive automata. The present work briefly introduces such formal tool and presents case studies showing how to use them in two very different situations: the first one, in the name management module of a multi-paradigmatic and multi-language programming environment, and the second one, in an application program implementing an adaptive automaton that accepts a context-sensitive language.

KEY WORDS

multi-language programming, multi-paradigm environment, adaptive automata.

1. INTRODUCTION

Existing programming paradigms usually adhere to particular classes of problems. In order to handle complex and interdisciplinary problem, it should be convenient to use more than one paradigm in the application. [1] Multi-paradigm environments may be used to allow users to handle different pieces of programs, written in a variety of paradigms and styles without leaving their programming environment. [2,3].

The same idea applies for maintenance tasks in which existing programs are to be modified or extended in their functionality. Then tasks usually involve inserting or substituting program parts with new pieces of code, written in more than one language.

For instance, by using the logical programming paradigms, in some module of its application programmers state them in terms of rules, facts, and a goal. Such an approach adheres strongly to declarative-style implementation of that module. [4]

In single-language programs, programmers model the solutions of problems by using constructs available in the implementation language. When such a language does not adhere to the needs of the problem being solved, the programmer may have troubles when trying to map the chosen solution into its syntactical constructs.

Therefore, the lack of a multi-paradigmatic programming environment or the non-availability of a language appropriate to solve a specific problem will force programmers to simulate constructs that are absent in the chosen available language. [5]

In our proposal, a set of primitive functions is made available for the application process to call.

Applications are made up of a set of processes that interact with the programming environment by means of system calls and information exchanges.

The proposed environment should manage all resources needed to execute the multi-paradigmatic application, e.g. the allocation and management of shared memory and name handling for variables and data areas.

By means of synchronizing and communication mechanisms, the environment assures the consistence of shared data concurrently referenced by processes.

Garbage collection and resource retrieving are also housekeeping functions that must be provided by the environment. [3]

In our proposal, multi-paradigmatic applications are run under our programming environment, which provides all the run-time facilities mentioned above.

This paper presents an implementation proposal for a programming environment that provides the system calls that are appropriate to the interoperability among the application parts.

Adaptive automata are introduced as a subjacent formal technique used, for illustration purposes, both in the implementation of one of the modules of our environment and as an application case study.

The first illustrative application describes an adaptive implementation of one environment's modules and the second one, a simple program running under our environment.

2. ADAPTIVE AUTOMATA

Adaptive automata is briefly introduced in this section in order to allow the reader to understand the mechanism chosen for the implementation of our system's name manager. Details on the formalism and its properties are available in [6].

Regular and Context Free languages are structurally simple, and may be easily accepted by popular models, e.g. finite-state and pushdown automata, respectively. [7]

Structured pushdown automata [8] are special forms of general pushdown automata that operate as a set of mutually recursive finite-state-like sub-machines. In this model, a pushdown store is used for holding return states whenever a submachine calls another one. This arrangement is particularly useful for improving efficiency and readability, and is employed as an underlying model for adaptive automata.

Unfortunately, there are useful languages that are not suitable to be accepted by structured pushdown automata. Adaptive automata have been proposed as a general formalism that has Turing-machine power, so they have power enough to accept virtually any language, despite its complexity.

So, context-dependent languages may be modeled by adaptive automata, which make use of the so-called adaptive rules in order to dynamically modify the set of rules defining the automation.

Most programming and natural languages may be described by means of context-sensitive formalisms, such as context-dependent grammars, Turing Machines, two-level grammars, attribute grammars and many others.

Adaptive automata [6] and adaptive grammars [11] have been developed in such a way that complex languages be described and accepted by means of formal models whose operation is similar to that of structured pushdown automata.

While no context-dependencies are exercised by the sentence being recognized, adaptive automata operate as a structured pushdown automaton.

Whenever any context dependency is detected, a self-modifying adaptive action is performed which may change the current set of rules defining the adaptive automation.

