Utilizando linguagens de programação orientadas a objetos para codificar programas adaptativos

P. R. M. Cereda e J. José Neto

***Abstract*—This paper presents concepts and techniques for writing adaptive code using a normal object-oriented programming language as reference. We aim at making adaptivity accessible to users via well-known object-oriented constructs, as a feasible alternative to the traditional development approaches.**

***Palavras-chave:*—Adaptatividade, linguagens de programação, orientação a objetos**

I. Introdução

Adaptatividade é o termo utilizado para denotar a capacidade de um dispositivo em modificar seu próprio comportamento, sem a interferência de agentes externos [1], [2]. A modificação espontânea ocorre em resposta ao histórico de operação e aos dados de entrada, resultando em comportamentos distintos ao longo do tempo [3]. A adaptatividade é adicionada como uma extensão aos formalismos já consolidados, aumentando seu poder de expressão [4].

De modo particular, a adaptatividade tem sido associada com sucesso às linguagens de programação, como pode ser observado em diversas publicações [5], [6], [7], [8], [9], [10]. Em [6], Freitas e José Neto, inspirados em um trabalho anterior de Rocha e José Neto [5], iniciaram o desenvolvimento de linguagens de programação orientadas a adaptatividade. Os autores enfatizam a necessidade de uma mudança de postura em relação às iniciativas existentes para o tratamento do fenômeno da adaptatividade juntamente com um estilo de programação. Linguagens de programação estendidas com construtos adaptativos permitem um fluxo de trabalho consistente para aplicações que requerem código auto-modificável em tempo de execução [6].

Um programa escrito em uma linguagem de programação com características adaptativas tem, no início de sua execução, um comportamento semelhante ao de um código não-adaptativo tradicional [7]. Tal programa inicia a partir de uma configuração de código inicial C_1 e evolui para configurações sucessivas $C_2 \ldots C_n$ no tempo através de ações adaptativas [1], conforme ilustra a Figura 1. A sequência de configurações é fortemente dependente dos dados de entrada; execuções do mesmo programa com dados diferentes consequentemente resultarão em sequências diferentes.

Inspirado nas recentes contribuições em linguagens de programação com características adaptativas e na expressão do próprio fenômeno da adaptatividade, este artigo apresenta conceitos e técnicas para a escrita de código adaptativo utilizando linguagens de programação orientadas a objetos convencionais. O atual estado da arte apresenta iniciativas baseadas em um subconjunto de código de montagem adaptativo [9] e especificações para linguagens de alto nível [8], [10]; este artigo contempla técnicas para a incorporação dos conceitos adaptativos em linguagens de programação orientadas a objetos já conhecidas.

Os autores podem ser contatados através dos seguintes endereços de correio eletrônico: paulo.cereda@usp.br e jjneto@usp.br.

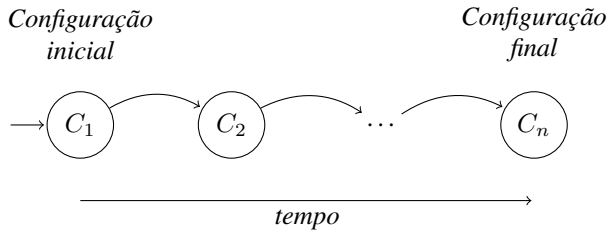


Figura 1. Evoluções sucessivas de código de um programa escrito em linguagem de programação com características adaptativas no tempo, através de ações adaptativas.

É importante observar que as técnicas apresentadas estão vinculadas ao estilo adaptativo e à própria adaptatividade. Em outras palavras, em algum nível, o desenvolvedor necessita entender como um código adaptativo funciona e como este código evoluirá entre configurações. Existe um trabalho em andamento para oferecer uma ferramenta gráfica com o objetivo de auxiliar o desenvolvedor na escrita de código adaptativo. A ideia é encapsular a adaptatividade em uma biblioteca de código, oferecendo uma interface fluente e transparente ao desenvolvedor, minimizando a necessidade de familiaridade com o estilo adaptativo como pré-requisito para a escrita de código adaptativo.

Este artigo está organizado da seguinte forma: a Seção II apresenta uma revisão da formulação geral para dispositivos adaptativos guiados por regras. A Seção III introduz duas técnicas para a escrita de código adaptativo, incluindo algumas discussões sobre implementação e trechos de código em Java e C#. A Seção IV apresenta um estudo de caso de duas implementações de um dispositivo adaptativo para o reconhecimento de cadeias na forma $a^n b^n c^n$, $n \in \mathbb{N}$, utilizando as técnicas apresentadas na seção anterior; vários testes são realizados nas implementações e os resultados são discutidos. As considerações finais são apresentadas na Seção V.

II. Dispositivos adaptativos

Esta seção apresenta uma revisão da formulação geral para dispositivos adaptativos guiados por regras, que são máquinas formais que permitem a alteração de seu comportamento, de modo dinâmico e espontâneo, como resposta a estímulos de entrada, sem a interferência de agentes externos.

