Visualizador gráfico para Redes de Petri Adaptativa

W. C. M. Gomes e A. R. Camolesi

***Resumo* — Este trabalho tem por objetivo apresentar os conceitos teóricos dos dispositivos Redes de Petri e Redes de Petri Adaptativa. Neste contexto é apresentado um visualizador gráfico para facilitar a visualização dos elementos de especificação das aplicações que usam tais dispositivos como linguagem de modelagem.**

***Palavras clave* — Tecnologia Adaptativa, Rede de Petri, Rede de Petri Adaptativa, Dispositivos Adaptativos**

I. INTRODUÇÃO

Camolesi nos diz em [1] que o projeto de aplicações adaptativas com suporte a multiformalismos exige, em geral, soluções relativamente complexas, nas quais é imprescindível o uso de um método bem definido, que dê suporte à especificação de aplicações e, posteriormente, à análise e à implementação das especificações produzidas. Fundamentado nos conceitos apresentados em [2] é proposto um método para o projeto de aplicações adaptativas constituído de três fases: especificação, análise e implementação.

Na fase de especificação, um projetista utiliza uma linguagem de especificação e gera modelos da aplicação desejada. Com base nos modelos obtidos, é realizada a fase de análise, que tem por objetivo verificar inconsistências na aplicação projetada. Nessa fase, o projetista submete estímulos ao modelo e obtém respostas que lhe permitem validar o modelo definido. Também nessa fase podem ser realizadas simulações do modelo, cujas respostas geradas auxiliarão o projetista na validação e análise da aplicação projetada. Ao serem detectadas inconsistências ou falhas, a especificação deve ser editada e, na seqüência, novamente analisada. Finalmente, depois de validada a aplicação, o modelo gerado poderá ser traduzido para uma linguagem de representação física.

Para auxiliar o projeto de aplicações que use tecnologia adaptativa, faz-se necessária a utilização de um ambiente que integra um conjunto de ferramentas que dê suporte aos projetistas na realização de seu trabalho. A Figura 1 apresentada em [1] ilustra a arquitetura geral de tal ambiente, que é organizado em 3 (três) grupos de ferramentas: Ferramentas de Edição, Ferramentas de Análise e Ferramenta de Implementação.

Valendo-se das Ferramentas de Edição, um projetista de aplicações utilizando-se de um dispositivo adaptativo específico poderá realizar a especificação de sua aplicação com o auxilio de um editor de texto qualquer ou de um editor gráfico.

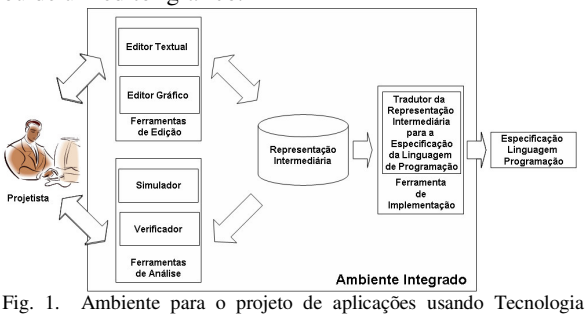


Fig. 1. Ambiente para o projeto de aplicações usando Tecnologia Adaptativa.

Para possibilitar o intercâmbio das especificações produzidas, os editores devem gerar objetos no formato da Representação Intermediária definida para suportar os elementos conceituais de dispositivos adaptativos. Caso a especificação seja produzida em um editor textual, esta deverá ser compilada para transformar a codificação realizada no formato definido para a representação intermediária.

Depois de realizada a especificação, o projetista de aplicações poderá utilizar as Ferramentas de Análise. Tais ferramentas utilizam a codificação da especificação (no formato da representação intermediária) como base e permitem a realização de uma análise do comportamento da aplicação adaptativa em desenvolvimento. Por fim, depois de especificada e analisada a representação de uma aplicação, o projetista pode se utilizar das Ferramentas de Produção de Representações Físicas e gerar uma representação de uma aplicação em um determinado padrão de linguagem de representação física e obter a aplicação desejada.

