Construção e simulação de modelos baseados em autômatos adaptativos em linguagem funcional

Ricardo Luis de Azevedo da ROCHA

Escola Politécnica da Universidade de São Paulo
Av. Luciano Gualberto, trav. 3, n.158 Cidade Universitária
CEP 05508-900 – São Paulo – Brasil
e-mail: rlarocha@usp.br
e
Universidade São Marcos
Av. Nazaré 900, Ipiranga
CEP 048-100 – São Paulo – Brasil
e-mail: rrocha@smarcos.br

João JOSÉ NETO

Escola Politécnica da Universidade de São Paulo
Av. Luciano Gualberto, trav. 3, n.158 Cidade Universitária
CEP 05508-900 – São Paulo – Brasil
e-mail: jjneto@pcs.usp.br

Área de Aplicação: Modelos Computacionais, Autômatos, Linguagem Funcional, Engenharia de Software

RESUMO

Neste trabalho busca-se analisar a construção de um simulador de modelos de autômato adaptativo, utilizado como parte de um dispositivo computacional [4, 5] que implementa um método de busca de soluções. O simulador foi construído usando-se uma linguagem de programação funcional.

CONSTRUÇÃO E SIMULAÇÃO DE MODELOS BASEADOS EM AUTÔMATOS ADAPTATIVOS EM LINGUAGEM FUNCIONAL

Ricardo Luis de Azevedo da Rocha; João José Neto

Escola Politécnica da Universidade de São Paulo
Av. Luciano Gualberto, trav. 3, n.158 Cidade Universitária
CEP 05508 – São Paulo – Brasil
e-mail: rlarocha@usp.br ou rrocha@smarcos.br; jjneto@pcs.usp.br

RESUMO

Neste trabalho busca-se analisar a construção de um simulador de modelos de autômato adaptativo, utilizado como parte de um dispositivo computacional [4, 5] que implementa um método de busca de soluções. O simulador foi construído usando-se uma linguagem de programação funcional.

ABSTRACT

The present paper analyzes the process of building a simulator for adaptive automata. This process has been used as a part of a computational device that implements a solution-searching method described in [4,5]. The simulator has been built in a functional programming language.

1. INTRODUÇÃO

Um modelo de autômato representado como um formalismo adaptável introduz novas modificações sobre o conceito de autômato, dotando-o da capacidade de se automodificar conforme as necessidades encontradas, o que lhe concede o poder de representação da Máquina de Turing [1]. Este formalismo pode ser também compreendido como um complemento à clássica teoria de autômatos [1]. Assim, este formalismo apresenta-se como uma formulação hierárquica do caso mais geral do autômato.

Esta pesquisa busca encontrar meios de construir e simular modelos baseados em autômatos adaptativos, a partir de alguma especificação inicial. Para isso, busca-se prosseguir com o trabalho iniciado em [4, 5]. Assim este trabalho se inicia com o estudo das possíveis ferramentas a serem usadas para a formulação da solução, passando então para a construção do dispositivo propriamente dito. Neste estudo são contempladas diversas ferramentas e também a possibilidade de implementação física em uma máquina mais poderosa (possivelmente paralela) qualquer, através de um esquema de implementação adequado.

Para tanto, foram levadas em consideração as possíveis estruturas computacionais em termos de linguagens, sistemas operacionais e sistemas de apoio ao desenvolvimento, de forma a se poder escolher um conjunto de técnicas e ferramentas computacionais de análise e programação convenientes, tais que venham a facilitar e agilizar o desenvolvimento do dispositivo proposto.

O estudo realizado levanta as características de cada ferramenta, levando em consideração a necessidade de se garantir a implementação do dispositivo em qualquer máquina, sem grandes alterações nos programas gerados.

O trabalho se encerra com o desenvolvimento de um protótipo para o dispositivo.

2. Formalismo adaptativo

Um formalismo adaptativo, apresentado em termos de autômatos, pode ser visto como uma máquina de estados representada por um grafo orientado inicial, que inicialmente se apresenta como um autômato finito ou de pilha, mas que, à medida que lhe são impostas alterações durante sua operação (alterações que são geradas pela própria movimentação do autômato e que resultam da aplicação de suas regras de transição), o grafo inicial pode ser modificado.
Com isso, estados e transições podem ser eliminados ou acrescentados ao modelo de autômato, em decorrência de cada um dos passos executados pelo mesmo durante sua operação.
Durante a execução de uma transição adaptativa, o autômato pode sofrer mudanças em sua própria topologia. Isto faz com que uma nova máquina de estados substitua a anterior, caracterizando, para o autômato, o percurso de um passo adicional, em uma trajetória no espaço das máquinas de estado.
A trajetória em um espaço de máquinas de estado é gerada porque a máquina inicial não está mais presente, portanto, o autômato começa seu trabalho em uma máquina específica e vai continuá-lo em outra máquina, descrevendo uma trajetória em um espaço de trabalho que contempla todas as configurações de autômato possíveis.

