Biblioteca para autômatos adaptativos com regras de transição armazenadas em objetos feita em C++

N. L. Werneck

***Abstract*—We created a library with the C++ language to simulate adaptive automata. Some implementations use a table to store the transition rules and adaptive actions to be performed. In our system, the informations relative to each state reside in separate objects in the program. The execution does not happen in a loop that analyzes the same transitions table repeatedly, but rather trough nested calls to methods of the different objects. We used the library to create an interpreter for a language that describes adaptive finite automata, and executed some examples.**

***Index Terms*—Autômatos finitos adaptativos (adaptive finite automata), programação orientada a objetos (object oriented programming)**

I. Introdução

NESTE artigo apresentaremos uma biblioteca desenvolvida na linguagem C++ capaz de simular o funcionamento de autômatos finitos adaptativos. Buscamos fazer uma modelagem orientada a objetos adequada das entidades envolvidas. Uma importante característica da arquitetura implementada é que optamos por uma forma um pouco menos usual de simulação de autômato finito, utilizando chamadas de subrotinas ao invés de uma tabela de transição.

Criamos ainda um simulador de autômatos finitos com esta biblioteca. Ele funciona a partir da descrição de uma máquina realizada com a linguagem SDMBA, definida por nós. Esta linguagem reflete as peculiaridades e nossa arquitetura, e acreditamos ser adequada para autômatos mais simples.

II. Teoria

A. *Arquiteturas para simulação de autômatos*

A aplicação da adaptatividade [1] a autômatos finitos (AF) dá origem aos autômatos finitos adaptativos (AFA) [2], ou autômatos finitos auto-modificáveis [3]. Estes autômatos possuem um poder computacional maior do que o dos AFs, porém preservam a mesma estrutura. Isto é atraente devido à adequação dos AFs a determinados problemas.

Existem duas formas básicas de se implementar AFs em programas de computador. Uma forma bastante popular é utilizar uma tabela que armazena todas as regras de transição. Um registrador guarda o estado em que a máquina se encontra. O programa possui um laço que modifica o registrador de acordo com seu valor, o símbolo na entrada, e as regras especificadas na tabela. Ao tornarmos adaptativo um sistema deste tipo, o controlador torna-se capaz de modificar também a tabela de transições de acordo com o estado da máquina a cada instante, seguindo as funções adaptativas especificadas. A Figura 1 ilustra este sistema.

Outra forma de se implementar um AF é escrever um programa onde existe uma efetiva passagem do controle da execução do programa entre diferentes subrotinas, que representam os estados da máquina. O primeiro tipo de implementação é geralmente considerado como mais próximo do paradigma de programação estruturada. Este segundo método, entretanto, possui algumas características que desejamos estudar. Criamos portanto um modelo orientado a objetos para AFAs inspirado neste modelo. Nossa implementação não utiliza uma tabela geral de transições acessada dentro de um laço, mas sim objetos que representam os estados do autômato, e que se alternam seqüencialmente no controle da execução do programa.

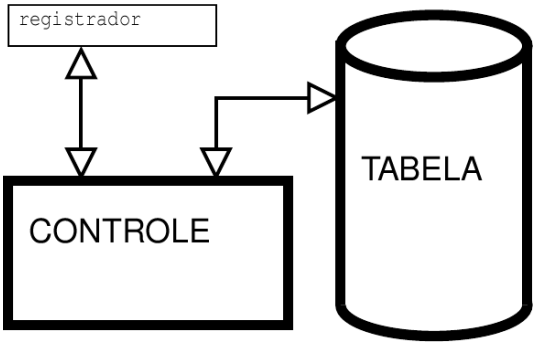

Fig. 1. Simulação de um AFA com registrador e tabela de transições.

B. *Justificativas*

Apesar de arquitetura com tabela funcionar, e possuir valor didático, ela se afasta da estrutura dos AFs. Máquinas de estados são modulares, com nodos e vértices. Não podemos ver isto bem ao misturar todas as transições numa só entidade.