Motivado pela definição de dispositivos e a construção de ferramentas é apresentado neste trabalho o formalismo Redes de Petri Adaptativa [3] e a estrutura de uma ferramenta para visualização dos objetos gráficos para tal dispositivo. O visualizador construído utilizará como base a biblioteca OpenJgraph [4], construída com base na linguagem Java e que permite a manipulação de grafos.

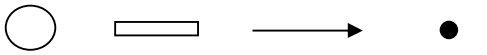
Este trabalho está organizado da seguinte forma: na seção II serão descritos os conceitos de Redes de Petri, na seqüência, é definido, na seção III, as Redes de Petri Adaptativas. Na seção IV é realizada a descrição da implementação do visualizador gráfico para Redes de Petri Adaptativa e, por fim, na seção V serão apresentadas as conclusões e sugeridos alguns trabalhos futuros.

W. C. M. Gomes (wilsoncmartins@gmail.com)
A. R. Camolesi (camolesi@femanet.com.br)

II. REDES DE PETRI

Redes de Petri é uma representação matemática e gráfica utilizada para modelagem de sistemas comunicantes e concorrentes [5].

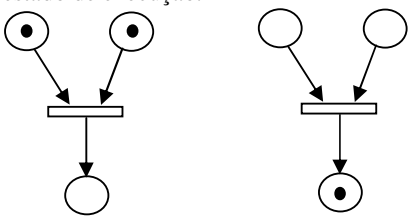
Uma Rede de Petri é uma quádrupla (*P, T, I, O*), onde *P* é o conjunto do lugar ou estado que representa a situação corrente da rede; T é conjunto de transições ou conexões, responsáveis por receber os sinais enviados pelos estados e os direcionar; *I* representa o conjunto de regras de entrada para ocorrência das transições da rede e o conjunto *O* representa as saídas, respostas de uma transição ocorrida na rede. Tais elementos podem ser representados graficamente conforme ilustrado na Figura 2.



a) Local b) Transição c) Arco d) Ficha

Fig. 2 Elementos gráficos Redes de Petri

Nas Redes de Petri uma ficha é responsável por marcar os estados em execução. Na Figura 3a é ilustrada uma Rede de Petri, na qual os estados *p1* e *p2* estão em execução aguardando o disparo da transição *t1*. Na Figura 3b observa-se a rede após o disparo da transição *t1*, onde *p1* e *p2* passaram para um estado de espera e *p3* passou para um estado de execução.



a) antes disparo b) após disparo

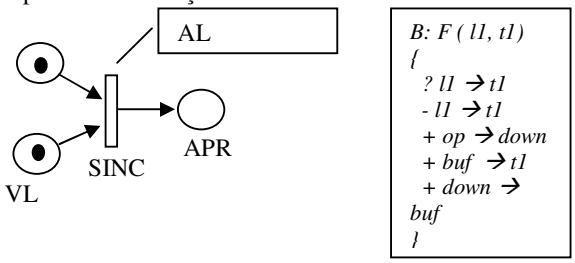
Fig. 3 – Exemplo Rede de Petri.

III. REDES DE PETRI ADAPTATIVA

Um dispositivo adaptativo é aquele que é capaz de se auto modificar em tempo de execução e sem influência externa [6, 7]. Para a transformação de uma Rede de Petri não adaptativa em uma Rede de Petri adaptativa, é acrescentada a esse formalismo uma camada adaptativa como apresentado na Figura 4, na qual após a execução das funções e ações adaptativas anteriores acontecem mudanças no comportamento do dispositivo, e após a execução das funções e ações adaptativas posteriores acontecem novas mudanças de comportamento.

Fig 4. Mecanismo Adaptativo de Rede de Petri.