Afterwards, the adaptive device will use the newly obtained set of states and transitions, which may drastically change the automaton's behavior in some cases.

The self-modifying approach showed by adaptive formalisms allows us to propose a new paradigm for software construction, which handles in a natural way the incrementally-changing behavior of some complex systems, e.g. intelligent (learning) software.

In order to illustrate the operation of adaptive automata, an example is given below for the implementation of an acceptor for the context-dependent language $a^n b^n c^n$, $n > 0$.

(sentences: abc, aabbcc, aaabbbccc,…)

Initially the adaptive automaton has the shape depicted in Fig-1.

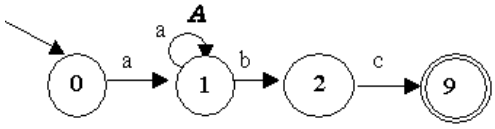

Fig-1 Adaptive automaton for $a^n b^n c^n$

Adaptive action *A* is responsible for modifying the shape of the automaton. Therefore in response to the receipt of each token "a" in the prefix of the sentence being accepted.

The adaptive automaton will add and eliminate adequate states and transitions in order to increment by 1 the number of both "b"'s and "c"'s accepted in sequence by the current set of transitions in the automaton.

An illustrating example of step-by-step use of the automaton above is the recognition of the sentence "aaabbbccc".

After consuming the first "a", the automaton evolves from state 0 to state 1 and remains unchanged (no adaptive action is executed in this transition).

After consuming the second "a", it executes the adaptive transition that brings it back to state 1, and executes adaptive action A, which changes the shape of the automaton to the configuration in Fig-2.

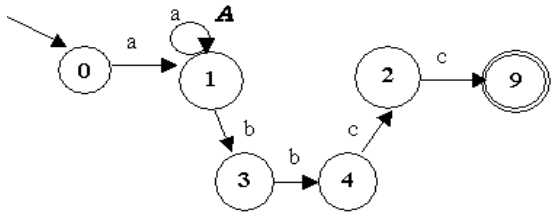

Fig.2 – Configuration after consuming "aa"

The next symbol "a" is then consumed and a similar operation is executed, resulting the configuration in Fig-3.

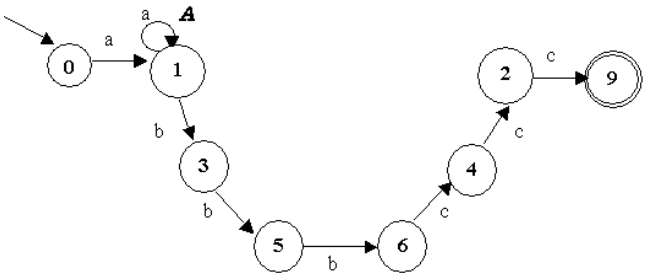

Fig.3 – Configuration after consuming "aaa"

The sentence may then be fully consumed by the automaton in this configuration, since no further adaptive actions are executed by the remaining transitions: b (1→ 3), b (3→ 5), b (5→ 6), b (6→ 4), c (4→ 2), c (2→ 9), and final state 9 is reached after full consumption of the input sentence.

3. A PROPOSAL FOR OUR MULTI-PARADIGM ENVIRONMENT

In this environment, multi-paradigm applications are structured as sets of processes, each corresponding to a different module. Each module may be written in different languages or paradigms. Some functions, implemented as system calls, are provided in order to integrate components. Concurrency is handled through a message-passing scheme and synchronization primitives. [9]

By allowing the creation of new processes and their synchronization, the environment also provides process control management.

With such an arrangement, it is advisable that applicatives be structured in such a way that each of its modules be responsible for some well-defined activity. Programmers must adequately insert system calls in order to guarantee proper execution flow.

Data are also exchanged by means of a shared memory scheme whose management is performed by our environment.