Autômatos adaptativos

O modelo de autômato adaptativo representa um caso especial de formalismo adaptativo [1], que serve como base para esta pesquisa.
Um autômato adaptativo M é constituído de:

- ω - Fita de entrada
- E_0 - Autômato inicial que implementa M
- E_m - Autômato final que implementa M após a aceitação da cadeia de entrada $\omega = \alpha_0\alpha_1 ... \alpha_m$ $(m \geq 0)$, onde cada α_i, $0 \leq i \leq m$, representa uma sub-cadeia de ω
- E_i - Autômato após a $i^{ésima}$ transição adaptativa, $0 \leq i \leq m$

Na aceitação de uma cadeia de entrada ω, composta de sub-cadeias α_k, $0 \leq k \leq m$, a operação do autômato M pode ser visualizada macroscopicamente como uma seqüência de reconhecimentos das sub-cadeias α_i pelas sub-máquinas correspondentes E_i $(0 \leq i \leq m)$.
Assim, M descreve um caminho $< E_0, \alpha_0 > \rightarrow < E_1, \alpha_1 > \rightarrow ... \rightarrow < E_m, \alpha_m >$, em que cada elemento $< E_i, \alpha_i >$ representa a aceitação de α_i pela sub-máquina correspondente E_i e a disposição de elementos $< E_i, \alpha_i >$, um após o outro, denota a seqüência de execução das transições adaptativas [1]. Uma transição adaptativa pode conter ações adaptativas de adição, remoção ou inspeção de transições.
Os tipos possíveis de transições são os seguintes:

- Transições entre submáquinas: Chamada/retorno de submáquina, são transições, executadas sem consumo de símbolos da cadeia de entrada, que permitem transferir o controle entre uma submáquina e outra;
- Transições internas: São transições que ocorrem dentro da mesma submáquina, com ou sem consumo de símbolos da cadeia de entrada;
- Transição em vazio: São transições internas sem consumo de símbolos;
- Transições adaptativas: São quaisquer das transições anteriores, às quais estejam associadas ações adaptativas especificando a alteração da do conjunto de transições do autômato adaptativo.

Uma transição adaptativa é descrita por uma quádrupla [1] $P_i = (t_i, A_i, u_i, B_i)$, em que:

- t_i - representa a situação do autômato, anterior à transição
- A_i - representa a ação adaptativa a ser executada antes da transição
- u_i - representa a situação do autômato, posterior à transição
- B_i - representa a ação adaptativa a ser executada depois da transição.

Operação do autômato adaptativo

A operação do autômato adaptativo dá-se através de sucessivas transformações em sua topologia, de forma que, partindo da topologia inicial, o autômato adaptativo pode sofrer modificações de ampliação ou de redução em seu conjunto de estados e transições iniciais. Estas transformações são provocadas pela execução das transições adaptativas.
O autômato adaptativo pode exibir um comportamento similar ao autômato finito ou de pilha estruturado, quando os problemas apresentados puderem ser representados e tratados por estes, isto é, para o autômato adaptativo a utilização de transições adaptativas não é compulsória. Entretanto, existe a possibilidade de aparecimento, na cadeia de entrada, de uma construção que não possa ser tratada pelos autômatos finitos ou de pilha, como é o caso, por exemplo, de uma construção dependente de contexto (no caso de reconhecedores de linguagens dependentes de contexto). Neste caso, o autômato adaptativo é capaz de resolver o problema, com o auxílio do recurso da alteração dos seus próprios conjuntos de estados e de regras de transição, de forma que o autômato modificado seja capaz de reconhecer a construção encontrada [1].
Algumas ações básicas podem ser executadas no autômato adaptativo, tais como, por exemplo, eliminação e inserção de transições, modificando assim o comportamento do autômato. Outra ação básica é a de inspeção, que permite obter a característica de um elemento específico na topologia corrente do autômato (após sucessivas modificações na topologia, pode ser necessário verificar uma determinada característica, que pode se encontrar diferente em relação à topologia inicial). Maiores detalhes podem ser encontrados em [1] e [10].

3. Possibilidades de Escolha para a implementação do Dispositivo

Para a concretização do dispositivo poder-se-ia optar por uma linguagem utilizável em quaisquer ambientes (inclusive sistemas de grande porte e sistemas paralelos) como “ADA”, “C” paralelo, “Fortran” de alta performance e outras [6], ou seguir a melhor escolha para fazer frente à implementação, isto é, escolher uma linguagem com um alto nível de abstração, facilitando a construção do protótipo, como, por exemplo, LISP, Prolog, etc., e posteriormente, se for o caso, transportar a implementação para outra linguagem.
Partindo da segunda opção, há ainda duas ferramentas que exploram o conceito de adaptatividade, inerente aos autômatos adaptativos: a STAD [7] e a RSW [9].
A escolha final foi utilizar uma linguagem de programação funcional bastante difundida na comunidade científica da área da Inteligência Artificial, que é a linguagem “LISP”. O uso desta linguagem permite o desenvolvimento mais rápido dos programas, já que seu nível de abstração é mais elevado, e opera naturalmente com processamento simbólico. Entretanto, o uso de um compilador para a linguagem “LISP” pura pode não ser tão vantajoso neste trabalho, já que não se teriam as facilidades de depuração, de geração de elementos gráficos, que se encontram em compiladores mais novos, os quais introduziram funções para orientação a objeto e uso de interface gráfica.

