AA4J: uma biblioteca para implementação de autômatos adaptativos

P. R. M. Cereda e J. José Neto

***Abstract*—Este artigo apresenta uma biblioteca para a implementação de autômatos adaptativos, utilizando a linguagem Java, de forma consistente e aderente à teoria. Aspectos técnicos do desenvolvimento da biblioteca são apresentados e discutidos, incluindo exemplos de implementação de autômatos para reconhecimento de linguagens regulares, livres de contexto e dependentes de contexto.**

***Palavras-chave:*—Autômato adaptativo, linguagens de programação, bibliotecas, adaptatividade.**

I. Introdução

Este artigo apresenta uma biblioteca chamada AA4J para a implementação de autômatos adaptativos, utilizando a linguagem Java (ou outras linguagens que rodam sobre a máquina virtual Java), de forma consistente e aderente à teoria original proposta por José Neto [1]. Espera-se que, através desta biblioteca, programas possam conter elementos adaptativos especificados em uma linguagem de alto nível.

A organização deste artigo é a seguinte: a Seção II apresenta uma breve revisão bibliográfica da teoria. A Seção III apresenta aspectos técnicos da biblioteca e discussões sobre implementação. Exemplos de implementação de autômatos utilizando AA4J são contemplados na Seção IV. As considerações finais são apresentadas na Seção V.

II. Conceitos iniciais

O *autômato adaptativo*, proposto por José Neto [1], é uma extensão do formalismo do autômato de pilha estruturado que permite o reconhecimento de linguagens do tipo 0, segundo a hierarquia de Chomsky. O termo *adaptativo*, neste contexto, pode ser definido como a capacidade de um dispositivo em alterar seu comportamento de forma espontânea. Logo, um autômato adaptativo tem como característica a possibilidade de provocar alterações em sua própria topologia durante o processo de reconhecimento de uma dada cadeia [2].

Essa capacidade de alteração do autômato faz-se possível através da utilização de ações adaptativas, que podem ser executadas antes ou depois de uma transição. A cada execução de uma ação adaptativa, o autômato tem sua topologia alterada, obtendo-se uma nova configuração. O objetivo de uma ação adaptativa é lidar com situações esperadas, mas ainda não consideradas, detectadas na cadeia submetida para reconhecimento pelo autômato [3]. Uma transição pode ter ações adaptativas associadas, que permitam a inclusão ou eliminação de estados e transições.

Ao executar uma transição que contém uma ação adaptativa associada, o autômato sofre mudanças, obtendo-se então uma nova configuração do autômato. Para a aceitação de uma determinada cadeia, o autômato percorrerá um caminho em um espaço de autômatos; em outras palavras, haverá um autômato E_0, que iniciará o reconhecimento de uma determinada cadeia; autômatos intermediários E_i, que serão criadas ao longo do reconhecimento; e um autômato final E_n, que corresponde ao final do reconhecimento da cadeia. Seja a cadeia $w = \alpha_0\alpha_1 \ldots \alpha_n$; então o autômato M descreverá um caminho de autômatos $\langle E_0, \alpha_0\rangle \rightarrow \langle E_1, \alpha_1\rangle \rightarrow \ldots \rightarrow \langle E_n, \alpha_n\rangle$, no qual E_i representa um autômato correspondente à aceitação da subcadeia α_i.

Definição 1 (autômato adaptativo)**.** Um *autômato adaptativo* M é definido por $M = (Q, S, \Sigma, \Gamma, P, q_0, Z_0, F)$, tal que Q é o conjunto finito de estados, $Q \subset Q^A$, Q^A é o conjunto de todos os estados possíveis, Q^A é enumerável, S é o conjunto finito de submáquinas, Σ é o alfabeto de entrada, $\Sigma \subset \Sigma^A$, Σ^A é o conjunto enumerável de todos os símbolos possíveis, Γ é o alfabeto da pilha, $\Gamma \subset \Gamma^A$, $\Gamma = Q \cup \{Z_0\}$, Γ^A é o conjunto enumerável de todos os símbolos possíveis da pilha, $\Gamma^A = Q^A \cup \{Z_0\}$, P é um mapeamento $P\colon Q^A \times \Sigma^A \times \Gamma^A \rightarrow Q^A \times (\Sigma^A \cup \{\epsilon\}) \times (\Gamma^A \cup \{\epsilon\}) \times H^0 \times H^0\}$, H^0 definido a seguir, $q_0 \in Q$ é o estado inicial, $Z_0 \in \Gamma$ é o símbolo inicial da pilha, $F \subset Q$ é o conjunto de estados finais. Os conjuntos enumeráveis Q^A, Σ^A e Γ^A (com A representando *All* – para todos) são convenientes porque as *funções adaptativas* podem *(a)* inserir novos estados q, $q \notin Q$ mas $q \in Q^A$, e *(b)* usar novos símbolos de pilha $\gamma \notin \Gamma$ mas $\gamma \in \Gamma^A$. Em resumo, as funções adaptativas podem modificar o autômato, mas os novos símbolos que elas introduzem estão todos nos conjuntos enumeráveis [4].

H^0 é o conjunto de todas as funções adaptativas no autômato adaptativo M. Define-se $H^0 = \{f \mid f\colon E \times G_1 \times G_2 \times \ldots \times G_k \rightarrow E\}$, tal que f é uma função, $k \in \mathbb{N}$ é o número de argumentos em f, e $G_i = Q^A \cup \Sigma^A \cup \Gamma^A$.

E é o conjunto de todos os autômatos adaptativos que têm o estado inicial q_0, o símbolo inicial de pilha Z_0 e o conjunto de estados finais F iguais aos do autômato adaptativo M. Define-se $E = \{N \mid N$ é um autômato adaptativo $N = (Q', \Sigma', \Gamma', P', q_0, Z_0, F)$, onde $Q' \subset Q^A$, $\Sigma' \subset \Sigma^A$, $\Gamma' \subset \Gamma^A$, $P'\colon Q^A \times \Sigma^A \times \Gamma^A \rightarrow Q^A \times (\Sigma^A \cup \{\epsilon\}) \times (\Gamma^A \cup \{\epsilon\}) \times H^0 \times H^0\}$. Observe que q_0, Z_0 e F são os mesmos em qualquer $N \in E$. □