Ao envolver uma Rede de Petri com o mecanismo adaptativo temos as funções e ações adaptativas associadas as transições. A seguir, é apresentado um exemplo de uma Rede de Petri Adaptativa que descreve a modelagem de uma aplicação de apresentação de áudio e vídeo sincronizados. Tal modelagem possui uma função adaptativa que permite que novos áudios sejam adicionados conforme as características de entrada da aplicação. Na Figura 5a os estados seriam representados por *AL* (Áudio Local), *VL* (Vídeo Local) e *APR* (Apresentação) enquanto que *SINC* (Sincronização) representa a transição do modelo.



a) Modelo aplicação b)Função Adaptativa *B*.

Fig. 5 Exemplo Rede de Petri Adaptativa Áudio e Vídeo Sincronizados

No exemplo apresentado, na ocorrência de uma ficha em *AL* e outra em *VL* ocorre a transição *SINC* que permite a apresentação das mídias de forma sincronizada na ocorrência de uma ficha no local *APR*. Na Figura 5b pode ser observado que a transição *SINC* possui a função adaptativa *B(..)* associada a mesma.

Para descrever uma função adaptativa devem ser elaboradas suas ações adaptativas que podem ser de três tipos: Consulta (?), Eliminação (-) e Adição (+). Uma ação de consulta verifica a existência de uma determinada regra que compõe o conjunto de regras de um modelo. Já a ação de eliminação é responsável por eliminar do modelo certa regra, enquanto, uma ação de adição serve para que novas regras sejam adicionadas ao modelo. Com base nos conceitos apresentados estrutura-se a função adaptativa *B* que ao ser executada recebe por exemplo, nos argumentos *l1* o valor *"AL"* e para o argumento *t1* o valor *"SINC"*. Inicialmente, é executada a ação adaptativa de consulta que verifica a existência de uma regra *AL* → *SINC*. Na seqüencia, caso a regra exista, ocorre a eliminação da mesma por meio de uma ação adaptativa de remoção e, por fim, executam-se as ações adaptativas de adição que adiciona novas regras ao modelo. Tais ações permitem que novas funcionalidades para obtenção de novos arquivos de áudio sejam adicionados e que estes áudios sejam disponibilizados ao sistema, para posterior apresentação. Desta forma, após a execução da função adaptativa *B* o novo comportamento da aplicação Áudio eVídeo, fica como ilustrado na Figura 6.

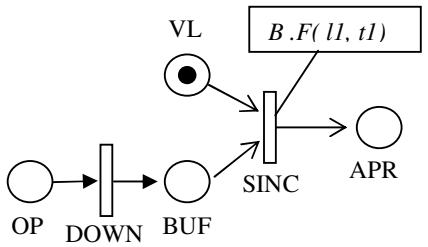


Fig 6. Aplicação Áudio e Vídeo após execução Função Adaptativa B

IV. IMLEMENTAÇÃO DO VISUALIZADOR GRÁFICO PARA REDE DE PETRI ADAPTATIVA

Para realizar a implementação do visualizador gráfico para Redes de Petri Adaptativa foi utilizada a biblioteca

OpenJgraph [4] que foi codificada usando a linguagem Java.

O primeiro passo para a realização do visualizador foi editar o modelo do vértice para cada um dos tipos de vértices necessários para a representação gráfica de Redes de Petri - o padrão do OpenJgraph são todos os vértices de forma gráfica quadrada -. A representação de cada forma de cada vértice da modelagem será de acordo com o valor que fica armazenado no modelo lógico representado pelo banco de dados da ferramenta. Para isso é necessário guardar o código do vértice e passá-lo por parâmetro no momento da instanciação do objeto *Vertex* e preparar o OpenJgraph para receber e retornar tal parâmetro. Essa etapa foi dividida nos seguintes passos descritos a seguir.

Inicialmente, na classe *VertexImpl.java* armazenada em *salvo/jesus/graph/VertexImpl.java* foi adicionado um novo parâmetro, do tipo inteiro, para armazenar o código de cada componente que é passado para o método. Na chamada do método *setLabel* passa-se a variável *cod_comp* por parâmetro. O *setLabel* está implementado na classe *LabeledGraphComponent Impl.java* e será descrito posteriormente. O código gerado é apresentado na Fgura 7.

```
public void setLabel(String label,int cod_comp)
   {
      //if (m_label != null) {
      //    throw new IllegalStateException("label cannot be
reassigned");
      //}
      m_cod_comp=cod_comp;
      m_label = label;
   }
```

Fig 7. Método *setLabel*.