Explicit data transfers may be needed among processes. Our environment automatically provides the allocation of data transfer areas. System calls allow users to inform the system on the source, destination and nature of the data to be transferred.

In order to keep track of the names needed to identify such data areas, the system uses the name manager described in section 4 and provides an adequate transfer protocol for the information which to be handled.

Data type compatibility is assured by this exchange mechanism by means of automatic format-conversion operations provided by our environment.

Of course, an adequate format must be chosen in order to guarantee proper data exchange among modules developed in different programming languages, since the corresponding abstract machines may work with different data representations.

In the present work, a simplification was made in order to reduce the complexity of our prototype: no complex data types are supported, but a string-passing mechanism has been chosen instead, since, without loss of generality, any data type may somehow be represented in string notation.

Anyway, all data are stored with an accompanying tag indicating the type it belongs to, for type-checking purposes at run-time.

From the implementation viewpoint, our environment first performs all needed allocations of shared resources (e.g. shared memory, system processes). Afterwards, control is passed to the application and the environment sleeps until some request is issued by the application (e.g. data transfers, control requests, program activates, process termination, etc.)

The data-transfer operations provided by our environment are listed below:

- heap writing (updating): a statically allocated named data block is registered into a shared memory space, and its name is kept by the environment by the name manager described in section 4.
- heap reading (non-destructive): a named data block is retrieved from the heap without changing its contents.
- writing into an ordered dynamically allocated data structure: a named data block is kept by the environment's name manager. Application processes may choose the ordering criterion to be followed by the retrieving operations.
- retrieving data from an ordered data structure: The dynamic data area is searched according to the proper ordering criterion (e.g. by name) for the specified datum to be retrieved, and the data item is erased from the data structure. A garbage collection procedure is used for housekeeping.
- setting, resetting and switching the value of system boolean variables: these operations allow programmers to use boolean flags for controlling the execution of their code. This flag data area is static, and their reading is not destructive.

Type checking is performed in a very simple way, so no type mixing is allowed in our current prototype. However, future versions should implement type checking and type conversion, under user's control.

From the operational point of view, the naming and type checking mechanisms are very similar to link-editing operations performed by the system.

Dynamic link-edition is also available in our prototype through the use of a set of calls to the environment system primitives that are responsible for import and export operations (whenever an application process needs to transfer data, these primitives are activated, and the corresponding data and name mapping is performed by the environment).

4. AN ADAPTIVE DESIGN FOR THE NAME MANAGER

The information exchange among different processes of the application system in our environment is more easily and securely performed with the aid of the environment's primitive operations.

Such interchange is provided by a parameter-passing mechanism that allows any of the modules of the applicative on the correct way data must be referred to.

In other words, the parameter-passing mechanism allow to specify how each data block is identified and tagged, so the environment is able avoid improper operations to be executed on them. (although this type-checking mechanism may be easily implemented, it has not been yet added to our running prototype).

This scheme creates a set of global names to be used by all modules within the application program. Obviously, such names must be unique in the system, so our environment avoids name duplication. Therefore, every data associated to one of these global names must be referenced as a pair (global name, value). Types may be retrieved from the data representation, since all stored data have an associated tag that tells its corresponding type.

In order to design an adaptive implementation for our name manager, adaptive automata have been used. Obviously, there are many other ways to implement name managers. Our option for an adaptive design has been made mostly for illustrating how adaptive techniques may be used in practical problem resolution

In our adaptive design, sequence of symbols in valid names corresponds, in an adaptive automaton, to possible paths from its initial state to any of its final states. Each transition in this automaton consumes the corresponding symbol from the input sequence. Common prefixes share the corresponding transitions of the automaton. Prefixes ending in any non-final state are not associated to valid names. All final states are tagged, indicating the corresponding name type. A value of the proper type is also associated to this state.

