An adaptive automata operational semantic

Ítalo Santiago Vega, *Professor, PUC/SP, FIRB/SP*

***Abstract*— This work presents a model of software of the semantics for the adaptive automata (AA) in terms of sequences of messages between objects. The elaboration of the model is based on the formularization proposal of J.J. Neto citeneto:aa:2001 as primary origin of the class objects and its respective behaviors. Several techniques of software engineering had been used throughout the work for the elaboration of the model, such as the domain modeling, behavior specification and design patterns. The resultant model complements the original formularization providing an operational semantics for the AA.**

***Keywords*— adaptive automata, software models, design patterns, UML, Ruby.**

I. Introdução

A TEORIA dos autômatos adaptativos (AA) foi criada por J. J. Neto como uma extensão de um trabalho anterior referente a autômatos estruturados [1], em uma linha de trabalho similar a [2] e [3]. Um dispositivo é dito adaptativo sempre que seu comportamento e topologia sofrerem alterações, dinamicamente, como resposta a algum estímulo de entrada, mesmo na ausência de agentes externos [4][17]. Do ponto de vista semântico, [5] descreve uma representação de tal categoria de autômatos na forma de tabelas de decisão as quais são tradicionalmente utilizadas para representar, declarativamente, a semântica de um modelo.

O presente trabalho divulga um modelo alternativo para a semântica da proposta geral de formulação de autômatos adaptativos baseada nas tabelas de decisão de [5]. O modelo, que pode ser classificado como sendo de semântica operacional, pressupõe uma particular implementação de autômato na forma de um conjunto de objetos. Com este enfoque, as trocas de mensagens entre eles são utilizadas para representar a operação de um AA.

Espera-se que este modelo operacional complemente a formulação geral apresentada por Neto em [5] introduzindo um novo modelo de execução de uma transição adaptativa sob um ângulo operacional de trocas de mensagens. Durante a construção deste modelo, procurou-se aplicar técnicas de engenharia de software com objetos visando a elaboração de uma arquitetura com suporte ao entendimento por segmentos do funcionamento operacional de um autômato adaptativo.

I. Vega é docente do Departamento de Ciência da Computação, Pontifícia Universidade Católica de São Paulo e das Faculdades Integradas Rio Branco, São Paulo, SP, Brazil. E-mail: `italo@pucsp.br`.

Diversas ferramentas foram empregadas ao longo da execução deste trabalho. Experimentalmente, o modelo da semântica operacional foi codificado na linguagem *Ruby* [9] [10]. Aspectos relevantes da arquitetura foram registrados em diagramas UML [11] [12] produzidos com as ferramentas *SDEdit* [14] e *UMLet* [13]. Finalmente, os diagramas do estudo de caso de um autômato adaptativo foram criados com a ferramenta *Graphviz* [15].

O restante do artigo tem a seguinte estrutura. A seção II caracteriza os elementos conceituais da teoria dos autômatos adaptativos relevantes para este trabalho, destacando-se a transição adaptativa, a transição subjacente e a função adaptativa. A seção III considera o modelo comportamental de um autômato adaptativo durante a execução dos movimentos que levam de uma configuração à seguinte. A seção IV introduz um padrão de projeto para resolver o problema de agrupar as diferentes formas de se realizar uma transição sem aumentar a complexidade ciclomática do modelo utilizado para representar o disparo. A seção V trata do modelo de uma função adaptativa, incluindo o efeito provocado pelas ações básicas. A seção VI combina o mecanismo de disparo com as funções adaptativas e transições subjacentes para representar o disparo de uma transição adaptativa. Finalmente, a seção VII ilustra o funcionamento do modelo de objetos na forma de um estudo de caso de operação de um autômato adaptativo, seguida de conclusões (seção VIII).

II. Elementos Estruturais de um Autômato Adaptativo

Sob o ponto de vista estrutural, um AA é constituído por um conjunto de transições adaptativas, cada uma delas contendo uma transição subjacente e, opcionalmente, uma função adaptativa *before* e uma função adaptativa *after*.

Uma típica transição adaptativa, identificada por `[1]`, é ilustrada na Fig 1. A transição subjacente leva o autômato do `P` para o estado `Q`, ao ser realizada, caso o símbolo da entrada seja `a`. Esta transição adaptativa possui, também, duas funções adaptativas, com nomes `X` e `Y`. A primeira ocupa o papel *before* e a segunda, o papel *after*.

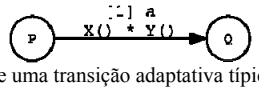


Fig. 1: Representação de uma transição adaptativa típica.

No modelo de objetos deste trabalho, a transição `[1]` é representada por objetos conforme o diagrama da Fig 2. Um objeto da classe `Transição Adaptativa` denota a transição `[1]`. Dois atributos caracterizam uma transição adaptativa: `corrente` e `símbolo`. O valor de cada um indica os correspondentes elementos ilustrados na Fig 2. Além disso, este objeto encontra-se conectado a outros três objetos, representando a transição subjacente e as duas funções adaptativas. O objeto `ts` é uma instância da classe `Transição Subjacente` e representa a transição subjacente. No seu modelo encontra-se o seguinte. O seu valor indica o correspondente elemento ilustrado na Fig 2. Os papéis `funçãoAB` e `funçãoAA` são ocupados por objetos da classe `Função Adaptativa`. O atributo nome de tais objetos possui um valor que coincide com o nome de uma função adaptativa existente na ilustração da Fig 2.

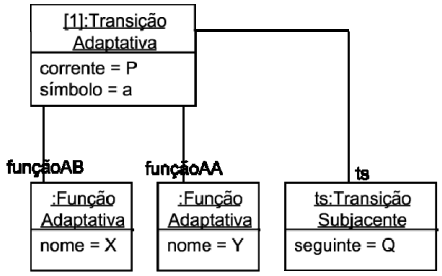


Fig. 2:. Representação da transição `[1]` (diagrama de objetos UML).

O modelo de objetos de uma transição adaptativa sugere uma estrutura de classes como aquela apresentada no diagrama da Fig 3. Na mesma figura pode-se observar que as transições adaptativas encontram-se organizadas na forma de uma estrutura do tipo lista no escopo do AA. Tal escopo ainda inclui uma outra estrutura do tipo mapa[1] de funções adaptativas. Cada transição adaptativa é representada por um objeto constituído por outro que simboliza a transição subjacente. Além disso, uma transição adaptativa oferece dois papéis para as funções adaptativas: `funçãoAB` e `funçãoAA`, denotando, respectivamente, a possível presença de uma função adaptativa *before* e uma função *after*.

