A dynamically variable code execution model, based on adaptive automata

E. J. Pelegrini, J. J. Neto

***Abstract*— This paper presents the proposal for an adaptive assembly language and its run-time mechanism. The language is design to build programs capable of modifying their own instruction while executing. In addition to the language proposal, the design of a language interpreter (built as a virtual machine) is described, based on adaptive technology. That avoids some difficulties to perform code modification existent in traditional run-time mechanisms. Some information on of the language performance and on its virtual machine prototype are reported. The present proposal is a starting point towards the implementation of a full environment supporting the implementation of high level adaptive programming languages.**

***Keywords*— Self-modifying code, Adaptive code, Adaptive programming language, Adaptive assembly, Adaptive Technology.**

I. INTRODUÇÃO

Código auto-modificável (*Self-Modifying Code*) é o nome dado ao código que permite a alteração de suas próprias instruções durante a execução. Programas com capacidade de se auto-modificar já foram aplicados na otimização de código [1], na proteção do código fonte [2], para esconder detalhes internos do programa [3], como mecanismo de interpretação em linguagens adaptativas [4], [5], etc.

Em máquinas nas quais não existe distinção entre a área de memória que armazena o programa e aquela que armazena os dados, e que não apresentam nenhum tipo de mecanismo de proteção ao código do programa, é possível utilizar linguagens de baixo nível (como a linguagem *assembly*) para escrever códigos que alteram suas instruções durante a execução. A alteração de uma instrução pode, nesse caso, ser feita por meio da execução de certas instruções de máquina (por exemplo, a instrução *mov*) que promovem a movimentação dos códigos desejados para o endereço que contenha a instrução a ser alterada (exemplo em [3]).

Apesar de ser possível a alteração do código de programas escritos na linguagem *assembly*, devido a diversas características desse tipo de linguagem e do seu mecanismo de execução, pode-se dizer que tanto a linguagem como o seu mecanismo de execução não foram obrigatoriamente projetados para suportarem tais alterações.

Por exemplo, pode-se citar a necessidade não apenas da realocação de posições de memória de parte do código, como também de relocação das referências contidas no próprio código, particularmente no caso da inserção de instruções em um ponto arbitrário do código.

Como decorrência da existência apenas de mecanismos nativos primitivos em linguagem de baixo nível para a alteração do código, a compreensão e depuração de um programa com códigos auto-modificáveis são consideradas atividades complexas, tornando-se por essa razão a auto-modificação um recurso pouco explorado [3]. Pode-se dizer que, pelo mesmo motivo, poucas linguagens de alto nível dão suporte, em seu nível, à escrita de códigos que se auto-modificam em tempo de execução. Alguns exemplos de linguagens de alto nível que suportam algum tipo de auto-modificação encontram-se descritas em [4]-[7].

No contexto deste artigo procura-se apresentar uma linguagem, baseada em instruções *assembly*, projetada para dar suporte à alteração robusta de código. Essa linguagem impõe algumas restrições ao processo de alteração, limitando-o a trechos específicos do código, explicitamente declarados pelo usuário, com o intuito de evitar a modificação de áreas consideradas inalteráveis do código do programa, tornando mais simples, através dessa disciplina, o manuseio e o entendimento desse tipo de código.

Associado a essa linguagem, deve, neste caso, existir um mecanismo de execução responsável pelo apoio à alteração do código e pela garantia da manutenção de sua consistência. Para a proposição deste mecanismo, foi utilizado um formalismo desenvolvido no contexto da tecnologia adaptativa [8], [9]. A tecnologia adaptativa lida com dispositivos e formalismos que possuem a capacidade de, quando em certas situações bem definidas, alterar o seu comportamento em tempo de execução por meio de operações de auto-modificação disparadas em resposta a determinados estímulos externos [8]. O formalismo utilizado neste caso foi o autômato finito adaptativo [8], [10], [11].

Como o mecanismo de execução proposto difere do modelo típico executado pela máquina física, a linguagem é proposta como uma linguagem intermediária, a ser executada por uma máquina virtual [12].

E. J. Pelegrini trabalha no LTA – Laboratório de Linguagens e Tecnologia Adaptativa. (eder.pelegrini@poli.usp.br).

João José Neto é o responsável pelo LTA – Laboratório de Linguagens e Tecnologia Adaptativa, vinculado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo – Av. Prof. Luciano Gualberto, trav. 3, No. 158 – Cidade Universitária – São Paulo - SP - Brasil. (joao.jose@poli.usp.br) fone: 55(11) 3091-5402

Este artigo está dividido em 8 seções. A seção II apresenta trabalhos correlatos. Em seguida, na seção III, é apresentado o formalismo adaptativo utilizado na definição do mecanismo de execução da linguagem proposta. A seção IV apresenta as motivações deste trabalho, descrevendo a problemática enfrentada no desenvolvimento de código auto-modificáveis com ênfase nos aspectos da alteração de códigos escritos em *assembly* tradicional. A linguagem proposta é apresentada na seção V. Em seguida, a seção VI descreve o mecanismo de execução e modificação do código. A seção VII apresenta alguns resultados experimentais, seguidos da conclusão (seção VIII), do apêndice (exemplo de programa adaptativo) e, por fim, das referências.

II. Trabalhos relacionados

Neto [11] e [13] apresenta o conceito de autômatos adaptativos. Autômato adaptativo é um formalismo, baseado no autômato de pilha estruturado, capaz de se auto-modificar. Esse formalismo foi usado por Pistori, em [8], como base para a definição de autômatos finitos adaptativos. O mecanismo adaptativo assim elaborado é usado para a proposição do mecanismo de execução descrito no presente trabalho.

Freitas e Neto [4] e [5] descrevem um modelo de linguagem funcional adaptativa. Esta proposta utiliza formalismos adaptativos como base para o projeto de uma linguagem de alto nível, aderente à tecnologia adaptativa, na qual é possível escrever programas capazes de se auto-modificar em tempo de execução segundo uma disciplina específica. Conceitos encontrados nesses trabalhos inspiraram alguns aspectos do desenvolvimento da linguagem apresentada no presente artigo.

Aycock [14] descreve um modelo de execução de códigos baseado em gramáticas dinâmicas livre de contexto. Neste modelo de execução, os terminais gramaticais são interpretados como as instruções de máquina e o programa é representado, inicialmente, pelo não terminal inicial da gramática e ao longo do tempo pelas formas sentenciais (denominada *sentencial code form*). A execução do programa é composta por duas etapas que se alternam: a de geração do código a ser executado, por meio das regras de derivação descritas pela gramática e a de execução das instruções geradas. O modelo de execução de códigos baseado em gramáticas serve de origem para o desenvolvimento do modelo apresentado neste artigo.

Já na linha da Engenharia de Software, Yoder e Johnson [15] descrevem uma arquitetura de objetos adaptativos. Este modelo trabalha com atributos, classes, relações e comportamentos através de metadados, os quais permitem que objetos, criados e empregados pelo usuário, possam ser alterados pelo programa de maneira controlada. Ainda nesta linha, Lieberherr [7] descreve o chamado método de Demeter, que permite a associação dinâmica entre classes e métodos. Desses trabalhos também foram aproveitados alguns conceitos usados na elaboração do presente artigo.

III. Aspectos teóricos: autômatos finitos adaptativos

Autômatos finitos adaptativos são dispositivos baseados em regras que apresentam capacidade de auto-modificação. Ao iniciar sua execução, um autômato finito adaptativo pode ser visto como um autômato finito [16] M_0. Durante o reconhecimento de uma cadeia de entrada $w_0w_1\ldots w_n$ ($n \geq 0$), ao se executar certas transições, o autômato se modifica, passando pelas formas M_1, depois M_2, e assim por diante [10], sendo que cada forma, por sua vez, também pode ser vista como um autômato finito.

O mecanismo que promove a auto-modificação é descrito por meio de abstrações denominadas funções adaptativas [11], as quais costumam ser paramétricas e declaradas a parte.