Definição 2 (submáquina do autômato adaptativo)**.** O conjunto de todas as submáquinas do autômato adaptativo M é representado por S. Cada *submáquina* s_i é definida como $s_i = (Q_i, \Sigma_i, P_i, q_{i0}, F_i)$, tal que $Q_i \subseteq Q$ é o conjunto de estados da submáquina s_i, $\Sigma_i \subseteq \Sigma$ é o alfabeto de entrada da submáquina s_i, $P_i \subseteq P$ é o mapeamento da submáquina s_i,

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

$q_{i0} \in Q_i$ é o estado inicial da submáquina s_i, e $F_i \subseteq Q_i$ é o conjunto de estados finais da submáquina s_i. □

Uma transição do autômato adaptativo é uma relação da forma $(q, a, \beta) \vdash (q', a', \beta')$ para $P(q, a, \beta) \rightarrow (q', a', \beta', A, B)$, A e B são funções adaptativas, definidas a seguir, $A, B \in H^0$. Se $\tilde{q}$, $\tilde{a}$ ou $\tilde{\beta}$ não pertencem ao autômato corrente, então $P(\tilde{q}, \tilde{a}, \tilde{\beta}) \rightarrow (\tilde{q}, \tilde{a}, \tilde{\beta}, I, I)$, tal que I é a função identidade em E.

As chamadas de funções adaptativas associadas a uma transição são funções de $E \times G_1 \times G_2 \times \ldots \times G_k$ em E. Isto é conveniente porque uma função adaptativa pode ser utilizada em mais de uma transição, com argumentos correspondentes a G_i diferentes. Ao definir uma transição $P(q, a, \beta) \rightarrow (q', a', \beta', A, B)$, é necessário fornecer os argumentos correspondentes aos conjuntos G_i. Estes argumentos podem variar de transição a transição; assim, uma função adaptativa $A\colon E \times G_1 \times G_2 \times \ldots \times G_k \rightarrow E$ pode ser utilizada em diversas transições.

Definição 3 (função adaptativa)**.** Uma *função adaptativa* é qualquer função $A \in H$, tal que $H = \bigcup_{k \geq 0} \{f \mid f\colon E \times G_1 \times \ldots \times G_k \rightarrow E\}$, $G_i = Q^A \cup \Sigma^A \cup \Gamma^A$, $i, k \in \mathbb{N}$. Em outras palavras, H é o conjunto de todas as funções que tomam um autômato adaptativo mais um conjunto de parâmetros em G (qualquer número) e retornam um autômato adaptativo. □

As funções adaptativas podem ser definidas a partir de ações adaptativas elementares de consulta, remoção e inclusão, definidas informalmente a seguir.

As funções adaptativas utilizam-se de variáveis e geradores para executar as ações de edição no autômato. As variáveis são preenchidas uma única vez na execução da função adaptativa. Os geradores são tipos especiais de variáveis, usados para associar nomes unívocos a estados recém-criados; os valores definidos são unívocos e identificados pelo símbolo $*$, por exemplo, g_1^*, g_2^*.

Os autômatos adaptativos possuem três tipos de ações adaptativas elementares utilizadas para edição de seu conjunto de regras [1]. O trecho de autômato da Figura 1 será utilizado para exemplificar as ações adaptativas elementares.

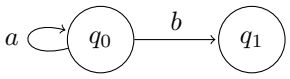

Figura 1. Trecho de um autômato utilizado para exemplificar as ações adaptativas elementares.

– *Ação adaptativa elementar de consulta*: realiza uma busca no autômato por produções cujos componentes sejam correspondentes aos valores passados a essa ação. É denotada por $?(q, a), A \rightarrow (q', a'), B$ ou $?(q, a, \beta), A \rightarrow (q', a', \beta'), B$, tal que q é o estado corrente, a é o símbolo a ser consumido, β é o topo da pilha, q' é o estado de destino, a' é o novo símbolo, podendo ser o mesmo símbolo a ser consumido ou vazio ($a' = a$ ou $a' = \epsilon$), β' é o novo topo da pilha, A é a função adaptativa a ser executada antes da transição, e B é a função adaptativa a ser executada após a transição.

As ações de consulta utilizam-se de variáveis para armazenar o resultado destas consultas. Assim, as variáveis armazenam um conjunto de estados ou transições que correspondem aos parâmetros consultados. Admitindo o trecho de autômato da Figura 1, e utilizando as variáveis $?x$ e $?y$, a Tabela I exemplifica as ações adaptativas elementares de consulta. Observe que, na terceira consulta, $?x$ e $?y$ contêm todos os símbolos possíveis.

Tabela I
EXEMPLOS DE AÇÕES DE CONSULTA.

Consulta	Significado	Resultado
$?(q_0, ?x) \rightarrow (q_1, \epsilon)$	Quais são os símbolos que, a partir do estado q_0, levam ao estado q_1?	$?x = \{b\}$
$?(q_0, b) \rightarrow (?x, \epsilon)$	A partir do estado q_0, qual é o estado de destino, consumindo o símbolo b?	$?x = \{q_1\}$
$?(q_0, ?x) \rightarrow (?y, \epsilon)$	A partir do estado inicial q_0, consumindo qualquer símbolo, quais estados de destino possíveis existem?	$?x = \{a, b\}$, $?y = \{q_0, q_1\}$

– *Ação adaptativa elementar de remoção*: remove uma produção de acordo com os valores passados a essa ação. É denotada por $-(q, a), A \rightarrow (q', a'), B$ ou $-(q, a, \beta), A \rightarrow (q', a', \beta'), B$, tal que q é o estado corrente, a é o símbolo a ser consumido, β é o topo da pilha, q' é o estado de destino, a' é o novo símbolo podendo ser o mesmo símbolo a ser consumido ou vazio ($a' = a$ ou $a' = \epsilon$), β' é o novo topo da pilha, A é a função adaptativa a ser executada antes da transição, e B é a função adaptativa a ser executada após a transição.

A ação adaptativa de remoção exclui elementos do autômato que correspondam aos parâmetros informados. Por exemplo, supondo o o trecho de autômato da Figura 1, a Tabela II exemplifica a execução da ação adaptativa de remoção.

Tabela II
EXEMPLO DE AÇÃO DE REMOÇÃO.

Remoção	Significado
$-[e, (q_0, a) \rightarrow (q_0, \epsilon)]$	Remova a transição que parte do estado q_0, consumindo o símbolo a, e que leva ao próprio estado q_0

O trecho de autômato modificado por essa ação de remoção é ilustrado na Figura 2.