3.1. Ferramentas para implementação

Após a escolha da linguagem de programação, partiu-se para a escolha do ambiente adequado, que integrasse as ferramentas de codificação, teste, documentação, etc., do projeto. Novamente

há diversas opções, comerciais ou livres, para cada tipo de sistema e equipamento que se pretenda utilizar. Como a opção para o equipamento foi seguir a linha de computadores pessoais, a de sistema operacional seguiu a mesma tendência, optando-se pelo sistema operacional gráfico mais difundido para o padrão (MS-Windows 98).
Outra possibilidade é a de se integrar ambientes diversos, como por exemplo, uma ferramenta de análise e projeto com uma ferramenta de programação, etc. Porém, como o objetivo da implementação é desenvolver um protótipo, optou-se por um único ambiente de programação, que permita gerar programas rapidamente e que possua capacidade gráfica.
Foi então efetuada uma busca na Internet para descobrir um ambiente de programação adequado. Alguns destes ambientes encontrados foram listados e estudados, antes de se optar por algum deles.

3.1.1. Ambiente de Programação usado na Implementação

Foram encontrados três ambientes gráficos baseados em LISP, disponíveis na Internet: Allegro CL Lite, versão 5.0 para Windows 95-98 (Pode ser encontrada no endereço eletrônico http://www.franz.com/, no qual há ainda um tutorial e manuais de apoio. Existe uma versão profissional, porém, é comercializada, não é livre); XLISP 2.1 (ambiente e compilador produzido para Windows 3.1 pelo laboratório de Inteligência Artificial, grupo de Linguagem Natural da New York University, e pode ser encontrada no endereço eletrônico http://www.nyu.edu/pages/linguistics/ling.html); e o LispWorks Personal Edition (versão para Windows 95, da empresa Harlequin, Inc., pode ser encontrada no endereço eletrônico http://harlequin.com/products/ads/lisp/).
A escolha do ambiente LISP mais propício deu-se com base nas seguintes características, encontradas Allegro CL Lite:
a) Ser mais recente e moderno (especialmente construído para Windows 95-98);
b) Incorporar o ambiente CLOS (“common LISP object system”);
c) Ser totalmente gráfico.

O trabalho neste ambiente assemelha-se a trabalhar em um ambiente como o Delphi, ou o C++ Builder. Com isso, a escolha deste ambiente se mostrou mais indicada para o dispositivo proposto, em relação às outras opções.

4. Construção do dispositivo

O dispositivo foi construído de forma a que seus componentes se comportem da seguinte maneira:
a) Há uma função de interpretação de autômato previamente codificada, que opera as construções de autômato (modelos) como uma máquina universal, executando os programas e dados que descrevem o comportamento de um modelo de autômato particular;
b) Os modelos de autômato vão sendo construídos e modificados à medida que o dispositivo computacional vai desenrolando seu processamento. Assim, a partir de uma entrada específica, os modelos são gerados para serem posteriormente interpretados. Este procedimento de geração–modificação e interpretação é realizado até que se obtenha uma resposta.

Para que o dispositivo possa efetuar seu processamento, é necessário que receba as informações que lhe permitirão gerar os modelos de autômato. Desta maneira, o primeiro elemento estruturado é a entrada, depois o controlador com suas funções, e por último o elemento de saída.

4.1.1. Análise do Dispositivo

Ao analisar o dispositivo proposto, é possível, por simples inspeção, definir cinco classes funcionais fundamentais [4]:

- a classe de entradas;
- a classe do controlador;
- a classe do interpretador de modelos de autômato;
- a classe dos modelos de autômato; e
- a classe das saídas.

A classe do interpretador de modelos de autômato possui como característica fundamental implementar um algoritmo de controle da topologia do modelo interpretado, para garantir a consistência e a coerência dos modelos e controlar o término da execução dos mesmos seja por finalização natural – computação terminada, ou por finalização abortada (computação parada pelo controlador por exceder um número limite de passos estabelecido).

Para este trabalho a classe que realmente importa é a classe do interpretador de modelos de autômato adaptativo, entretanto, maiores detalhes sobre as demais classes podem ser encontrados em [4].