Seguindo a formulação definida para o autômato adaptativo [13], as funções adaptativas especificam coleções de operações básicas. Existem três tipos de tais operações, denominadas ações adaptativas elementares: (i) ação de consulta, que retorna um conjunto de transições que obedecem a algum padrão estabelecido; (ii) ação de eliminação, que remove do conjunto de transições uma transição especificada; e (iii) ação de inserção, que acrescenta uma transição ao conjunto de transições do autômato.

Ativações das funções adaptativas denominam-se ações adaptativas, que podem ser associadas às transições do autômato. As ações adaptativas podem ser executadas antes e/ou depois da execução da transição a qual estão associadas [11].

Está provado em [17] que utilizar apenas um tipo de chamada de função adaptativa (antes ou depois da execução da transição) não reduz o poder de expressão do formalismo. O esquema ilustrativo da equivalência do poder de expressão se encontra na Fig. 1. A transição $(q_i, x) \rightarrow q_j$ [B●A] (Fig. 1(a)) – que utiliza chamada de funções adaptativas antes (B) e depois (A) da transição – pode ser reescrita como uma seqüência de duas transições $(q_i, \varepsilon) \rightarrow q_k$ [●B] ; $(q_k, x) \rightarrow q_j$ [●A] (Fig. 1(b)) – que utilizam apenas chamadas de funções adaptativas depois da execução de cada transição.

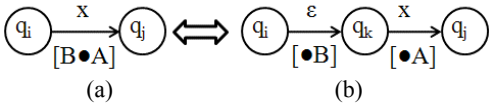


Fig. 1. Ilustração do esquema da equivalência. (a) Transição com chamada de funções adaptativas antes (B) e depois (A) da execução de uma transição. (b) Seqüência de transições equivalente a (a), que utilizam apenas chamada de função adaptativa depois da execução de cada transição.

Prova-se também que os autômatos finitos adaptativos, assim como os autômatos adaptativos, possuem poder de expressão equivalente ao da Máquina de Turing [10], [11].

IV. Auto-modificação e linguagem *assembly*

Conforme mencionado anteriormente, a motivação desta proposta está relacionada com o fato de as linguagens de máquina usuais, bem como os mecanismos de execução

tradicionais, não se apresentarem como mecanismos nativos práticos para códigos auto-modificáveis. Em especial, existem alguns problemas que dificultam significativamente a construção de códigos adaptativos.

Os mecanismos de execução tradicionais operam considerando o programa como um vetor de instruções. Exceto no caso de instruções de desvio e as de chamada e retorno de procedimentos, a próxima instrução a ser executada é, implicitamente, a instrução contida na posição seguinte do vetor (de fato, no endereço que dista da instrução corrente um número de posições de memória igual ao comprimento da que foi executada [18]). Este modelo de execução dificulta a inserção e remoção de instruções, uma vez que isso exige a realocação das instruções movimentadas e a correção das referências a elas feitas pelo programa.

Para ilustrar tal situação, um exemplo é mostrado na Fig. 2(a). Para facilitar o entendimento, apenas referências a endereços de instrução e valores imediatos estão expressos numericamente (as instruções, registradores e referências a dados são aqui mantidos em notação simbólica). Por hipótese, endereços e valores imediatos ocupem apenas uma posição de memória.

End.	Instrução
0	mov eax,5
3	mov ebx,3
6	add eax,ebx
9	cmp eax,8
12	je 17
14	add eax,1
17	nop

(a)

End.	Instrução
0	mov eax,5
3	add eax,1
6	mov ebx,3
9	add eax,ebx
12	cmp eax,8
15	je 20
17	add eax,1
20	nop

(b)

End.	Instrução
0	je eax
2	mov ebx,3
5	add eax,ebx
8	cmp eax,8
11	je 16
13	add eax,1
16	nop

(c)

Fig. 2. Exemplo. (a) Original. (b) Inserindo uma nova instrução. (c) Alterando a primeira instrução para je eax.

Caso se deseje inserir uma nova instrução entre as duas primeiras instruções do trecho de código da Fig. 2(a), ou seja, entre as instruções que começam no endereço 0 e no endereço 3, será necessário realocar todo o código contido nas posições 3 e seguintes, bem como atualizar os endereços de destino das instruções de desvio (ex.: instrução je), cujo endereço de destino seja superior ao endereço modificado, como se pode observar na Fig. 2(b) (o endereço de destino da instrução je 17 – Fig. 2(a) – passou a ser 20 – Fig. 2(b)).

Outro ponto importante reside no fato de a arquitetura do *hardware* em questão não oferecer uma instrução específica para a alteração de código, sendo usada também para isso a instrução de movimentação de dados (mov). Apesar de ser suficiente para alterar o conteúdo de uma única posição de memória, esta instrução não é adequada para inserções e remoções de grupos de instruções, nem garante a coerência do programa resultante. Por exemplo, considere-se o caso de mover o código da instrução je para o endereço 0 (que armazena o código de operação da instrução mov). Esta operação resultaria em código incoerente, dado que as primeiras três posições de memória conteriam je, eax, e a constante 5, que não faz parte da instrução recém-movida, pois uma instrução je ocupa apenas 2 *bytes*. Para evitar esse efeito, a substituição deve ser acompanhada da correta realocação do código, que consiste na remoção de três posições de memória, e da inserção de duas (e nas devidas realocações das demais instruções), resultando em um código coerente, como mostra a Fig. 2(c) – je eax.

As constantes necessidades de realocações dificultam o controle sobre as auto-modificações. As realocações alteram os endereços das instruções, sendo necessário acompanhar cuidadosamente as modificações para possibilitar futuras alterações corretas. Por exemplo, caso se deseje alterar a segunda instrução (mov ebx,3) após a alteração da Fig. 2(b), não se deve esquecer que após a primeira realocação, o endereço da segunda instrução do código original passa de 3 (Fig. 2(a)) para 6 (Fig. 2(b)).

V. Linguagem simbólica (*assembly*) adaptativa

Este trabalho apresenta uma proposta de linguagem intermediária adaptativa destinada a servir de linguagem de baixo nível a ser gerada como resultado da compilação de linguagens adaptativas de alto nível. Um código escrito em uma linguagem de programação adaptativa qualquer consiste inicialmente no código-fonte LF_0. Conforme se executam funções adaptativas, esse código do programa vai se modificando para LF_1, LF_2, ..., LF_n [4], [5], [19].

Tal linguagem pode também ser usada para criar programas em que se desejem ocultar detalhes internos, proteger o código-fonte, etc. De fato, código adaptativo é aquele código auto-modificável cujas alterações são sempre executadas mediante a ativação de funções adaptativas.

A proposta desta linguagem busca aproveitar as instruções típicas das linguagens simbólicas (como instruções do tipo *mov*, *add*) e, ao mesmo tempo, formular um modelo de alteração de código que se mostre prático quando comparado ao que se utiliza em linguagens *assembly* tradicionais.

No ensaio aqui relatado, tal linguagem se compõe de um subconjunto das instruções simbólicas da arquitetura IA-32 [20], [21], com a ressalva que à instrução mov não é permitido acessar a área de programa. Para realizar a alteração do código, três instruções (tratadas pelo interpretador da linguagem intermediária) foram criadas: mark, emark, adapt.

As primeiras duas permitem ao programador marcar como passível de alteração um trecho do código, que deve atender a critérios específicos (convenciona-se que somente trechos de código assim marcados podem ser alterados). A tal trecho é associado um identificador único, para ser usado no momento da alteração do código, eliminando a necessidade de o programador conhecer o endereço físico em que se encontra o trecho de programa a ser alterado.

A terceira instrução é adapt, a qual recebe o nome associado ao trecho a ser alterado e o da posição de memória (na área de dados) que contém o novo trecho a ser inserido. Ao ser executada esta operação, o interpretador da linguagem intermediária monta o trecho de programa, escrito na área de dados, e com ele substitui o conteúdo existente no local ocupado pelo trecho indicado. Este processo destrói o trecho

declarado, removendo também as instruções mark e emark. O objetivo de remover o trecho é permitir que um trecho de código passível de alteração perca tal propriedade. Entretanto, novos trechos alteráveis podem ser declarados como parte do código a ser inserido.