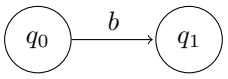

Figura 2. Trecho do autômato da Figura 1, modificado pela ação adaptativa elementar de remoção da Tabela II.

– *Ação adaptativa elementar de inclusão*: inclui uma produção de acordo com os valores passados a essa ação. É denotada por $+(q, a), A \rightarrow (q', a'), B$ ou $+(q, a, \beta), A \rightarrow (q', a', \beta'), B$, tal que q é o estado corrente, a é o símbolo a ser consumido, β é o topo

da pilha, q' é o estado de destino, a' é o novo símbolo podendo ser o mesmo símbolo a ser consumido ou vazio ($a' = a$ ou $a' = \epsilon$), β' é o novo topo da pilha, A é a função adaptativa a ser executada antes da transição, e B é a função adaptativa a ser executada após a transição.
A ação elementar de inclusão insere elementos novos no autômato, como transições e estados novos, de acordo com os argumentos fornecidos. Para a criação de um estado novo, utiliza-se um gerador, o qual receberá um identificador único que será o nome desse estado recém-criado. Por exemplo, supondo o trecho de autômato da Figura 1, e utilizando um gerador g_1^*, a Tabela III ilustra as execuções de ações adaptativas de inclusão.

Tabela III
EXEMPLO DE AÇÕES DE INCLUSÃO.

Inclusão	Significado
$+(q_1, b) \rightarrow (q_1, \epsilon)$	Insira uma transição que parta do estado q_1, consumindo o símbolo b, e que leve ao próprio estado q_1 (*loop*)
$+(q_1, c) \rightarrow (g_1^*, \epsilon)$	Insira uma transição que parta do estado q_1, consumindo o símbolo c, e que leve a um novo estado criado g_1^*

O trecho de autômato, modificado pelas ações de inclusão, é ilustrado na Figura 3.

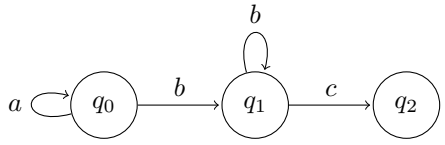

Figura 3. Trecho do autômato da Figura 1, modificado pelas ações adaptativas elementares de inclusão da Tabela III.

Graficamente, é possível representar se uma função adaptativa será executada antes ou depois da transição, através da notação apresentada na Figura 4. Em *(a)*, a função adaptativa A é executada antes da transição, enquanto que em *(b)* a função adaptativa B é executada após a transição. Em *(c)*, as duas funções são executadas, sendo A antes da transição, e B após.

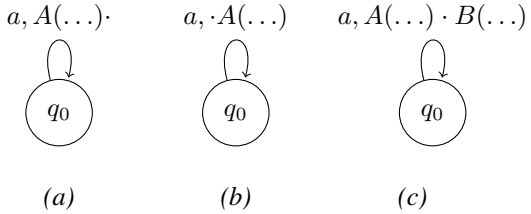

Figura 4. Notação gráfica para determinar o momento de execução das funções adaptativas.

A relação de transição $\vdash$ entre configurações do autômato adaptativo é similar à empregada no autômato de pilha estruturado, sendo que aqui deve ser considerada a presença das funções adaptativas. A linguagem aceita por um autômato adaptativo M é dada por $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q_f, \epsilon, Z_0)$, onde $q_f \in F\}$.

III. ASPECTOS TÉCNICOS E IMPLEMENTAÇÃO

A biblioteca `AA4J` permite a implementação de autômatos adaptativos, utilizando a linguagem Java, de acordo com a teoria apresentada na Seção II. Esta seção apresenta os aspectos técnicos e discussões sobre implementação.

A. Estados e símbolos

Estados e símbolos constituem os elementos básicos para a implementação de autômatos adaptativos utilizando a biblioteca `AA4J`. São disponibilizadas duas classes abstratas, `State` e `Symbol`, que devem ser estendidas pelo programador, incluindo os métodos abstratos apresentados na Figura 5. Optou-se pela obrigatoriedade da reescrita dos métodos de comparação, código *hash* e representação textual para garantir que os estados e símbolos possam ser manipulados corretamente durante o processo de reconhecimento (é comum que programadores não atentem-se à necessidade de reescrita de tais métodos).

```
public abstract boolean equals(Object o);
public abstract int hashCode();
public abstract String toString();
```

Figura 5. Métodos abstratos das classes `State` e `Symbol`. Estes métodos devem, obrigatoriamente, ser reescritos nas classes estendidas.

Estados e símbolos podem conter desde tipos primitivos até objetos complexos, proporcionando expressividade ao modelo. A seguir, são apresentados dois exemplos (Exemplos 1 e 2) de classes estendidas contendo diferentes tipos de dados (os códigos utilizam, para fins de simplificação, classes utilitárias da biblioteca Apache Commons Lang[1]).

Exemplo 1. Considere o autômato da Figura 6, no qual seus estados e símbolos são representados por cadeias de caracteres (equivalentes ao tipo de dado `String` em Java). A implementação de tais classes estendidas é apresentada nas Figuras 7 e 8.

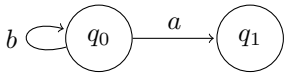

Figura 6. Estados e símbolos são representados por cadeias de caracteres.

A implementação dos estados e símbolos do autômato da Figura 6 utilizando as classes `StrState` e `StrSymbol` das Figuras 7 e 8 é apresentada na Figura 9.

É importante observar que o programador é livre para incluir funcionalidades adicionais às classes, de acordo com a necessidade. □

Exemplo 2. Considere o autômato da Figura 10, no qual seus estados são representados por números inteiros e seus símbolos são representados por valores reais (equivalentes aos tipos de dados `int` e `double` em Java, respectivamente). A implementação de tais classes estendidas é apresentada nas Figuras 11 e 12.

