Estudo preliminar sobre a aplicação da adaptatividade em linguagens de programação imperativas

D. Queiroz and R. L. A. Rocha

***Abstract*— The later decades emerged a subtle interest in the production of self-modifying code in order to achieve a higher level of protection over the source code. Besides hiding the internals of a program, the adaptivity can be seen as a new resource to programmers to develop software that can rearrange its instructions or even perform new tasks. Based on this assumption, several works propose extensions to current programming languages to facilitate the writing of adaptive code. This work represents a preliminary survey of proposals of imperative programming languages suitable for adaptive programming, presenting a comparison between two forms of implementation.**

***Keywords*— Programming languages. Adaptive programming. Imperative programming.**

I. Introdução

Sob uma perspectiva generalista, o atributo "*adaptativo*" é dado a tudo àquilo que possua a capacidade de se adaptar, isto é, de se ajustar para atender novas necessidades ou desempenhar novas funções. Tal adaptação geralmente é realizada com base em regras pré-definidas que atuam como resposta a novas informações obtidas por algum elemento sensorial do objeto adaptativo.

No contexto apresentado por esse trabalho, entende-se por *programação adaptativa* a prática de construir programas com capacidade adaptativa, isto é, programas com mecanismos que alterem seu comportamento inicial ou *programas adaptativos*. Tal alteração de comportamento pode ser realizada através da modificação do (i) fluxo de execução do programa; ou (ii) das instruções definidas em sua listagem original. Essa definição é particularmente relevante devido às divergências pelas quais a literatura categoriza trabalhos que aderem ao modelo de adaptatividade proposto por José Neto [1].

Nessa interpretação, uma linguagem é considerada adequada ao desenvolvimento de programação adaptativa se ela possui recursos que corroboram com a elaboração de mecanismos que alteram o comportamento do programa.

Considerando que a introdução da adaptatividade não aumenta, em sua teoria, a classe de expressividade do programa (dado a sua equivalência com máquinas de Turing [2]), é possível que um programa adaptativo seja transformado tanto em código adaptativo, que efetivamente produz as alterações programadas na linguagem para obter o resultado desejado; como em código não-adaptativo, que sem a necessidade de alterar seu comportamento realiza o mesmo processamento sobre um objeto e, consequentemente, produz o mesmo resultado que o código adaptativo.

Dessa forma, esse trabalho tem como objetivo analisar duas formas de introdução de mecanismos adaptativos em linguagens de programação imperativas, de forma a promover a programação adaptativa.

II. Programação adaptativa

Alguns estudos mostram que há necessidade que as linguagens de programação possuam meios de controlar, não apenas o resultado do processamento de um programa, mas também a sua estrutura e o seu fluxo de execução. Com isso, alguns trabalhos surgiram propondo linguagens de programação de alto nível que tentam satisfazer essa necessidade.

Segundo Anckaert et al. [3], desde a década de 80 a programação adaptativa foi utilizada para ocultar características internas de programas, citando jogos virtuais onde instruções eram produzidas na memória após a inicialização do programa para evitar que engenharia reversa indesejada comprometesse seus mecanismos de proteção contra cópia não autorizada.

Além desse tipo de uso, inúmeras outras tarefas podem ser realizadas através de programação adaptativa: um programa pode, por exemplo, persistir dados entre suas execuções alterando em disco o código de inicialização de suas variáveis internas.

Dada essa difusão, convém avaliar o quão expressivas e poderosas são as linguagens de programação que se apresentam adequadas a essa necessidade. Nas próximas seções apresentaremos algumas propostas de linguagens de programação e a forma como elas se introduzem à programação adaptativa.

III. Programação adaptativa baseada em blocos

Uma das dificuldades que é inerente à criação de programação adaptativa está na necessidade de garantir a coerência daquilo que será efetivamente executado pelo programa (do ponto de vista semântico). Muitas vezes a alteração da sequência de execução de um código não é factível, pois existe dependência da ordem de execução do código.

Para lidar com esse problema, uma proposta de adaptatividade é a de descrever blocos de código que executem operações autocontidas, exercendo a adaptatividade sobre o fluxo de execução desses blocos.

D. Queiroz, Escola Politécnica da Universidade de São Paulo (USP), São Paulo, SP, Brasil, diego.queiroz@usp.br

R. L. A. Rocha, Escola Politécnica da Universidade de São Paulo (USP), São Paulo, SP, Brasil, rlarocha@usp.br

Essa forma de organização dos programas remete a estrutura dos autômatos adaptativos, onde cada bloco de código representa um estado e as condições que determinam o fluxo de execução entre esses blocos são as transições do autômato. Assim como o autômato, ações adaptativas podem estar ou não associadas a essas transições, de modo a permitir a alteração do fluxo original de execução, criação de novos blocos ou a modificação/substituição dos blocos existentes.