Mecanismos com estruturas conexionistas mais poderosos do que AFs já foram abordados por diversos pesquisadores, incluíndo Shannon, e o próprio Turing [4]. O conexionismo tornou-se um tema de pesquisa especialmente popular nos anos 1980 [4,5,6], mas permanece um pouco sub-

Este trabalho foi financiado pela CAPES.

N. L. Werneck é aluno de doutorado em Engenharia Elétrica na EPUSP, onde integra o Laboratório de Técnicas Inteligentes (LTI) do PCS (telefone: 55-11-3091-5397; e-mail: nwerneck@gmail.com).

utilizado fora do meio acadêmico, sendo ainda algo mais comum como tema de pesquisa do que como ferramenta.

Queremos evidenciar a natureza conexionista dos autômatos adaptativos, buscando uma implementação descentralizada onde os nodos possuem contato apenas com seus vizinhos. Pretendemos assim explorar a possibilidade dos AFAs poderem às vezes atuar como uma "ponte" entre mecanismos conexionistas e monolíticos, devido justamente a estas duas formas alternativas de implementação desta entidade em programas de computador.

Além do conexionismo, queremos aproximar nossa implementação de outra das inspirações dos AFAs: os programas auto-modificáveis.

Programas auto-modificáveis são aqueles que modificam a própria área de memória em que se encontram, explorando a memória compartilhada da arquitetura de von Neumann [8]. Esta técnica de programação tornou-se um pouco marginalizada após o desenvolvimento de computadores melhores e de linguagens de programação mais sofisticadas, além da preocupação com o problema prático da segurança.

Por poderem às vezes ser difíceis de analisar, programas auto-modificáveis conflitam com alguns conceitos da ciência da computação, como a programação estruturada. Este paradigma é defendido na famosa carta de Dijkstra [9], que fala sobre as limitações cognitivas de humanos para compreender programas. Estas limitações justificariam o uso de linguagens com estruturas bem-definidas, o que permitiria uma melhor compreensão dos programas criados com elas.

Algumas pessoas definem a programação estruturada apenas como sendo a eliminação da instrução `go to` de programas e linguagens. Mas esta é uma visão muito superficial do problema, que não beneficia a discussão elaborada por Dijkstra, e ignora a existência de algumas interessantes estruturas que são bem descritas utilizando-se o `go to`. Nosso desejo é investigar as estruturas oferecidas pelo formalismo de AFAs, e determinar em que condições elas podem beneficiar a descrição de programas, e qual seu impacto na eficiência. Knuth menciona em um artigo [10] programas simples onde o `go to` é utilizado de uma forma benéfica e insubstituível. Queremos averiguar se não existem programas auto-modificáveis igualmente interessantes, utilizando AFAs como ponto de partida.

Por outro lado, temos também interesse no poder de expressão das AFAs, ou seja, em conhecer que sistemas são expressos de forma simples por AFAs, por mais que estas estruturas simples possam ser difíceis de se compreender.

Ao questionar políticas simplórias de programação, como a eliminação do `go to`, e ao procurar estudar estruturas de computação sem concepções rígidas sobre o que é fácil ou não de se compreender, nosso trabalho ainda se alinha ao desejo de outros programadores que compõem uma minoria [11] que busca questionar certos dogmas sendo adotados por alguns programadores de forma muito irrestrita, sem atenção ao contexto. Acreditamos que estas práticas podem estar não apenas trazendo benefícios a certas práticas de programação, mas também malefícios, principalmente ao restringir desnecessariamente a visão de pesquisadores.

Na pesquisa em Sistemas Operacionais o estudo de algoritmos de baixo-nível auto-modificáveis e adaptativos têm apresentado resultados interessantes. Existem trabalhos que demonstram a possibilidade de se obter grandes melhorias de eficiência nos sistemas utilizando programas de baixo nível sofisticados mais flexíveis e com adaptatividade [12,13,14].

III. A ARQUITETURA

A Figura 1 acima ilustra a arquitetura utilizada em algumas implementações de AFAs [15]. Um registrador armazena o estado atual da máquina, e o processo continua até atingir algum critério de parada, como o término da cadeia de entrada.

