An Adaptive Approach for Error-Recovery in Structured Pushdown Automata

J. J. Neto, H. Pistori, A. A. Castro Jr. and M. R. Borth

***Abstract* — In this paper, we introduce a practical way to implement an efficient error-recovery scheme intended to handle incorrect input received by devices based on finite-state and structured pushdown automata. We start proposing an exact recovery scheme for single errors in deterministic finite-state devices. Then, we extend this approach to perform local recovery in the individual sub-machines of structured pushdown automata, and we complement the proposed single error-recovery scheme by considering non-deterministic devices and interactions among sub-machines. Finally, we add a panic-mode scheme to the proposed method in order to take into account even multiple-error inputs. The chosen adaptive approach used in this strategy leaves untouched the underlying original automaton until an error is detected in its input stream. At this moment, an adaptive action is taken that adequately extends the automaton in order to perform the needed recovery in response to the detected error. Afterwards, the transitions included in the extension are executed according to the contents of the input stream, and when the underlying automaton is reached anyway, the extension is discarded, restoring the automaton to its original shape.**

***Keywords* — Structured Pushdown Automata, Adaptive Automata, Error Recovery, Compiler Construction.**

I. INTRODUCTION

Language acceptors are expected to recognize symbol sequences belonging to the language they define. In practice, strings given as input to language acceptors often include misspellings and grammatical errors, and acceptors must reject them in this case. When used within a language processor, an acceptor should not simply reject the string, but bypass the error in order to proceed searching for further errors in its input.

In general, error-recovery is a complex subject, even when focusing low-complexity devices such as finite-state automata [1, 2, 3, 4, 5, 6, 9, 10, 11]. Two adjacent errors are called simple when they are sufficiently far apart that their effects in the behavior of the automaton are not superposed, otherwise they are said to be multiple. By observing practical programs, one may remark that simple errors are significantly more likely to occur than multiple ones, and that for most practical situations no complex methods are needed for handling them. Nevertheless, multiple errors do occur, and by handling them as exceptions, it is possible to concentrate efforts in recovering simple errors.

The presence of an error in the input text is detected whenever the acceptor cannot perform any valid transition in response to some input symbol. Symptoms of the presence of an error seldom occur in its neighborhood, leading the acceptor to mistakenly perform a sequence of transitions before rejecting the input string. Although being inadequate for the particular input string being handled, note that those transitions would be perfectly valid for other correct input sentences. Such unavoidable imprecision usually make automatic error-repair procedures far worse than a manual correction performed by the text's author. Therefore, instead of viewing error recovery as an approach to fix the input text errors, we will use it for re-synchronizing the acceptor with its wrong input string, and proceed searching for further errors.

An adaptive automaton is an adaptive rule-driven device whose subjacent mechanism is a structured pushdown automaton. This formalism preserves intact most part of the clean syntax of structured pushdown automata, allowing a very intuitive view for the resulting Turing-powerful formalism as an automata with a dynamically changeable set of transitions. Such modifications are expressed through a very simple language where queries, deletions and insertions of transitions are specified by adaptive functions to be executed just before or after the execution of some transition. The self-modification required to model an error-recovery behavior can be elegantly captured by adaptive automata. The efficiency of such method is also discussed and illustrated in this work, through the analysis of an implemented scheme of adaptive error-recovery.

The next section provides some notational revision for structured pushdown automata and adaptive automata. Afterwards, a classical error-recovery mechanism is presented. Section 7 presents in details our adaptive error-recovery approach, describing also some implemented examples. Last section gives some conclusions and future work proposals.

II. NOTATION

A. Structured Pushdown Automata Revisited

A structured pushdown automaton (SPA) may be viewed as a set of finite state machines with special transitions for passing the execution control back and forth among machines. Formally, a structured pushdown automaton is a tuple $M = (S, Q, \mu, \Sigma, \Gamma, Q_0, F, \delta^i, \delta^e)$ where:

- S is a finite non-empty set of sub-machines.
- Q is a finite non-empty set of states.