A. Dispositivos guiados por regras

Um dispositivo guiado por regras é todo e qualquer dispositivo formal cujo comportamento depende, de forma exclusiva, de um conjunto finito de regras que definem o mapeamento entre configurações [11]. Em outras palavras, um dispositivo guiado por regras pode ser definido pela sêxtupla $ND = (C, NR, S, c_0, A, NA)$, onde ND é um dispositivo guiado por regras, C é o conjunto de todas as possíveis configurações, $c_0 \in C$ é a configuração inicial, S é o conjunto de todos os possíveis eventos que são estímulos de entrada do dispositivo, $\epsilon \in S$, $A \subseteq C$ é o subconjunto de todas as configurações de aceitação (da mesma forma, $F = C - A$ é o subconjunto das configurações de rejeição), NA é o conjunto de todos os possíveis símbolos de saída de ND como efeito da aplicação das regras do dispositivo, $\epsilon \in NA$, e NR é o conjunto de regras que definem ND através de uma relação $NR \subseteq C \times S \times C \times NA$. Uma regra $r \in NR$ tem a forma $r = (c_i, s, c_j, z)$, $c_i, c_j \in C$, $s \in S$ e $z \in NA$, indicando que, em resposta a um estímulo s, r muda a configuração corrente c_i para c_j, consome s e gera z como saída [11]. Uma regra $r = (c_i, s, c_j, z)$ é dita compatível com a configuração corrente c se, e somente se, $c_i = c$ e s é vazio ou igual ao estímulo de entrada corrente; neste caso, a aplicação da regra r move o dispositivo para a configuração c_j, denotada por $c_i \Rightarrow^s c_j$, e adiciona z ao fluxo de saída.

Um fluxo de estímulos de entrada $w = w_1 w_2 \dots w_n$, $w_k \in S - \{\epsilon\}$, $k = 1, \dots, n$, $n \geq 0$, é aceito por um dispositivo ND quando $c_0 \Rightarrow^{w_1} c_1 \Rightarrow^{w_2} \dots \Rightarrow^{w_n} c$ (de modo resumido, $c_0 \Rightarrow^w c$), e $c \in A$. Da mesma forma, w é rejeitado por ND quando $c \in F$. A linguagem descrita pelo dispositivo guiado por regras ND é representada pelo conjunto $L(ND) = \{w \in S^* \mid c_0 \Rightarrow^w c, c \in A\}$.

B. Dispositivos adaptativos guiados por regras

Um dispositivo guiado por regras $AD = (ND_0, AM)$ é considerado adaptativo sempre que, para todos os passos de operação $k \geq 0$ (k é o valor de um contador embutido T iniciado em zero e que é incrementado de uma unidade toda vez que uma ação adaptativa não nula é executada), AD segue o comportamento do dispositivo subjacente ND_k até que a execução de uma ação adaptativa não nula inicie o passo de operação $k + 1$ através de mudanças no conjunto de regras; em outras palavras, a execução uma ação adaptativa não nula em um passo de operação $k \geq 0$ faz o dispositivo adaptativo AD evoluir do dispositivo subjacente ND_k para ND_{k+1}.

O dispositivo adaptativo AD inicia sua operação na configuração c_0, com o formato inicial definido por $AD_0 = (C_0, AR_0, S, c_0, A, NA, BA, AA)$. No passo k, um estímulo de entrada move AD para uma configuração seguinte e inicia seu passo de operação $k + 1$ se, e somente se, uma ação não-adaptativa for executada; dessa forma, estando o dispositivo AD no passo k, com o formato $AD_k = (C_k, AR_k, S, c_k, A, NA, BA, AA)$, a execução de uma ação adaptativa não nula leva a $AD_{k+1} = (C_{k+1}, AR_{k+1}, S, c_{k+1}, A, NA, BA, AA)$, onde $AD = (ND_0, AM)$ é um dispositivo adaptativo com um dispositivo subjacente inicial ND_0 e um mecanismo adaptativo AM, ND_k é o dispositivo subjacente de AD no passo de operação k, NR_k é o conjunto de regras não adaptativas de ND_k, C_k é o conjunto de todas as configurações possíveis para ND no passo de operação k, $c_k \in C_k$ é a configuração inicial no passo k, S é o conjunto de todos os eventos possíveis que são estímulos de entrada para AD, $A \subseteq C$ é o subconjunto as configurações de aceitação (da mesma forma, $F = C - A$ é o subconjunto de configurações de rejeição), BA e AA são conjuntos de ações adaptativas (ambos contendo a ação nula, $\epsilon \in BA \cap AA$), NA, com $\epsilon \in NA$, é o conjunto de todos os possíveis símbolos de saída de AD como efeito da aplicação de regras do dispositivo, AR_k é o conjunto de regras adaptativas definido pela relação $AR_k \subseteq BA \times C \times S \times C \times NA \times AA$, com AR_0 definindo o comportamento inicial de AD, AR é o conjunto de todas as possíveis regras adaptativas para AD, NR é o conjunto de todas as possíveis regras não-adaptativas subjacentes de AD, e AM é o mecanismo adaptativo, $AM \subseteq BA \times NR \times AA$, a ser aplicado em um passo de operação k para cada regra em $NR_k \subseteq NR$. Regras adaptativas $ar \in AR_k$ são da forma $ar = (ba, c_i, s, c_j, z, aa)$ indicando que, em resposta a um estímulo de entrada $s \in S$, ar inicialmente executa a ação adaptativa anterior $ba \in BA$; a execução de ba é cancelada se esta elimina ar do conjunto AR_k; caso contrário, a regra não-adaptativa subjacente $nr = (c_i, s, c_j, z)$, $nr \in NR_k$ é aplicada e, finalmente, a ação adaptativa posterior $aa \in AA$ é executada [11].

Ações adaptativas podem ser definidas em termos de abstrações chamadas funções adaptativas, de modo similar às chamadas de funções em linguagens de programação usuais [11]. A especificação de uma função adaptativa deve incluir os seguintes elementos: *(a)* um nome simbólico, *(b)* parâmetros formais que referenciarão valores passados como argumentos, *(c)* variáveis que conterão valores de uma aplicação de uma ação elementar de inspeção, *(d)* geradores que referenciam valores novos a cada utilização, e *(e)* o corpo da função propriamente dita.