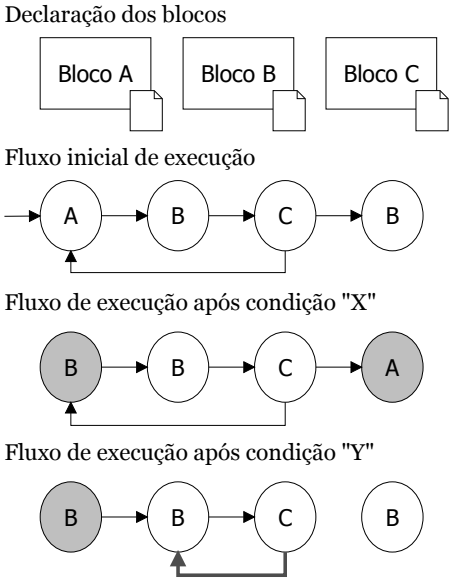

Figura 1. Organização de um programa adaptativo baseado em blocos.

Nessas linguagens, pontos do programa são marcados (através de rótulos ou numeração das instruções) de forma que seja possível indicar a próxima instrução a ser executada, mesmo que essa não esteja no fluxo usual de execução do código.

Tal alteração no fluxo de execução de um programa é facilmente obtida em linguagens de programação cuja construção esteja próxima da linguagem da máquina, como o Assembly, por exemplo, onde o fluxo de execução é determinado por saltos condicionais e o processamento dos dados é realizado por intermédio de "variáveis globais", isto é, os registradores.

No entanto, as linguagens de programação populares, como o C/C++ e Java, restringem essa forma de programação com suas construções estruturadas livres de desvios arbitrários e variáveis dependentes de escopo (locais). Em suma, não é coerente, do ponto de vista dessas linguagens, desviar o fluxo de execução de uma sub-rotina para outra sub-rotina sem questionar os efeitos dessa transição (o que ocorre com os dados?).

Para contornar essa limitação, Sabaliauskas e Rocha [4] propuseram uma extensão da linguagem de programação Oberon que aplica o conceito de programação adaptativa em blocos através do uso de sub-rotinas adaptativas. Na definição apresentada, a sub-rotina adaptativa é formada por blocos de código nomeados, chamados de *fragmentos*. Dessa forma, cada sub-rotina adaptativa possui seu escopo de variáveis, de modo que todos os fragmentos possam compartilhá-lo (as variáveis são declaradas fora dos fragmentos). A Figura 2 apresenta uma implementação da função fatorial definida nessa linguagem.

```
ADAPTIVE fatorial(n: INTEGER): INTEGER;
VAR
  result: INTEGER;

FRAGMENT init;
  result := 1
END init;

FRAGMENT calc;
  result := result * n;
  n := n - 1
END calc;

ACTIONS
  ACTION diferenteZero;
    AFTER calc
      RETURN result CASE n = 0,
      GOTO calc OTHERWISE
  END diferenteZero;

CONNECTIONS
  AFTER init
    RETURN result CASE n = 0,
    PERFORM diferenteZero OTHERWISE

END fatorial;
```

Figura 2. Exemplo de função fatorial desenvolvida na linguagem proposta por Sabaliauskas e Rocha [4].

```
ADAPTIVE MAIN [ NAME=nome, ENTRY=ini, EXIT=fim ]
IS

  CODE init  : <
     ...
  >;
  CODE calc  : <
     ...
  >;
  CODE fim   : < >;

  CONNECTION FROM ini (
    CASE 0 : TO fim
    OTHERWISE TO calc
      PERFORM adapt(ini) BEFORE
  );

  ADAPTIVE FUNCTION adapt(x)[ GENERATORS g ]: {

    REMOVE [ CONNECTION FROM x TO fim ];

    INSERT [ CODE g <
                ...
             > ];

    INSERT [ CONNECTION FROM x (
               CASE 0 : TO calc
               OTHERWISE TO ini
             ) ];
  }

END MAIN
```

Figura 3. Estrutura e sintaxe de um programa escrito na linguagem proposta por Silva e José Neto [5][6].

Após a definição dos blocos, são declaradas regras que podem ser aplicadas durante a execução da sub-rotina com o intuito de redefinir o fluxo de execução dos blocos declarados. Por fim, são estabelecidas as conexões iniciais entre os blocos declarados. As expressões condicionais que definem as regras de transição dessa linguagem (tanto das conexões iniciais quanto das ações adaptativas) também podem fazer uso das variáveis dentro do escopo da sub-rotina ou de seus parâmetros.