J. J. Neto, Polytechnic School of the University of São Paulo (EPUSP), São Paulo, Brazil, joao.jose@poli.usp.br

H. Pistori, Dom Bosco Catholic University (UCDB), Campo Grande, Mato Grosso do Sul, Brazil, pistori@ucdb.br

A. A. Castro Jr., Federal University of Mato Grosso do Sul (UFMS), Ponta Porã, Mato Grosso do Sul, Brazil, amaury.ufms@gmail.com

M. R. Borth, Federal Institute of Mato Grosso do Sul (IFMS), Ponta Porã, Mato Grosso do Sul, Brazil, marceloborth@gmail.com

- μ: $Q \rightarrow S$ is a function. The inverse of μ, μ', induces a partition on set Q, with each part corresponding to the set of states of each sub-machine. Given a sub-machine $s \in S$, $\mu'(s) \subseteq Q$ is the set of states of s.
- Σ is a finite non-empty input alphabet.
- Γ is a finite set that is called the stack alphabet.
- $Q_0 \subseteq Q$ is a set of initial states where, for each $s \in S$, $|\ \mu'(s) \cap Q_0\ | = 1$ (Each sub-machine has one, an only one, initial state).
- F is a set of final states, which also corresponds to sub-machine return transition.
- δ^i is a relation over $\mu'(s) \times \Sigma \cup \{\epsilon\} \times \mu'(s)$, where $s \in S$. Each element of this relation is called an internal transition.
- δ^e is a relation over $\mu'(s) \times S \times \mu'(s)$. Each element $(q, m, q') \in \delta^e$ is denominated an external transition or a sub-machine call, and can be interpreted as a sub-machine call from state q to the sub-machine m pushing the state q' onto a pushdown store.

Configurations in SPA are triples (q, x, z) where $q \in Q$ is the current state, $x \in \Sigma^*$ is the part of input string yet to be read and $z \in Q^*$ is the content of the pushdown store, holding a sequence of sub-machine return addresses. When $\delta^e = \emptyset$ and $|S| = 1$ the SPA specializes to an ϵ-FSA, operating as such (the third element is not used). Otherwise, the step relation, $\vdash$, which determines the machine operation, is extended to comprise the sub-machine call and return dynamics. Formally, given a SPA $M = (S, Q, \mu, \Sigma, \Gamma, Q_0, F, \delta^i, \delta^e)$, $q, q' \in Q$, $\sigma \in \Sigma \cup \{\epsilon\}$, $x \in \Sigma^*$, $y, y' \in Q^*$ we have $(q, \sigma x, y) \vdash_M (q', x, y')$ if and only if one of these three conditions apply:

1. $(q, \sigma, q') \in \delta^i$, $y = y'$ and $\mu(q) = \mu(q')$. [Internal transition]
2. $(q, s, p) \in \delta^e$, $q' \in Q_0 \cap \mu'(s)$, $y' = py$, $\mu(q) = \mu(p)$ and $\sigma = \epsilon$. [Sub-machine call]
3. $q \in F$, $y = q'y'$ and $\sigma = \epsilon$. [Sub-machine return]

B. Adaptive Automata Revisited

An adaptive automaton is an adaptive device $A_D = (S_M, A_M)$ whose subjacent mechanism is a structured pushdown automaton $SM = (S, Q, \mu, \Sigma, \Gamma, Q_0, F, \delta^i, \delta^e)$, as defined above; and the adaptive mechanism $A_M = (A_F, A_C)$, comprises a set of adaptive functions, A_F, and a function $A_C : \delta^i \cup \delta^e \rightarrow (A_F)^2$. The function A_C links each transition to a pair of adaptive functions to be executed before and after the transition. The set A_F includes the special symbol ϵ, representing the empty adaptive function. The elements of A_C are called adaptive actions, or adaptive function calls.