4.1.1.1. Definição e Geração dos Modelos

Um modelo de autômato, para o dispositivo, é representado por uma lista composta de transições organizadas também como listas. Cada uma das transições constitui-se de um símbolo a ser consumido, um estado de partida, um estado de destino. Cada transição pode ainda conter, após o estado de destino, uma lista na qual há a descrição de uma função adaptativa [composta de ações adaptativas simples, dispostas em forma de lista, como, por exemplo: ((+ (a 1 2) (- (a 3 2))))].
Portanto, um modelo de autômato é composto por uma estrutura em forma de lista de transições. Através destas transições, o dispositivo pode avaliar e simular os modelos gerados. Como os modelos serão simulados, o dispositivo poderá gerá-los sem a necessidade de verificar se em todos os estados há transições previstas para todos os símbolos. Com isso, se durante a simulação, algum estado alcançado não tiver uma transição para o símbolo corrente, este será considerado rejeitado, encerrando o processamento do modelo. Ao final, havendo a necessidade de se completar o modelo, a tarefa somente será executada no(s) modelo(s) que satisfizer(em) a todas as condições impostas (passado nos testes e com a menor complexidade dentre os demais).
Para todos os modelos gerados, o estado inicial é o primeiro da lista de transições. Com essas simplificações na descrição dos modelos, o número de transições geradas diminui consideravelmente, como também o espaço ocupado pelos modelos. Para completar a representação do modelo de autômato, resta acrescentar dois elementos à estrutura que o descreve, que permitirão sua simulação em outra etapa:

a) Lista de transições criadas por funções adaptativas (criadas durante o processo de simulação);
b) Estado corrente de execução.

Voltando à etapa de geração, os modelos são gerados seguindo as regras de produção recebidas pelo elemento de entrada. Considerando que as regras de produção especificadas podem ser de três tipos, três são as ações possíveis à função de geração:

a) Gerar uma transição para consumo de cada símbolo.
b) Gerar uma transição entre o estado corrente e o estado de origem (partida) de uma regra recursiva, consumindo um símbolo terminal.
c) Gerar uma função adaptativa, caso haja alguma construção dependente de contexto.

Estas ações são tomadas à medida que as regras forem encontradas, durante o processo de geração. Ao final do processo, tem-se uma lista contendo todas as transições geradas. Estas transições compõem um modelo inicial, que deve, posteriormente, ser trabalhado por uma outra função de geração, para se poder encontrar os modelos de solução.

A primeira ação tomada pelo controlador, ao receber a lista de transições, é efetuar uma nova chamada de função, desta vez para ordená-la e organiza-la. A função chamada devolve a lista de transições, organizada por estado e ordenada.
Com essa lista, o controlador pode proceder à geração dos modelos e, ao mesmo tempo, efetuar uma combinação entre as transições. Já que dispõe de todas as transições organizadas e ordenadas, o controlador efetua nova chamada de função, para que seja realizada uma combinação entre as transições recebidas, gerando o produto cartesiano das transições existentes. Assim, diversos modelos diferentes são gerados. Após esta combinação, o controlador elimina da lista de transições, em cada modelo gerado, as transições que partem de estados não-alcançáveis pelas demais transições.

4.1.1.2. Execução dos Modelos

Usando os modelos gerados pelas funções anteriores, o controlador efetua, então, uma chamada à rotina de simulação de casos válidos. Esta rotina recebe a lista de modelos, a lista de casos de treinamento, e trabalha efetuando, para cada caso de treinamento, uma simulação de cada um dos modelos em ordem lexicográfica. Isto é, para cada símbolo de uma cadeia de treinamento a ser exercitada, a rotina efetua uma repetição, passando por todos os modelos, na qual tenta executar uma transição que seja capaz de consumir o símbolo corrente da cadeia.
A execução de uma transição é, na realidade, uma busca dentro do modelo em execução, por uma transição que, partindo do estado corrente do modelo (armazenado), seja capaz de consumir o símbolo corrente da cadeia recebida (esta transição pode estar na lista de transições criadas por ações adaptativas). Se o símbolo puder ser consumido por um modelo, a função de simulação troca o valor do estado corrente deste modelo para o valor correspondente ao próximo estado da transição. Caso o símbolo não possa ser consumido, a função pára a execução deste modelo, indicando que este falhou, através da mudança do valor de seu estado corrente para vazio (nil).
Caso uma função adaptativa seja encontrada, esta pode conter as ações básicas de inspeção, adição e remoção de transições. Neste caso, o simulador trabalha diretamente com a lista de transições adaptativas de cada modelo, realizando as ações designadas pela função, isto é, acrescentando transições à lista de transições adaptativas, ou removendo transições (que estão na lista ou no modelo), ou então inspecionando o modelo.
Através destas especificações, a função de simulação consegue exercitar todos os modelos e garantir que não haverá falhas de execução, porque o dispositivo finaliza um modelo quando este não é capaz de realizar uma transição. Com isso, a lista de modelos, ao final da simulação, pode ser muito menor do que a lista inicial recebida.
Ao reduzir a lista de modelos gerados, na qual somente estão presentes os modelos que efetivamente foram aprovados nos testes, a função de simulação restringiu as opções de escolha dentre os modelos, facilitando com isso a tarefa da função que elegerá o modelo mais apto.