A arquitetura que implementamos é ilustrada pela Figura 2. Cada estado da máquina é representado por um objeto da classe `Estado`. Esta classe possui um método principal que deve ser executado no instante da entrada da máquina em cada estado. Cada objeto possui sua própria tabela de transições, implementada utilizando um *template* `map` da STL [16], que mapeia símbolos de entrada a ponteiros para objetos do tipo `Estado`, e do tipo `Funcao`. O método é responsável pela leitura da entrada, execução de funções adaptativas, e por executar o método do objeto que representa o próximo estado.

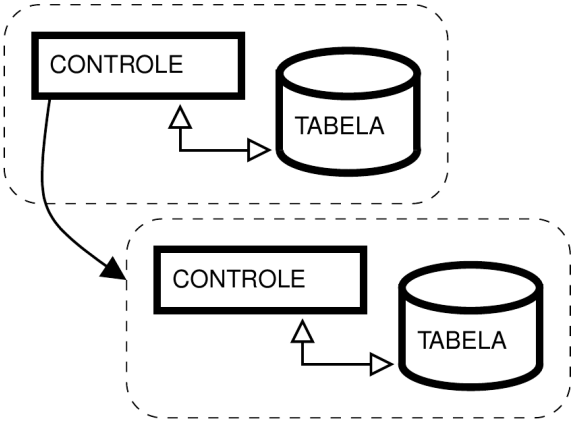

Fig. 2. Arquitetura implementada. Estados são objetos que executam os seguintes recursivamente.

Os ponteiros para os objetos criados são todos armazenados em uma lista implementada a partir do *template* padrão `list`. A entrada da máquina é feita pela entrada-padrão do sistema operacional, acessada por um objeto do tipo `istream`.

O programa resultante possui portanto diversos objetos do tipo `Estado`, cada um relativo a um estado da máquina, e estes podem criar e destruir livremente outros estados, além de modificar suas regras de transição. Ao invés de utilizarmos um laço que realiza as transições iterativamente, o funcionamento da máquina foi representado por chamadas recursivas do método principal da classe `Estado`.

Para realizar uma implementação adequada desta arquitetura é preciso utilizar um compilador que suporte chamadas de funções "irmãs", ou *sibling calls* ou ainda *tail calls* [17]. Neste tipo de chamada a sub-rotina retorna diretamente à função superior da que realizou a chamada.

Caso isto não corra, o programa resultante pode ficar lento, e até mesmo estourar a área de memória reservada ao programa, devido aos sucessivos empilhamentos de endereços de retorno. Em nosso sistema utilizamos a versão 4.2.1 do compilador GNU GCC [18], que utiliza este tipo de chamada como uma técnica de otimização do código.

A. *Funções adaptativas*

Cada transição pode possuir uma função adaptativa, modelada pela classe `Funcao`. Uma função em nosso sistema é uma lista de ações adaptativas que são executadas seqüencialmente. Existem dois tipos de ação: a criação de um novo estado, e a determinação de uma transição. Não implementamos ações explícitas de consulta ou de remoção.

Ao determinar uma transição, uma ação adaptativa só é capaz de acessar o último estado criado, o estado atual, e outros estados alcançáveis a partir destes. Isto é feito fornecendo-se uma cadeia de caracteres que deve ser seguida a partir do estado de partida (o último estado criado, ou o atual). Na versão atual do sistema, não é possível portanto fazer referência a estados de maneira global, ou fazer consultas em toda a máquina, mas apenas percorrer a sua estrutura momentânea como em um autômato finito, seguindo as cadeias de caracteres definidas dentro da função. Apesar de não podermos remover transições, podemos modificar livremente o destino das transições existentes. Na versão atual do sistema também é impossível apagar estados.

Estas restrições impostas às ações adaptativas que podemos realizar facilitam muito a tarefa de implementação de nossa arquitetura, e provém naturalmente de suas características. É preciso ainda estudar o impacto destas restrições no funcionamento das máquinas e na complexidade de suas descrições. Note que é possível superar algumas das dificuldades surgidas com táticas como criar transições com símbolos reservados para acessar estados específicos a partir de qualquer estado, ou ainda criar sempre pares de transições de sentidos opostos, permitindo navegar nos dois sentidos.