Each adaptive function $f \in A_F$ is a set of elementary adaptive actions, of one of three kinds: search, erase and insert elementary action. Search actions are patterns used for selecting the transitions that erase and insert actions operate on. Patterns, in this context, are transition-shaped structures whose states, input symbols and adaptive actions may be replaced by variables and generators. In this work we choose to use implicit variable declarations. Each occurrence of a variable is prefixed with a question mark. Generators designate a symbol not used elsewhere, which is intended to be used in the dynamic creation of states in the automaton. Like variables, generators, denoted with the asterisk prefix, are implicitly declared. Implicit declarations are useful for graphical representation. Since graphics are essentially non-linear, they may be read from any directions (explicit declarations would eliminate this freedom). Finally, a reserved variable, *?c*, is used for referencing the current state during automaton operation. This amendment may be used, in many cases, to replace the formerly defined [8] adaptive action parameter passing mechanism, which is not used here, so it has been omitted for simplicity.

Figure 1 represents an adaptive automaton that recognizes the classical context-dependent language $a^n b^n c^n$. The subjacent mechanism, S_M, keeps reading the symbols a and calling the adaptive function F, denoted $[.F]$ (the dot in the notation indicates that F is called after the execution of the state transition), for each symbol a read. The adaptive function just seeks for the ϵ transition (here working as a mark) and replaces it by a pair of transitions that reads the substring bc. The ϵ transition mark is kept between the transitions that read b and c, so that when the i-th a is input, the automaton will have a sequence of transitions that are able to consume the sequence $b^i c^i$.

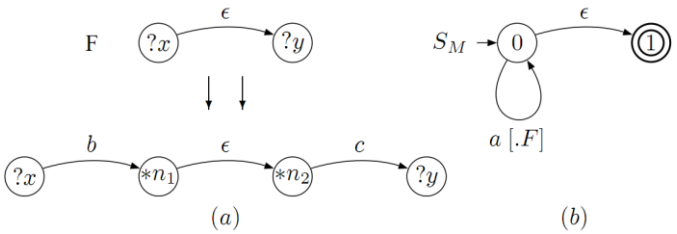

Figure 1. Adaptive Automaton that Recognizes $a^n b^n c^n$. (a) Adaptative Mechanism (b) Subjacent Mechanism.

C. Auxiliary Concepts and Definitions

Definition Let $M = (S, Q, \mu, \Sigma, Q_0, F, \delta^i, \delta^e)$ be an SPA and $q \in Q$. *First*: $Q \rightarrow 2^Q$ is a function where $q' \in$ *First(q)* if and only if $\exists \sigma \in \Sigma\ |\ (q, \sigma x) \vdash^*_M (q', x)$, $x \in \Sigma^*$ [1]. It calculates the set of states reachable from q, after reading one symbol.

Definition *SFirst* : $Q \rightarrow 2^\Sigma$ is a function where, for each $q \in Q$, $\sigma \in$ *SFirst*(q) if and only if $(q, \sigma x) \vdash^*_M (q', x)$, $x \in \Sigma^*$, $q' \in$ *First*(q).

Definition Given a state $q \in Q$ from an SPA M, let *First*$(q) = \{q_0, q_1, ..., q_n\}$. The function *Second* : $Q \rightarrow 2^Q$ is defined for each element $q \in Q$ as:

$$Second(q) = \bigcup_{0 \le i \le n} First(q_i)$$

Given some state $q \in Q$, the reserved symbol θ will denote the set $\Sigma - SFirst(q)$. Assume that each state q will carry a special transition, the error transition that is automatically activated whenever, being in q, the automaton reads a symbol in θ. The destination of such transition is a trap, non-final,

[1] $\vdash^*$ denotes the transitive closure of the step relation, $\vdash$

error state that will consume all the remainder of the input string.

III. Classic Simple Error Recovery

Given a finite state automaton, an error-recovery strategy should extend the automaton to allow it to keep consuming the input string despite the error detected. Omission, insertion and substitution of a symbol in the input string are the only sources of simple errors, which may be recovered through the reinsertion of an omitted symbol, the omission of an inserted symbol or the substitution of a wrong symbol by the correct one. The elimination of incorrect symbols may be done by adding transitions consuming the wrong symbols in the current state and leading the automaton to a specific error recovery state.

In order to complete our simple-error recovery mechanism, two further transitions are needed, replicating the normal transitions into the added extension: the first one departs from the new recovery state and the second, from the original state. A special care must be taken for preserving the structure of the original device: instead of inserting error-recovery transitions directly to the automaton, an empty-transition is added from the state in which the error is detected to an auxiliary error-recovery state. In operation, such empty-transition is activated only when no other normal transition is allowed. That is achieved by imposing greater priority to normal transitions than to empty ones, and its effect is that the recovery extension is activated only in case of errors.