Depois de armazenadas as informações referentes ao código do componente no OpenJgraph, o próximo passo é o desenvolvimento de métodos para retornar estes valores. E, para tal, criou-se na classe *VertexImpl.java* um outro método que foi chamado *getCodigo* e descrito na Figura 8.

```
public int getCodigo(){
    return this.labelDelegator.getCodigo();
  }
```

Fig 8. Método *VertexImpl.Java.*

Na classe *VertexImpl.Java* foi declarado o atributo *getCodigo* para ser, posteriormente, utilizado na classe *LabeledGraphComponentImpl.java.*

Na classe *LabeledGraphComponentImpl.java* que se encontra no pacote *salve/jesus/graph/LabeledGraph ComponentImpl.java*, foi criado um novo atributo para armazenar o código do componente (*m_cod_comp),* do tipo inteiro privado. No método *public void setLabel*, acrescentou-se um parâmetro do tipo inteiro *cod_comp* para receber a passagem de parâmetro do *m_cod_comp*, conforme descrito na Figura 9.

```
public void setLabel(String label,int cod_comp)
   {
      m_cod_comp=cod_comp;
      m_label = label;
   }
```

Fig 9. Método setLabel.

Por fim realizou-se o método *getCodigo()* conforme representado na Figura 10. Tal método permite o retorno do código de um componente.

```
public int getCodigo()
   {
      return m_cod_comp;
   }
```

Fig 10. Implementação getCodigo().

Na classe *AbstractVisualGraphComponent.java* pertencente ao pacote *salve/jesus/graph/visual/Abstract VisualGraphComponent.java*, definiu-se o método *getCodigo()* responsável por retornar o código do componente, como ilustrado na Figura 11.

```
public int getCodigo(){
    return this.component.getCodigo();
    }
```

Fig 11. Implementação método getCódigo().

Na classe *VisualVertex.java* localizada no pacote *salve/jesus/graph/visual/VisualVertex.java*, definiu-se um novo atributo para armazenar o tipo de forma gráfica do vértice. Tal atributo é do tipo de dados inteiro *(public int tipo;)*. Nesse caso, conforme o código do tipo de componente eu foi armazenado no banco de dados, tem que criar uma conexão com o banco de dados e fazer uma busca comparando o código do banco com o campo *getCodigo* que armazena o valor do código do componente, no OpenjGraph. Encontrando o componente desejado, armazena-se o seu código de tipo na variável *tipo* e as coordenadas que esse vértice deve ser apresentado na interface do visualizador (tais coordenadas também estão armazenadas no banco de dados). Ainda nessa classe foi acrescentado um comando condicional para verificar qual o tipo do vértice. Caso o tipo seja igual a quatro *(tipo==4)*, sua cor de fundo será branca (desta mesma forma pode-se alterar a cor de fundo dos vértices). Com essas alterações o método ficou conforme apresentado na Figura 12.

```
public VisualVertex( Vertex vertex, VisualGraph vGraph ){
   this.component = vertex;
   this.painter =
vGraph.getVisualVertexPainterFactory().getPainter( this );
   this.visualGraph = vGraph;
   this.setFont( new Font( "Lucida Sans", Font.PLAIN, 10 ));
   Componentes.listar("graph_x");
   Componentes.primeiro();
   do {

if(Componentes.getCampoInt("codigo_cpt")==vertex.getCodigo()){
        x=Componentes.getCampoInt("graph_x");
        y=Componentes.getCampoInt("graph_y");
        tipo=Componentes.getCampoInt("cod_tipo_cpt");
        Componentes.ultimo();
        }
     }while (Componentes.proximo());
   this.initLocation();
   // Force adjustment of width and height.
   this.rescale();
   this.setOutlinecolor( Color.black );
   if(tipo==4)this.setFillcolor( new Color( 255, 255, 255 ));
   else this.setFillcolor( new Color( 0, 225, 255 ));
  }
```