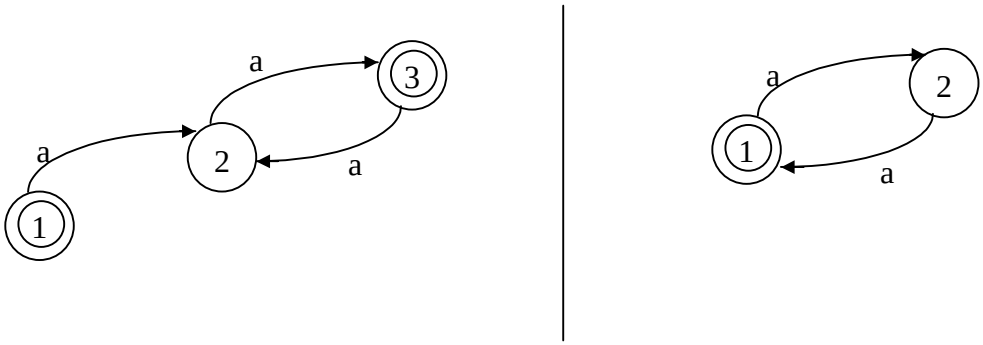
5. Execução do Simulador

Qualquer problema a ser solucionado pelo dispositivo necessita de um conjunto de informações inicial, que lhe sirva como entrada. Conforme mencionado anteriormente estas informações são passadas, nesta implementação experimental, em forma de especificação de linguagem.

5.1.1. Exemplo de linguagem regular

Para ilustrar o funcionamento do dispositivo, o exemplo a seguir propõe a geração de um modelo de autômato mínimo para o seguinte problema de linguagens regulares: gerar um modelo para reconhecer a linguagem (a a)*, ou seja, o conjunto de cadeias formado de um número par (eventualmente nulo) de símbolos “a”. A utilização do dispositivo tem início com a especificação da entrada [4, 5].

Neste exemplo tem-se a linguagem regular composta por quantidades pares de elementos “a”, $L = (a^2)^*$, ou usando-se a notação adotada L = (a a)*, cuja especificação fica: Símbolo inicial: s; Lista de Símbolos Terminais: (a); Lista de Símbolos Não-Terminais: (s aa a*); Lista de Produções: ((s -> a*) (a* -> () (aa aa a*)) (aa -> a)).
A segunda regra da lista de regras de produção: (a* -> () (aa aa a*)) é interpretada como um “ou” entre o símbolo de regra vazia “()” e a lista (aa aa a*).
O dispositivo solicita novas informações a serem passadas, assim, uma tela solicitando casos para treinamento ao usuário aparece e deve ser preenchida. Os casos de teste (ou de treinamento), que permitirão exercitar os modelos aptos e, a partir do exercício, determinar quais são o(s) mais apto(s), o(s) melhor(es). Utilizando o mesmo exemplo anterior, as listas são do tipo: Casos válidos: ((a a) (a a a a)); Casos não-válidos: ((a) (a a a)).
Os modelos gerados neste caso são os seguintes:



A partir destes modelos – no caso ambos estão corretos – o sistema simula-os e gera uma lista de modelos válidos. Como neste exemplo não há mudança em relação à lista inicial, a solução somente aparece quando se busca o modelo de menor complexidade através da função de comparação de complexidades. Neste caso, segundo o critério adotado para a função de medição, o valor de complexidade para o primeiro modelo é seis, e o valor de complexidade para o segundo modelo é quatro. Assim sendo, o segundo modelo é o melhor.

5.1.2. Exemplo de linguagem dependente de contexto

De forma idêntica à anterior podem ser consideradas as linguagens livres de contexto. Já no caso das dependentes de contexto são utilizadas funções adaptativas.
Para simplificar o exemplo a ser apresentado, aqui se estará tratando uma linguagem livre de contexto como dependente de contexto, através da alteração de parte do código da função de entrada. Desta maneira, pode-se representar uma linguagem do tipo a^nb^n como se fosse uma linguagem dependente de contexto.
Procedendo assim, deve-se entrar com os dados de entrada da seguinte forma: ((s -> a*) (a* -> () (aa a* bb)) (aa -> a) (bb -> b)). O dispositivo trata uma produção deste tipo através da criação de transições adaptativas. No caso exemplificado, ao se dar entrada na produção a*, através da regra aa, o dispositivo gera uma função adaptativa que, quando executada, gera uma nova transição que conecta o estado corrente a ele mesmo, consumindo o símbolo indicado pela regra aa. Ao sair da produção a*, através da regra bb, o dispositivo gera uma função adaptativa que, quando executada, elimina uma transição que conecta o estado anterior a ele mesmo, e consome o símbolo indicado pela regra bb. A seqüência para o dispositivo fica:

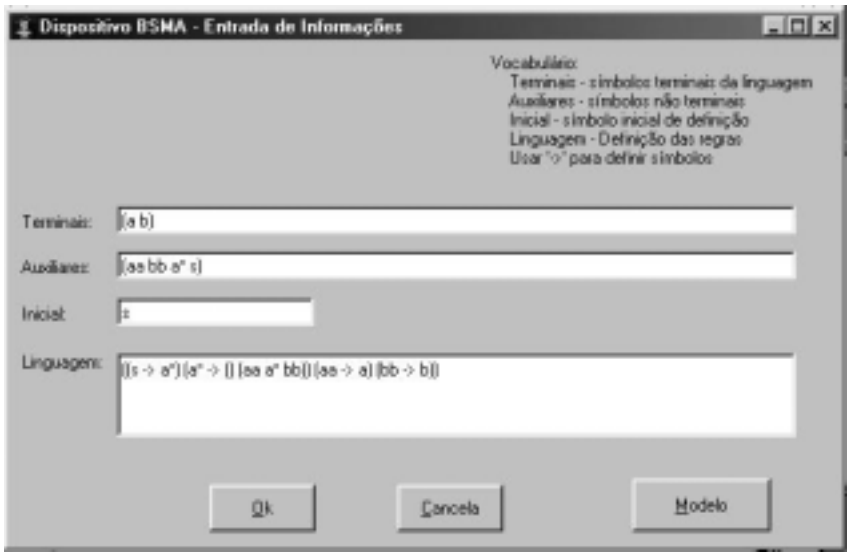


Figura 1 - Tela de Entrada

Após a entrada dos dados o sistema solicita os casos de treinamento, que, neste caso, são do tipo: Casos válidos ((a b) (a a b b)); Casos não-válidos ((a) (b) (a a b)).

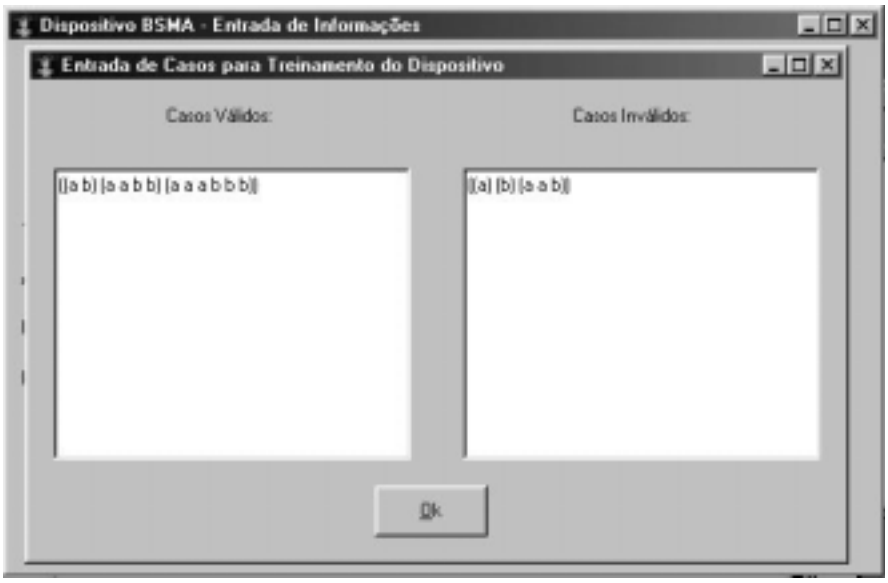


Figura 2 – Entrada de casos de treinamento

Partindo, então, da especificação da linguagem e dos casos de treinamento, o dispositivo gera uma lista de transições que inicia o processo. As informações são todas internas e o usuário não tem acesso. A ferramenta de programação utilizada permite a verificação das variáveis internas, através da janela de rastreamento a seguir:

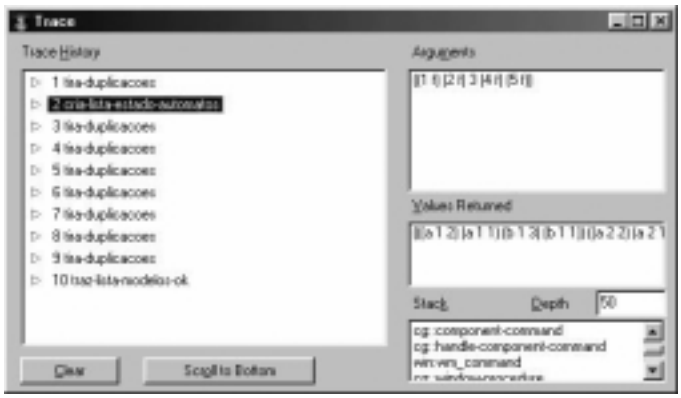


Figura 1 - Janela de Rastreamento

Os modelos podem ser observados através da seguinte janela (inacessível ao usuário):

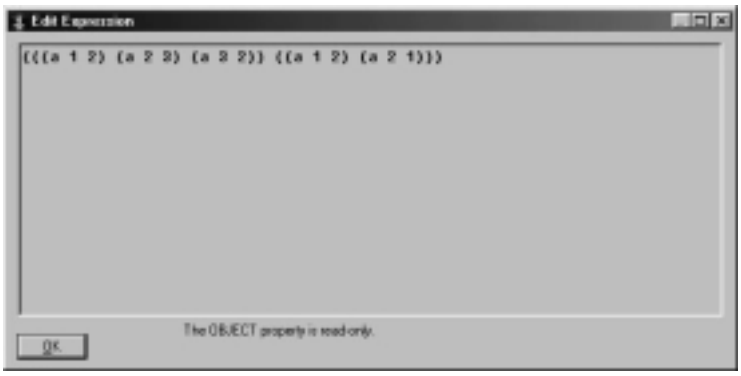


Figura 2 - Modelos gerados para simulação

O sistema então simula os modelos e gera uma saída, com os modelos válidos. A resposta é enviada ao usuário da seguinte forma:

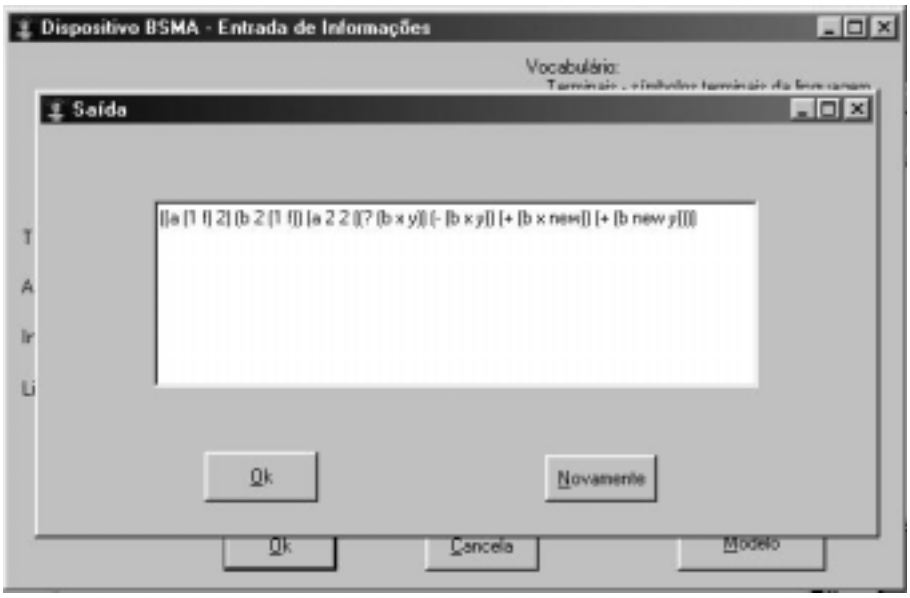
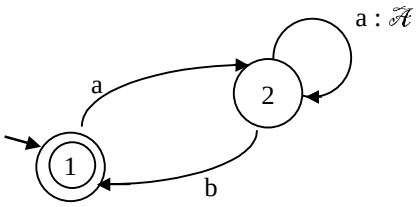


Figura 3 – Saída gerada pelo dispositivo

Assim tem-se o seguinte modelo:



A função adaptativa $\mathscr{A}$ executa basicamente quatro ações adaptativas, a primeira de inspeção, identificando a transição que consome o símbolo b e conecta um estado a ser determinado ao estado final (1). Após a identificação, a função adaptativa elimina esta transição, e em seu lugar repõe duas outras, a primeira conectando o estado determinado na ação de inspeção a um novo estado (gerado), e a segunda conectando o estado gerado ao estado final (1), ambas consomem o símbolo b. Assim, a cada execução da função adaptativa $\mathscr{A}$ um novo estado intermediário (não final) é gerado, e passa a haver uma transição a mais no modelo em execução.

6. Considerações sobre a pesquisa

Os experimentos realizados foram simples, com baixo número de estados na maior parte dos casos, e com baixa complexidade. Porém, o objetivo foi exatamente mostrar a viabilidade de geração de soluções e o custo associado a essa tarefa. Desta maneira, acredita-se que o objetivo foi cumprido, tendo sido obtidos alguns resultados que possibilitam tecer alguns comentários finais.

6.1.1. Resultados

Observou-se que, mesmo com um número menor de modelos gerados, a aplicação da medida de complexidade como fator direcionador para encontrar a melhor resposta mostrou-se válida. Ainda que se considere um caso ligeiramente mais complexo, como por exemplo, uma linguagem regular composta por (ab ∪ ba)*. Nestes casos, o total de modelos é maior, e se fosse necessário limitar o total de modelos gerados e simulados, mesmo assim os melhores modelos estariam presentes.

Para o dispositivo a forma utilizada para limitar a quantidade de modelos é a escolha baseada na medida de complexidade. Nos casos estudados, ao limitar o espaço de busca, ainda assim, no espaço remanescente, os melhores estavam presentes. Isto não quer dizer que com qualquer quantidade de modelos este resultado seja válido, porém, indica que esta medida de complexidade é um elemento direcionador e seu uso pelo dispositivo auxilia a melhor escolha, mesmo que se necessite eliminar alguns modelos por falta de espaço, ou limitar uma explosão combinatória.

O mapeamento das produções de uma linguagem em um modelo de autômato não-determinístico é realizado levando-se em consideração as descrições existentes, ou seja, através de um conjunto de definições. As transformações realizadas nos modelos gerados são produzidas por combinação (produto cartesiano).