[1]Disponível em `https://commons.apache.org/proper/commons-lang/`.

```
public class StrState extends State {

  private String value;

  public StrState(String value) { this.value = value; }

  public boolean equals(Object object) {
     if (object == null) { return false; }
     else {
        if (!(object.getClass().equals(StrState.class))) {
           return false;
        }
        else {
           return new EqualsBuilder().append(this.getValue(),
                 ((StrState) object).getValue()).isEquals();
  }}}

  public String getValue() { return value; }

  public void setValue(String value) { this.value = value; }

  public int hashCode() { return new
       HashCodeBuilder().append(this.getValue()).hashCode(); }

  public String toString() { return value; }

}
```

Figura 7. Implementação da classe estendida `StrState`, cujo estado é representado por uma cadeia de caracteres.

```
public class StrSymbol extends Symbol {

  private String value;

  public StrSymbol(String value) { this.value = value; }

  public boolean equals(Object object) {
     if (object == null) { return false; }
     else {
        if (!(object.getClass().equals(StrSymbol.class))) {
           return false;
        }
        else {
           return new EqualsBuilder().append(this.getValue(),
                 ((StrSymbol) object).getValue()).isEquals();
  }}}

  public String getValue() { return value; }

  public void setValue(String value) { this.value = value; }

  public int hashCode() { return new
       HashCodeBuilder().append(this.getValue()).hashCode(); }

  public String toString() { return value; }

}
```

Figura 8. Implementação da classe estendida `StrSymbol`, cujo símbolo é representado por uma cadeia de caracteres.

```
State q0 = new StrState("q0");
State q1 = new StrState("q1");
Symbol a = new StrSymbol("a");
Symbol b = new StrSymbol("b");
```

Figura 9. Implementação dos estados e símbolos do autômato da Figura 6 utilizando as classes `StrState` e `StrSymbol` das Figuras 7 e 8.

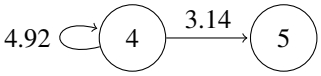

Figura 10. Estados são representados por números inteiros e símbolos são representados por números reais.

A implementação dos estados e símbolos do autômato da Figura 10 utilizando as classes `IntState` e `DoubleSymbol` das Figuras 11 e 12 é apresentada na Figura 13.

Classes mais complexas devem implementar os métodos de comparação, código *hash* e representação textual de acordo com a lógica do modelo. Uma classe que implementa *tokens*, por exemplo, pode comparar apenas as classes gramaticais e ignorar os valores propriamente ditos. □

B. Ações semânticas e adaptativas

A biblioteca `AA4J` disponibiliza uma classe abstrata `Action` para definição de ações. Em linhas gerais, as ações podem ser puramente semânticas (por exemplo, para geração de código ou verificação da existência de um símbolo na tabela de símbolos), adaptativas (definidas somente em termos de ações adaptativas elementares), ou híbridas, contemplando uma combinação de ambas. É necessário estender a classe `Action` e incluir o método abstrato apresentado na Figura 14.

De acordo com a Figura 14, o método abstrato `execute(...)` tem como parâmetros o mapeamento do autômato, a transição corrente que disparou a ação (para referência), e um vetor de objetos como parâmetros formais da ação. Este último permite que as ações sejam parametrizadas quando definidas no escopo de uma transição.

Cada ação deve receber um nome unívoco, que será seu identificador. O nome da ação permitirá que esta seja referenciada posteriormente, durante o processo de reconhecimento de uma cadeia de símbolos. O Exemplo 3 apresenta a implementação de duas ações semânticas.

Exemplo 3. Considere duas ações semânticas: a primeira imprime tão somente uma mensagem no terminal, e a segunda exibe o resultado de um cálculo matemático de acordo com os valores fornecidos como parâmetros. A Figura 15 apresenta tal implementação.

Observe que é necessário realizar a conversão de objetos quando a ação utiliza seus parâmetros formais. Na segunda ação semântica da Figura 15, os objetos devem ser convertidos para valores inteiros (utilizando o conceito de *unboxing*), de modo que o cálculo matemático especificado torne-se, portanto, válido. □

Ações adaptativas elementares são disponibilizadas através de uma classe utilitária chamada `ElementaryActions`. É

```
public class IntState extends State {

  private int value;

  public IntState(int value) { this.value = value; }

  public boolean equals(Object object) {
     if (object == null) { return false; }
     else {
        if (!(object.getClass().equals(IntState.class))) {
           return false;
        }
        else {
           return new EqualsBuilder().append(this.getValue(),
                 ((IntState) object).getValue()).isEquals();
  }}}

  public int getValue() { return value; }

  public void setValue(int value) { this.value = value; }

  public int hashCode() { return new
       HashCodeBuilder().append(this.getValue()).hashCode(); }

  public String toString() { return String.valueOf(value); }

}
```

Figura 11. Implementação da classe estendida `IntState`, cujo estado é representado por um número inteiro.

```
public class DoubleSymbol extends Symbol {

  private double value;

  public DoubleSymbol(double value) { this.value = value; }

  public boolean equals(Object object) {
     if (object == null) { return false; }
     else {
        if (!(object.getClass().equals(DoubleSymbol.class))) {
           return false;
        }
        else {
           return new EqualsBuilder().append(this.getValue(),
                 ((DoubleSymbol)
                      object).getValue()).isEquals();
  }}}

  public double getValue() { return value; }

  public void setValue(double value) { this.value = value; }

  public int hashCode() { return new
       HashCodeBuilder().append(this.getValue()).hashCode(); }

  public String toString() { return String.valueOf(value); }

}
```

Figura 12. Implementação da classe estendida `DoubleSymbol`, cujo símbolo é representado por um número real.

```
State s1 = new IntState(4);
State s2 = new IntState(5);
Symbol v1 = new DoubleSymbol(3.14);
Symbol v2 = new DoubleSymbol(4.92);
```

Figura 13. Implementação dos estados e símbolos do autômato da Figura 10 utilizando as classes `IntState` e `DoubleSymbol` das Figuras 11 e 12.

```
public abstract void execute(Mapping transitions, Transition
     transition, Object... parameters);
```

Figura 14. Método abstrato da classe `Action`. Este método deve, obrigatoriamente, ser reescrito nas classes estendidas.

necessário instanciar um objeto dessa classe, fornecendo o mapeamento do autômato como parâmetro ao construtor. Após a instanciação, é possível utilizar os métodos do objeto que implementam as ações adaptativas elementares (apresentados na Figura 16).