Being an adaptive device, our automaton will evolve during its operation, so that, for all names it handles, prefixes accepted by the existing transitions will not affect the shape of the automaton, while symbols for which no path is found in the current automaton will create the lacking transition, so the automaton will evolve by adding new transitions in order to accept the new sequence thereafter.

Therefore, throughout the operation of our environment, this adaptive automaton will represent all valid active names in the system.

Names not needed anymore are excluded from the automaton by searching the associated path backwards, for states with more than one departing transition (consider final states as having a special extra departing transition), and eliminate all transitions with only one departing transition. [10]

The following example illustrates how the name manager works. Starting from a single non-final state without any departing transitions, this adaptive automaton accepts sequences of letters, and incorporates transitions as needed to add the currently accepted sequence to the set of valid names in the system.

In order to achieve such a behavior, all symbols in the name that have a corresponding consuming transition in the automaton will not change its shape, while all symbols not corresponding to existent consuming transitions will add such a transition to the automaton, and will also prepare the new path to accept new foreign suffixes.

The resulting data structure depicted by this automaton is, of course, a growing tree whose root node is the initial state of the adaptive automaton, and its final states correspond to the leaves of the tree. The sequence of figures 4,5,6 and 7 illustrate the growth of this tree-shaped name automaton after accepting the names, abc, ad and ab, respectively.

Fig.4 – Initial Situation

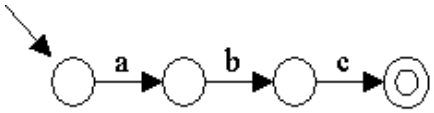

Fig. 5 – Situation after consuming the name abc

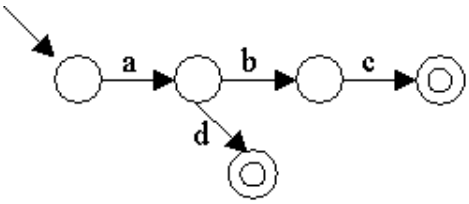

Fig. 6 – Situation after consuming abc, ad

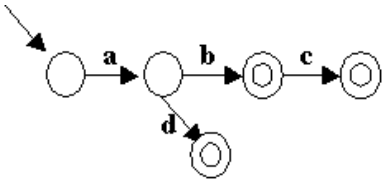

Fig. 7 – Situation after consuming abc, ad, ab

5. RUNNING A MULTIPARADIGMATIC ADAPTIVE APPLICATION UNDER OUR ENVIRONMENT

In this simple illustrating example, a multi-paradigmatic implementation of the adaptive automaton described in section 2 is shown, running under our environment prototype. The overall structure of this application is sketched in Fig. 8.