São definidos três tipos de ações adaptativas elementares que realizam testes nas regras ou modificam regras existentes, a saber:

1) *ação adaptativa elementar de inspeção:* a ação não modifica o conjunto de regras, mas permite a inspeção deste para a verificação de regras que obedeçam um determinado padrão.
2) *ação adaptativa elementar de remoção:* a ação remove regras que correspondem a um determinado padrão do conjunto corrente de regras.
3) *ação adaptativa elementar de inclusão:* a ação insere uma regra que corresponde a um determinado padrão no conjunto corrente de regras.

Tais ações adaptativas elementares podem ser utilizadas no corpo de uma função adaptativa, incluindo padrões de regras que utilizem parâmetros formais, variáveis e geradores disponíveis.

Como exemplo, considere dispositivo adaptativo, mais precisamente um autômato adaptativo M, que reconhece cadeias pertencentes à linguagem $L(M) = \{w \in \{a, b\}^* \mid w =$

$a^n b^n, n \in \mathbb{N}, n \geq 1\}$, apresentado na Figura 2. A função adaptativa $\mathcal{F}$ é apresentada no Algoritmo 1.

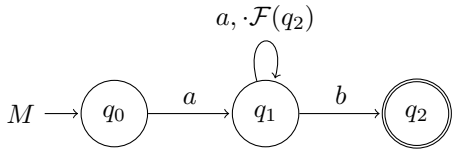


Figura 2. Autômato adaptativo M que reconhece cadeias na forma $a^n b^n$, com $n \in \mathbb{N}$, $n \geq 1$.

Algoritmo 1 Função adaptativa $\mathcal{F}(p)$

função adaptativa $\mathcal{F}(p)$
variáveis: $?x$
geradores: $g^\star$
// procura pelo estado que possui uma transição
// consumindo o símbolo b até p e remove essa transição
$?(?x, b) \rightarrow p$
$-(?x, b) \rightarrow p$
// remove o laço existente em q_1
$-(q_1, a) \rightarrow q_1, \cdot\mathcal{F}(p)$
// adiciona as novas transições
$+(?x, b) \rightarrow g^\star$
$+(g^\star, b) \rightarrow p$
// adiciona um novo laço em q_1
$+(q_1, a) \rightarrow q_1, \cdot\mathcal{F}(g^\star)$
fim da função adaptativa

Durante o processo de reconhecimento de uma cadeia w, M sofrerá alterações em sua topologia para acomodar mais símbolos. No caso da linguagem $L(M)$, a cada ocorrência de um símbolo a adicional (indicada por um laço em q_1), a função adaptativa posterior $\mathcal{F}$ será executada para que M possa receber um símbolo b adicional. A Figura 3 ilustra o autômato M resultante após o reconhecimento da cadeia $w = aabb$.

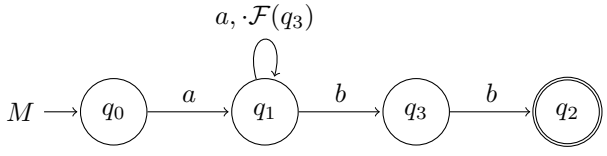


Figura 3. Autômato adaptativo M resultante após o reconhecimento da cadeia $w = aabb$.

É importante destacar que o mecanismo adaptativo pode ser visto como uma simples extensão do dispositivo não-adaptativo subjacente, o que preserva a integridade e as propriedades deste [11].

III. Técnicas para a escrita de código adaptativo

Esta seção apresenta duas técnicas para a escrita de código adaptativo utilizando uma linguagem de programação como referência. As linguagens Java e C# foram escolhidas para fins de ilustração, mas é importante mencionar que qualquer linguagem de programação pode ser utilizada, desde que alguns requisitos mínimos estejam presentes nas especificações da linguagem escolhida.

A. Reflexão estrutural

A primeira técnica para a escrita de código adaptativo explora reflexão, uma característica que permite a análise e modificação de classes em tempo de execução. Todas as informações sobre uma classe, incluindo seus métodos e atributos, estão disponíveis como metadados que são preservados durante a compilação. Considere um exemplo de uma classe simples, apresentada na Figura 4.

Classe em Java

```
public class Bar {

  private int a;
  private String b;
  private double c;

  public int sum(int n, int m) {
    return n + m;
  }
}
```

Classe em C#

```
public class Bar {

  private int a;
  private string b;
  private double c;

  public int sum(int n, int m) {
    return n + m;
  }
}
```

Figura 4. Exemplo de uma classe.

É possível escrever um trecho de código que, utilizando reflexão, analisa a classe `Bar` da Figura 4 e retorna todas as informações sobre seus atributos e métodos, como visto na Figura 5. É importante observar que tal análise da classe ocorre em tempo de execução.

A execução do trecho de código apresentado na Figura 5 retorna uma lista de atributos e métodos definidos na classe `Bar` (Figura 4) através de classes utilitárias que oferecem reflexão para as duas linguagens.