B. *Algumas considerações*

Nosso projeto foi desenvolvido a partir da idéia de possuir diferentes subrotinas executando umas às outras com regras variantes no tempo. Entretanto, o sistema compilado ao final, na presente forma, não possui exatamente este formato. A execução de um método de um objeto em C++ nada mais faz do que executar uma mesma função na memória que recebe o endereço do novo objeto como parâmetro [16].

O que realizamos ao final foi uma troca do laço utilizado na outra arquitetura pela execução de uma função recursiva que modifica iterativamente os parâmetros da chamada. Estes parâmetros são determinados a partir das regras de transição, que ao invés de serem lidos em uma grande lista são agora obtidos de uma estrutura semelhante a uma árvore encadeada.

Conceitualmente, estamos realizando uma troca do controle de execução do programa entre métodos de diferentes objetos, o que se assemelha um pouco com o que acontece em programas auto-modificáveis. Mas para obtermos isto no baixo nível, com diferentes rotinas, seria necessário definir diferentes classes para os estados.

Uma possibilidade para se criar esta alternância de chamadas de rotinas variante no tempo seria a ativação colateral de funções dentro de cada estado. Isto poderia ser obtido com o uso de funções do tipo *callback* e de sinais [19]. Por um lado isto não seria muito diferente de executar uma grande estrutura `case` para realizar algo dado cada estado, mas isto seria similar à substituição de nosso armazenamento de dados em árvore encadeada por uma lista, o que estraga a característica mais importante de nosso sistema: a concentração dos dados de cada estado em objetos separados.

Note ainda que estamos sempre realizando apenas modificações nos valores de variáveis do tipo ponteiro. Para realmente criarmos um programa auto-modificável seria necessário efetivamente escrever sobre o código, e não simplesmente alterar endereços de instruções de desvio e chamada de sub-rotina. A técnica de programas auto-modificáveis foi apenas uma inspiração para o desenvolvimento de nossa arquitetura. Mas ainda assim ela sai um pouco do paradigma de programação estruturada.

IV. SIMULADOR PROGRAMÁVEL E LINGUAGEM DESCRITIVA

Utilizando nossa biblioteca, criamos um programa capaz de simular AFAs projetados pelo usuário. Este programa lê a descrição da máquina a partir de um arquivo, lê a cadeia de entrada a partir da entrada-padrão, e ao final da cadeia ele avisa se esta foi aceita ou não, de acordo com a característica do estado no momento.

Decidimos definir uma nova linguagem para a descrição dos AFAs para deixar mais explícito as características de nossa arquitetura. Ela foi batizada de SDMBA—Sintaxe para Descrição de Máquinas da Biblioteca Adaptóide.

A linguagem permite nomear estados e funções, porém estes nomes não são utilizados posteriormente no funcionamento da máquina, quando apenas ponteiros são utilizados. Nomes não precisam ser declarados antes de ser utilizados.

```
programa = ( regra )*
regra = ( função | transição
        | ( "FINAL" | "INIT ") identificador ";" )
transição = nome símbolo nome (nome)? ";"
função = "FUN" ("POS" | "PRE") nome ":"
          (ação)* "END" ";"
estado_fun = ("T"|"*")(simbolo)*
ação = ( estado_fun símbolo estado_fun (nome)? ";"
       | "GEN" ";")
```

Nossa linguagem possui a seguinte gramática:

Um programa é uma seqüência de regras, que podem ser uma definição de estado final ou do inicial, uma transição da máquina inicial, ou uma declaração de função adaptativa. Transições podem ou não possuir funções associadas.

Na declaração das funções é que se define se ela é do tipo “pré” ou “pós”, executada antes ou depois da transição. O corpo da declaração possui uma lista de ações. O comando `GEN` cria um novo estado, que pode ser referenciado pelo símbolo `*` nas definições das ações.