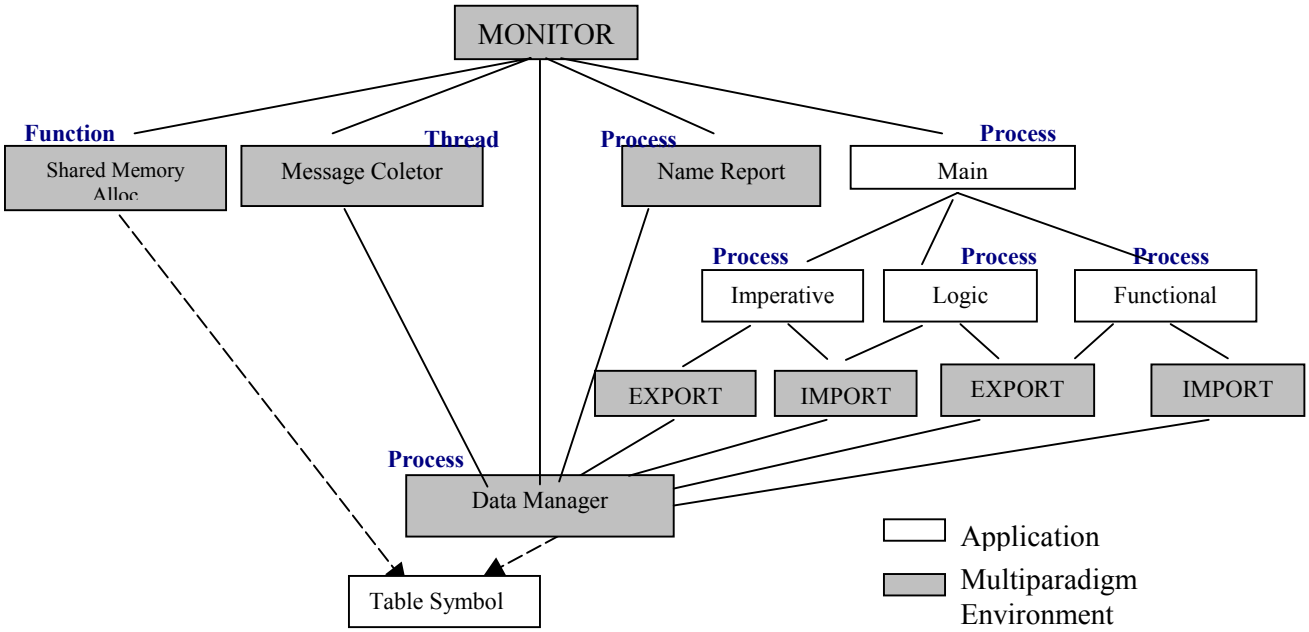

Fig. 8 – Structure Application Sketch

Besides the system modules already described in section 3, and the name manager described in section 4, the functionality and structure of the modules that compose the application program are described in this section.

In this case study on the implementation of adaptive automata, a multi-paradigm and multi-language program has been developed for running under our environment.

Three major modules have been developed: an imperative-paradigm module, for interfacing purposes; a functional-paradigm, for the static part of the simulation process; and a logical-paradigm, for the dynamic part of the simulation process.

Shared memory is used for holding data that are used by more than one module. The imperative module of the program operates as an interface between the program and the external environment.

It is responsible for: acquiring data from input files or devices; analyzing the input; decomposing it into elementary segments classified by category; and performing formatting and printing operations.

It is also responsible by the lexical analysis of the input stream, and by feeding the functional module with the tokens extracted from the input media whenever needed.

Other operations, such as collecting statistical information, generating traces of the operation, and performing control functions have been included in this module as well.

The imperative module also reads from an input device the description of the automaton to be simulated, and stores it into a shared area for being interpreted by the functional module, and modified by the logical module.

The functional module of the program has been designed to simulate all basic non-adaptive operation of the automaton being implemented. It implements a general interpreter of a set of rules describing the adaptive automaton to be simulated.

This description may either be always the same or it may change from run to run. So, in the later case, the description of the desired automaton must be input prior to starting the simulation. This function is performed by the imperative module, which passes the description (read from the input) to the functional module, so allowing the simulation to begin.

Simulation operates conventionally by searching the description for rules that adhere to the current configuration of the automaton and to the current input to be processed. Once a rule is selected, it is applied by the algorithm and the result is a new configuration for the automaton. In the case that the selected rule makes use of adaptive actions, that may result in modifications to the set of rules itself, resulting a new shape for the adaptive device. Further transitions will occur by applying the new set of rules until another adaptive action is executed.

The logical module of the program is activated by the functional module whenever an adaptive action is requested in the simulation.

The main purpose of this module is to perform all self-modifying operations corresponding to the request received from the functional module in response to the interpretation of an adaptive transition.