A possibilidade de uma linguagem de programação usar reflexão permite a construção de programas auto-modificáveis, isto é, programas capazes de alterar seu próprio código. Entretanto, a maioria das linguagens de programação, incluindo Java e C#, apresentam restrições em suas classes utilitárias de reflexão: apenas ações de introspecção são permitidas, isto é, a capacidade de analisar estruturas [12], [13], [14]. Assim, diversas extensões para linguagens foram propostas para oferecer suporte à reflexão comportamental – interceptar uma operação tal como uma invocação de método e mudar o

Reflexão em Java

```
for (Field field :
      Bar.class.
      getDeclaredFields()) {
  System.out.println(field.
      toString());
}
for (Method method :
      Bar.class.
      getDeclaredMethods()) {
  System.out.println(method.
      toString());
}
```

Reflexão em C#

```
Type type = typeof(Bar);
foreach (FieldInfo field in
      type.GetFields(
      BindingFlags.NonPublic |
      BindingFlags.Instance)) {
  Console.WriteLine(field);
}
foreach (MethodInfo method in
      type.GetMethods()) {
  System.Console.
      WriteLine(method);
}
```

Figura 5. Trecho de código que analisa a classe `Bar` e retorna todas as informações sobre seus atributos e métodos.

comportamento dessa operação [15] – e à reflexão estrutural – oferecer a possibilidade de modificar estruturas de dados [16].

A seguir, é apresentada uma iniciativa interessante do uso de reflexão estrutural em Java através da biblioteca Javassist [17], [16], [18]. Tal característica é oferecida através de uma transformação de bytecode que pode acontecer tanto em tempo de compilação quanto em tempo de carga. As modificações não são permitidas em tempo de execução por causa de uma limitação da linguagem: uma vez que uma classe é instanciada na memória, todas as tentativas de modificar sua estrutura lançarão uma exceção [12], [13]. Entretanto, é possível utilizar o conceito de herança do paradigma de programação orientada a objetos para estender a classe-alvo (já instanciada), alterar a estrutura da nova classe em tempo de execução, instanciá-la e definir uma nova referência para a variável contendo o objeto ascendente original, como ilustrado na Figura 6. É importante mencionar que o mesmo conceito utilizado na linguagem Java aplica-se para a linguagem C#.

De acordo com a Figura 6, todas as modificações são realizadas na nova classe, tendo a classe original como sua antecedente. Agora é possível retornar um novo objeto da nova classe que é compatível com as especificações originais, mas seus atributos e métodos podem estar modificados.

A Figura 7 ilustra a criação de uma nova classe, `Baz`, que herda da classe original `Bar` (Figura 4), e sua modificação posterior, adicionando um novo atributo. A biblioteca Javassist [17], [16], [18] foi utilizada para facilitar a modificação

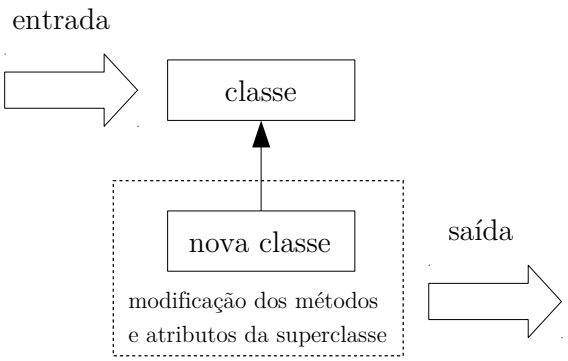


Figura 6. Estendendo a classe-alvo e alterando a nova classe.

de código.

```
ClassPool pool = ClassPool.getDefault();
CtClass clazz = pool.get(Bar.class.
    getCanonicalName());
CtClass newClazz = pool.makeClass("Baz");
newClazz.setSuperclass(clazz);
newClazz.addField(new CtField(pool.get(
    Float.class.getCanonicalName()),
     "d", newClazz));
newClazz.toClass();
Class createdClazz = Class.
    forName("Baz");
```

Figura 7. Criação de uma nova classe `Baz` através de herança e reflexão estrutural.

Como visto na Figura 7, através de herança, `Baz` tem todos os atributos e métodos da classe original `Baz` com a adição de um novo atributo de ponto flutuante `d`. É possível repetir a extensão de classes exaustivamente através da modificação de atributos e métodos, quando apropriado.

A reflexão estrutural é uma característica interessante para o desenvolvimento de código adaptativo, pois permite modificações em tempo de execução. Entretanto, uma deficiência do uso de reflexão é o tempo de execução e consumo de memória consideráveis devido à ausência de otimizações do compilador – código injetado via reflexão não existe em tempo de compilação – eao fato de que cada modificação simples requer descoberta, isto é, toda a estrutura de classe requer uma inspeção completa de modo a encontrar quais são os pontos modificáveis. Além disso, quando existe a utilização de bibliotecas de manipulação de bytecode, como Javassist, deve-se considerar também o processo de reconstrução de bytecode ao criar novas classes.

B. Blocos de código

Como uma alternativa à reflexão estrutural e suas deficiências, a segunda técnica aplica um conceito utilizado por simuladores e máquinas virtuais: um conjunto de blocos de código, conectados por ligações simbólicas, e um executor

que roda cada bloco, um por vez. Linguagens de programação modernas possuem suporte a threading, um componente utilitário que permite a representação de blocos de código. A motivação para as interfaces de threading surge da necessidade de oferecer um protocolo comum para objetos executarem um determinado trecho de código durante seus ciclos de vida. A Figura 8 ilustra um esboço da estrutura de execução de um programa implementado com blocos de código.