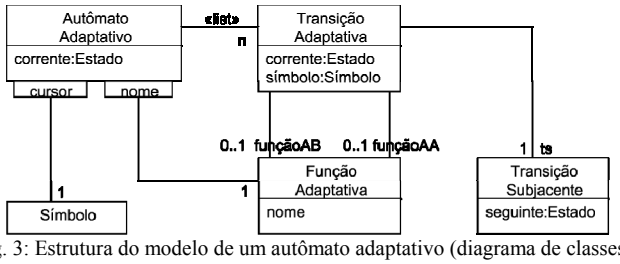


Fig. 3: Estrutura do modelo de um autômato adaptativo (diagrama de classes UML).

O atributo `nome` permite acessar as funções adaptativas que se encontram no mapa do autômato. Este mapa é definido na instanciação da classe `Autômato Adaptativo` não sendo posteriormente alterado. Este não é o caso da lista de transições adaptativas que pode ser modificada por uma ação elementar de uma função adaptativa, como será discutido na seção V.

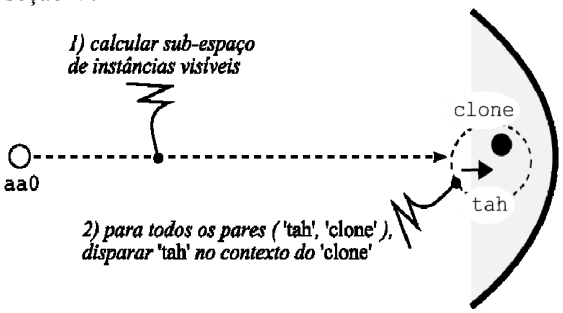


Fig. 4: Estratégia para a realização de uma transição adaptativa.

Finalmente, o símbolo da cadeia de entrada a ser analisada por um objeto da classe `Autômato Adaptativo` é indicado pelo qualificador `cursor`. O símbolo indicado pelo cursor e o valor do estado `corrente` são chave para que um `Autômato Adaptativo` calcule o seu sub-espaço de clones, a ser discutido na próxima seção.

III. Sub-espaço de Autômatos Visíveis

Um autômato adaptativo move-se de uma configuração para a seguinte ao realizar uma transição adaptativa. O efeito produzido pela transição se revela pela eventual troca de estado do autômato (com ou sem consumo de um símbolo da cadeia de entrada). A troca de estado provoca uma mudança no valor do atributo `corrente` de um `Autômato Adaptativo`. O consumo do símbolo de entrada se reflete no valor do `cursor`. Mas o efeito de uma transição adaptativa também pode ser sentido em uma possível alteração na lista de transições do autômato.

O modelo de execução de movimentos proposto neste trabalho envolve dois passos:

1. o cálculo do sub-espaço de instâncias de autômatos visíveis em uma determinada configuração, cada um deles vinculado a uma transição adaptativa pronta para ser realizada e
2. a subseqüente realização das transições habilitadas, considerando-se o sub-espaço anteriormente calculado.

O primeiro passo envolve a identificação de todas as transições adaptativas que podem ser realizadas pelo autômato. A efetiva realização de tais transições ocorrerão em clones do contexto do autômato no qual elas se encontram habilitadas à realização. Desta forma, o conjunto de pares transição adaptativa habilitada-clone constitui o que se entende por sub-espaço de instâncias de autômatos visíveis ou, simplesmente, sub-espaço de clones visíveis. Na Fig 4, o autômato `aa0` primeiro calcula o seu sub-espaço de clones visíveis (como aquele identificado por `clone`) vinculando a cada um deles uma transição adaptativa habilitada `tah` pronta para ser realizada.

Em um segundo momento, o autômato `aa0` envia uma mensagem para cada clone do seu sub-espaço, solicitando para que ele realize a transição vinculada dentro seu próprio contexto.

Ao término da realização da transição, cada clone, por sua

[1] Também referido como dicionário ou *array* associativo.

vez, calcula o seu sub-espaço de clones visíveis, vinculando-os a transições a serem realizadas. Esta dinâmica origina um efeito de "ondas de sub-espaços de clones", conforme sugerido pela ilustração da Fig 5.

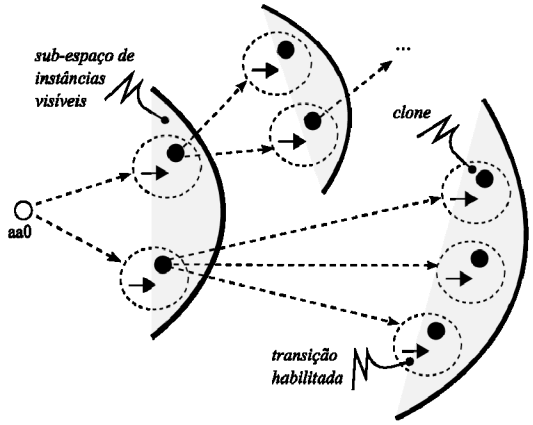


Fig. 5: Ilustração das ondas de sub-espaços de clones.

As ondas se propagam até que, eventualmente, um clone não se encontre em alguma configuração que admita a realização de uma transição. Neste caso, a onda de clones não se forma pois o cálculo do seu sub-espaço produz um conjunto vazio de pares transição-autômato.

A. Transições Habilitadas

O primeiro passo do modelo de execução de movimentos de um autômato envolve o cálculo do sub-espaço de clones. Este cálculo tem início na determinação de quais transições adaptativas encontram-se habilitadas para o disparo no contexto do autômato. Uma transição adaptativa `t` está habilitada para o disparo se:

1. `t.corrente` for o estado `corrente` do seu AA, e
2. `t.símbolo` for o símbolo da entrada pelo `cursor` do AA (exceto no caso de uma transição em vazio).

Em Ruby [10], esta condição pode ser expressa na classe `Transição Adaptativa` como[2]:

```
# TransiçãoAdaptativa:
def habilitada?( automato )
   corrente = automato.corrente
   simbolo = automato.simboloLido()
   # Condição de habilitação:
 return (@condicao == [ corrente, simbolo ])
end
```

A variável `@condicao` denota um *array* contendo os valores dos atributos `corrente` e `símbolo` da classe `Transição Adaptativa` (Fig 3). Ruby suporta a comparação direta entre dois *array*s, conforme se observa no comando que retorna a condição de habilitação.

B. Cálculo do Sub-espaço de Instâncias Visíveis

O cálculo do sub-espaço pode ser implementado no método `calcularVisiveis`() da classe `Autômato Adaptativo`:

```
# AutômatoAdaptativo:
def calcularVisiveis()
   subEspacoAutomatosVisiveis = {}
   @contextoFA.listaTAs.each do |t|
     if( t.habilitada?( @contextoTS ) )
       subEspacoAutomatosVisiveis[ t ] = clonar()
     end
   end
   return subEspacoAutomatosVisiveis
end
```

Inicialmente, considera-se o sub-espaço vazio (implementado como um *hash* vazio em Ruby). Em seguida, para cada uma das transições `t` do AA[3], se ela estiver habilitada então o autômato cria um clone, vinculando-o à transição `t` no sub-espaço de autômatos visíveis. Neste sentido, a variável `subEspacoAutomatosVisiveis` mantém um mapa indexado pelas transições adaptativas habilitadas no contexto do autômato que conduz o cálculo do sub-espaço.