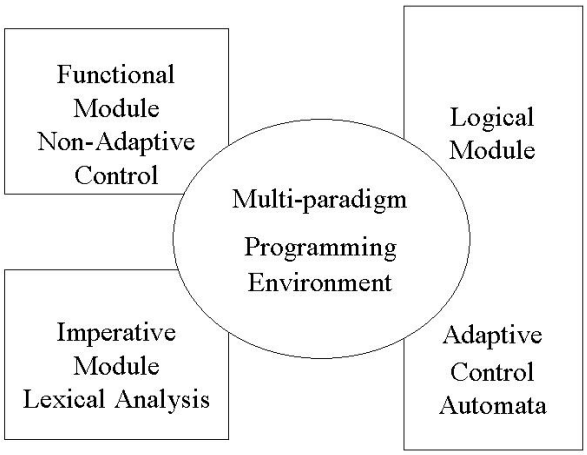

Fig. 9 – Modules Application Sketch

It works by reading the currently used description of the automaton and performing the requested edit operations on the set of rules it represents.

In order to select the proper changes to the set of rules, the logical module extracts the corresponding commands from a description of the adaptive function to be applied.

Formal parameters are first replaced by actual arguments of the adaptive function being called, then the set of editing commands is applied to the set of current rules.

After being updated, the set of rules describing the new shape of the adaptive automaton is released for use, and command is passed back to the functional module for proceeding with the simulation.

Errors and malfunctions in the program are reported by the imperative module whenever detected by any of the modules that simulate the adaptive automaton.

In the cases that some module detects any error situation, that module will activate the imperative module by sending to it a request for reporting the error situation.

6. CONCLUSIONS

Multi-paradigm and multi-language programming allow developers to flexibly express the implementation of their application programs by using a mix of different languages and even paradigms.

Among the advantages of this technique we can mention: Programmers are allowed to use the most important features of each language and paradigm; They are able to choose the most adequate language for each part of the application being developed. In the case multiple programming groups are used in the development of the project, the best of the skills and knowledge in each team may be used in the development of the final product.

In the work described in this paper, a prototype of such a programming environment has been used as a reference, and adaptive automata have been employed as a formal mathematical model in two very different situations: in the development of the name manager of the environment and in the construction of a multi-language and multi-paradigmatic application case study that implements an adaptive acceptor for the famous context-sensitive language $a^n b^n c^n$.

Besides validating most of the initially proposed arguments, these experiments have confirmed the effectiveness of both the multi-paradigmatic/multi-language approach in programming and the adaptive techniques used to implement solutions for complex problems.

7. REFERENCES

[1] Zave, Pamela, A compositional Approach to Multiparadigm Programming, AT&T Laboratories, IEEE Software – September 1989.

[2] Hailpern Brent, Multiparadigm Research: A Survey of Nine Projects" - IEEE Software - January 1986.

[3] Spinellis, Diomidis D., Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming", February 1994, A thesis submitted for the degree of Doctor of Philosophy of the University of London.

[4] Mellish, C.S. and Clocksin, W.F., Programming in Prolog, Springer-Verlag Berlin Heidelberg, - 1994.

[5] Budd, Timothy A., Multiparadigm Programming in LEDA, Oregon State University, Addison-Wesley Publishing Company, Inc, 1995.

[6] Neto, J.J., Adaptive automata for context-dependent languages, ACM SIGPLAN Notices, Volume 29, Número 9, Setembro 1994.

[7] Lewis, Harry R. and Papadimitriou, Christos H., Elements of the Theory of Computation". Second Edition. Prentice-Hall Inc. 1998.

[8] Neto, J. J., Introdução à Compilação, Editora Livros Técnicos e Científicos Ed., Rio de Janeiro, 1987.

[9] Freitas, A. V., Aspectos de Implementação de Ambientes Multilinguagens de Programação, Dissertação de Mestrado, EPUSP, 2000.

[10] Neto, J. J. and Freitas, A. V., Aspectos do Projeto e Implementação de Ambientes Multilinguagens de Programação,CACIC 2000–VI Congreso Argentino de Ciencias de la Computación –2000 , Argentina.

[11] Iwai, M. K., Um formalismo gramatical adaptativo para linguagens dependentes de contexto. São Paulo 2000. Doctoral Thesis. Escola Politécnica USP.