De acordo com a Figura 16, a classe `ElementaryActions` apresenta vários métodos que implementam as ações adaptativas elementares conforme a teoria apresentada na Seção II. Classes utilitárias são utilizadas como suporte para a execução dos métodos que implementam as ações adaptativas elementares:

- `Variable`: representa uma variável de consulta nas ações elementares e pode armazenar um conjunto de valores. Por definição, assim que a variável recebe um ou mais valores (no momento de sua instanciação ou através da aplicação de uma ação elementar de consulta), esta torna-se imutável, isto é, não é possível alterar seu conteúdo.
- `ActionQuery`: contempla um objeto de representação de ações. Uma ação pode ser identificada univocamente através de seu nome (construtor com apenas um parâmetro) ou através de seu nome seguido por seus parâmetros (construtor com múltiplos parâmetros).
- `SubmachineQuery`: contempla um objeto de representação de submáquinas. Uma submáquina é identificada univocamente através de seu nome.

```
Action message = new Action("message") {
  public void execute(Mapping transitions, Transition
       transition, Object... parameters) {
    System.out.println("Hello world!");
  }
};

Action add = new Action("add") {
  public void execute(Mapping transitions, Transition
       transition, Object... parameters) {
    int a = (int) parameters[0];
    int b = (int) parameters[1];
    System.out.println(a + b);
  }
};
```

Figura 15. Implementação de duas ações semânticas.

```
public void query(Variable source, Variable symbol, Variable target);
public void query(Variable source, SubmachineQuery submachine, Variable target);
public void query(Variable source, SubmachineQuery submachine, Variable target, ActionQuery postAction);
public void query(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target);
public void query(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target, ActionQuery
    postAction);
public void query(Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void query(ActionQuery priorAction, Variable source, Variable symbol, Variable target);
public void query(ActionQuery priorAction, Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void remove(Variable source, Variable symbol, Variable target);
public void remove(ActionQuery priorAction, Variable source, Variable symbol, Variable target);
public void remove(Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void remove(ActionQuery priorAction, Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void remove(Variable source, SubmachineQuery submachine, Variable target);
public void remove(Variable source, SubmachineQuery submachine, Variable target, ActionQuery postAction);
public void remove(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target);
public void remove(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target, ActionQuery
    postAction);
public void add(Variable source, Variable symbol, Variable target);
public void add(ActionQuery priorAction, Variable source, Variable symbol, Variable target);
public void add(Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void add(Variable source, SubmachineQuery submachine, Variable target, ActionQuery postAction);
public void add(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target, ActionQuery
    postAction);
public void add(ActionQuery priorAction, Variable source, Variable symbol, Variable target, ActionQuery postAction);
public void add(ActionQuery priorAction, Variable source, SubmachineQuery submachine, Variable target);
public void add(Variable source, SubmachineQuery submachine, Variable target);
```

Figura 16. Métodos da classe `ElementaryActions` que implementam as ações adaptativas elementares.

As classes utilitárias permitem que as ações elementares sejam executadas de modo consistente. O Exemplo 4 apresenta a implementação de uma função adaptativa não-parametrizada utilizando a composição de ações adaptativas elementares.

Exemplo 4. Considere o trecho de autômato da Figura 17, no qual uma função adaptativa $\mathcal{A}$ (Algoritmo 1) realiza alterações em sua topologia, removendo a transição $(q_1, b) \rightarrow q_0$ e inserindo uma nova transição $(q_0, a) \rightarrow q_1$. A implementação de tal função adaptativa é apresentada na Figura 18 (admita a utilização das classes `StrState` e `StrSymbol` introduzidas nas Figuras 7 e 8, respectivamente).

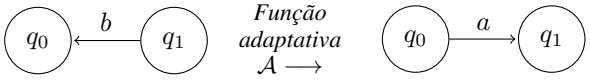

Figura 17. Trecho de autômato que ilustra uma função adaptativa removendo a transição $(q_1, b) \rightarrow q_0$ e inserindo uma nova transição $(q_0, a) \rightarrow q_1$.

Algoritmo 1 Função adaptativa $\mathcal{A}$

função adaptativa $\mathcal{A}$
$-(q_1, b) \rightarrow q_0$
$+(q_0, a) \rightarrow q_1$
fim da função adaptativa

□

É importante observar que a utilização de variáveis contendo múltiplos valores como parâmetros das ações adaptativas elementares resulta na aplicação da operação sobre o produto cartesiano dos parâmetros.

```
Action a = new Action("A") {
  public void execute(Mapping transitions, Transition
      transition, Object... parameters) {
    ElementaryActions ea = new ElementaryActions(transitions);
    State q0 = new StrState("q0");
    State q1 = new StrState("q1");
    Symbol a = new StrSymbol("a");
    Symbol b = new StrSymbol("b");
    ea.remove(new Variable(q1), new Variable(b), new
        Variable(q0));
    ea.add(new Variable(q0), new Variable(a), new
        Variable(q1));
  }
};
```

Figura 18. Implementação da função adaptativa $\mathcal{A}$ do Algoritmo 1.

C. Transições

Transições do autômato são implementadas através da classe `Transition`, que disponibiliza os métodos listados na Figura 19. É possível especificar transições contendo consumo de símbolos, em vazio e chamadas de submáquinas; adicionalmente, cada transição pode conter ações anteriores e posteriores, juntamente com seus parâmetros.

De acordo com a Figura 19, os métodos disponibilizados permitem uma especificação consistente do tipo de transição. Para definir uma transição em vazio, opta-se por utilizar a constante `EPSILON` ou a referência `null` como símbolo a ser consumido. O Exemplo 5 apresenta a implementação das transições do trecho de autômato da Figura 20.

Exemplo 5. Considere o trecho de autômato da Figura 20,

```
public void setSourceState(State sourceState);
public void setSymbol(Symbol symbol);
public void setTargetState(State targetState);
public void setSubmachineCall(String submachineCall);
public void setPriorActionCall(String priorActionCall);
public void setPriorActionArguments(Object[]
    priorActionArguments);
public void setPostActionCall(String postActionCall);
public void setPostActionArguments(Object[]
    postActionArguments);
public void setTransition(State sourceState, Symbol symbol,
    State targetState);
public void setSubmachineCall(State sourceState, String
    submachineCall, State targetState);
```

Figura 19. Métodos da classe `Transition`, que implementa as transições do autômato.

contendo transições com consumo de símbolos, em vazio e chamada de submáquina. A implementação de suas transições é apresentada na Figura 21 (admita a utilização das classes `StrState` e `StrSymbol` introduzidas nas Figuras 7 e 8, respectivamente).

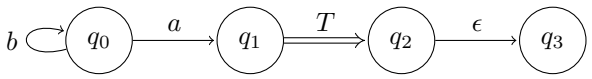

Figura 20. Trecho de autômato contendo transições com consumo de símbolos, em vazio e chamada de submáquina.

```
State q0 = new StrState("q0"); State q1 = new StrState("q1");
State q2 = new StrState("q2"); State q3 = new StrState("q3");
Symbol a = new StrSymbol("a"); Symbol b = new StrSymbol("b");

Transition t1 = new Transition();
t1.setTransition(q0, b, q0);
Transition t2 = new Transition();
t2.setTransition(q0, a, q1);
Transition t3 = new Transition();
t3.setSubmachineCall(q1, "T", q2);
Transition t4 = new Transition();
t4.setTransition(q2, EPSILON, q3);
```

Figura 21. Implementação das transições do trecho de autômato da Figura 20.