Cada ação é executada imediatamente quando acionada. Isto significa que um estado referenciado por `T100`, por exemplo, pode referenciar um estado no começo da lista mas outro ao final, quando uma das transições no caminho for alterada.

V. TESTES

Utilizamos nosso simulador para implementar um AFA que reconhece números triangulares, ou seja, da série

O alfabeto utilizado possui apenas um símbolo, e os números foram representados em base unária. Este autômato está ilustrado na Figura 3. Inicialmente a máquina possui apenas um estado, que é o estado final, e uma transição para si próprio que coleta um símbolo e executa a função adaptativa posterior *H*. Esta função apenas cria um novo estado intermediário para o qual é apontado o estado de origem. Este novo estado possui uma nova transição para o estado final, executando a função *H*. O resultado é que o autômato cresce com uma estrutura de anel, ganhando mais um estado a cada volta.

Utilizando nossa linguagem, descrevemos a máquina da seguinte maneira:

```
FUN POS H:
  GEN;
  * 1 T1 H;
  T 1 *; END;
q1 1 q1 H;
FINAL q1; INIT q1;
```

Este programa pode ser modificado para realizar o reconhecimento de números quadrados, bastando modificar a função para que se acresça dois novos estados ao invés de um. Também é possível reconhecer potências de dois: basta executar a função *H* em ambas as transições criadas.

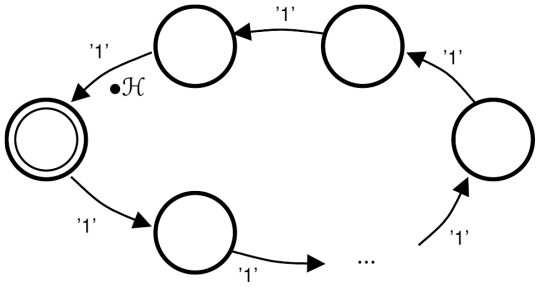

Fig. 3. Autômato finito adaptativo reconhecedor de números triangulares.

Um outro exemplo que implementamos com sucesso foi uma máquina capaz de reconhecer a linguagem `A^nB^nC^n`. Uma possibilidade de implementação pode ser descrita em nossa linguagem da seguinte maneira:

```
FUN POS F1: T a T F2;
  GEN; T b * G;
  * c T o; * a Tc; * b Tc; END;
FUN POS F2: T a T F3;
  GEN; Tb c * o;
  * b Tb; * a Tc; * c T x;
  T b *; END;
FUN POS F3:
  GEN; Tb c * x;
  * b Tb; * a Tc; * c T x;
  T b *; END;
FUN PRE x: T c Ta; END;
FUN POS o: Tc c Tcc o; END;

FINAL q1; INIT q1;
q1 a q1 F1; q1 b err; q1 c err;
err a err; err b err; err c err;
```

O estado `err` recebe transições referentes a palavras não-reconhecidas. Caracteres `a` no estado inicial geram uma cadeia de *n* estados com transições de ida para o símbolo `b`, e de volta para o símbolo `c`. As funções *x* e *o* atuam prevenindo que a quantidade de `b`s seja menor do que a de `a`s.

Os códigos-fonte da biblioteca e do simulador se encontram em nossa página, localizada em:

VI. CONCLUSÃO

`http://www.lti.pcs.usp.br/~nwerneck/adaptoide`

A arquitetura proposta foi utilizada com sucesso para implementar AFAs em programas em C++. Também pudemos criar um simulador de AFAs descritas por uma linguagem definida por nós. Demonstramos assim a aplicabilidade do modelo, e ainda da linguagem C++ em sua implementação.

Nossa arquitetura foi concebida de uma forma diferente da técnica popular de se implementar AFs, com tabelas de transição e um laço que realiza as leituras e transições. Entretanto, o programa que geramos ao final não ficou muito longe disto. No nível mais baixo, nosso sistema se diferencia apenas por ter trocado o laço de controle por uma função recursiva, e a estrutura de dados com as regras de transição passou a ser uma árvore encadeada. Porém, estas características estão implícitas em nosso programa, que foi construído em um nível de abstração mais alto, com objetos representando os estados da máquina, e passando o controle da execução do programa de um para outro.