Isso garante a coerência do código escrito, uma vez que o trecho a ser inserido deve ser montado (em caso de erro, a montagem falhará e, conseqüentemente, a execução será interrompida).

Como esta instrução é responsável pela modificação do código a instrução adapt constitui o operador adaptativo desta linguagem.

Para exemplificar o funcionamento do mecanismo de alteração de código, considere-se o programa da Fig. 3(a). As instruções mark 1 e emark 1 não englobam nenhuma instrução, o que indica que é um trecho vazio de código, ao qual é possível acrescentar novas instruções.

Instrução	Instrução
mov eax,5	mov eax,5
mark 1	add eax,1
emark 1	mov ebx,3
mov ebx,3	add eax,ebx
add eax,ebx	cmp eax,8
cmp eax,8	je label1
je label1	add eax,1
add eax,1	label1:
label1:	adapt 1,dado
adapt 1,dado	
(a)	(b)

Fig. 3. Programa Exemplo em *Assembly* adaptativo. (a) Código Original. (b) Código após modificação (comando adapt).

A última instrução (adapt 1, dado), pode ser descrita como: montar o programa escrito (na área de dados) a partir da posição indicada por dado e inserir este código em lugar do trecho 1. Suponha-se que essa área de dados contenha o seguinte trecho de código: add eax,1 endp (endp é uma pseudo-instrução simbólica utilizada pelo processo de montagem, que indica o final físico do código simbólico a ser montado). Após essa alteração do programa, obtém-se um código similar ao apresentado em Fig. 3(b).

Para suportar tais alterações, o mecanismo de execução realiza um modelo baseado na premissa de que a próxima instrução a ser executada é sempre determinada pela instrução corrente, empregando-se para isso um autômato finito adaptativo, a cujos estados se associam as correspondentes instruções de máquina de que se compõe o programa. Em outras palavras, os programas se assemelham, quando representados graficamente (por exemplo, na Fig. 4), à máquina de Moore, na qual as instruções são associadas aos estados como se fossem as operações de saída da máquina.

O modelo de execução do programa da Fig. 3(a) é apresentado na Fig. 4(a). Ao se executar a ação de alteração de código (adapt 1,dado), o código resultante é o representado na Fig. 4(b).

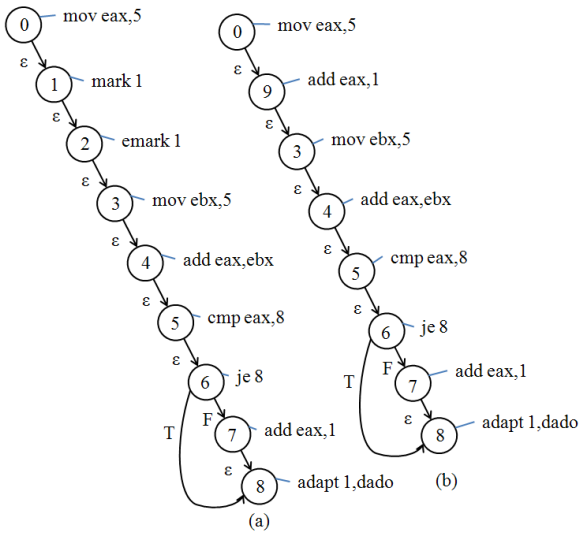


Fig. 4. Programa exemplo, formato de execução do código – endereçamento por autômatos. (a) Original (b) Após a execução da ação adaptativa.

VI. Mecanismo de Execução e Alteração

Apesar de as instruções serem baseadas em um subconjunto das instruções da linguagem *assembly* referentes à arquitetura IA-32, conforme mencionado anteriormente, a linguagem proposta não é executada diretamente sobre a máquina, necessitando de um mecanismo de máquina virtual (interpretador responsável pela execução do programa), similar à máquina virtual de linguagens tais como Java e C#. Esta seção descreve o mecanismo de execução, sem considerar a alteração de código, e, em seguida, esta é estendida para proporcionar recursos para a alteração do código.

A. Considerações a respeito do mecanismo de execução

Para a correta execução do programa, a máquina virtual que interpreta a linguagem provê algumas funcionalidades. Detalhes internos referentes a estas não serão aqui explicitados, dado que o foco do trabalho é descrever o mecanismo de execução das instruções e o de alteração de código. Dentre tais funcionalidades, destacam-se:

- Interpretação de instruções: responsável pela execução propriamente dita das instruções suportadas pela máquina virtual (como por exemplo, uma instrução de soma).
- Gerência de memória: responsável por controlar o processo de alocar memória e de reaver áreas de memória alocada, tanto para dados quanto para programas. Outra finalidade desta é a de controlar a quantidade de memória utilizada por um programa, interrompendo-o caso ultrapasse um tamanho máximo pré-estabelecido (preocupação inexistente em programas não-adaptativos, ao menos para a área de código executável).
- Proteção à área de programa: protege o código executável do programa, permitindo que apenas a instrução de alteração adaptativa de código (adapt) modifique o seu

conteúdo.

- Mecanismo adaptativo: responsável por prover operações de apoio à alteração do código, tais como: processar e executar as operações relacionadas a adaptatividade (mark, emark, adapt – que implementam o mecanismo de alteração do código executável), tratar casos de erros de alteração de código (como, por exemplo, alterar um trecho inexistente), tratar os casos excepcionais (como, por exemplo, alterar o trecho em execução), manter as informações que forem necessárias acerca dos trechos de código executável passíveis de serem alterados.

B. *Modelo de endereçamento e execução de instruções*

Conforme foi previamente comentado, o esquema de execução de programa consiste em um modelo em que a instrução seguinte é explicitamente apontada pela instrução corrente, exibindo assim uma topologia similar à de uma lista ligada de instruções. Este esquema de execução permite que novos códigos possam ser inseridos com facilidade e que trechos de código existentes sejam facilmente removidos, sem a necessidade de realocação nem tampouco de alteração, de qualquer natureza, no restante do código do programa.

O funcionamento deste modelo se assemelha ao modelo de execução, baseado em gramáticas dinâmicas, apresentado em [14] (apesar deste último não ter sido concebido com o propósito de oferecer suporte à alteração de código).

Na proposição do mecanismo de execução descrito neste trabalho foi utilizado, entretanto, o conceito de autômatos finitos adaptativos. Isso é possível, levando-se em conta que formalismos dinâmicos da classe dos autômatos e das gramáticas são equivalentes e ambos têm o poder de expressão das Maquinas de Turing. Em [22] faz-se um estudo detalhado das gramáticas desta categoria, e em [17] mostra-se com converter gramáticas adaptativas em autômatos adaptativos.

Cada procedimento declarado na linguagem é associado a um autômato finito estendido, implementando assim a forma de endereçamento das instruções anteriormente esboçada.

A extensão do autômato, acima mencionada, consiste na execução de instruções, pertencente à linguagem, as quais são interpretadas pela máquina virtual, associadas aos estados do autômato. Em outras palavras, no estabelecimento de um conjunto de instruções I, formado de instruções suportadas pela máquina virtual, e na associação, a cada estado do autômato, de uma instrução pertencente ao conjunto I, assim constituído.

Este autômato é declarado pela 7-upla $(Q, \Sigma, \delta, q_0, F, I, W)$, com:

- Q representando o conjunto finito e não vazio de estados do autômato finito;
- Σ representando um alfabeto binário de entrada do autômato: $\Sigma = \{T, F\}$;
- δ representando o conjunto das transições do autômato. Cada estado declarado aceita ou somente uma transição em vazio, ou então duas transições (a primeira, rotulada com o símbolo T e a outra, com o símbolo F);
- q_0 representando o estado inicial do autômato, que contém a primeira instrução do programa representado;
- F representando o conjunto finito e não-vazio de estados finais do autômato. Este conjunto é composto por todos os estados que estiverem associados à instrução de fim de execução do procedimento;
- I representando o conjunto de todas as instruções aceitas pela linguagem (incluem-se nesta definição as instruções mark, emark e adapt) e que podem ser associadas a um estado (para simplificar a descrição, no presente trabalho as instruções são apresentadas por meio da correspondente notação simbólica usada na linguagem *assembly*);
- W representando a associação de cada um dos estados do autômato a alguma instrução suportada pela máquina. É um conjunto de pares ordenados da forma (q_i, i_j), onde $q_i \in Q$ e $i_j \in I$. Como a cada estado fica associada apenas uma instrução, o conjunto W deve conter apenas um elemento para cada estado existente.