Uma vez calculado o seu sub-espaço, o autômato percorre os clones produzidos solicitando a realização da transição habilitada, mas dentro do contexto de cada um deles.

IV. Realização de Transições

A. Realização de uma Transição Não-Adaptativa

A realização de uma transição não-adaptativa habilitada provoca uma possível mudança no estado do AA, eventualmente avançando o cursor sobre um símbolo da cadeia de entrada. Para confinar o efeito produzido pela realização de uma transição não-adaptativa, decidiu-se pela criação de uma nova classe chamada `ContextoTS`, responsável por encapsular as variáveis que podem ser afetadas pela realização de uma transição subjacente: estado `corrente` e `cursor` do AA. Assim, a ação do método `realizarTS()` da classe `TransicaoSubjacente` pode ser descrita da seguinte forma[4]:

```
# TransicaoSubjacente:
def realizarTS( contextoTS )
   if( @seguinte != "." )
      contextoTS.corrente = @seguinte
   end
   if( @movimentoCursor != "." )
      contextoTS.executar( @movimentoCursor )
   end
end
```

O novo estado `corrente` do AA (memorizado no `contextoTS`) corresponde ao estado `seguinte` definido pela transição subjacente. O avanço do cursor ocorre se for especificado na transição subjacente (seu valor deve ser diferente de `"."`).

O diagrama UML apresentado na Fig 6 destaca a dependência dinâmica entre as classes `Transição Subjacente` e `contextoTS`. O método `realizarTS()` de

[2] Por questões de simplificação do modelo, pressupõe-se apenas transições subjacentes determinísticas.

[3] As variáveis de instância `@contextoFA` e `@contextoTS` armazenam, respectivamente, a lista de transições do AA e o par constituído pelo estado corrente e posição do cursor na cadeia de entrada. A justificativa para tal decisão não será discutida neste artigo.

[4] O valor `"."` denota uma invariância de estado ou do movimento do cursor.

uma transição subjacente pode alterar o estado `corrente` e o `cursor` do `ContextoTS` que ela recebe como argumento.

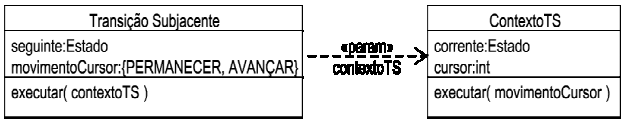


Fig. 6: Dependência entre uma relação subjacente o contexto do seu efeito (diagrama de classes UML).

A execução deste método, por sua vez, é desencadeada pela realização de uma transição adaptativa sem funções adaptativas. Com o objetivo de facilitar a comparação entre os modelos de realização de transições adaptativas e subjacentes, decidiu-se pela introdução de uma nova classe chamada `MecanismoRealizaçãoTransições`, conforme ilustrado na Fig 7.

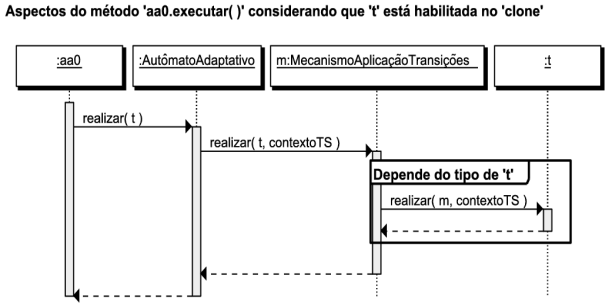


Fig. 7: Papel do mecanismo de realização de transições (diagrama de seqüências de mensagens UML).

O efeito da mensagem enviada pelo mecanismo `m` para o objeto `t` depende do tipo da transição envolvida.

B. Mecanismo de Realização de Transições Não-Adaptativas

O mecanismo de realização de transições, ao ser ativado por um autômato adaptativo, envia uma mensagem envolvendo a operação `realizar()` para o objeto `t` da classe `TransiçãoSubjacente` (Fig 8). A transição `t` reage enviando a mensagem `realizarTS()` para o mecanismo `m`. Finalmente, este objeto solicita que a transição execute o método `realizarTS()` considerando o `contextoTS`, conforme descrito na seção IV-A.

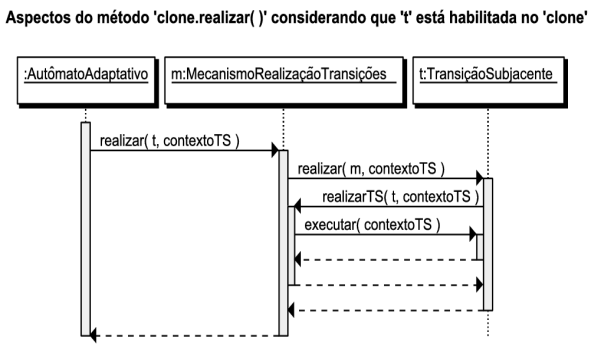


Fig. 8: Realização de uma transição subjacente não-adaptativa (diagrama de seqüências de mensagens UML).

Em Ruby, esta parte do modelo pode ser implementada da seguinte forma:

```
# MecanismoRealizaçãoTransições:
def realizar( t, contextoTS )
   t.realizar( self, contextoTS )
end
```

O modelo comportamental representado pelo diagrama da Fig 8 e decorrente da execução do método `realizar()` anteriormente apresentado estabelece uma dependência dinâmica cíclica entre as classes dos objetos `m` e `t`.

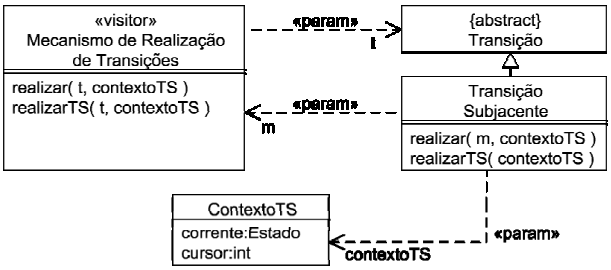


Fig. 9: Dependências na estrutura de transições não-adaptativas (diagrama de classes UML).

Com o objetivo de encapsular as diferentes estratégias de realização de transições entre os estados de um autômato e também desacoplar (estaticamente) as classes envolvidas na dependência cíclica, optou-se pela utilização de um padrão conhecido por `Visitor` [8][6][7]. Este padrão de projeto faz uso do tipo de dois argumentos para selecionar o método a ser executado, permitindo separar a estrutura de uma coleção de objetos (da classe `Transição Adaptativa`, neste caso) das operações sobre eles.