The following algorithm implements the recovery scheme described:

```
Input: FSA M = (Q, Σ, q0, F, δ)
Output: M with Simple Error Recovery

For each state q ∈ Q Do
  Add two new states, e1 and e2, to Q
Add the empty transition (q, ϵ, e1) to δ //Isolate
Error States

For each c ∈ Q – {First(q) ∪ Second(q)} do
  Add the transition (e1, c, e2) to δ //Consume
Wrong Symbol
For each qs ∈ Second(q) Do
  Let b be the symbol of Σ that guarantee the
presence of qs in Second(q)
  Add the transition (e1, b, qs) to δ //Elimination
Error
  Add the transition (e2, b, qs) to δ //Substitution
Error
For each qf ∈ First(q) Do
  Let a be the symbol of Σ that guarantee the
presence of qf in First(q)
  Add the transition (e2, a, qf) to δ //Insertion
Error
If q is a final state then e2 must be made final too
```

Assuming $\Sigma = \{a, b, c\}$, Figure 2 shows the application of the error recovery mechanism to state p. Added transitions appear in dotted lines.

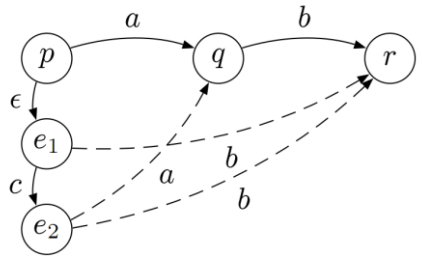

Figure 2. Simple Error Recovery

IV. panic Mode Error Recovery

Further transitions may be added to the described extension, allowing multiple errors to be handled too. A simple way to include multiple-error-recovery consists in successively eliminating symbols from the input data until some special symbol is found that allows the automaton to proceed to some corresponding synchronization state. Although being forceful, this technique produces good results in most practical cases.

By inspecting the language, a set of synchronizing symbols may be chosen such that their presence in the input text determines the start, the end or some significant point of the sentence. In order for this technique to be effective, we should choose synchronizing symbols that be likely to occur in typical input texts. In practice, it is usual to include symbols or keywords that delimit commands, expressions or groupings, as well as operators, separators and punctuation symbols. The best synchronization symbols are those that do not belong to more than one syntactic construct that are likely to occur simultaneously anywhere in the input text. From each new state created in the error-recovery extension, corresponding to the points where unsuitable symbols are discarded:

- Add a set of transitions for eliminating all non-synchronizing symbols, holding the automaton in the same state, until any synchronizing symbol is found.
- Add a set of transitions for consuming any synchronizing input symbol, moving the automaton to the state it would reach in the original automaton upon finding such symbol in correct sentences

This practice simulates replacing the discarded part of the input string by another one the automaton would expect to find instead. This technique has a wide application since the less the requirements on the use of the input string in case of multiple errors, the simpler its implementation will be. The following algorithm, which could be easily optimized if executed along the simple-error recovery, summarizes the multiple error recovery mechanism.

```
Input: FSA M with Simple Error Recovery
Output: M with Simple and Multiple Error Recovery
Let S be the set of synchronizing symbols
For each state e2 ∈ Q Do
  For each s' ∉ S Do
    Add the transition (e2, s', e2) to δ
  For each s ∈ S Do
    Let qs be the destination state of the transition
      consuming s in M
    Add the transition (e2, s, qs) to δ
```

Obviously, the symbols in *S* that are more adequate for this purpose are those that, at least in the context of recovery have unique corresponding q_s. Although the above technique has been use to complement the handling of simple errors, it may be used alone with the original automaton, especially for situations in which no rigorous recovery is needed.