Embora não exista uma implementação dessa linguagem para utilização, ela nos permite avaliar uma possível forma de estruturar o código para programação adaptativa.

Silva e José Neto [5][6] propuseram uma abordagem semelhante, com a ideia de uma linguagem de alto nível que pudesse acoplar qualquer outra linguagem imperativa. Nesse trabalho é proposta uma estrutura independente de linguagem que define blocos nomeados de código. Esses blocos carregariam, a princípio, código escrito em uma linguagem qualquer, chamada de *linguagem hospedeira*. Assim, de forma semelhante ao trabalho de Sabaliauskas e Rocha, são definidas as regras de transição entre os blocos (conexões) e as ações adaptativas associadas a essas transições. A Figura 3 apresenta a sintaxe e a estrutura do código dessa linguagem de programação.

O grande diferencial nessa definição é que, além da capacidade de alterar a forma de transição entre os blocos previamente definidos, ela permite a criação de novos blocos de código através de construções dinâmicas associadas às ações adaptativas, chamadas de *geradores*. Assim, é possível conectar um bloco existente a código recém-gerado através da referência ao seu gerador.

IV. Programação adaptativa baseada em recomputação

Outra forma de lidar com as alterações no programa é determinar previamente quais elementos do código estão sujeitos a modificações, de forma que ações sejam tomadas com base nessas modificações.

Para exemplificar essa ideia, Hammer et al. propuseram uma nova linguagem de programação baseada na linguagem C, chamada CEAL [7]. Nessa proposta é introduzido o conceito de *referência modificável*, que nada mais é do que uma posição de memória cujo conteúdo pode ser lido e alterado, semelhante a um ponteiro. No entanto, ao invés de serem diretamente acessadas, a construção da linguagem exige que o conteúdo armazenado em uma referência modificável seja acessado por intermédio das funções read(modref_t) e write(modref_t, void).

Assim, sempre que uma referência modificável possui seu valor atualizado, todos os cálculos e processamentos realizados com base no valor anterior daquela referência são automaticamente recomputados para refletir suas alterações.

Para tanto, as sub-rotinas que fazem uso dessas referências precisam ser declaradas com a palavra reservada "*ceal*" (Figura 4), indicando que seu conteúdo deve ser processado novamente caso alguma das referências modificáveis utilizadas pela rotina tenham sido alteradas desde sua última execução.

A Figura 5 apresenta uma sequência de instruções que exemplifica o funcionamento da linguagem CEAL: nas duas primeiras linhas, a referência modificável "x" é inicializada. Na linha 3, a sub-rotina "adapt" é invocada passando a referência modificável "x" como parâmetro. Na linha 4, a referência "x" é alterada e, por fim, na linha 5, a função propagate() é invocada, causando a recomputação da sub-rotina "adapt".

Como é possível observar, a chamada das sub-rotinas "ceal" não é feito diretamente, mas por intermédio da função run_core. De forma semelhante, a recomputação das sub-rotinas não é realizada automaticamente sempre que as referências modificáveis são alteradas, mas apenas após a chamada da função propagate().

Essa abordagem, apesar de diferir funcionalmente dos autômatos adaptativos ou da ideia de que o código adaptativo deve obrigatoriamente alterar sua estrutura de execução, traz a ideia de que o código precisa se ajustar automaticamente de acordo com a introdução de novas informações, redefinindo as estruturas de dados previamente produzidas.

```
ceal adapt(modref_t *param)  {

  // processa "param"

}
```

Figura 4. Definição de uma sub-rotina na linguagem CEAL.

```
1  modref_t x = modref();
2  write( x, obtemEntrada() );
3  run_core( adapt, x );
4  modify( x, obtemEntrada() );
5  propagate();
```

Figura 5. Exemplo de chamada de sub-rotina e propagação das modificações na linguagem CEAL.

V. Análise dos modelos de programação adaptativa

O modelo de programação adaptativa baseado em blocos satisfaz diretamente a ideia de programas auto modificáveis, por permitir diretamente que a estrutura do código seja remanejada pelo programador. Essa característica torna o modelo ideal quando há a necessidade prévia de determinar como o código produzido deve ser organizado (com o intuito de esconder algum recurso interno do programa, por exemplo).

Por outro lado, a liberdade dada ao programador permite que o programa resultante seja inconsistente (do ponto de vista semântico), devido a alterações não controladas que tenham sido produzidas no código original. Essa característica exige uma maior atenção do programador ou que limitadores de funcionalidade sejam impostos pela linguagem de programação com o propósito de evitar que inconsistências sejam acidentalmente produzidas.

Já o modelo de programação adaptativa baseado em recomputação tem a característica de fornecer um mecanismo de automação ao programa.