O esquema de execução deste autômato estendido consiste em, partindo do estado que contém a primeira instrução do código (estado inicial do autômato relacionado ao procedimento principal do código), executar a instrução associada ao estado e transitar para o próximo estado (próxima instrução). Este processo se repete até atingir a última instrução (estado final do autômato relacionado ao procedimento principal do código).

O modelo de execução apresentado trabalha com um esquema de endereçamento de instruções baseado nos estados do autômato, onde as transições são responsáveis por determinar a próxima instrução a ser executada. A construção da(s) transição(ões) que partem de um determinado estado depende da instrução associada a ele. Isto se deve ao fato de existir um grupo de instruções na linguagem *assembly*, denominado em [18] de instruções para fluxo de controle, que determinam a escolha da próxima instrução a ser executada. Quatro diferentes tipos de instruções para fluxo de controle podem ser distinguidos: desvios condicionais, desvio incondicionais, chamadas de procedimentos e retorno de procedimento [18].

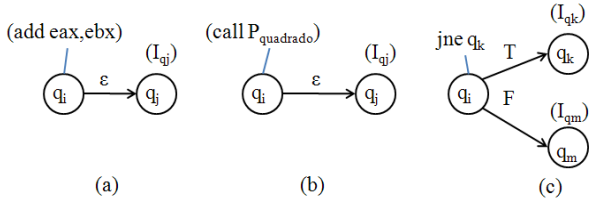


Fig. 5. Esquema ilustrativo de execução de instruções. (a) esquema da instrução de soma add eax,ebx. (b) esquema da invocação da sub-rotina quadrado. (c) esquema da instrução de desvio condicional jne q_k (salta para q_k se um determinado bit de controle for igual a zero).

Com o intuito de detalhar o esquema de obtenção da próxima instrução, o conjunto de instruções I é dividido em três subgrupos: instruções de rotinas, de desvio e as instruções

que não alteram o fluxo de execução (demais instruções). O exemplo relacionado a cada grupo se encontra na Fig. 5.

As instruções que não alteram o fluxo de execução são aquelas cuja operação não influência na determinação da próxima instrução a ser executada, como: instrução de soma (add), de multiplicação (mult), operações de pilha de sistema (push e pop). Um estado que contenha este tipo de instrução indica a próxima instrução (outro estado) através de uma transição em vazio. Um exemplo é a transição $(q_i, \varepsilon) \rightarrow q_j$ que se encontra na Fig. 5(a).

As instruções de rotinas são aquelas que invocam um procedimento (ex.: call) e que retornam de um procedimento (ex.: ret). Ao se entrar em um estado (q_i) que contenha uma chamada de procedimento, deve-se salvar o valor do próximo estado indicado pela transição em vazio (por exemplo, no caso da Fig. 5(b), deve-se salvar o estado q_j, estado destino da transição $(q_i, \varepsilon) \rightarrow q_j$) e o identificador do autômato corrente. Executa-se o autômato relacionado ao procedimento e, ao término desta execução (instrução ret), restaura-se o contexto salvo, representado pelo par (estado, autômato). Este processo é similar a chamada de sub-máquina, definido no autômato de pilha estruturado [23] e no autômato adaptativo [11].

As instruções de desvio são aquelas que saltam ou podem saltar para alguma instrução especifica (o destino é indicado como operando da instrução). Neste esquema, uma instrução de desvio condicional (desvia caso alguma condição seja satisfeita) apenas testa a condição e, se for satisfeita, escreve T na cadeia de entrada, caso contrário, escreve F [13]. De um estado que está associado a uma instrução deste tipo (por exemplo, o estado q_j da Fig. 5(c)), partem duas transições: (i) uma do tipo $(q_i, T) \rightarrow q_k$, onde $q_i, q_k \in Q$ e q_k é o endereço da instrução a ser executada caso ocorra desvio; e (ii) outra do tipo $(q_i, F) \rightarrow q_m$, onde $q_i, q_m \in Q$ e q_m é o endereço da instrução a executar caso não ocorra o desvio (exemplo Fig. 5(c)).

Os desvios incondicionais (ou saltos) não são necessários, uma vez que cada instrução indica, explicitamente, à próxima. Entretanto, como este tipo de instrução existe no *assembly* adaptativo (devido à forma de escrita do código, similar ao *assembly* tradicional, exemplo na Fig. 3), tal instrução foi implementada. Durante a interpretação, está instrução não realiza absolutamente nada. A próxima instrução a ser executada é aquela apontada pela transição vazia $(q_i, \varepsilon) \rightarrow q_j$, onde $q_i, q_j \in Q$ e q_j é o endereço da instrução a ser executada. Processos de otimização de código permitem eliminar estas instruções do modelo de execução, aumentando a eficiência do código.

C. Modelo de alteração do código

Um dos motivos de propor um modelo de execução baseado em autômatos finitos adaptativos foi o fato deste formalismo possuir um mecanismo que prove a capacidade de se auto-modificar, o que permite aproveitar idéias e conceitos previamente validados para definir o processo de alteração do código. O modelo de alteração de código é composto de duas partes: o elemento de alteração e o mecanismo de alteração.

1) Elemento de alteração

O elemento básico de alteração do código consiste em um trecho de código que pode não conter nenhuma instrução, uma, duas, e no caso extremo, todas as instruções de um procedimento.

A escolha de marcar áreas de código que contenham um número inteiro de instruções justapostas é motivada por: (i) simplicidade da notação: devido à forma de escrita de códigos *assembly*, o processo de demarcar dados pertencentes a uma instrução dificultaria o entendimento do código, dado que a escrita da marcação do elemento sobreporia a própria escrita da instrução; e (ii) ao se trabalhar com instruções, é possível evitar situações problemáticas, como a substituição do código de operação de uma instrução por outro que precisa de um número diferente de operandos (exemplo da troca do valor de mov para je, descrita na seção IV).

Os trechos de códigos passíveis de alteração são marcados por meio de duas instruções: mark id e emark id, onde o termo id refere-se ao identificador do trecho do código. A instrução mark id é utilizado para iniciar um trecho de código que pode ser alterado, enquanto a instrução emark id encerra tal trecho. É permitido marcar um trecho de código passível de alteração dentro de outro (aninhamento de marcação).

Como comentado anteriormente, apenas trechos marcados podem ser modificados. Entretanto, alguns requisitos devem ser seguidos durante a criação do trecho. São eles:

- O trecho marcado deve ser interno a um procedimento (portanto, não é possível criar novos procedimentos).
- Uma instrução fora de um determinado trecho não pode desviar para uma instrução dentro deste trecho. Apenas a instrução anterior ao inicio do trecho (instrução mark) pode transitar para a primeira instrução dentro do trecho.
- Com exceção da última instrução do trecho (instrução emark), nenhuma outra instrução dentro do trecho pode transitar para uma instrução fora do trecho.

Tais restrições evitam que as operações de modificação do código acarretem a desatualização de referências a posições internas a parte modificada.

Embora não seja permitido criar novos procedimentos com este esquema de alteração, apenas alterar os existentes, isso não reduz a expressividade deste esquema, visto que sempre é possível criar um procedimento contendo um comando de múltipla escolha (desvios), no qual cada escolha corresponderia a um procedimento diferente, e tais elementos possam ser criados dinamicamente, simulando assim a criação de tantos procedimentos quantos forem necessários.

A aparente restrição acerca das limitações impostas aos desvios entre trechos pode ser contornada fragmentando-se ou incorporando outras instruções adequadamente aos trechos, de acordo com a necessidade das referências externas e da localização das instruções de desvio.