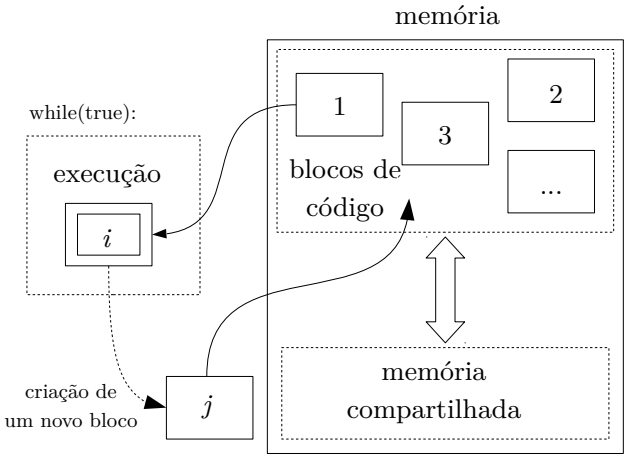


Figura 8. Esboço da estrutura de execução de um programa implementado com blocos de código.

A Figura 8 esboça uma execução típica de um programa implementado com blocos de código, consistindo de um laço principal e a execução sequencial de cada bloco até que uma instrução de parada seja encontrada. Uma possível implementação de um bloco de código é apresentada na Figura 9. Para o escopo do artigo, optou-se pela implementação da interface `Runnable` em Java e a utilização da classe `Thread` com `delegate` em C#.

Um bloco de código em Java

```
Runnable r = new Runnable() {
  public void run() {
    // implementation
  }
};
```

Um bloco de código em C#

```
Thread t = new Thread(
  delegate() {
    // implementation
  }
);
```

Figura 9. Uma possível implementação de um bloco de código.

Uma implementação mais sofisticada de um bloco de código em Java e sua utilização é ilustrada na Figura 10. A classe `CodeBlock` implementa parcialmente a interface `Runnable` e define um mapa de memória e um identificador único para distinguir o bloco corrente dos outros. A implementação completa de um bloco de código ocorre no programa principal, através da especificação do corpo do método `run()`.

```
public abstract class CodeBlock
     implements Runnable {
  private HashMap<String,
     Object> memory;
  private String identifier;
  public abstract void run();
  public CodeBlock(
       HashMap<String,
       Object> memory,
       String identifier) {
    this.memory = memory;
    this.identifier = identifier;
  }
...
}
...
CodeBlock q0 = new CodeBlock(
     memory, "q0") {
  @Override
  public void run() {
    System.out.
        println("Hello world!");
    }
};
```

Figura 10. Uma implementação mais sofisticada de um bloco de código em Java e sua utilização.

O bloco de código `q0`, como visto na Figura 10, simplesmente imprime uma mensagem de saudação no terminal. É importante notar que blocos de código podem conter definições aninhadas de outros blocos de código em suas especificações. Assim, novos blocos podem ser criados e novas ligações dinâmicas podem ser estabelecidas.

Usar blocos de código na representação de componentes de um programa adaptativo é uma solução viável para implementar características adaptativas em linguagens de programação que suportam threading. Entretanto, é importante observar que tal solução requer um alto nível de especificação do que seu equivalente reflexivo, isto é, blocos de código são definidos com operações básicas ao invés de métodos completos, podendo gerar uma quantidade considerável de blocos para representar um algoritmo.

IV. Experimentos

Esta seção apresenta um estudo de caso consistindo em duas implementações de um dispositivo adaptativo, mais precisamente um autômato adaptativo, que reconhece cadeias na forma $a^n b^n c^n$, com $n \in \mathbb{N}$, $n \geq 1$ [1], ilustrado na Figura 11. A função adaptativa $\mathcal{A}$ presente em M é apresentada no Algoritmo 2. A primeira implementação utiliza reflexão estrutural para realizar as ações adaptativas, disparadas por uma função adaptativa definida no conjunto de regras do dispositivo. A segunda implementação usa blocos de código para representar cada estado do dispositivo, e ações adaptativas são disparadas através da criação de novos blocos e da modificação das ligações dinâmicas existentes entre blocos em tempo de execução.

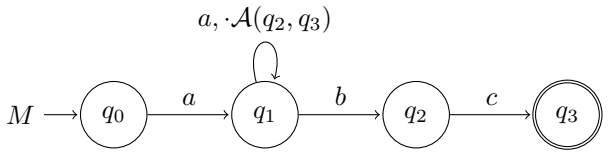
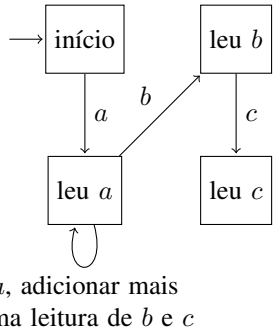


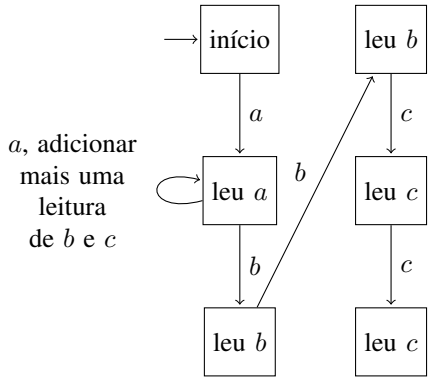
Figura 11. Autômato adaptativo M que reconhece cadeias na forma $a^n b^n c^n$, com $n \in \mathbb{N}$, $n \geq 1$.