A Fig 9 ilustra a estrutura estática envolvendo as classes `MecanismoRealizaçãoTransições` e `Transição`. O padrão de projeto `Visitor` resolve a dependência cíclica pela introdução de uma classe abstrata. No modelo proposto neste trabalho a classe abstrata recebeu o nome de `Transição`. Além disso, a classe `MecanismoRealizaçãoTransições` encapsula a operação `realizarTS()`, utilizada para implementar a parte do modelo referente à semântica de uma transição subjacente não-adaptativa.

O diagrama também destaca que a ação de uma transição subjacente se restringe à instância da classe `ContextoTS`, ou seja, a realização de uma transição subjacente pode afetar os valores dos atributos estado `corrente` e `cursor` do contexto.

C. Realização de uma Transição Adaptativa

O disparo de uma transição adaptativa do autômato envolve a sua função adaptativa *before*, a sua transição subjacente e a sua função adaptativa *after*. No caso de transições adaptativas, o método `realizarTA()`é executado pelo mecanismo conforme o comportamento proposto no formalismo de um AA (Fig 10).

O mecanismo envia a mensagem `aplicarFAB()`, provocando a realização da ação da função adaptativa *before*. Caso esta função não elimine a transição `t`, o mecanismo comanda a execução da transição subjacente e, em seguida, a envia a mensagem `aplicarFAA()`.

Em Ruby, o método `realizarTA()` da classe `MecanismoRealizaçãoTransições` implementa a semântica de uma transição adaptativa da seguinte maneira:

```
# MecanismoRealizaçãoTransições:
def realizarTA( t, listaTAs, contextoTS )
   t.aplicarFAB()
   if( listaTAs.include?( t ) )
      t.realizarTS( contextoTS )
      t.aplicarFAA()
   end
end
```

Este modelo dinâmico também é resolvido pelo padrão `Visitor` conforme ilustrado no diagrama da Fig 11. Ao ser realizada, uma transição adaptativa atua sobre as variáveis do `contextoTS` (quando executa a sua transição subjacente) e sobre a lista de transições adaptativas do AA (quando aplica as suas funções adaptativas).

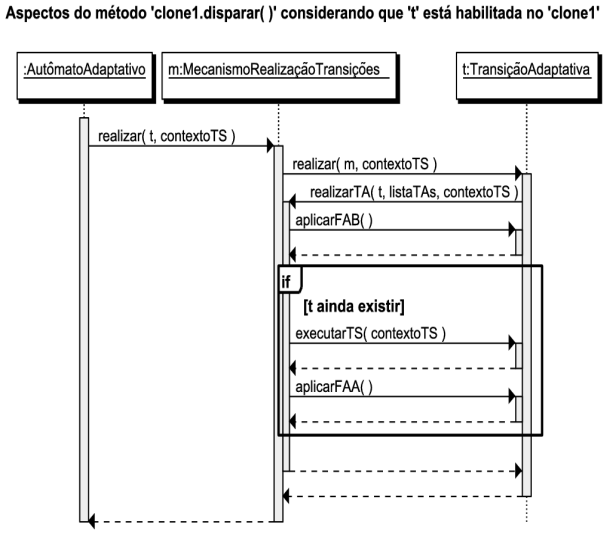


Fig. 10: Realização de uma transição adaptativa (diagrama de seqüências de mensagens UML).

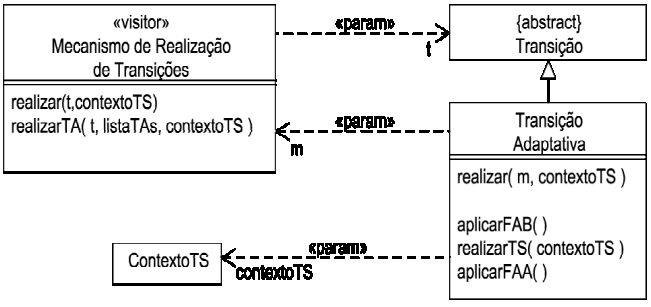


Fig. 11: Dependências na estrutura de transições adaptativas (diagrama de classes UML).

V. Comportamento de uma Função Adaptativa

Em termos de comportamento, a aplicação de uma função adaptativa pode alterar a lista de transições adaptativas do autômato. Mais especificamente, a execução das ações elementares do seu corpo pode provocar mudanças nas transições adaptativas do AA. No escopo deste artigo, apenas as funções de remoção e de inclusão serão discutidas.

A remoção de transições adaptativas da lista do autômato é resultado da aplicação de funções adaptativas contendo ações elementares de remoção. Uma ação elementar de remoção exclui uma particular transição adaptativa do contexto da função adaptativa aplicada. No modelo proposto neste trabalho, a classe `ContextoFA` encapsula a lista de transições adaptativas, delimitando o escopo de atuação de uma função adaptativa, conforme a estrutura representada pelo diagrama da Fig 12. O `corpo` de uma função adaptativa pode ser programado para retirar ou adicionar uma transição da `listaTAs` do `contextoFA` do autômato. Os parâmetros das operações `retirar()` e `adicionar()` são utilizados para identificar a transição adaptativa a ser retirada ou a estrutura da transição a ser adicionada na lista.

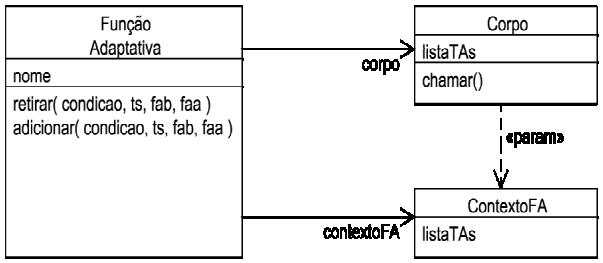


Fig. 12: Dependência entre uma função adaptativa e o contexto do seu efeito (diagrama de classes UML).

No modelo em Ruby, a remoção feita por uma função adaptativa é codificada por:

```
# FunçãoAdaptativa:
def retirar( condicao, ts, fab, faa )
   @contextoFA.retirarTA( condicao, ts )
end
```

De modo similar, a adição de uma nova transição adaptativa é codificada por:

```
# FunçãoAdaptativa:
def adicionar( condicao, ts, fab, faa )
  @contextoFA.adicionarTA( condicao, ts, fab, faa )
end
```

VI. Método de Execução de Movimentos de um Autômato Adaptativo

O método `executar()` compõe os elementos do modelo apresentado nas seções anteriores, implementando o comportamento principal de um objeto da classe `Autômato Adaptativo`[5]:

```
# AutômatoAdaptativo:
def executar()
   automatosGerados = [ self ]
   visiveis = calcularVisiveis()
```

[5] A variável `automatosGerados` mantém uma lista contendo todos os autômatos adaptativos gerados por todas as ondas de sub-espaços.

```
   while( visiveis != {})
      << realizar as transições habilitadas >>
      << calcular novo sub-espaço de instâncias >>
      automatosGerados.concat( visiveis.values )
   end
   return automatosGerados
end
```

Inicialmente, o autômato calcula o sub-espaço de instâncias visíveis, armazenando-o na variável `visiveis`. Enquanto existir alguma transição habilitada neste sub-espaço, o autômato encaminha a sua realização para o respectivo clone:

```
<< realizar as transições habilitadas >>=
visiveis.each do |raHabilitada, clone|
   clone.realizar( raHabilitada )
end
```

Por fim, ele faz a união dos sub-espaços visíveis dos clones e reinicia o processo:

```
<< calcular novo sub-espaço de instâncias >>=
novoSubEspaco = {}
visiveis.values.each do |clone|
  subEspacoDoClone = clone.calcularVisiveis()
  novoSubEspaco.update( subEspacoDoClone )
end
visiveis = novoSubEspaco
```

A operação `update()` atualiza o mapa `novoSubespaco` com novos pares transição-clone, tornando-se o sub-espaço de instâncias visíveis da próxima onda.