The procedure described so far is enough to recover errors in a finite state automaton. However, it generates too many states and transitions, and its behavior is often non-deterministic. Additionally, error handling occurs only when there are errors in the input text, making the resulting extension remain unused in all normal cases. However, the extensions referring to each original state are mutually independent and independent of the original automaton, so it is possible to consider each of them separately, and to activate the proper one exclusively when the corresponding specific error is detected. Such independence creates an option for the designer, allowing that only the desired parts of the extension mechanism to be used. A practical option consists in pre-building all recovery extensions without physically inserting them into the original automaton, and activating them from disk only when an actual error detection occurs. Another good option is the subject of this paper, and consists of building and executing the extensions strictly at run-time, when the error is actually detected.

V. Error Recovery in SPA

Many authors address error recovery in traditional LR and LL pushdown automata [9, 4]. Being structured pushdown automata deeply based on finite-state devices, we may adapt the methods described above in order to recover errors in structured pushdown automata. The following cases have to be considered in this case:

- At internal transitions not involving final states all methods used for finite-state automata may be applied without change.
- At sub-machine call the contents of the pushdown store change and some action must take place in response.
- At sub-machine return, when a sub-machine finishes its operation, there is complementary change in the pushdown store, requiring some corresponding action.

Final states in sub-machines conceptually differ from those in finite-state automata, since the former represent the end of the syntactic construct defined by the sub-machine while the later indicate the end of the whole sentence. Therefore, while the detection of an error at the final state of a finite-state automaton initiates some error-recovery procedure, a similar situation in a sub-machine must be interpreted as a valid condition for returning to its caller sub-machine. Therefore, on the detection of an error at some final state of a sub-machine that is not the final state of the automaton, we must verify the behavior of the automaton in all states reachable after returning and consuming the next input symbol. In addition to recovering errors corresponding to internal transitions we will now consider interrelations among sub-machines in our recovery strategy. As the definition of first and second successors are based on the step relation of structured pushdown automata, a careful reading reveals that they already apply to the cases described above. However, the algorithm must be slightly modified by replacing internal transitions to sub-machine calls, when some first or second successor does not belong to the same sub-machine.

A. Multiple-error recovery in SPA

This scheme follows the one presented before for finite-state automata. In addition to the intricate methods needed for finite-state error recovery, the contents of the pushdown store affect the behavior of the automaton, therefore two cases must be considered: errors detected while the sub-machine to which the destination states of the error-recovery transitions belong, and the more complex case in which such state belongs to the calling machine. For further cases, we will adopt empirical recovery criteria.

In the first case, recovery may be locally done, since it considers only transitions internal to the sub-machine and independent of the pushdown store. In the second case, we need transitions from one sub-machine to another, conditioned to the pushdown store contents. In this case, we choose recovery transitions with destination states in the same sub-machine the top of the pushdown store refers to. Error recovery involving the current sub-machine and some other external one must provide the elimination of information previously stacked in the pushdown store, so that some reference to that sub-machine is found in the pushdown store, which is also popped out.

VI. Adaptive Error Recovery

Implementing error-recovery with adaptive automata may be achieved as follows: attach to each error transition one adaptive function, say E_1, which will perform structural transformations on the automata in order to absorb the error. This adaptive function will implement the error-recovery strategy described so far as an intrinsic part of the formalism. Some special transitions created by this adaptive function will call another adaptive function, E_2, which erases all transitions created by E_1. The adaptive solution also lowers space cost, since all transitions related to error-recovery, which should be replicated at all normal transitions in the classic solution are created and destroyed just as needed. Figure 3 illustrates adaptive functions E_1 and E_2.