Estas instruções (mark e emark) são usadas durante dois momentos distintos:

- Durante a montagem do programa (conversão da linguagem simbólica para o formato de execução baseado em autômatos), as instruções são usadas para formar uma tabela de controle dos trechos, que será usada pelo interpretador durante a execução. Esta estrutura de controle tem como objetivo agilizar a localização do trecho, bem como tratar os casos de referência indireta aos trechos marcados (o indexador associado ao trecho marcado pode ser passado de forma indireta para a instrução de alteração de código).
- Em tempo de execução, estas instruções permitem que o interpretador identifique que trechos de códigos (passível de alteração) se encontram em execução. Esta informação é particularmente importante para os casos em que a instrução de alteração do código modifica o próprio trecho marcado de código que a contém.

Um exemplo ilustrativo de marcação e de sua estrutura de controle é fornecido na Fig. 6. O trecho que começa no estado 1 e vai até o 5 corresponde ao trecho de código CA1 (indexado por 1), que pode sofrer alterações. Observe que a estrutura de controle armazena o nome símbolo do trecho, o indexador do trecho (este indexador é fornecido durante a montagem do código simbólico), bem como a transição de entrada e de saída do trecho, no caso da Fig. 6, a transição de entrada é (0, ε) → 1 e a de saída é (5, ε) → 6.

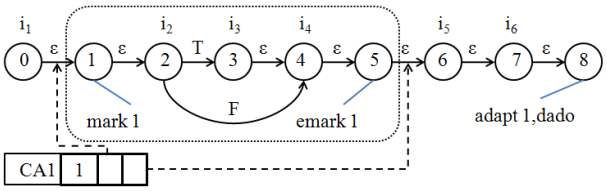


Fig. 6. Elemento de alteração e sua estrutura de controle - Trecho de código CA1 (indexado por 1).

Por fim, ressalta-se que cada trecho marcado possui apenas uma transição de entrada e uma de saída. A unicidade da transição de saída é garantida pelo fato da última instrução de um trecho marcado ser emark e ser proibido que uma instrução dentro do trecho passível de alteração salte para fora do trecho. Já em relação à unicidade da transição de entrada, esta é garantida por meio de um artifício na montagem do código.

Este artifício é usado apenas em caso de existência de desvios para a primeira instrução do trecho passível de alteração, o que implicaria na existência de mais de uma transição de entrada, e consiste na inserção de uma instrução nop (*no operation*) antes da primeira instrução do trecho marcado (instrução mark).

2) Mecanismo de alteração

O mecanismo responsável por executar a alteração do código baseia-se na camada adaptativa proposta para os autômatos adaptativos, ou seja, nas funções adaptativas. Ao contrario do autômato adaptativo, onde cada função adaptativa deve ser declarada a parte, no modelo descrito neste artigo existe apenas uma função adaptativa declarada, de caráter generalista. Ou seja, esta função adaptativa declarada é capaz de realizar todos os tipos permitidos de alteração de código, evitando, assim, a necessidade de declarar outras funções adaptativas.

Esta função, descrita a seguir, é ativada por meio de uma instrução com o seguinte formato: adapt bloco,cod. Durante a execução desta instrução (adapt), os operandos desta instrução são passados como parâmetro para a função adaptativa. O primeiro argumento da função adaptativa (associado ao operando bloco) refere-se ao indexador do trecho de código marcado como passível de alteração (a passagem do argumento pode ser de maneira direta – próprio identificador – ou indireta – posição de memória que contenha o identificador desejado). O segundo argumento (associado ao operando cod) refere-se à posição de memória da área de dados que contém o código a ser montado e inserido ao programa. Caso um desses dois argumentos não seja válido (como, por exemplo, o identificador do trecho não exista), a execução do programa é abortada.

O funcionamento desta ação adaptativa pode ser dividido em três passos:

- Primeiro Passo: Este passo possui um comportamento similar ao de uma ação elementar de consulta indexada. Consiste em buscar as transições de entrada e saída do trecho marcado, a partir do parâmetro referente ao indexador do trecho a ser alterado, como exemplificado na Fig. 7.

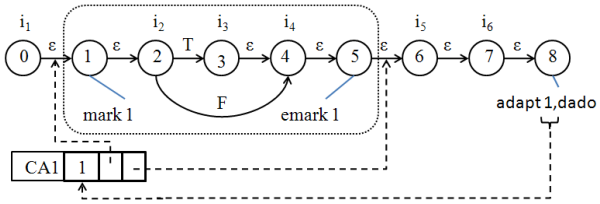


Fig. 7. Passo 1 - Procura do trecho de código a ser alterado (no caso indexado por 1).

- Segundo Passo: Remover o código existente no trecho de código que vai ser modificado (incluindo as instruções de marcação), salvando as informações das transições de entrada (ts) e a de saída (te) deste trecho (exemplo: Fig. 8). Ao se remover o código, tanto às instruções do trecho quanto as informações de controle que o interpretador possui são destruídas (caso exista um ou mais trechos declarados dentro do trecho a ser removido, estes também serão removidos);

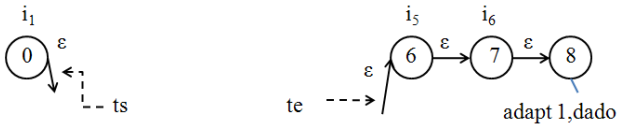


Fig. 8. Passo 2 - Remoção do trecho a ser modificado

- O terceiro e último passo consiste na inserção de novas instruções no lugar do trecho apagado. Caso o parâmetro referente à posição de memória onde se encontra o trecho de código a ser inserido seja igual a zero, este passo não será executado (tal situação é uma das maneiras de representar o processo de exclusão de um trecho). Esta etapa é subdivida em duas partes:
  - A primeira parte consiste em montar o trecho de código armazenado na área de dados (segundo parâmetro da função), conforme o ilustrado no exemplo da Fig. 9. O código armazenado pode conter instruções que criam um ou mais trechos de códigos com capacidade de alteração. As instruções devem estar escritas na linguagem *assembly* proposta (para facilitar a escrita de códigos) e, portanto, deverão ser montadas e convertidas para a representação interna do interpretador. Caso a montagem do código existente da área de dados falhe, o interpretador da linguagem recebe um aviso de erro e efetua o devido tratamento (até o presente momento, este tratamento consiste no término da execução do código).

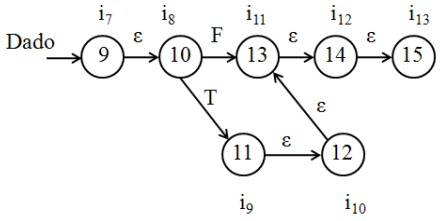


Fig. 9. Passo 3.1 - Programa escrito em dado (parâmetro) montado.

  - A segunda parte remete a modificação em si: inserir o novo trecho no lugar do trecho removido durante o segundo passo (como no exemplo da Fig. 10). Para realizar esta inserção, é atribuído um novo estado de destino (estado associado à primeira instrução do trecho montado) a transição salva como ts e um novo estado de origem (estado associado à última instrução montada) para a transição salva como te. Caso a instrução de adapt esteja contida dentro do trecho que altera, a próxima instrução a ser executada é aquela contida no estado de destino da transição te (por exemplo, para a situação descrita na Fig. 10, a próxima instrução seria a referente ao estado 6).

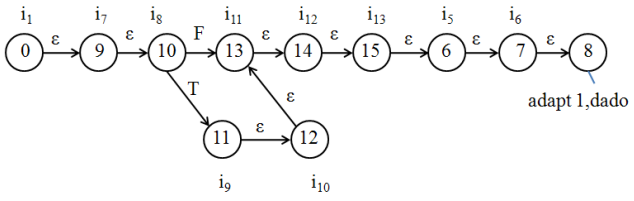


Fig. 10. Programa após alteração de código.

Todo este processo de alteração de código descrito não é visível ao programador, sendo visível apenas a chamada da função adaptativa, por meio da instrução adapt. Espera-se, com isso, facilitar o uso desta linguagem.

Uma vez que a função adaptativa é associada ao estado ao invés da transição torna-se necessário provar que este modelo está condizente com os desenvolvidos em [8], [11], [13].

A prova consiste em demonstrar que invocar as funções adaptativas no estado é equivalente a invocá-las após executar uma transição. O esquema de equivalência usado na prova se encontra na Fig. 11 (Fig. 11(a) é equivalente a Fig. 11(b)).