Algoritmo 2 Função adaptativa $\mathcal{A}(p_1, p_2)$

função adaptativa $\mathcal{A}(p_1, p_2)$
variáveis: $?x, ?y$
geradores: $g_1^\star, g_2^\star$
// procura pelo estado que possui uma transição
// consumindo o símbolo b até p_1 e remove essa transição
$?(?x, b) \rightarrow p_1$
$-(?x, b) \rightarrow p_1$
// procura pelo estado que possui uma transição
// consumindo o símbolo c até p_2 e remove essa transição
$?(?y, c) \rightarrow p_2$
$-(?y, c) \rightarrow p_2$
// remove o laço existente em q_1
$-(q_1, a) \rightarrow q_1, \cdot\mathcal{A}(p_1, p_2)$
// adiciona as novas transições
$+(?x, b) \rightarrow g_1^\star$
$+(g_1^\star, b) \rightarrow p_1$
$+(?y, c) \rightarrow g_2^\star$
$+(g_2^\star, c) \rightarrow p_2$
// adiciona um novo laço em q_1
$+(q_1, a) \rightarrow q_1, \cdot\mathcal{A}(g_1^\star, g_2^\star)$
fim da função adaptativa



Considere o reconhecimento da cadeia $aabbcc$. Ao ler o segundo a da cadeia, o programa realizará modificações para acomodar novas situações de leitura de símbolos.



Após a leitura do segundo símbolo a da cadeia, o programa adicionou duas leituras em seu fluxo de execução. Caso mais um símbolo a seja lido, o programa repetirá a adição de mais leituras.

Figura 12. Lógica do programa de reconhecimento de cadeias na forma $a^n b^n c^n$, com $n \in \mathbb{N}$, $n \geq 1$.

As duas implementações propostas seguem a lógica apresentada na Figura 12: quando há a ocorrência de um símbolo a adicional, o programa sofrerá modificações em seu fluxo de execução para acomodar as novas situações de leitura de símbolos.

Dois experimentos foram executados para cada implementação: o primeiro consistiu na medição, em milissegundos, do tempo de reconhecimento de cada cadeia w, $w = a^n b^n c^n$, com $1 \leq n \leq 310$, durante 1000 iterações. O segundo experimento consistiu na medição, em bytes, do tamanho de objeto de cada componente implementando o motor adaptativo, e o consumo de memória para cada cadeia w, $w = a^n b^n c^n$, com $1 \leq n \leq 310$, também durante 1000 iterações. Os experimentos foram realizados em um ambiente Linux de 64 bits, utilizando uma máquina virtual Oracle JRockit. Cada valor de medição obtido a partir de cada iteração foi coletado para um processamento posterior, com a geração de estatísticas.

Os resultados do primeiro experimento são sintetizados na Figura 13. O programa reflexivo apresentou uma linha mais íngreme, mesmo para valores pequenos de n, do que seu equivalente não-reflexivo. Para $n = 310$, obteve-se um tempo de $8395{,}13\,\mathrm{ms}$ do programa reflexivo em comparação a $56{,}01\,\mathrm{ms}$ do programa utilizando blocos de código.

De acordo com a Figura 13, o programa reflexivo requer muito mais tempo do que seu equivalente não-reflexivo. No estudo de caso proposto, existem $n-1$ chamadas de uma função adaptativa para cada valor n, o que significa que, ao longo do tempo, mudanças no código tornam-se extremamente onerosas (com inclinação $= 2{,}4239 \times 10^1$). O programa utilizando blocos de código obteve um desempenho significativamente melhor (com inclinação $= 1{,}744 \times 10^{-1}$).

Os resultados do segundo experimento são exibidos nas Figuras 14 e 15. A primeira parte do experimento consistiu na medição do tamanho de objeto, em bytes, de cada componente implementando o motor adaptativo, enquanto que a segunda parte analisou o consumo de memória.

De acordo com a Figura 14, é possível notar que o programa reflexivo produz um tamanho de objeto menor do que seu equivalente não-reflexivo no início do experimento, para valores muito pequenos de n, mas termina com uma taxa de crescimento mais acentuada ao longo do tempo (com inclinação $= 9{,}95 \times 10^2$). O programa utilizando blocos de código teve um crescimento aparentemente linear durante as iterações, gerando um modelo de objeto mais conciso

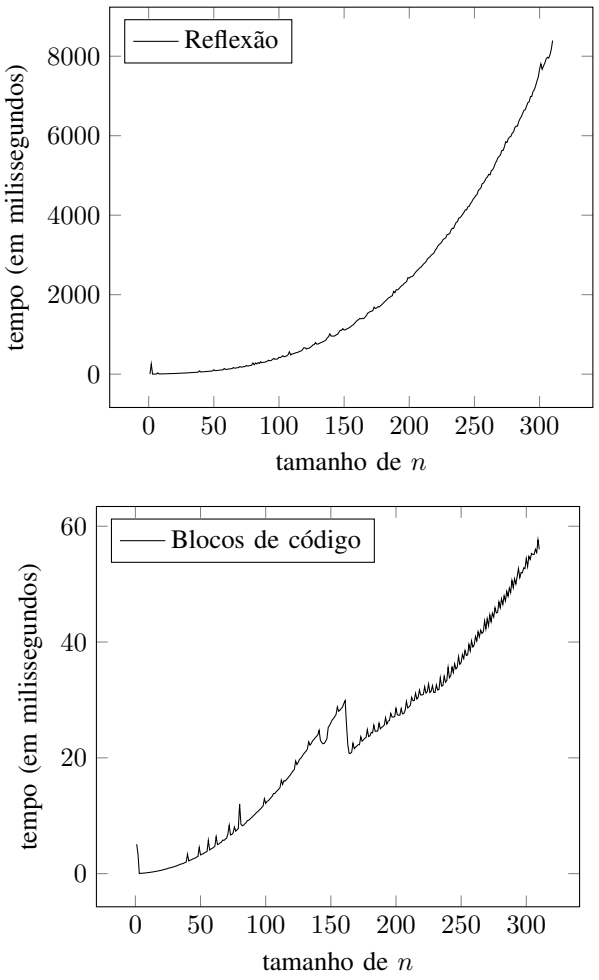


Figura 13. Primeiro experimento, medindo o tempo de reconhecimento de cada cadeia w, $w = a^n b^n c^n$, com $1 \leq n \leq 310$, durante 1000 iterações.