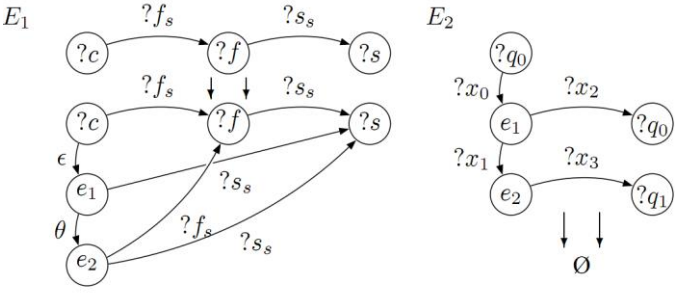

Figure 3. Adaptive Functions for Error Recovery

The algorithms presented in this paper were implemented with AdapTools 1.1[2], a software that offers a graphical environment where adaptive automata can be designed, implemented and tested. AdapTools embodies debugging and visualization tools, as well as a set of examples that may be executed by a special virtual machine included in the package. Among these examples is a compiler-compiler that produces a SPA-based syntactic acceptor, with the adaptive error-recovery described in this paper, from a Wirth notation grammar specification of the language. Figure 4 shows the AdapTools code that implements the example in Figure 2 using adaptive functions. E1 and E2 are the adaptive function shown in Figure 3. Using AdapTools we tested this automaton with different input string and could verify that it can recover from simple errors, growing in size only during the recovering process, returning to the initial size afterwards (due to the E2 adaptive function). Experiments comparing the adaptive and non-adaptive solutions were not conducted for this paper.

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	S	0	a	1	nop	nop	.E1(1,b,2)
2	S	1	b	2	nop	nop	E2.
3	S	2	c	0	fin	nop	nop
4	+E1	$\%_1$	a	9990	nop	[ERRO-Ins	nop
5	+E1	9990	$\%_2$	$\%_3$	nop	]	.E2
6	+E1	$\%_1$	eps	9990	nop	[Erro	nop
7	+E1	9990	eps	$\%_3$	nop	-Rem]	.E2
8	?E2	?x1	?x2	9990	?x3	?x4	?x5
9	?E2	9990	?y1	?y2	?y3	?y4	?y5
10	-E2	?x1	?x2	9990	?x3	?x4	?x5
11	-E2	9990	?y1	?y2	?y3	?y4	?y5

Figure 4. Code for adaptive automata with error recovery in AdapTools

VII. Conclusions and Future Work

In this paper, we presented a very practical approach to error-recovery, based on adaptive automata. This approach places the error-recovery scheme at the same formalization level of the subjacent structure, allowing error handling as a part of the machine. For instance, an adaptive automaton could be easily projected to dynamically turn the error-recovery procedures on and off, in response to adaptive actions.

The time and space complexity of our adaptive error-recovery approach is constant, both over the input string and the size of Q (states). The complexity depends on the transitions departing from a specific state, however, in practical problems, it does not lower the automata overall performance, since the size of the input alphabet is usually very small when compared to the size of the states set.

Some future experiments using adaptive error-recovery include the use of the adaptive automata self-transformation power to create a more sophisticated error-recovery mechanism. For instance, the error-recovery adaptive function may, interacting with the user or based on previous run information, detect the most likely corrections for specific errors, and change the structure permanently, so that further errors of the same kind will be automatically corrected. Another interesting research topic would be the extension of the adaptive error-recovery approach to deal with sequences of errors and the application of such extension to problems related to the edit-distance of two strings [7]. Error recovery is recently gaining attention from the computer vision community as some groups are reviving the syntactical pattern recognition approach augmented with recent advances in feature extraction techniques and more powerful machines [12, 13, 14].

[2] Freely available at https://code.google.com/p/adaptools/

References

[1] A. V. Aho and J. D. Ullman, “The Theory of Parsing, Translation and Compiling”, vol. 1 and 2. Prentice Hall, 1979.

[2] R. C. Backhouse, “Syntax of Programming Languages - Theory and Practice”. Prentice Hall, 1979.

[3] W. A. Barrett and J. D. Couch, “Compiler Construction - Theory and Practice”. SRA, 1979.

[4] K. J. Gough, “Syntax Analysis and Software Tools”. Addison-Wesley, 1988.

[5] R. Hunter, “The Design and Construction of Compilers”. John Wiley and Sons, 1981.

[6] S. Llorca and G. Pascual, “Compiladores - Teoría y Construcción”. Paraninfo, 1986.

[7] M. Mohri, “Edit-distance of weighted automata”. Conference on Implementation and Application of Automata - CIAA 2002 (July 2002), 7–29.

[8] J. J. Neto, “Adaptive automata for context-sensitive languages”. SIGPLAN NOTICES 29, 9 (September 1994), 115–124.

[9] J. P. Tremblay and P. G. Sorenson, “The Theory and Practice of Compiler Writing”. McGraw-Hill, 1985.

