Biblioteca para Autômatos Adaptativos com regras de transição armazenadas em objetos feita em C++

Nicolau Leal Werneck
LTI — PCS — EPUSP
Universidade de São Paulo
05508–900 São Paulo, SP, Brasil
Email: nwerneck@gmail.com
Telephone: +55-11-3091-5397
Fax: +55-11-3091-5294

***Resumo*—We created a library with the C++ language for the simulation of Adaptive Automaton. While traditional implementations generally use a table to store the transition rules and adaptive actions to be performed, in the system we present the informations relative to each state reside in separate objects. The execution does not happen in a loop that repeatedly analyzes the same transitions table, bur rather trough nested calls to methods of the different objects. A simple example was implemented.**

I. Introdução

A. Contexto

A aplicação do conceito de adaptatividade [1] a Autômatos Finitos dá origem aos Autômatos Finitos Adaptativos [2], ou Autômatos Finitos Auto-Modificáveis [3]. A modificação destas estruturas simples aumenta seu poder de computação mantendo sua adequação para certas atividades.

Implementações deste mecanismo em computadores convencionais geralmente utilizam tabelas centralizadas que armazenam as regras de transição. Um registrador armazena o estado atual em que a máquina se encontra, e o programa possui um laço que modifica o seu conteúdo de acordo com as transições da máquina, e ainda a tabela de transições de acordo com as regras de modificação (regras adaptativas).

B. Inspiração

Apesar da implementação com tabela funcionar adequadamente, e possuir valor didático, ela se afasta um pouco da forma original do sistema. Máquinas de estados possuem uma estrutura de nodos e vértices, e seria interessante buscar uma implementação mais descentralizada, onde os nodos possuem contato apenas com seus vizinhos. Além disso, a implementações com tabela centralizada também nada possui em comum com uma das maiores inspirações dos Autômatos Adaptativos: os programas auto-modificáveis.

Programas auto-modificáveis são códigos que modificam a própria área de memória em que se encontram, explorando a mais significativa característica da arquitetura de von Neumann: a memóra compartilhada [4]. O desenvolvimento de linguagens de programação modernas e compiladores sofisticados, além do problema prático da segurança, causaram o surgimento de uma antipatia a técnicas como esta e outras que só se observam em programação de baixo nível.

A antipatia por certas técnicas praticáveis apenas em programação de baixo nível costuma ainda tentar se justificar no movimento pelas linguagens estruturadas, simbolizado principalmente pela famosa carta de Edsger Dijkstra [5].

Enquanto uns se deixaram levar pelo forte título daquele artigo, chegando a parecer defender a própria obliteração do conceito de desvio na análise de algoritmos, outros levantaram questionamentos [6]. Por mais que a programação estruturada mais simples possua um grande poder, é inegável que apenas com o conhecimento da operação de máquinas em mais baixo nível podemos criar novas estruturas de programação, como por exemplo os blocos do tipo *try–catch*, que apenas recentemente se tornaram comuns em linguagens mais abstratas.

Dijkstra buscava entretanto apenas estimular o uso de conceitos de mais alto nível na atividade da programação, conceitos que devem ser naturalmente propostos sobre os de nível mais baixo. Nosso trabalho busca justamente explorar o conceito mais abstrato de Autômata Adaptativo, sem perder de vista a forma subjacente de sua implementação.

Nosso trabalho ainda se alinha ao desejo de outros programadores que compõem uma minoria que busca questionar certos dogmas que são adotados por alguns [7], e que acreditamos poderem estar não apenas trazendo benefícios durante certas práticas de programação, mas também malefícios, restringindo desnecessariamente a visão de pesquisadores. Este movimento questiona a antipatia a conceitos como programas auto-modificáveis, tornando-os novamente objetos de estudo regulares. Existem ainda trabalhos recentes que demonstram a possibilidade de se obter grandes melhorias de eficiência em sistemas operacionais utilizando programas de baixo nível mais flexíveis e com habilidades adaptativas [8], [9].

II. Arquitetura

A arquitetura para autômatos adaptativos tradicional é ilustrada pela Figura 1. Ela possui uma tabela de transições que é lida por um programa controlador central. O registrador armazena a identificação do estado atual do sistema. O programa controlador é basicamente um laço que realiza as transições de estado iterativamente, até que se atinja algum critério de parada, como o término da cadeia de entrada.

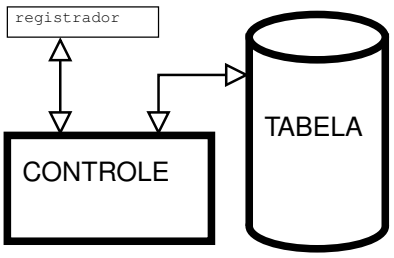

Figura 1. Arquitetura tradicional com registrador e tabela de transições.

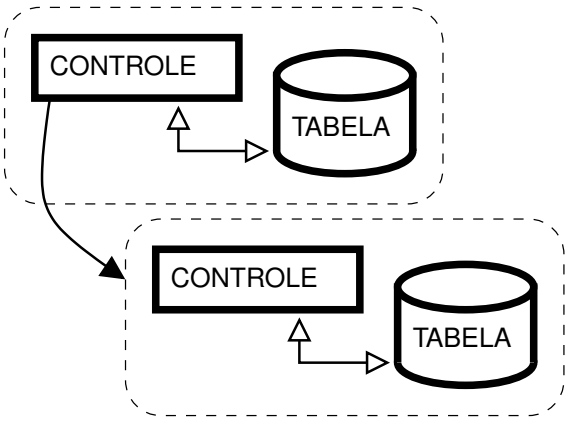

Figura 2. Arquitetura implementada. Estados são objetos que executam os seguintes recursivamente.