Observe que a implementação apresentada na Figura 21 pode ser reescrita utilizando outros métodos da classe `Transition` (Figura 19), o que confere liberdade ao programador para escolher a forma que melhor lhe agradar. □

Internamente, a biblioteca utiliza um tipo especial de transição que realiza o retorno de uma chamada de submáquina. Em linhas gerais, quando a submáquina termina seu reconhecimento em um estado de retorno, ocorre a remoção do estado de destino do topo da pilha e a execução prossegue a partir do novo endereço; a biblioteca realiza então uma transição de retorno, que segue em vazio a partir do estado de retorno da submáquina chamada até o estado que estava no topo da pilha.

D. Submáquinas

A biblioteca `AA4J` considera que todo autômato tem, ao menos, uma submáquina (por exemplo, um autômato finito M possui uma única submáquina, que é ele próprio). Submáquinas são disponibilizadas através da classe `Submachine`, cujo construtor é apresentado na Figura 22. Observe que os parâmetros do construtor são o nome da submáquina, que será utilizado para identificar univocamente a submáquina durante o processo de reconhecimento de uma cadeia de símbolos, o conjunto de estados que compõem a submáquina, o estado inicial (ou de entrada) da submáquina, e o conjunto de estados de aceitação (ou de retorno) da submáquina. O Exemplo 6 apresenta a implementação da especificação da submáquina T da Figura 23.

```
public Submachine(String name, Set<State> states, State
    initialState, Set<State> acceptingStates);
```

Figura 22. Construtor da classe `Submachine` que implementa uma submáquina.

Exemplo 6. Considere a submáquina T da Figura 23, tal que q_0 é o estado inicial e os estados q_1 e q_2 são de aceitação. A implementação da submáquina T é apresentada na Figura 24 (admita a utilização da classe `StrState` introduzida na Figura 7).

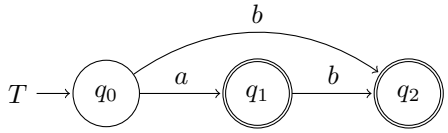

Figura 23. Submáquina T.

```
State q0 = new StrState("q0");
State q1 = new StrState("q1");
State q2 = new StrState("q2");

Set<State> Q = new HashSet<>();
Q.add(q0); Q.add(q1); Q.add(q2);

Set<State> F = new HashSet<>();
F.add(q1); F.add(q2);

Submachine T = new Submachine("T", Q, q0, F);
```

Figura 24. Implementação da especificação da da submáquina T da Figura 23 utilizando a classe `Submachine`.

É importante destacar que os estados da submáquina T são exclusivos a ela, isto é, outras submáquinas não poderão compartilhar estados entre si. □

E. Autômato adaptativo

Uma autômato adaptativo é implementado através da classe abstrata `AdaptiveAutomaton`, que deve ser estendida, incluindo seu método abstrato, para permitir a especificação dos símbolos, estados, transições, submáquinas e ações. O método a ser implementado na classe estendida é apresentado na Figura 25.

```
public abstract void setup();
```

Figura 25. Método da classe abstrata `AdaptiveAutomaton` a ser implementado na classe estendida para permitir a especificação dos símbolos, estados, transições, submáquinas e ações.

É necessário seguir uma seqüência de passos para a definição dos símbolos, estados, transições, submáquinas e ações no corpo do método `setup()`, de tal forma que o autômato seja configurado corretamente. Tais passos são descritos a seguir:

- definir inicialmente todos os símbolos e estados componentes do modelo do autômato, conforme ilustra o exemplo da Figura 9;
- definir todas as ações semânticas, adaptativas ou híbridas pertencentes ao modelo do autômato, conforme ilustram os exemplos das Figuras 15 e 18;
- inserir todas as ações definidas no passo anterior em uma lista de ações chamada `actions`, que é uma variável protegida da classe `AdaptiveAutomaton`; considerando `action` como uma ação definida, a sintaxe para inseri-la na lista é `actions.add(action);` no corpo do método;
- definir todas as submáquinas do modelo do autômato, conforme ilustra o exemplo da Figura 24.
- inserir todas as submáquinas definidas no passo anterior em uma lista de submáquinas chamada `submachines`, que é uma variável protegida da classe `AdaptiveAutomaton`; considerando `submachine` como uma submáquina definida, a sintaxe para inseri-la na lista é `submachines.add(submachine);` no corpo do método;
- definir todas as transições do modelo do autômato, conforme ilustra o exemplo da Figura 21.
- inserir todas as transições definidas no passo anterior em uma lista de transições chamada `transitions`, que é uma variável protegida da classe `AdaptiveAutomaton`; considerando `transition` como uma transição definida, a sintaxe para inseri-la na lista é `transitions.add(transition);` no corpo do método; e
- definir qual será a submáquina principal do autômato, através do método `setMainSubmachine(name)`, no qual `name` é o nome da submáquina principal.

A classe `AdaptiveAutomaton` disponibiliza os métodos concretos apresentados na Figura 26. O método `recognize(...)` é utilizado para realizar o processo de reconhecimento da lista de símbolos, retornando um valor lógico referente ao resultado do reconhecimento. O método `setStopAtFirstResult(...)` permite definir se autômato deve interromper o processo de reconhecimento caso já exista um resultado disponível (no caso de não-determinismo). Os métodos `getRecognitionPaths()` e `getRecognitionMap()` retornam uma lista e um mapa, respectivamente, contendo todos os caminhos de reconhecimento percorridos pelo autômato durante o processo de reconhecimento (em caso de um reconhecimento determinístico, haverá apenas um caminho).

```
public boolean recognize(List<Symbol> input);
public void setStopAtFirstResult(boolean flag);
public List<RecognitionPath> getRecognitionPaths();
public Map<Integer, RecognitionPath> getRecognitionMap();
```

Figura 26. Métodos concretos da classe `AdaptiveAutomaton`.

É importante destacar que os objetos da classe `AdaptiveAutomaton` sempre retornarão à sua configuração inicial após o processo de reconhecimento de uma cadeia de entrada (não há persistência de configurações). Exemplos completos da implementação de autômatos adaptativos utilizando a classe `AdaptiveAutomaton` são apresentados em detalhes na Seção IV.