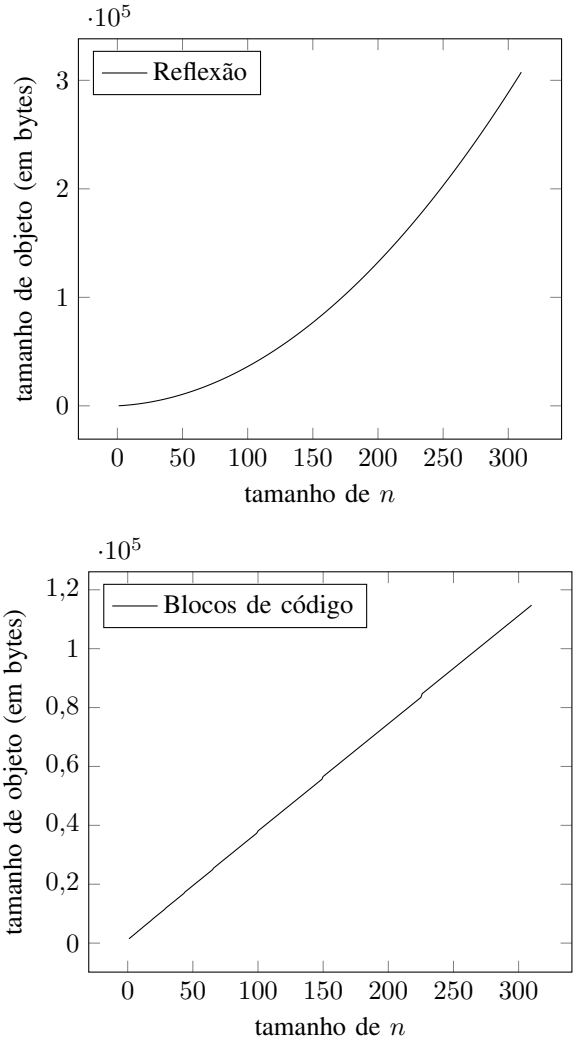


Figura 14. Primeira parte do segundo experimento, medindo o tamanho de objeto, em bytes, de cada componente implementando o motor adaptativo para cada cadeia w, $w = a^n b^n c^n$, com $1 \leq n \leq 310$, durante 1000 iterações.

(com inclinação $= 3{,}6774 \times 10^2$). A análise do consumo de memória é apresentada na Figura 15.

É possível observar, de acordo com os resultados da Figura 15, que o programa reflexivo teve um consumo de memória íngreme (com inclinação $= 3{,}1503 \times 10^9$) devido à enorme quantidade de buscas na estrutura da classe para definir os pontos de entrada e realizar as modificações. O programa utilizando blocos de código teve um consumo de memória significativamente menor (com inclinação $= 1{,}4171 \times 10^8$), mesmo para grandes valores de n. É importante notar que o processo de coleta de lixo (*garbage collection*) ocorre frequentemente durante a execução do programa utilizando blocos de código e, portanto, reduzindo o consumo de memória de tempos em tempos; no programa reflexivo, devido à utilização do conceito de herança, toda a hierarquia de classes deve ser preservada, portanto a coleta de lixo não pode ser realizada.

V. Considerações finais

O estudo de caso apresentado na Seção IV indica a viabilidade da implementação de programas adaptativos através da inserção de trechos de código, escritos em linguagens de programação convencionais com características de automodificação. As implementações podem ter a inspiração de técnicas tais como reflexão estrutural ou particionamento por blocos de código, apresentadas neste artigo. Apesar da reflexão ter problemas de desempenho, evidenciados nos testes realizados, ainda assim é uma solução viável para a inspeção de código já existente e injeção de blocos de código com características de adaptatividade quando necessário.

Entretanto, é importante observar que tais técnicas dependem fortemente de conceitos oriundos do estilo adaptativo [6] e da própria adaptatividade. Para minimizar tal dependência e permitir um fluxo de trabalho mais simplificado para a escrita de código adaptativo, uma ferramenta gráfica integrada ao ambiente de desenvolvimento de software está em estudo e análise. A proposta é tornar as características de adaptatividade encapsuladas na forma de bibliotecas adaptativas, escritas em linguagens de programação existentes, permitindo que usuários escrevam códigos adaptativos sem a necessidade imediata de conhecer os aspectos de implementação do estilo adaptativo.