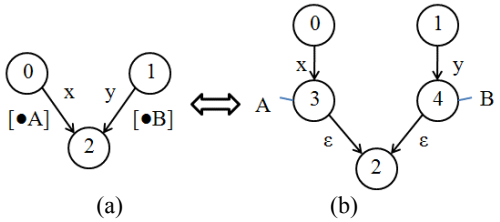


Fig. 11. Esquema de demonstração de equivalência (a execução de (a) é equivalente a de (b)). (a) Modelo Original de execução: A transição (0, x)→2 [•A] representa transição interna com o símbolo x, invocando a ação adaptativa A após a execução da transição. (b) Modelo de invocação das ações adaptativas nos estados.

D. Problemas relacionado a cache

Máquinas modernas costumam utilizar hierarquia de memória e cache [2], [18], com o objetivo de aumentar o desempenho de execução dos códigos. O modelo desenvolvido neste trabalho apresenta problemas relacionados a utilização de mecanismos de cache tipicamente associados a execução de código. Tais problemas, em paralelo com o *overhead* introduzido pela máquina virtual, podem degradar o desempenho de execução.

Associado ao modelo de execução tradicional existe o chamado princípio da localidade espacial [18], base para a construção do mecanismo de cache dos processadores modernos. Este princípio, em suma, diz que em um pequeno intervalo de tempo tende-se a acessar, com um alto índice de probabilidade, endereços próximos [18], ou seja, o endereço da próxima instrução tipicamente se encontra próximo ao endereço da última instrução executada. Como na proposta descrita neste artigo a próxima instrução a ser executada é explicitamente indicada pela instrução corrente, tem-se um modelo que independe da distribuição das instruções na memória, o que implica na inadequação de mecanismos de cache baseados neste princípio.

Supondo que o mecanismo de cache funcionasse, ainda existe outro problema relacionado ao mesmo. Ao se alterar um trecho de código que se encontra armazenado no cache (o que acontece, por exemplo, quando a instrução corrente altera a próxima instrução), é necessário atualizá-lo, para manter a coerência entre o código armazenado na memória principal e o armazenado no cache. Este problema é relatado em outros trabalhos sobre código com capacidade de se auto-modificar, como em [1], [2].

Entretanto, grande parte dos processadores da família x86 possui mecanismos que garantem tal coerência [2] e, portanto, não sofrem com o problema acima descrito.

VII. Resultados experimentais: adaptcod

Com o intuito de exercitar os conceitos trabalhados ao longo deste artigo, foi desenvolvido um protótipo da máquina virtual, denominado *AdaptCod Machine*, que serve de ambiente de execução para a linguagem *assembly* adaptativa denominada *AdaptCod*.

Este ambiente processa o código, monta o programa na forma de autômato descrito nas seções anteriores e o executa.

Para analisar o desempenho da *AdaptCod Machine* em relação à máquina física, foram utilizados três programas. Devido ao fato de a *AdaptCod Machine* ser um protótipo experimental, com quantidade limitada de instruções, optou-se por desenvolver ensaios simples em lugar de usar programas pré-existentes.

Para garantir que os programas fossem escritos de maneira similar, foi montado um pequeno compilador que gera o programa tanto para a linguagem de máquina do PC quanto para a do interpretador (portanto, tais programas são estáticos e não realizam auto-modificações). Os programas foram compilados tanto para a linguagem *Assembly* do PC quanto para o *AdaptCod*.

O primeiro programa (Teste 1) calcula os fatoriais de maneira recursiva, para os números de 1 a 10, apresentando os resultados na tela; o segundo (Teste 2) realiza um milhão de loops, cada qual consiste de cinco operações: duas de soma (somando uma variável a ela mesma), um decremento (da variável que controla o salto) e uma operação de apresentação de dados na tela; e o terceiro programa (Teste 3) é similar ao anterior (Teste 2), exceto pelo fato de não fazer a apresentação de dados na tela.

Os programas foram executados em um computador com a seguinte configuração: AMD 64, 3.2 Ghz, com 1 GB de RAM. Foram coletados: o tempo de execução total – o tempo de carregar e executar o programa, até o seu término, denotado por T_{total} – e o tempo de ocupação da CPU pelo programa, denotado por T_{proc}.

Para garantir a consistência dos resultados, foram realizadas dez medições para cada programa. Os valores médios dos tempos coletados estão apresentados na Tabela I.

TABELA I
Desempenho de Execução

	Assembly		*AdaptCod*	
	T_{total} (ms)	T_{proc} (ms)	T_{total} (ms)	T_{proc} (ms)
Teste 1	40,60	15,63	42,20	42,20
Teste 2	69235,50	3651,56	71517,10	31525,00
Teste 3	62,50	50,00	12907,70	12773,44

Para os programas que realizam operações de saída (apresentam dados na tela), Teste 1 e Teste 2, o tempo de execução total do *AdaptCod* tende a se aproximar do obtido na versão *assembly*. Isto se deve ao fato de que as operações de entrada/saída costumam ser lentas, quando comparadas aquelas exclusivamente dependentes do processador [18], bem como ao fato de estas operações serem tratadas de maneira relativamente similar (a máquina virtual que interpreta o *AdaptCod* não trabalha com nenhum esquema sofisticado relacionado à entrada/saída, tal como virtualização de periféricos). Entretanto, ao se analisar o tempo que estes programas ocupam no processador, é possível perceber um acréscimo de mais de 60% no tempo de utilização, causado pelo processo de interpretação de instruções e das transições entre os estados do autômato.

Esta situação se confirma nos resultados referentes ao Teste 3 (programa que faz uso intensivo de processador). Observa-se que o tempo total referente à execução deste programa na versão *AdaptCod* é significativamente maior que na versão *assembly* do PC. A elevada utilização do processador é esperada, devido ao *overhead* ocasionado pela máquina virtual, destacando-se o fato de que o interpretador de instruções foi construído em linguagens de alto nível. Outro fator importante que degrada o desempenho é a ausência de mecanismo de cache na implementação da máquina virtual.

No intuito de averiguar se tal queda de desempenho é efeito colateral do uso de um interpretador de instruções da máquina virtual ou do tempo associado à determinação da próxima instrução (transição do autômato referente ao modelo de execução), o código da máquina virtual foi alterado, para não interpretar as instruções, apenas percorrer o mesmo caminho (i.e. passar pelo mesmo número de estados do autômato). O resultado foi uma queda no tempo de execução total, que caiu de aproximadamente 13 mil milisegundos para 856 milisegundos, o que indica a necessidade de uma otimização da implementação do interpretador de instruções.

Um último teste de desempenho foi feito, desta vez para averiguar o desempenho da operação de auto-modicação / adaptação. Em um novo programa, com 135 instruções, foi feito uma substituição de código de 32 instruções por um trecho outras 32 instruções. O processo de alteração demora, aproximadamente, 87 milisegundos, sobre um tempo de execução de 193,8 milisegundos. Ou seja, 45% do tempo de execução foi gasto para alterar 23,7% do código. Este elevado percentual deve-se ao processo de montagem do código (conversão da linguagem simbólica para a representação interna) do código a ser inserido, antes de aplicar a modificação.

VIII. Conclusões

Este artigo apresenta uma proposta de código *assembly* adaptativo, bem como seu mecanismo de execução. Esta proposta procura contornar algumas dificuldades enfrentadas ao se escrever um código que se auto-modifica na linguagem *assembly* tradicional. Para facilitar ao programador utilizar os mecanismos de alteração de código, disponibiliza-se para ele três operações, que implementam as mais complexas dificuldades da tarefa de alteração do código.

Foi proposta uma nova forma de execução de código, baseada em formalismos adaptativos. As dificuldades do processo de alteração em si não são visíveis ao programador, que pode alterar o código baseado em um mecanismo simples de delimitação de trechos de seu programa e no uso da

instrução de modificação do código.

O principal objetivo dessa linguagem é servir como uma linguagem padronizada para propiciar o desenvolvimento de linguagens de alto nível adaptativas que servirão para implementações de programas adaptativos. Espera-se que a proposta deste *assembly* adaptativo motive a busca e a convergência de mecanismos adaptativos.