VII. Estudo de Caso

A Fig 13 ilustra a situação inicial de um autômato adaptativo. Estados são representados por círculos. Um círculo preenchido denota o estado corrente (`I`, nesta figura). Círculos com linhas espessas destacam estados finais. Transições são representadas por arcos dirigidos. Assim, o arco ligando os círculos dos estados `I` e `J` representa a transição adaptativa `[1]`. As transições `[1]` e `[5]` envolvem funções adaptativas, enquanto as transições `[3]` e `[4]` não. As transições `[2]`, `[6]`, `[7]` e `[8]` denotam transições em vazio.

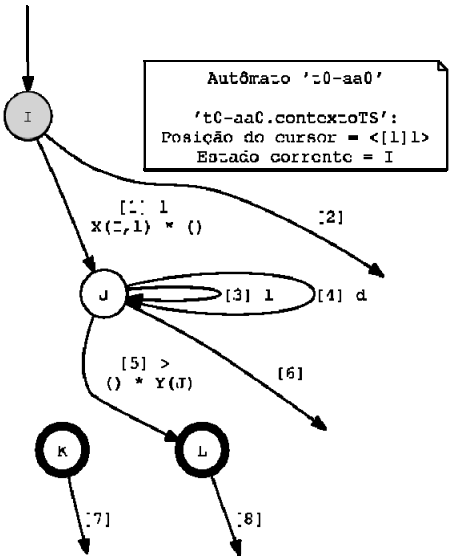


Fig. 13: Configuração do `aa0` no instante `t0`.

Em Ruby, pode-se instanciar o objeto `@aa0` da classe `AutomatoAdaptativo`, iniciando-o com a cadeia `<ll>` da seguinte forma:

```
@aa0 = AutomatoAdaptativo.new()
@aa0.iniciar( "<ll>", "I" )
```

Neste caso, as transições adaptativas são assim codificadas:

```
@aa0.adicionarTA(
["I","l"], ["J","v","."], ["X", ["I","l"]], [])
@aa0.adicionarTA(
["I", "." ], ["." , "." , "x"], [], [] )
@aa0.adicionarTA(
["J", "l" ], ["J", "v", "."], [], [] )
@aa0.adicionarTA(
["J", "d" ], ["J", "v", "."], [], [] )
@aa0.adicionarTA(
["J",">" ], ["L","."," ."], [], ["Y",["J"] ])
@aa0.adicionarTA(
["J", "." ], ["." , "." , "x"], [], [] )
@aa0.adicionarTA(
["K", "." ], ["." , "." , "v"], [], [] )
@aa0.adicionarTA(
["L", "." ], ["." , "." , "v"], [], [] )
```

O corpo da função adaptativa `X` é definido de tal forma a retirar uma transição adaptativa, inserindo cinco outras, de acordo com os argumentos que lhe são passados no momento da ativação:

```
corpo = proc do |x, args|
   p1 = args[0]
   p2 = args[1]
   g1 = x.contextoFA.gerador.g()
   x.retirar( [p1, p2], ["J","v","."],
      ["X", [p1, p2]], [])
   x.adicionar( [p1, p2], [g1,"v","."],
      [], [] )
   x.adicionar( [g1,"l"], ["J","."," ."],
      ["X", [g1,"l" ] ], [] )
   x.adicionar( [g1,"d"], ["J","."," ."],
      ["X", [g1,"d" ] ], [] )
   x.adicionar( [g1,">"], ["L","."," ."],
      [], ["Y", [ g1 ] ] )
   x.adicionar( [g1,"."], ["."," ."," x"],
      [], [] )
end
@aa0.adicionarFA( "X", corpo )
```

A função adaptativa `Y`, por outro lado, retira e adiciona transições adaptativas como especificado por:

```
corpo = proc do |y, args|
   q1 = args[0]
   y.retirar( [ q1, ">"], ["L", ".", "."],
      [], [ "Y", [ q1 ] ] )
   y.adicionar( [ q1, ">"], ["K", ".", "."],
      [], [] )
end
@aa0.adicionarFA( "Y", corpo )
```

A. *Situação* `t0`

Considerando que o `cursor` do `contextoTS` do autômato `@aa0` indique o símbolo `l` e que o autômato se encontre no estado `I`, apenas a transição `[1]` está habilitada para o disparo. Neste caso, o autômato `aa0` cria o clone `clone1`, vinculando-o à transição `[1]` no subespaço de instâncias visíveis: `[1]` → `clone1`. Concluída esta etapa, o autômato

`aa0` solicita que o `clone1` realize a transição `[1]`. (Tal transição deverá ser realizada no contexto do `clone1`.)

B. *Instante* `t1`

Ao realizar a transição `[1]` a topologia do `clone1` se altera da seguinte forma. A função adaptativa `X` remove a própria transição `[0]` e adiciona cinco novas transições: `[9]`, `[10]`, `[11]`, `[12]`, `[13]` (Fig 14).

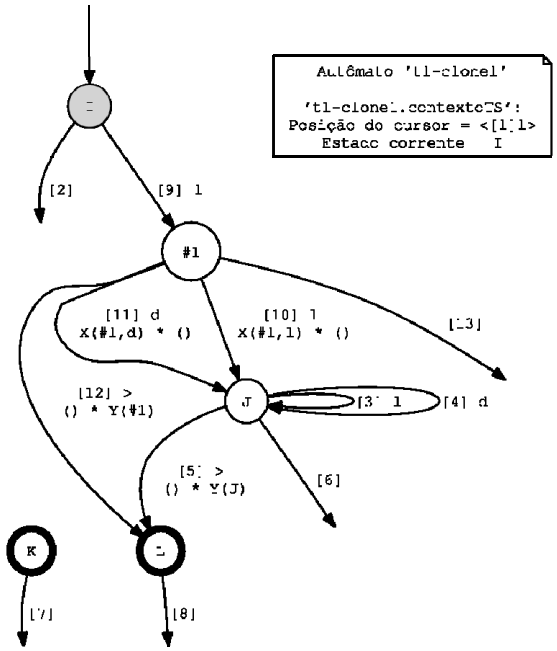


Fig. 14: Configuração do `clone1` no instante `t1`.