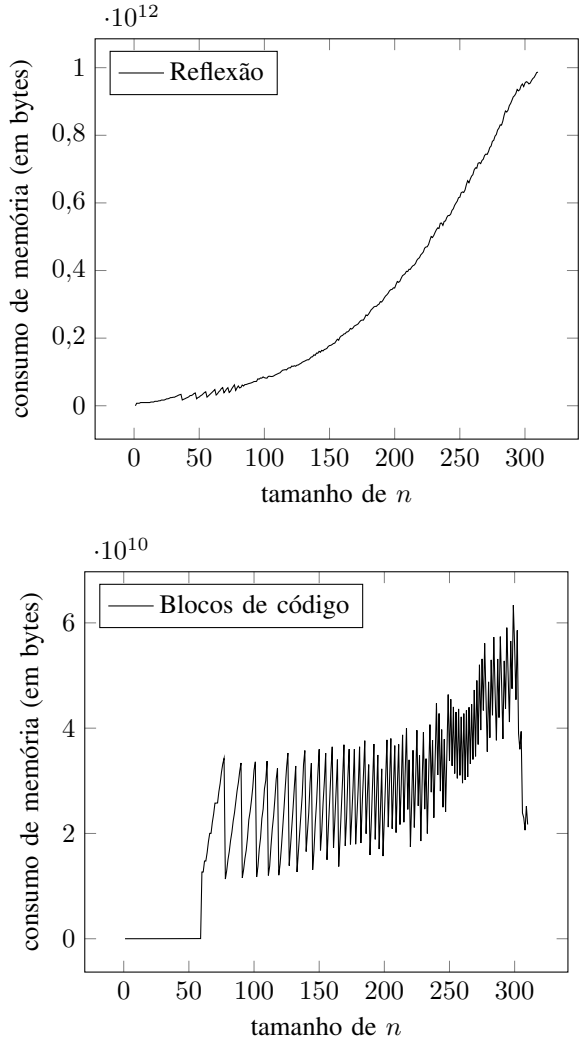


Figura 15. Segunda parte do segundo experimento, medindo o consumo de memória de cada cadeia w, $w = a^n b^n c^n$, com $1 \leq n \leq 310$, durante 1000 iterações.

REFERÊNCIAS

[1] J. José Neto, "Contribuições à metodologia de construção de compiladores," Thesis (Livre-docência), Escola Politécnica, Universidade de São Paulo, 1993.

[2] ——, "Adaptive automata for context-dependent languages," *SIGPLAN Notices*, vol. 29, no. 9, pp. 115–124, 1994.

[3] ——, "Um levantamento da evolução da adaptatividade e da tecnologia adaptativa," *IEEE Latin America Transactions*, vol. 5, pp. 496–505, 2007.

[4] H. Pistori, "Tecnologia em engenharia de computação: estado da arte e aplicações," PhD thesis, Escola Politécnica, Universidade de São Paulo, 2003.

[5] R. L. A. Rocha and J. José Neto, "Uma proposta de linguagem de programação funcional com características adaptativas," in *IX Congreso Argentino de Ciencias de la Computación*, 2003.

[6] A. V. Freitas and J. José Neto, "Adaptive languages and a new programming style," in *WSEAS International Conference on Applied Computer Science*, Tenerife, Canary Islands, Spain, 2006.

[7] A. V. Freitas, "Considerações sobre o desenvolvimento de linguagens adaptativas de programação," PhD thesis, Escola Politécnica, Universidade de São Paulo, 2008.

[8] A. A. Castro Junior, "Aspectos de projeto e implementação de linguagens para codificação de programas adaptativos," PhD thesis, Escola Politécnica, Universidade de São Paulo, 2009.

[9] E. J. Pelegrini, "Códigos adaptativos e linguagem de programação adaptativa: conceitos e tecnologias," Master's thesis, Escola Politécnica, Universidade de São Paulo, 2009.

[10] S. R. B. Silva, "Software adaptativo: método de projeto, representação gráfica e implementação de linguagem de programação," Master's thesis, Escola Politécnica, Universidade de São Paulo, 2011.

[11] J. José Neto, "Adaptive rule-driven devices: general formulation and case study," in *International Conference on Implementation and Application of Automata*, 2001.

[12] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java language specification, Java SE 7 edition," Oracle America, Inc., Tech. Rep., 2011.

[13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java virtual machine specification, Java SE 7 edition," Oracle America, Inc., Tech. Rep., 2011.

[14] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# Language Specification*. Addison-Wesley Longman Publishing, 2003.

[15] Y. Hassoun, R. Johnson, and S. Counsell, "Reusability, open implementation and Java's dynamic proxies," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, ser. PPPJ'03. New York, NY, USA: Computer Science Press, Inc., 2003, pp. 3–6.

[16] S. Chiba, "Load-time structural reflection in Java," in *ECOOP 2000, Object-Oriented Programming*, 2000.

[17] ——, "Javassist, a reflection-based programming wizard for Java," in *Proceedings of the ACM OOPSLA, Workshop on Reflective Programming in C++ and Java*, 1998.

[18] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators," in *International Conference on Generative Programming and Component Engineering*, 2003.

Paulo Roberto Massa Cereda é graduado em Ciência da Computação pelo Centro Universitário Central Paulista (2005) e mestre em Ciência da Computação pela Universidade Federal de São Carlos (2008). Atualmente, é doutorando do Programa de Pós-Graduação em Engenharia de Computação do Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, atuando como aluno pesquisador no Laboratório de Linguagens e Técnicas Adaptativas do PCS. Tem experiência na área de Ciência da Computação, com ênfase em Teoria da Computação, atuando principalmente nos seguintes temas: tecnologia adaptativa, autômatos adaptativos, dispositivos adaptativos, linguagens de programação e construção de compiladores.

João José Neto é graduado em Engenharia de Eletricidade (1971), mestre em Engenharia Elétrica (1975), doutor em Engenharia Elétrica (1980) e livre-docente (1993) pela Escola Politécnica da Universidade de S ão Paulo. Atualmente, é professor associado da Escola Politécnica da Universidade de São Paulo e coordena o LTA – Laboratório de Linguagens e Técnicas Adaptativas 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.