6.1.2. Comentários Finais

O uso do autômato adaptativo para representação de problemas difíceis [11] já mostrou ser adequado. Além disso, esta pesquisa ilustra uma representação do autômato adaptativo, em linguagem funcional, construída de forma simples e, ainda assim, propiciou a execução de um experimento que lida com linguagens dependentes de contexto. Este experimento somente foi possível pela maneira com que o autômato adaptativo é capaz de lidar com estas construções.

A construção do protótipo em uma linguagem funcional como a linguagem LISP, que permite maior abstração, tornou a tarefa de codificação menos penosa, e, ao mesmo tempo, propiciou um melhor acompanhamento do processo.
Pode-se ainda tecer um comentário final sobre o ambiente de desenvolvimento utilizado, que, além de funcional é também gráfico e orientado a objeto. Ao unir as características de programação funcional com orientação a objeto em um ambiente gráfico, o ambiente permite a geração de produtos mais atraentes e mais confiáveis.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[1] JOSÉ NETO, J. Adaptive automata for context-dependent languages. *ACM SIGPLAN Notices*, v. 29, n. 9, p. 115-124, Sep. 1994.

[2] LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elements of the theory of computation*. New Jersey, Prentice-Hall, Inc, 1998.

[3] LI, M; VITÁNYI, P. *An introduction to Kolmogorov complexity and its applications*. 2nd. ed., New York, Springer-Verlag, 1997.

[4] ROCHA, R. L. A. *Um método de escolha automática de soluções usando tecnologia adaptativa*. São Paulo, 2000. 211p. Tese (Doutorado) – Escola Politécnica, Universidade de São Paulo.

[5] ROCHA, R. L. A.; JOSÉ NETO, J. , Uma proposta de método adaptativo para a seleção automática de soluções, *Proceedings of the VI International Congress on Information Engineering – ICIEY2K*, Buenos Aires, Argentina, 2000, [CD-ROM].

[6] BEN-ARI, M. *Principles of concurrent and distributed programming*. Cambridge (UK), Prentice-Hall International, 1990.

[7] ALMEIDA JÚNIOR, J. R. de. *STAD - Uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos*. São Paulo, 1995. 202p. Tese (Doutoramento) - Escola Politécnica, Universidade de São Paulo.

[8] CHAITIN, G. J. The limits of mathematics. *Journal of Universal Computer Science*, 2, n. 5, p. 270-305, 1996.

[9] PEREIRA, J. C. D.; JOSÉ NETO, J. Um ambiente de desenvolvimento de reconhecedores sintáticos baseados em autômatos adaptativos. Apresentado no *SBLP´97, II Simpósio Brasileiro em Linguagens de Programação*, Instituto de Computação, Universidade Estadual de Campinas, 3-5, p. 139-150, Set., 1997.

[10] JOSÉ NETO, J. *Contribuições à metodologia de construção de compiladores*. São Paulo, 1993. 272p. Tese (Livre-Docência) - Escola Politécnica - Universidade de São Paulo.

[11] JOSÉ NETO, J. Solving Complex Problems Efficiently with Adaptive Automata - paper submitted to the *5th International Conference on Implementation and Application of Automata - CIAA 2000* - London - Ontario - Canada - July 24-26, 2000.

MINI-CURRÍCULOS DOS AUTORES:

Ricardo Luis de Azevedo da Rocha

Formado em Eng. Elétrica – modalidade Eletrônica pela PUC-RJ em 1982.
Mestre em Eng. Elétrica – Sistemas Digitais - EPUSP - 1995
Doutor em Eng. Elétrica – Computação e Sistemas Digitais - EPUSP – 2000
Áreas de interesse: Autômatos e Linguagens formais, Engenharia de Software, Tecnologia Adaptativa, Inteligência Artificial, Métricas de Software, Teoria da Computação, Complexidade.
Pesquisa Corrente em: metodologias para aplicação de tecnologias adaptativas à engenharia de software
Atividade Didática: Prof. Universitário de Engenharia de Software, Autômatos e linguagens formais, Compiladores e Teoria da Computação.

João José Neto

Formado em Eng. Elétrica – modalidade Eletrônica pela Escola Politécnica da USP em 1971.
Mestre em Eng. Elétrica – Escola Politécnica da USP – 1975
Doutor em Eng. Elétrica – Escola Politécnica da USP – 1980
Livre Docente do Departamento de Engenharia de Computação e Sistemas Digitais - Escola Politécnica da USP – 1993.
Atividade Didática: Professor do curso de graduação em engenharia de computação da Escola Politécnica da USP: Projeto e Técnicas de Construção de Compiladores, Software Básico, Sistemas Operacionais.
Atividade de Pós-graduação: Professor e Orientador do programa de Pós-graduação em engenharia elétrica, área de sistemas digitais: Teoria da Computação, Autômatos e Linguagens Formais, Linguagens de Programação.
Pesquisa Corrente em: metodologias para aplicação de tecnologias adaptativas em Engenharia de Computação.
Áreas de interesse: Teoria da Computação, Autômatos e Linguagens Formais, Linguagens de Programação, Software Básico, Construção de Compiladores, Ferramentas de Auxílio à Compilação, Tecnologia Adaptativa, Engenharia de Software.