Observa-se que a aplicação da função adaptativa `X`, elimina a transição `[1]`. Em vista disso, o `clone1` encerra sua ação permanecendo no estado `I`, sem consumir o símbolo de entrada.

Por questões de simplicidade, a Fig 15 ilustra a topologia do `clone1` sem as transições em vazio. A transição `[9]` corresponde à transição `[1]` sem a função adaptativa *before* `X`. É esta transição que se encontra habilitada no instante `t1`.

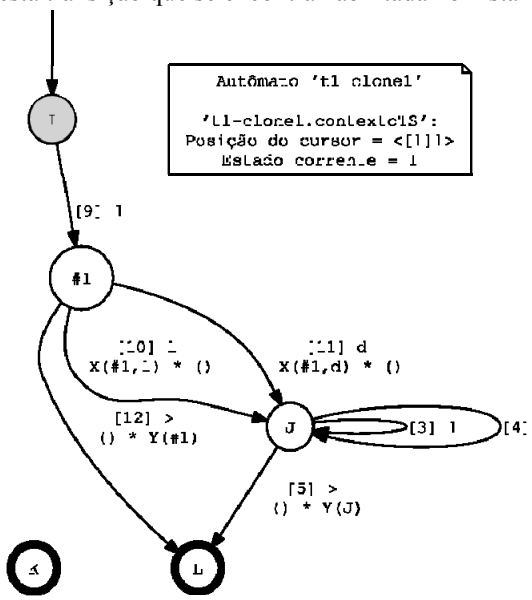


Fig. 16: Configuração do `clone1` no instante `t1` (sem as transições em vazio).

C. *Instante* `t2`

Ao realizar a transição `[9]` a topologia do `clone2` não se altera pois tal transição não possui funções adaptativas. No contexto do `clone2`, a transição `[9]` do `clone1` passa a ter identidade `[8]`. Ao ser disparada, o estado do `clone2` muda para `#1` com consumo do símbolo de entrada (Fig 16). Após o disparo da transição `[8]`, o `clone2` calcula um novo subespaço de instâncias visíveis. Cada novo clone deve estar associado a uma única transição habilitada na configuração corrente do `clone2`. Neste caso, apenas a transição `[9]` pode disparar, sendo a ela associada o `clone3`.

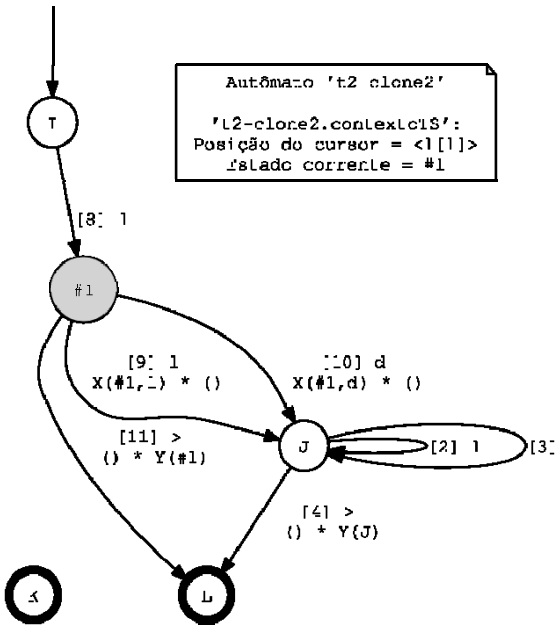


Fig. 17: Configuração do `clone2` no instante `t2` (sem as transições em vazio).

D. *Instante* `t3`

No contexto do `clone3`, apenas a transição `[9]` está habilitada. A aplicação da sua função adaptativa *before* (identificada por `X`) desencadeia alterações na topologia do autômato (Fig 17). A própria transição `[9]` é eliminada, sendo adicionadas as transições `[13]`, `[14]`, `[15]`, `[16]` e `[17]` (esta última trata-se de uma transição em vazio).

Considerando a sua nova configuração, o autômato `clone3` deve calcular as transições habiliatadas e vinculá-las a clones próprios. Como o disparo da transição `[9]` provoca a sua eliminação, o `clone3` não troca de estado, nem consome símbolos da entrada. Assim, apenas a transição `[13]` está habilitada, sendo associada ao `clone4`.

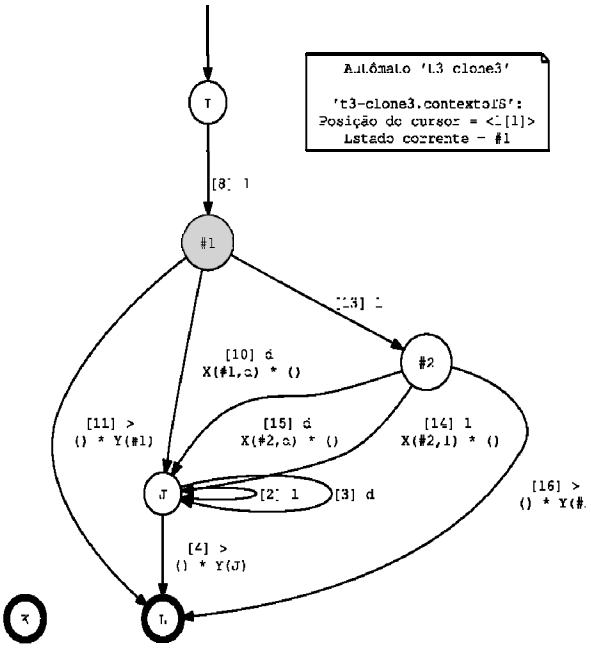


Fig. 18: Configuração do `clone3` no instante `t3` (sem as transições em vazio).

E. *Instante* `t4`

No contexto do `clone4`, a transição `[13]` recebe a identificação `[12]` (Fig 18). Por se tratar de uma transição sem funções adaptativas, o seu disparo apenas provoca uma mudança de estado e o consumo do símbolo de entrada. Nenhuma alteração é feita nas transições deste autômato.

Após o disparo da transição `[12]`, o autômato passa para o estado `#2`, consumindo o símbolo `1` e avança o cursor para o símbolo `>`. Neste momento, o `clone4` deve calcular o novo subespaço de instâncias visíveis. A única transição habilitada neste caso é identificada por `[15]` e possui uma função adaptativa *after* chamada `Y`.