Acreditamos que nossa arquitetura se compara à alternativa da mesma forma que listas encadeadas se comparam a vetores, e comprometimentos similares aos do uso destas estruturas de memória devem ocorrer na operação com ambas as propostas. Em especial, acreditamos que há uma maior dificuldade em nossa arquitetura de se realizar modificações envolvendo estados arbitrários ou distantes do atual, o que não é problema com uma tabela centralizada de transições. Tabelas entretanto possuem problemas, como menor flexibilidade.

Procuramos fazer o melhor uso possível de bibliotecas-padrão em nossos programas, esperando facilitar a integração com outros programas e obter mais eficiência. Um desejo nosso era ainda utilizar a programação orientada a objetos sem perder de vista a forma final do sistema no nível mais baixo, e acreditamos ter cumprido esta meta.

Esperamos no futuro introduzir diversas melhorias ao sistema, como empurrar símbolos na entrada utilizando métodos da biblioteca-padrão, implementar as funções adaptativas utilizando sinais de *callback*, engatilhar chamadas de função externas em cada estado, e testar a inicialização de múltiplas *threads* para simular indeterminismo.

REFERENCES

[1] J. J. Neto, *Adaptive rule-driven devices - general formulation and case study*, in CIAA, ser. Lecture Notes in Computer Science, B. W. Watson and D. Wood, Eds., vol. 2494. Springer, 2001, pp. 234-250.

[2] ----, *Adaptive automata for context-dependent languages*, SIGPLAN Notices, vol. 29, no. 9, pp. 115124, 1994.

[3] R. S. Rubinstein and J. N. Shutt, *Self-modifying finite automata: An introduction*, Inf. Process. Lett., vol. 56, no. 4, 1995, pp. 185190.
[4] C. Teuscher, Turing's Connectionism. An Investigation of Neural Network Architectures. London: Springer-Verlag, 2002.
[5] M. L. Minsky, S. A. Papert, *Perceptrons*. MIT Press, 1969.
[6] Wikipedia, C*onnection Machine*, Wikipedia, 15-Janeiro-2007.
[7] Wikipedia, *Inmos Transputer*, Wikipedia, 17-Dezembro-2007.
[8] Wikipedia, *von Neumann Architecture*, Wikipedia, 17-Dezembro-2007.
[9] E. W. Dijkstra, *Letters to the editor: go to statement considered harmful*, Commun. ACM, vol. 11, no. 3, pp. 147148, 1968.
[10] D. E. Knuth *Structured programming with go to statements*. ACM Computing Surveys 6(4), 1974, pp. 261301.
[11] U. Boccioni, P. Haeberli, B. Karsh, R. Fischer, P. Broadwell, and T. Wicinski, *The manifesto of the futurist programmers*, Internet, June 1991. Disponível em: `www.graficaobscura.com/future/futman.html`
[12] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, E. G. Sirer. *Spin - an extensible microkernel for application-specific operating system services*. Operating Systems Review, vol. 29(1), 1995, pp. 74-77.
[13] H. Massalin, *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, New York, NY, USA, 1992.
[14] H. Massalin, C. Pu, *Reimplementing the synthesis kernel*. In: *USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX, 1992, pp. 177186
[15] D. G. Santos, L. de Jesus, *Aspectos de projeto e implementação do não-determinismo no adaptools e seus impactos no aperfeicoamento da ferramenta*. Tech. rep., Universidade Católica Dom Bosco, 2006.
[16] B. Stroustrup, *The C++ Programming Language*, Second Edition. Addison-Wesley, 1991.
[17] Wikipedia, *Tail recursion*, Wikipedia, 18-Dezembro-2007.
[18] GCC manual, "GCC command options." [Online]. Disponível em: `gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
[19] A. Pereira and M. Cumming, *Libsigc++*, 2004, Disponível em: `libsigc.sourceforge.net/libsigc2/docs/manual/html`