Na arquitetura que implementamos, ilustrada pela Figura 2, cada estado da máquina é representado por um objeto da classe `Estado`. Esta classe possui um método que deve ser executado no instante da entrada da máquina naquele estado. Este objeto possui sua própria tabela de transições, implementada utilizando um template `map` da STL. O método é responsável pela leitura da entrada, execução de funções adaptativas, e por executar o método do próximo estado selecionado.

Os ponteiros para os objetos criados são todos armazenados numa mesma lista encadeada, implementada a partir do template padrão `list`. A entrada da máquina é realizada através da entrada-padrão oferecida pelo sistema operacional, acessada por um objeto da biblioteca padrão do tipo `istream`.

O programa resultante possui portanto diversos objetos, cada um relativo a um estado, e estes podem criar e destruir livremente outros estados, e modificar suas regras de transição. Ao invés de possuir um laço que realiza as transições iterativamente, o funcionamento da máquina é 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", em que sub-rotinas retornam diretamente à função superior. Caso isto não ocorra o programa resultante pode ficar lento, e até mesmo estourar a área de memória reservada ao programa.

III. Testes

Utilizamos nossa biblioteca para implementar um autômato adaptativo que reconhece números triangulares, ou seja, da série $T_n = T_{n-1} + n = \left(n^2 + n\right)\frac{1}{2}$. O alfabeto utilizado possui apenas um símbolo, e os números foram representados em base unária. Este autômato está representado 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 $\mathcal{H}$. Esta função cria um novo estado para o qual ocorre a transição do estado de origem. Este novo estado possui uma nova transição para o estado final, executando a função $\mathcal{H}$. O autômato cresce em um estado a cada vez que se retorna ao estado final.

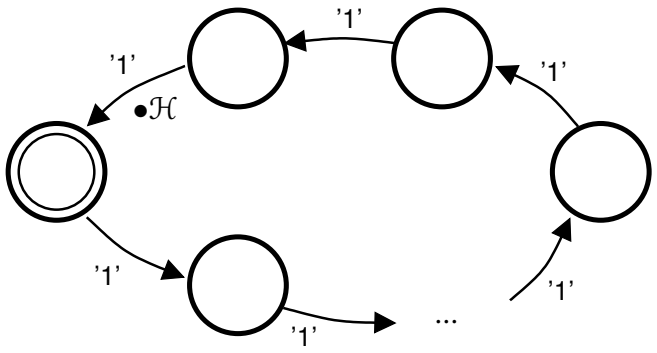

Figura 3. Autômato finito adaptativo reconhecedor de números triangulares.

O programa criado funcionou com sucesso, recebendo a entrada através da entrada-padrão do sistema operacional.

IV. Conclusão

A arquitetura proposta foi implementada apenas em um exemplo simples, mas que serviu para mostrar a viabilidade da nossa proposta. Não só demonstramos o sucesso do modelo, mas também do uso da linguagem C++, e da versão atual do compilador GNU GCC [10], que é capaz de otimizar o código realizando chamadas de funções "irmãs".

Acreditamos que nossa arquitetura se compara à tradicional da mesma forma que listas encadeadas se comparam a vetores, e acreditamos que comprometimentos similares aos do uso destas estruturas de memória ocorram na operação com as duas propostas.

Procuramos fazer o melhor uso possível de bibliotecas-padrão, esperando facilitar a integração com outros programas, além de possivelmente obter ganhos de eficiência.

Esperamos poder testar diversas melhorias ao sistema no futuro, como empilhar símbolos na entrada utilizando a biblioteca-padrão, implementar as funções adaptativas utilizando sinais de *callback*, engatilhar chamadas de função externas cada estado, implementar um interpretador para a linguagem AdapMap, e testar a a inicialização de múltiplas *threads* para simular indeterminismo.

Acknowledgment

O autor foi financiado pela CAPES.

Referências

[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. 115–124, 1994.
[3] R. S. Rubinstein and J. N. Shutt, “Self-modifying finite automata: An introduction,” *Inf. Process. Lett.*, vol. 56, no. 4, pp. 185–190, 1995.
[4] Wikipedia, “Von neumann architecture — wikipedia, the free encyclopedia,” 2007, [Online; accessed 17-December-2007].
[5] E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” *Commun. ACM*, vol. 11, no. 3, pp. 147–148, 1968.
[6] F. Rubin, “Letters to the editor: ‘GOTO considered harmful’ considered harmful,” *Commun. ACM*, vol. 30, no. 3, pp. 195–196, 1987.
[7] U. Boccioni, P. Haeberli, B. Karsh, R. Fischer, P. Broadwell, and T. Wicinski, “The manifesto of the futurist programmers,” Internet, June 1991. [Online]. Available: www.graficaobscura.com/future/futman.html
[8] H. Massalin, “Synthesis: an efficient implementation of fundamental operating system services,” Ph.D. dissertation, Columbia University, New York, NY, USA, 1992.
[9] C. Pu, “A retrospective study of the synthesis kernel.” [Online]. Available: citeseer.ist.psu.edu/169975.html
[10] GCC manual, “GCC command options.” [Online]. Available: gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html