F. Registro de eventos

A biblioteca `AA4J` registra todos os eventos de suas classes internas para cada chamada do processo de reconhecimento de uma cadeia de símbolos. Assim, é possível analisar a execução de forma incremental. A granularidade do registro está definida para capturar apenas eventos graves, mas é possível ajustar os valores através da edição do arquivo de configuração chamado `log4j2.xml`, utilizado pela biblioteca Apache Log4j[2], responsável pelo gerenciamento do registro.

O registro de eventos torna-se útil para automatizar o processo de validação de um modelo utilizando autômatos adaptativos. Adicionalmente, é possível utilizar os métodos `getRecognitionPaths()` e `getRecognitionMap()` diretamente no código-fonte para analisar o comportamento da execução (esta técnica, entretanto, não oferece detalhes complementares).

G. Não-determinismo

A biblioteca `AA4J` permite a definição e execução de modelos não-determinísticos, tal que todos os caminhos válidos sejam inspecionados e executados simultaneamente. Por padrão, a biblioteca aguarda o término de todas as ramificações do processo de reconhecimento de uma cadeia de símbolos, mas é possível alterar tal comportamento através do método `setStopAtFirstResult(...)`, que permite interromper o processamento quando um dos caminhos já encerrou-se.

Para determinar se o processo de reconhecimento de uma cadeia de símbolos foi determinístico, é suficiente verificar o tamanho da lista de caminhos de reconhecimento retornada pelo método `getRecognitionPaths()`; caso a lista possua apenas um caminho de reconhecimento, a execução foi determinística (é possível realizar o mesmo teste com o mapa retornado pelo método `getRecognitionMap()`, verificando o tamanho do conjunto de chaves).

H. Ciclos em vazio

A versão corrente da biblioteca `AA4J` não detecta ciclos em vazio potenciais (triviais ou não) no modelo do autômato adaptativo, podendo causar uma execução infinita; assim, é altamente recomendável que estes ciclos sejam detectados e removidos *a priori* para evitar problemas de execução.

[2]Disponível em `http://logging.apache.org/log4j/`.

IV. EXEMPLOS

Esta seção apresenta três exemplos de autômatos para ilustrar o funcionamento da biblioteca `AA4J`. Todas as implementações apresentadas utilizam as classes `StrState` e `StrSymbol` introduzidas nas Figuras 7 e 8, respectivamente. Considere também a existência de um método estático `List<Symbol> convert(final String text)` que converte um texto para uma lista de símbolos.

Exemplo 7. Considere um autômato finito M_1 que reconhece cadeias pertencentes à linguagem regular $L_1 = \{w \in \{a,b\}^* \mid w = (ab)^+\}$, de acordo com a Figura 27. A implementação de tal autômato é apresentada na Figura 28.

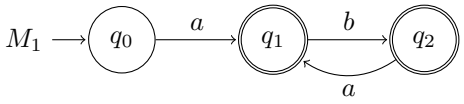

Figura 27. Autômato finito M_1 que reconhece cadeias pertencentes à linguagem regular $L = \{w \in \{a,b\}^* \mid w = (ab)^+\}$.

```
AdaptiveAutomaton aa = new AdaptiveAutomaton() {
 public void setup() {

   State q0 = new StrState("q0");
   State q1 = new StrState("q1");
   State q2 = new StrState("q2");

   Symbol a = new StrSymbol("a");
   Symbol b = new strSymbol("b");

   Set<State> Q = new HashSet<>();
   Q.add(q0); Q.add(q1); Q.add(q2);

   Set<State> F = new HashSet<>();
   F.add(q2);

   Submachine M = new Submachine("M", Q, q0, F);

   Transition t1 = new Transition();
   t1.setTransition(q0, a, q1);

   Transition t2 = new Transition();
   t2.setTransition(q1, b, q2);

   Transition t3 = new Transition();
   t3.setTransition(q2, a, q1);

   submachines.add(M);

   transitions.add(t1);
   transitions.add(t2);
   transitions.add(t3);

   setMainSubmachine("M");
 }
};

System.out.println(aa.recognize(convert("ab")));
System.out.println(aa.recognize(convert("abab")));
System.out.println(aa.recognize(convert("aba")));
```

Figura 28. Implementação do autômato finito M da Figura 27.

O resultado da execução do trecho de código-fonte apresentado na Figura 28 é: `true`, `true`, `false`. □

Exemplo 8. Considere um autômato de pilha estruturado M_2 que reconhece cadeias pertencentes à linguagem livre de contexto $L_2 = \{w \in \{a,b\}^* \mid w = a^n b^n, n \in \mathbb{Z}\}$, de acordo com a Figura 29. A implementação de tal autômato é apresentada na Figura 30.

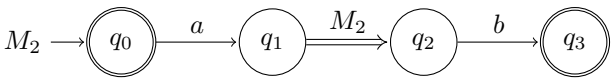

Figura 29. Autômato de pilha estruturado M_2 que reconhece cadeias pertencentes à linguagem livre de contexto $L_2 = \{w \in \{a,b\}^* \mid w = a^n b^n, n \in \mathbb{Z}\}$.

```
AdaptiveAutomaton aa = new AdaptiveAutomaton() {
 public void setup() {

   State q0 = new StrState("q0");
   State q1 = new StrState("q1");
   State q2 = new StrState("q2");
   State q3 = new StrState("q3");

   Symbol a = new StrSymbol("a");
   Symbol b = new StrSymbol("b");

   Set<State> Q = new HashSet<>();
   Q.add(q0); Q.add(q1); Q.add(q2); Q.add(q3);

   Set<State> F = new HashSet<>();
   F.add(q0); F.add(q3);

   Submachine M = new Submachine("M", Q, q0, F);

   Transition t1 = new Transition();
   t1.setTransition(q0, a, q1);

   Transition t2 = new Transition();
   t2.setSubmachineCall(q1, "M", q2);

   Transition t3 = new Transition();
   t3.setTransition(q2, b, q3);

   submachines.add(M);

   transitions.add(t1);
   transitions.add(t2);
   transitions.add(t3);

   setMainSubmachine("M");
 }
};

System.out.println(aa.recognize(convert("ab")));
System.out.println(aa.recognize(convert("aab")));
System.out.println(aa.recognize(convert("aabb")));
```

Figura 30. Implementação do autômato de pilha estruturado M_2 da Figura 29.