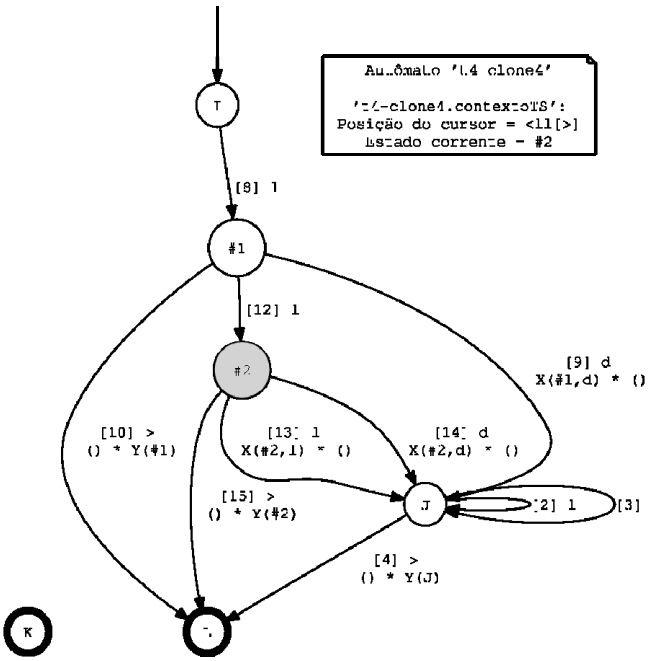
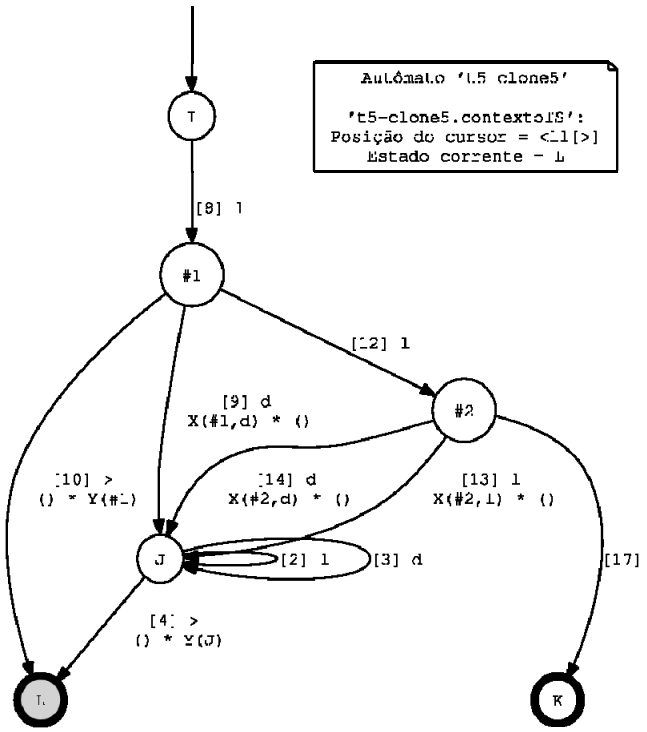


Fig. 19: Configuração do `clone4` no instante `t4` (sem as transições em vazio).

F. *Instante* `t5`

O tratamento do disparo da transição `[15]` é feito pelo `clone5`. Inicialmente, ocorre uma mudança no seu estado: ele passa do estado `#2` para o estado `L`, consumindo o símbolo `>`, mas sem avanço do cursor. Em seguida, ele aplica a função adaptativa *after* denominada `Y` com argumento `#2`. Esta função retira a transição `[15]` e adiciona a transição `[17]` (Fig 19).

Após a transformação, o `clone5` atinge o estado `L` e nenhuma outra transição se torna habilitada. Por conseguinte, o cálculo do subespaço de instâncias visíveis retorna o conjunto vazio encerrando a "onda de clones".



Configuração do `clone5` no instante `t5` (sem as transições em vazio).

A Fig 20 apresenta uma vista do `clone5` com as transições em vazio. Supondo-se que este autômato possa ser reiniciado no estado `I` com a entrada `<ll>`, uma nova solicitação de reconhecimento não provocará alterações na lista de transições adaptativas. Em particular, este autômato percorrerá os estados `I`, `#1`, `#2` e `K`, acusando o reconhecimento da cadeia de entrada.

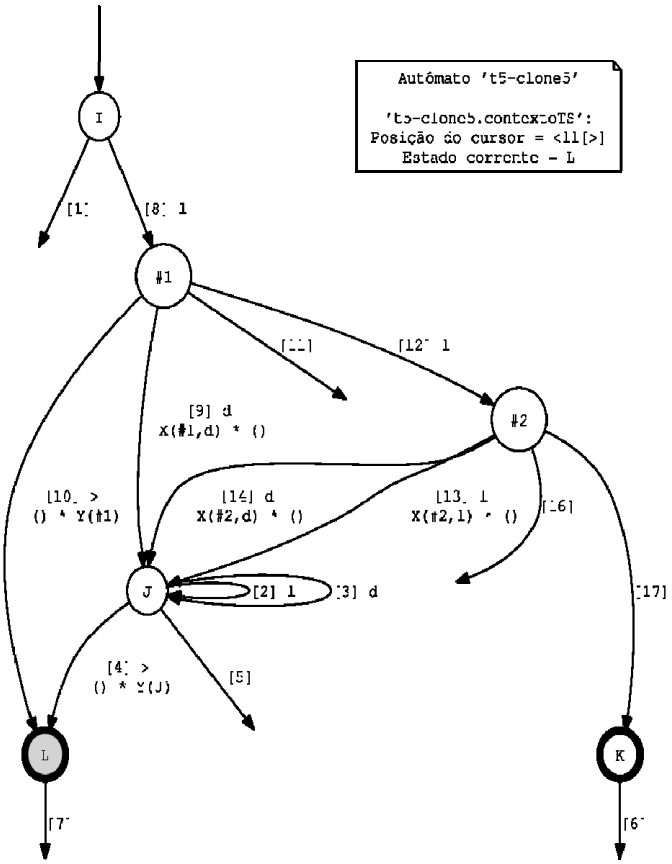


Fig. 20 Configuração do `clone5` no instante `t5` (sem as transições em vazio).

G. Vista da Onda de Clones

A Fig 21 apresenta uma ilustração das ondas de clones do caso estudado. Observa-se que, por ser um reconhecimento determinístico, o `aa0` desencadeia uma série de ondas, cada uma delas envolvendo um único clone. Isto ocorre pois os passos de evolução das configurações originam-se a partir de uma transição habilitada por onda.

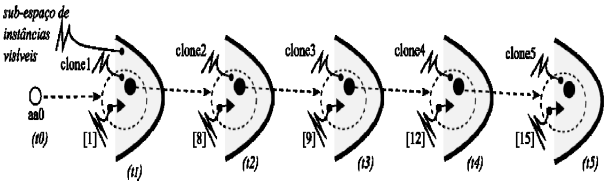


Fig. 21 Ilustração das ondas de clones do caso estudado.