Para ilustrar a funcionalidade do recurso, pode-se considerar a atualização dos componentes visuais de uma aplicação gráfica qualquer, cuja invocação é frequentemente necessária para refletir as alterações realizadas pelo programa ao usuário. No modelo de programação não adaptativo, essa chamada geralmente é realizada por meio de eventos ou de consultas periódicas sobre o estado das variáveis.

Contudo, através da modelo baseado em recomputação seria possível, por exemplo, determinar que as rotinas de atualização fossem invocadas automaticamente sempre que os dados relacionados ao componente fossem modificados, sem a necessidade de criação de eventos explícitos para essa finalidade, tornando o modelo de programação reativo e auto ajustável. O

Quadro **1** apresenta uma síntese sobre os modelos de programação adaptativa apresentados nesse trabalho.

Quadro 1. Comparação entre os modelos de programação adaptativa baseado em blocos e baseado em recomputação.

	Blocos	**Recomputação**
Permite a geração de novos programas?	Sim	Não
Permite a alteração do fluxo de execução?	Sim	Não
Permite a persistência dos cálculos realizados?	Sim	Não
Resposta automática a modificações?	Não	Sim
Permite a geração de código inconsistente?	Sim	Não

VI. CONCLUSÃO

Esse trabalho representa um estudo preliminar sobre o projeto de linguagens de programação destinadas à programação adaptativa.

Em resumo, o modelo de programação adaptativa baseado em blocos fornece ao programador uma maior liberdade de estabelecer a forma como o programa deve interagir consigo mesmo para obter o processamento desejado, assim como também fornece uma maior clareza sobre o código que deve ser produzido caso o processo de compilação procure traduzir as alterações no código em alterações reais do programa durante a sua execução.

Por outro lado, o modelo de programação adaptativa baseado em recomputação, apesar de não fornecer recursos para alterar o código ou o fluxo de execução original do programa, fornece uma nova camada de abstração sobre o programa, permitindo que estruturas de dados previamente computadas respondam automaticamente a alterações de seus parâmetros sem a intervenção direta do programa.

Por fim, apesar dessa análise comparativa, é preciso salientar que os modelos de adaptatividade apresentados nesse trabalho não são auto excludentes, isto é, é possível que uma linguagem incorpore os dois modelos de programação simultaneamente, embora os autores não tenham conhecimento de nenhum trabalho já realizado que proponha tal fusão.

Em trabalhos futuros pretende-se adicionar a esse estudo diferentes tipos de abordagem a essa forma de programação, não se limitando apenas ao paradigma imperativo.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da CAPES, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil.

REFERÊNCIAS

[1] J. José Neto, "Contribuições à metodologia de construção de compiladores," Universidade de São Paulo, São Paulo, 1993.

[2] R. L. de A. da Rocha and J. José Neto, "Autômato Adaptativo, limites e complexidade em comparação com a Máquina de Turing," in Proceedings of the second Congress of Logic Applied to Technology (LAPTEC'2000), 2001, vol. 1, pp. 33–48.

[3] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," Inf. Hiding, vol. 4437, no. 1, pp. 232–248, 2007.

[4] J. A. Sabaliauskas and R. L. de A. da Rocha, "Project and Implementation for a Programming Language Suitable to Express Adaptive Algorithms," IEEE Lat. Am. Trans., vol. 9, no. 6, pp. 969–973, Oct. 2011.

[5] S. R. B. da Silva and J. José Neto, "Proposal of a High-Level Language for Writing Self Modifying Programs," IEEE Lat. Am. Trans., vol. 9, no. 2, pp. 192–198, Apr. 2011.

[6] S. R. B. da Silva, "Software adaptativo: método de projeto, representação gráfica e implementação de linguagem de programação," Universidade de São Paulo, 2011.

[7] M. Hammer, U. A. Acar, and Y. Chen, "CEAL: a C-based language for self-adjusting computation," ACM Sigplan Not., pp. 25–37, 2009.

Diego Queiroz é graduado em Ciência da Computação pela Universidade Nove de Julho e Mestre em Ciências pela Escola Politécnica da Universidade de São Paulo. Atualmente atua no Laboratório de Linguagens e Técnicas Adaptativas como aluno de doutorado.

Ricardo L. A. Rocha é natural do Rio de Janeiro-RJ e nasceu em 29/05/1960. Graduou-se em Engenharia Elétrica modalidade Eletrônica na PUC-RJ, em 1982. É Mestre e Doutor em Engenharia de Computação pela Escola Politécnica da USP (1995 e 2000, respectivamente). Suas áreas de atuação incluem Tecnologias Adaptativas, Fundamentos de Computação e Modelos Computacionais. Dr. Rocha é membro da ACM, IEEE e SBC.