O resultado da execução do trecho de código-fonte apresentado na Figura 30 é: `true`, `false`, `true`. □

Exemplo 9. Considere um autômato adaptativo M_3 que reconhece cadeias pertencentes à linguagem dependente de contexto $L_3 = \{w \in \{a,b,c\}^* \mid w = a^n b^n c^n, n \in \mathbb{Z}, n \geq 1\}$, de acordo com a Figura 31. A função adaptativa $\mathcal{A}$ é apresentada no Algoritmo 2. A implementação de tal autômato é apresentada na Figura 32. Considere a existência de um método estático `generateState()` que retorna novos estados (através de um contador interno, por exemplo).

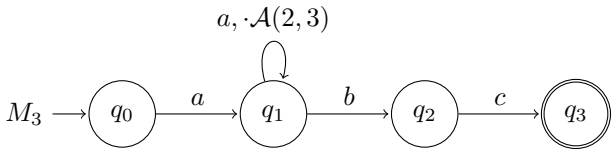

Figura 31. Autômato adaptativo M_3 que reconhece cadeias pertencentes à linguagem dependente de contexto $L_3 = \{w \in \{a,b,c\}^* \mid w = a^n b^n c^n, n \in \mathbb{Z}, n \geq 1\}$.

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

função adaptativa $\mathcal{A}(p_1, p_2)$
variáveis: $?x, ?y$
geradores: $g_1^\star, g_2^\star$
$?(?x, b) \rightarrow p_1$
$-(?x, b) \rightarrow p_1$
$?(?y, c) \rightarrow p_2$
$-(?y, c) \rightarrow p_2$
$-(q_1, a) \rightarrow q_1, \cdot\mathcal{A}(p_1, p_2)$
$+(?x, b) \rightarrow g_1^\star$
$+(g_1^\star, b) \rightarrow p_1$
$+(?y, c) \rightarrow g_2^\star$
$+(g_2^\star, c) \rightarrow p_2$
$+(q_1, a) \rightarrow q_1, \cdot\mathcal{A}(g_1^\star, g_2^\star)$
fim da função adaptativa

O resultado da execução do trecho de código-fonte apresentado na Figura 32 é: `false`, `true`, `true`. □

V. Considerações finais

O código-fonte da biblioteca `AA4J` está disponível para consulta e download em `https://github.com/cereda/aa`, sob a licença GPLv3[3]. O bytecode gerado é compatível com Java 7 ou versões superiores. Espera-se que esta contribuição possa proporcionar subsídios para a implementação de programas que apresentem características adaptativas.

A biblioteca apresentada neste artigo pode contribuir para a implementação de autômatos adaptativos utilizando a linguagem Java de forma consistente e aderentes à teoria. Adicionalmente, programas escritos em outras linguagens que rodam sobre a máquina virtual Java (tais como Groovy, Scala e Clojure) podem compartilhar o mesmo *bytecode* e utilizar as funcionalidades apresentadas.

[3]Disponível em `http://www.gnu.org/licenses/gpl-3.0.html`.

Referências

[1] J. José Neto, "Contribuições à metodologia de construção de compiladores," Tese de livre docência, Escola Politécnica da Universidade de São Paulo, São Paulo, 1993.
[2] ——, "Adaptive automata for context-sensitive languages," *SIGPLAN Notices*, vol. 29, no. 9, pp. 115–124, set 1994.
[3] ——, "Adaptive rule-driven devices: general formulation and case study," in *International Conference on Implementation and Application of Automata*, 2001.
[4] P. R. M. Cereda, "Modelo de controle de acesso adaptativo," Dissertação de mestrado, Departamento de Computação, Universidade Federal de São Carlos, 2008.

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.

```
AdaptiveAutomaton aa = new AdaptiveAutomaton() {
 public void setup() {

  State q0 = new StrState("q0"); State q1 = new StrState("q1");
  State q2 = new StrState("q2"); State q3 = new StrState("q3");

  Symbol a = new StrSymbol("a"); Symbol b = new StrSymbol("b"); Symbol c = new StrSymbol("c");

  Set<State> Q = new HashSet<>(); Q.add(q0); Q.add(q1); Q.add(q2); Q.add(q3);

  Set<State> F = new HashSet<>(); F.add(q3);

  Submachine M = new Submachine("M", Q, q0, F);

  submachines.add(M);

  Transition t1 = new Transition(); t1.setTransition(q0, a, q1);
  Transition t2 = new Transition(); t2.setTransition(q1, b, q2);
  Transition t3 = new Transition(); t3.setTransition(q2, c, q3);
  Transition t4 = new Transition(); t4.setTransition(q1, a, q1);

  t4.setPostActionCall("A");
  t4.setPostActionArguments(Variable.values(q2, q3));

  transitions.add(t1); transitions.add(t2); transitions.add(t3); transitions.add(t4);

  Action adapt = new Action("A") {
   public void execute(Mapping transitions, Transition transition, Object... parameters) {

    Symbol a = new StrSymbol("a"); Symbol b = new StrSymbol("b");
    Symbol c = new StrSymbol("c"); State q1 = new StrState("q1");

    ElementaryActions ea = new ElementaryActions(transitions);

    Variable p1 = new Variable(parameters[0]); Variable p2 = new Variable(parameters[1]);
    Variable g1 = new Variable(generateState()); Variable g2 = new Variable(generateState());
    Variable x = new Variable(); Variable y = new Variable();

    ea.query(x, new Variable(b), p1); ea.remove(x, new Variable(b), p1);
    ea.query(y, new Variable(c), p2); ea.remove(y, new Variable(c), p2);

    ea.remove(new Variable(q1), new Variable(a), new Variable(q1), new ActionQuery(new Variable("A"), p1, p2));

    ea.add(x, new Variable(b), g1); ea.add(g1, new Variable(b), p1);
    ea.add(y, new Variable(c), g2); ea.add(g2, new Variable(c), p2);
    ea.add(new Variable(q1), new Variable(a), new Variable(q1), new ActionQuery(new Variable("A"), g1, g2));
   }
  };

  actions.add(adapt);

  setMainSubmachine("M");
 }
};

System.out.println(aa.recognize(convert("aabbc")));
System.out.println(aa.recognize(convert("aabbcc")));
System.out.println(aa.recognize(convert("aaabbbccc")));
```

Figura 32. Implementação do autômato adaptativo M_3 da Figura 31.