Assim, inicialmente o autômato `aa0` contém apenas a transição `[1]` habilitada, provocando a criação do clone `clone1`. Este, por sua vez, ao executar a transição `[1]`, habilita apenas a transição `[8]`. Por conseguinte, o `clone1` precisa produzir apenas o `clone2`. Este comportamento se repete até que o `clone5`, ao executar a transição `[15]` atinge uma configuração na qual nenhuma transição fica habilitada e o processo de geração de clones se encerra, finalizando os movimentos originados pelo autômato `aa0`.

Uma situação de reconhecimento mais geral poderia apresentar diversos clones por onda, e várias ondas, uma para cada um dos clones quando estes executam a sua transição habilitada, como sugerido na Fig 5.

VIII. Conclusões

Este trabalho apresentou um modelo da semântica operacional dos autômatos adaptativos, complementando a representação em tabelas de decisão de [5]. Ao longo do processo de elaboração do modelo, técnicas de engenharia de software foram aplicadas visando a criação de uma arquitetura que suportasse o estudo modular do comportamento de um AA. Em particular, a estratégia de distribuição de responsabilidades foi diretamente influenciada pelos trabalhos de [6] e [7]. A proposta e discussão do processo de concepção do modelo operacional é um aspecto pouco encontrado em trabalhos congêneres.

Neste artigo, em particular, foram discutidos os aspectos relevantes das classes `TransiçãoAdaptativa`, `TransiçãoSubjacente`, `FunçãoAdaptativa` e `MecanismoRealizaçãoTransições`, centrais para o modelo da semântica operacional. A classe `Mecanismo RealizaçãoTransições` encapsula o modelo de realização de transições adaptativas e subjacentes. Neste sentido, ela incorpora dependências das classes `TransiçãoAdaptativa` e `TransiçãoSubjacente`. Estas, por sua vez, decidem quais operações devem ser executadas pelo `Mecanismo RealizaçãoTransições`, estabelecendo uma dependência cíclica. O padrão de projeto `Visitor` [8] foi utilizado para resolver a dependência cíclica, preservando a encapsulação do modelo comportamento das diferentes naturezas de transição na classe `MecanismoRealização Transições`.

Um sub-produto da elaboração deste modelo da semântica dos autômatos adaptativos foi a introdução dos conceitos *transição habilitada* e *ondas de clones*. Com estes conceitos a evolução de estados no espaço de estados pode ser apresentada e discutida sob um ângulo alternativo em relação ao trabalho original [5].

As ferramentas utilizadas foram de grande valia na concepção do modelo elaborado neste trabalho. Destaque especial para a linguagem Ruby como ambiente de exploração executável e das ferramentas UMLet e SDedit, que apoiaram, de forma ágil, a criação dos diagramas UML do modelo. A ferramenta Graphviz teve papel fundamental na visualização do autômato do caso estudado, gerando, de forma automática, os grafos em instantes pré-determinados da evolução do espaço de estados.

No estudo de caso, foi possível observar o comportamento do modelo da semântica operacional de um AA em uma implementação utilizando a linguagem de programação Ruby. Os diversos diagramas de estados ilustraram os clones e as transições habilitadas ao longo do processamento de uma cadeia de entrada. Os resultados produzidos pela realização experimento envolvendo a implementação em Ruby coincidiram com aqueles relatados no exemplo discutido em [5].

Agradecimentos

O autor agradece ao Dr. João J. Neto, amigo de longa data, pela paciência em esclarecer os elementos teóricos

relacionados aos autômatos adaptativos. Sua clareza na expressão dos conceitos teve contribuição decisiva na elaboração deste trabalho. Também agradece a influente ajuda do colega e amigo Marcus Vinícius M. Ramos nas longas discussões sobre a modelagem dos autômatos adaptativos com objetos e respectiva implementação em Ruby. Finalmente, o autor agradece aos revisores pelas sugestões e recomendações que muito contribuiram para o aprimoramento da versão final deste artigo.

REFERÊNCIAS

[1] Neto, J. J.; *Introdução à Compilação*. Editora LTC, Rio de Janeiro, 1987.

[2] Conway, M. E., *Design of a separable transition-diagram compiler*. Commun. ACM 6, 7 (Jul. 1963), pp. 396-408. DOI= http://doi.acm.org/10.1145/366663.366704.

[3] Lomet, D. B. 1973. *A Formalization of Transition Diagram Systems*. J. ACM 20, 2 (Apr. 1973), pp. 235-257. DOI= http://doi.acm.org/10.1145/321752.321756.

[4] Neto, J. J.; *Adaptive Automata for Context-Sensitive Languages*. SIGPLAN NOTICES, Vol. 29, n. 9, September, 1994, pp. 115-124.

[5] Neto, J. J.; *Adaptive rule-driven devices – general formulation and case study.* In: International Conference on Implementation and Application of Automata - CIAA 2001, 6th, Pretoria, South Africa, July 2001. Lectures Notes on Computer Science 2494. Berlin: Springer-Verlag, p.234-250.

[6] Wirfs-Brock R.; McKean, A.; *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2003. ISBN 0201379430.

[7] Larman, C.; *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, 3a ed., 2004. ISBN 0131489062.

[8] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. M.; *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994. ISBN 0201633612.

[9] Thomas, D.; Fowler, C.; Hunt, A.; *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004. ISBN 0974514055.

[10] *Ruby, a programmer's best friend – Documentation*. Disponível na Internet. URL: http://www.ruby-lang.org/en/documentation/, 2007.

[11] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003. ISBN 0-321-19368-7.

[12] *UML Resource Page*. Disponível na Internet. URL: http://www.uml.org/, 2007.

[13] *Free UML Tool for Fast UML Diagrams*. Disponível na Internet. URL: http://www.umlet.com/, 2007.

[14] *Quick Sequence Diagram Editor*. Disponível na Internet. URL: http://sdedit.sourceforge.net, 2007.

[15] *Graphviz - Graph Visualization Software*. Disponível na Internet. URL: http://sdedit.sourceforge.net, 2007.

[16] *Inkscape Draw Freely*. Disponível na Internet. URL: http://www.inkscape.org/index.php?lang=en, 2007.

[17] J. J. Neto, *WTA 2007 - Adaptive Technology Tutorial*, IEEE LATIN AMERICA TRANSACTIONS, vol. 5, No. 7, pp. 495-495, Nov. 2007.

Ítalo Santiago Vega recebeu os títulos de Mestre e Doutor em Engenharia de Software na Escola Politécnica da Universidade de São Paulo, Brasil, em 1993 e 1998, respectivamente. Realiza pesquisas na área de desenvolvimento de sistemas de software, com ênfase em modelagem por objetos, processos ágeis e métodos formais. Atua como Assessor de Políticas Tecnológicas e como Professor Associado do Departamento de Ciência da Computação na Pontifícia Universidade Católica de São Paulo, das Faculdades Integradas Rio Branco e como professor-colaborador nos cursos de pós-graduação do Senac/SP.