[10] W. M. Waite and G. Goos, “Compiler Construction”. Springer-Verlag, 1984.

[11] F. Almeida, J. Urquiza, Á. Velázquez, "Educational visualizations of syntax error recovery", 1st Annual IEEE Engineering Education Conference (EDUCON 2010), 2010, pp. 1019-1027.

[12] H. Pistori, P. Flach, A. Calway, “A new strategy for applying grammatical inference to image classification problems”, IEEE-ICIT International Conference on Industrial Technology (February 2013), 2013.

[13] B. Yao, X. Yang, L. Lin, M. Lee, and S. Zhu, “I2t: Image parsing to text description,” Proceedings of IEEE, vol. 98, no. 8, pp. 1485–1508, August 2010.

[14] R. Damaševičius, “Structural analysis of regulatory DNA sequences using grammar inference and support vector machine” Neurocomputing, vol. 73, pp. 633–638, January 2010

João José Neto holds a BS in Electrical Engineering (1971), Master in Electrical Engineering (1975), Ph.D. in Electrical Engineering (1980) and full professor (1993) from the Polytechnic School of the University of São Paulo. He is currently associate professor at the Polytechnic School of the University of São Paulo and coordinates LTA – Laboratory of Languages an Adaptive Technology of PCS – Department of Computer Engineering and Digital Systems EPUSP. Has experience in the area of Computer Science, with emphasis on Fundamentals of Computer Engineering, acting on the following topics: adaptive devices, adaptive technology, adaptive automata, and its applications to computer engineering, particularly in adaptive decision making systems, analysis and processing of natural languages, compiler construction, robotics, computer-assisted teaching, modeling of intelligent systems, machine learning processes and inferences based on adaptive technology.

Hemerson Pistori (Três Lagoas, MS, Brazil, 1970) coordinates the R&D&i computer vision group, INOVISAO, at the Dom Bosco Catholic University, UCDB, where he works since 1993. Professor Hemerson was a founder of the department of Computer Engineering at UCDB and has held the position of department head from 1998 to 2001. He also served as the chairperson of UCDB's scientific committee, as the director of research and since 2009 is the Dean of Research and Graduate Studies of this institution. He holds a permanent professor position at the Biotechnology graduate program and recently helped to implement a new graduate program on Environment Science and Agriculture Sustainability at UCDB. His BA and MA in Computer Science were obtained from the UFMS and UNICAMP universities, respectively, and the Ph.D. in Computer Engineering was held at USP. From 2011 to 2012 he stayed 6 months with the University of Bristol, UK, as a visiting researcher.

Amaury Antônio de Castro Junior is graduated in Computer Science at Federal University of Mato Grosso do Sul (1997), Masters in Computer Science at Federal University of Mato Grosso do Sul (2003) and Ph.D. at Polytechnic School of the University of Sao Paulo (2009). Currently holds the position of Regional Secretary of the SBC (Brazilian Computer Society), responsible for the state of Mato Grosso do Sul. Works since 2006 as associate professor at Federal University of Mato Grosso do Sul (UFMS) - Campus Ponta Pora (CPPP). Was the founder of Ponta Porã's Robotic Laboratory (LaRPP) and coordinates several projects of research and extension. Currently working with applied research on robotics and related areas. Has experience in Computer Science, with an emphasis Theory of Computing, acting on the following subjects: Adaptive Technologies, Adaptive Automata, Design of Programming Languages and Computer Model.

Marcelo Rafael Borth is Ph.D. candidate in Environmental Sciences and Agricultural Sustainability at Dom Bosco Catholic University - UCDB, masters in Computer Science at State University of Maringá - UEM and graduated in Information Systems at Paranaense University. Works as Professor at Federal Institute of Education, Science and Technology of Mato Grosso do Sul (IFMS - Ponta Porã). Awarded as outstanding student by SBC (Brazilian Computer Society) in 2006. Provided consultancy in software projects in Brazil and abroad. Has 3 Java certifications, OCJA, OCPJP and OCPWCD. Has experience in Computer Science, with emphasis on Pattern Recognition, Machine Learning, Semantic Web and Information Retrieval.