Não obstante, também é esperado que esta linguagem seja usada para a escrita de códigos auto-modificáveis que possuam outras finalidades, como as previamente citadas (por exemplo, programas que desejem ocultar detalhes internos).

Atualmente, está sendo elaborada uma nova versão do ambiente de execução, procurando melhorar seu desempenho para tornar esta proposta mais atrativa, bem como tornar mais abrangente o conjunto de instruções aceitas pela máquina virtual que interpreta a linguagem.

APÊNDICE: EXEMPLO DE PROGRAMA ADAPTATIVO

Este apêndice apresenta um exemplo de código que realiza auto-modificação com o intuito de servir de apoio a explicação do processo de modificação do código baseado nas instruções mark, emark e adapt.

Este exemplo consiste no procedimento que calcula o fatorial de um número positivo (caso não seja, retorna 1). Este procedimento foi escolhido por sua simplicidade e tamanho reduzido do código.

Para a escrita deste código, foram usadas as mesmas hipóteses simplificadoras adotadas na escrita de códigos em linguagem *assembly* tradicional (veja seção IV). Entretanto, neste exemplo, os endereços referenciados pelas instruções de desvios serão substituídos por nomes que os representam. Por fim, foi considerado que os códigos a serem inseridos tenham sido previamente gravados na área de dados do programa.

O código do procedimento é apresentado na Fig. 12.

Área de Programa

```
FAT:            //procedimento fatorial(n)
  pop n         //desempilha n (fat (n))
  mov eax,1     //acumulador ← 1
  cmp n,0       //compara n com 0
  // para os casos em que N <= 0, desvia
  // para o fim do procedimento, retornando
  // o valor 1.
  jle X         //desvia para X se n <= 0
  // caso n > 0, modifica o código para
  // calcular o fatorial
  adapt 1,step //adapta trecho 1
  mark 2        //trecho adaptativo 2
  mark 1        //trecho adaptativo 1 (vazio)
  emark 1       //fim de trecho adaptativo 1
  emark 2       //fim de trecho adaptativo 2
X:
  mov res,eax  //res ← eax
  adapt 2,base //adapta trecho 2
  ret           //retorna
```

Área de dados

```
//Códigos aqui apresentados são códigos não
//montados (estão na linguagem assembly
//adaptativo)
Dado base
// endereço base aponta para o local
// onde está armazenada a primeira
// instrução (mark 2) do dado abaixo.
  mark 2
  mark 1
  emark 1
  emark 2
  endp          //endp fim de montagem
Dado step  //similar ao caso anterior
  mul eax,n     //eax ← eax*n
  dec n         // n ← n-1
  cmp n,0
// o rótulo Y abaixo é um label cujo endereço
// não está resolvido. O endereço a que se
// refere somente será resolvido durante a
// montagem deste código (passo 3 do
// processo de modificação do código).
  jz Y          //desvia para Y se n == 0
  // modifica o código para continuar o
  // cálculo do fatorial
  adapt 1,step //adapta trecho 1
  mark 1
  emark 1
Y: // label Y (endereço a ser resolvido
   // durante a montagem)
  endp
```

Fig. 12. Procedimento fatorial de n adaptativo

Este programa utiliza a auto-modificação para criar a série de multiplicações (do tipo n * (n-1) * ... *1) que calcula o fatorial do número n. A área de dados apontada por step gera o código responsável por multiplicar o resultado parcial (por exemplo, n * (n-1)) pelo próximo valor (por exemplo, (n-2)) e, em seguida, decrementar n. Também é responsável por gerar o próximo passo da multiplicação (caso n seja diferente de 0), por meio da instrução `adapt 1,step` e da marcação de um novo trecho marcado, indexado por 1.

Para detalhar o funcionamento, considere o caso do cálculo do fatorial de 3 (n = 3). A Fig. 13 apresenta a situação do código após as modificações que calculam o fatorial e antes da execução da instrução `adapt 2,base`.

Inicialmente, o acumulador (EAX) recebe o valor 1. Como n é maior que zero, a instrução `adapt 1,step` é executada, gerando a multiplicação (EAX ← 3*1 = 3) e decrementando n (n=2). Como n é diferente de 0, o código inserido no programa faz com que o programa se modifique novamente, inserindo os códigos responsáveis pela multiplicação (EAX ← 3*2 = 6) e decrementando n (n=1). Novamente, n ainda é diferente de zero e, portanto, o programa se modificará uma terceira vez, completando o cálculo do valor do fatorial, armazenando-o no acumulador (EAX ← 6*1) e decrementando n (n=0). Como agora n é igual a zero, a instrução jz Y''' faz com o que a execução salte para o endereço associado a Y''' (Fig. 13), interrompendo processo de modificação e encerrando o cálculo do fatorial.

Área de Programa

```
FAT:            //procedimento fatorial(n)
  pop n         //desempilha n (fat (n))
  mov eax,1     //acumulador ← 1
```

```
  cmp n,0       //compara n com 0
  jle X         //desvia para X se n <= 0
  adapt 1,step  //adapta
  mark 2
     mul eax,n  //eax ← 1*3 = 3
     dec n      //n ← 3-1=2
     cmp n,0
     jz Y'      //n != 0, não desvia
     adapt 1,step  //adapta
        mul eax,n //eax ← 3*2 = 6
        dec n     //n ← 2-1=1
        cmp n,0
        jz Y''    //n != 0, não desvia
        adapt 1,step  //adapta
           mul eax,n //eax ← 6*1 = 6
           dec n     //n ← 1-1=0
           cmp n,0
           jz Y'''   // n==0, desvia
           adapt 1,step  //não executa
           mark 1
           emark 1
     Y''': // Y''' = 3ª instância de Y
   Y'': // Y'' = 2ª instância de Y
 Y':  // Y' = 1ª instância de Y
  emark 2
X: //as instruções abaixo ainda não foram
   //executadas
  mov res,eax //res ← 6
  adapt 2,base
  ret
```

Fig. 13. Exemplo da inserção de código para o cálculo do fatorial de 3. A endentação do código é utilizada para diferenciar os códigos do programa original, bem como os códigos inseridos em momentos distintos (no lugar de algum trecho adaptativo). As setas com tracejado sólido indicam que instrução gerou o código relacionado a outro nível de endentação. A seta pontilhada indica desvio realizado durante a execução.

Após armazenar o valor do fatorial em res (instrução `mov res,eax`), o procedimento realiza uma última auto-modificação (instrução `adapt 2,base`). Esta auto-modificação faz com que o procedimento FAT retorne à sua forma original, removendo as instruções inseridas pelas instruções `adapt 1,base` (Fig. 14), para que em uma próxima chamada possa funcionar corretamente e com o intuito de evitar o consumo desnecessário de memória.

Área de Programa

```
FAT:          //procedimento fatorial(n)
  pop n       //desempilha n (fat (n))
  mov eax,1   //acumulador ← 1
  cmp n,0     //compara n com 0
  jle X       //desvia para X se n <= 0
  adapt 1,step  //adapta
      mark 2
      mark 1
      emark 1
      emark 2
X:
  mov res,eax //res ← eax = 6
  adapt 2,base
  ret
```

Fig. 14. Procedimento FAT (calculando fatorial de 3) após execução da instrução `adapt 2, base`. A endentação do código é utilizada para diferenciar os códigos do programa original, bem como os códigos inseridos em momentos distintos (no lugar de algum trecho adaptativo). A seta com tracejado sólido indica que instrução gerou o código relacionado a outro nível de endentação.

É importante destacar os seguintes pontos deste exemplo:

- Enquanto a instrução `adapt 1,step` remove o trecho marcado, indexado por 1, também cria um novo trecho marcado, com o mesmo indexador. Este processo funciona porque o mecanismo de alteração primeiramente apaga o trecho marcado referenciado e depois monta e insere o código existente em step.
- Durante a montagem do código que começa em step, o *label* Y (endereço de destino caso desvie) é resolvido em função da declaração do *label* existente no próprio trecho a ser inserido. Portanto, a cada inserção (montagem), o endereço relacionado ao *label* Y fica associado a um endereço diferente (na Fig. 13, esta distinção se faz alterando-se Y para os nomes Y′, Y′′ e Y′′′, instâncias diferentes de Y que representam diferentes endereços resolvidos durante as respectivas montagens).

É importante ressaltar que existem outras formas de resolver este problema por meio de auto-modificação de códigos. Outra possível abordagem consiste em gerar, em uma área de dados (por exemplo, iniciada no endereço calc), a seqüência de instruções de multiplicações necessárias para resolver o fatorial e, em seguida, inserir e executar este código.

Por exemplo, para calcular o fatorial de 3, este procedimento atribuiria o valor 1 para EAX. Em seguida, escreveria na área de dado apontada por calc, inicialmente vazia, o trecho de código descrito na Fig. 15.

```
Área de dados
Dado calc
// endereço calc aponta para o local
// onde está armazenada a primeira
// instrução (mul eax,3) do dado abaixo.
  mul eax,3  //eax ← 3*1
  mul eax,2  //eax ← 3*2
  mul eax,1  //eax ← 6*1
  endp       //endp fim de montagem
```

Fig. 15. Trecho de dados relacionado ao endereço calc, que calcula fatorial de 3.

Nesta abordagem, após o término do cálculo, o procedimento deve armazenar o resultado e realizar outra modificação para retornar o código do procedimento a sua forma original (similar ao feito na abordagem anterior), bem como limpar a área de dados apontada por calc.

REFERÊNCIAS

[1] H. Massalin, "Synthesis: An Efficient Implementation of Fundamental Operating System Services," PhD Thesis dissertion, faculty advisor by Calton Pu, Columbia University, New York, NY, 1992. Available: http://portal.acm.org/citation.cfm?id=143219

[2] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening Software Self-Checksumming via Self-Modifying Code," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 23-32, 2005.

[3] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," in *Information Hiding*, vol. 4437/2007, 2006, pp. 232-248, Springer-Verlag Berlin Heidelberg 2007.

[4] A. V. de Freitas and J. J. Neto, "Adaptive Device With Underlying Mechanism Defined By a Programming Language," in *Proceedings of the 4th WSEAS International Conference on Information Security*, pp.423-428, Tenerife, Spain, 2005.

[5] A. V. de Freitas and J. J. Neto, "Conception of adaptive programming languages," in *MS 2006: Proceedings of the 17th IASTED international conference on Modelling and simulation*, pp. 453-458, 2006, ACTA Press.

[6] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, nº 6, pp. 990-1034, 2006. Available: http://doi.acm.org/10.1145/1186632.1186634

[7] K. J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method* with *Propagation Patterns*, PWS Publishing Company, Boston, 1996.

[8] H. Pistori, "Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações," in Departamento de Computação e Sistemas Digitais (PCS) - Escola Politécnica, Dissertação de Doutorado, orientada por J. J. Neto, Universidade de São Paulo (USP), São Paulo, SP, 2003.

[9] J. J. Neto, "Um Levantamento da Evolução da Adaptatividade e da Tecnologia Adaptativa," *IEEE LATIN AMERICA TRANSACTIONS*, vol. 5, n.º 7, pp. 496-505, nov. 2007 (to appear).

[10] R. L. A. Rocha and J. J. Neto, "Autômato adaptativo, limites e complexidade em comparação com máquina de Turing," in *Proceedings of the second Congress of Logic Applied to Technology - LAPTEC'2000*, São Paulo: Faculdade SENAC de Ciências Exatas e Tecnologia, 2000.

[11] J. J. Neto and C. Bravo, "Adaptive Automata - a Revisited Proposal," in *Lecture Notes in Computer Science. J.M. Champarnaud, D. Maurel (Eds.): Implementation and Application of Automata 7th International Conference, CIAA 2002*, Vol.2608, pp. 158-168, 2003, Springer-Verlag Berlin Heidelberg 2003. Available: http://www.springerlink.com/content/x2b04jgeuvu431w0/

[12] R. Nair and J. E. Smith, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, 2005.

[13] J. J. Neto, "Adaptive Automata for Context-Sensitive Languages," *SIGPLAN NOTICES*, vol. 29, nº 9, pp. 115-124, 1994.Available: http://doi.acm.org/10.1145/185009.185033

[14] J. Aycock, "A Program Execution Model Based on Generative Dynamic Grammars," in *CST 2003: Proceedings of IASTED International Conference on Computer Science and Technology*, pp. 411-416, 2003, ACTA Press. Available: http://www.actapress.com/Abstract.aspx?paperId=14060

[15] J. W. Yoder and R. Johnson, "The Adaptive Object-Model Architectural Style," in *Proceedings of the IFIP 17th World Computer Congress - Tc2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, Deventer*, The Netherlands, pp. 3-27, 25 - 30, Aug., 2002.

[16] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation,* Prentice-Hall, Inc., 1981.

[17] M. K. Iwai, "Um formalismo gramatical adaptativo para linguagens dependentes de contexto," in Departamento de Computação e Sistemas Digitais (PCS) - Escola Politécnica, Dissertação de Doutorado, orientada por J. J. Neto, Universidade de São Paulo (USP), São Paulo, SP, 2000.

[18] J. L. Hennessy and D. A. Patterson, A*rquitetura de Computadores: Uma Abordagem Quantitativa*, 3ª ed: Editora Campus, 2003.

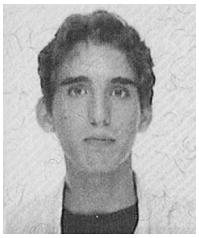
[19] A. V. de Freitas and J. J. Neto, "Linguagens de Programação aderentes ao Paradigma Adaptativo," *IEEE LATIN AMERICA TRANSACTIONS*, vol. 5, n.º 7, pp. 522-526, nov. 2007 (to appear).

[20] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2A: Instruction Set Reference, A-M," 2006. Available: http://www.intel.com/design/processor/manuals/253666.pdf

[21] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2B: Instruction Set Reference, N-Z," 2006. Available: http://www.intel.com/design/processor/manuals/253667.pdf

[22] C. A. B. Pariente, "Gramáticas Livres de Contexto Adaptativas com Verificação de Aparência," in Departamento de Computação e Sistemas Digitais (PCS) - Escola Politécnica, Dissertação de Doutorado, orientada por J. J. Neto, Universidade de São Paulo (USP), São Paulo, SP, 2004.

[23] J. J. Neto and M. E. Magalhães, "Reconhecedores Sintáticos - Uma Alternativa Didática para Uso em Cursos de Engenharia," presented at XIV Congresso Nacional de Informática, São Paulo, 1981.



Éder José Pelegrini graduado em Engenharia de Eletricidade (2005) pela Escola Politécnica da Universidade de São Paulo.

Atualmente, é aluno do curso de mestrado da Escola Politécnica da Universidade de São Paulo, sendo orientado pelo professor João José Neto, coordenador do LTA - Laboratório de Linguagens e Tecnologia Adaptativa do PCS - Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP. Tem experiência e interesse nos seguintes temas: dispositivos adaptativos, tecnologia adaptativa, autômatos adaptativos, linguagens adaptativas e analise de desempenho.

João José Neto graduado em Engenharia de Eletricidade (1971), mestrado em Engenharia Elétrica (1975) e doutorado em Engenharia Elétrica (1980), e livre-docência (1993) pela Escola Politécnica da Universidade de São Paulo.

Atualmente é professor associado da Escola Politécnica da Universidade de São Paulo, e coordena o LTA - Laboratório de Linguagens e Tecnologia Adaptativa do PCS - Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP. Tem experiência na área de Ciência da Computação, com ênfase nos Fundamentos da Engenharia da Computação, atuando principalmente nos seguintes temas: dispositivos adaptativos, tecnologia adaptativa, autômatos adaptativos, e em suas aplicações à Engenharia de Computação, particularmente em sistemas de tomada de decisão adaptativa, análise e processamento de linguagens naturais, construção de compiladores, robótica, ensino assistido por computador, modelagem de sistemas inteligentes, processos de aprendizagem automática e inferências baseadas em tecnologia adaptativa.