Fig 12. Método Visual Vertex

Finalizando, foi acrescentado no método *private void initLocation*" um comando de decisão *(switch-case)* para verificar o valor da variável tipo, para realizar a escolha da forma de representação gráfica do vértice, nesse exemplo o tipo do vértice pode variar de 0 à 17 que permite representar diversas formas geométricas. Nas

últimas linhas deste método são passadas as coordenadas de visualização do vértice na interface do visualizador, capturadas nos métodos definidos acima e armazenadas nos atributos *x* e *y*. O referido método é apresentado na Figura 13.

```
private void initLocation(){
   StringTokenizer   sttokenizer;
   int               height = 0, width, maxwidth = 0;
   int               lineheight;
   switch(tipo){
      case 0: drawpath = new GeneralPath( (Shape) new
Rectangle.Double( 2, 2, 2, 2 ));break;
      case 1: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      case 2: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      case 3:  drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      case 4: drawpath = new GeneralPath( (Shape) new
Rectangle.Double(10,10,10,10)); break;
      case 5: drawpath = new GeneralPath( (Shape) new
Rectangle.Double( 5, 5, 10, 10 )); break;
      case 6: drawpath = new GeneralPath( (Shape) new
Rectangle.Double( 5, 5, 10, 10 )); break;
      case 7: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Arc2D.Double(16.0, 10.0,
30.0,30.0,90.0,180.0,Arc2D.CHORD));break;
      case 8: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      case 16: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      case 17: drawpath = new GeneralPath( (Shape) new
java.awt.geom.Ellipse2D.Double( 5, 5, 10, 10 )); break;
      default: drawpath = new GeneralPath( (Shape) new
Rectangle.Double( 2, 2, 2, 2 ));break;
    }
   if(x==0){
    this.setLocation( rand.nextInt( 500 ), rand.nextInt( 400 ) );
   }
   else{this.setLocation( x, y ); }
  }
```

Fig 13. Método initiLocation().

Depois de realizadas as alterações na biblioteca OpenJGraph é possível visualizar os vértices em sua forma gráfica. Tal fato comprova-se na Figura 14.

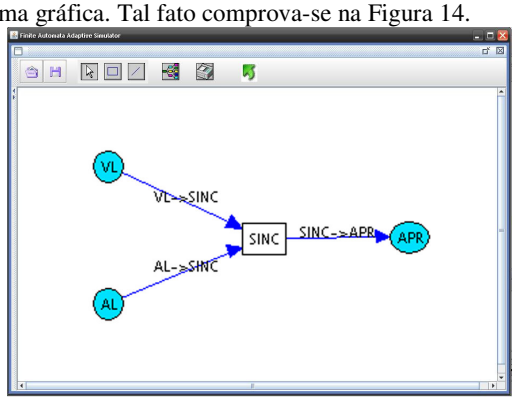


Fig 14. Interface do Visualizador gráfico para Redes de Petri Adaptativa.

V – CONCLUSÕES

Neste trabalho foram apresentados os conceitos de uma Rede de Petri Adaptativa e da arquitetura e construção de um Visualizador Gráfico para tal dispositivo.

Pode-se observar que a biblioteca OpenJGraph utilizada para a representação gráficas dos elementos conceituais do dispositivo estudado foi adequada e que a mesma permite a adição de novos elementos e estruturas para suportar outros dispositivos.

Este trabalho também permitiu comprovar que o modelo lógico definido para representação de dispositivos adaptativos representa de forma adequada tantos os elementos conceituais e de representação gráfica dos dispositivos nele definidos.

O visualizador implementado e integrado com a ferramenta de especificação de aplicações usando o modelo lógico (AdapSim) facilita o trabalho de simulação e o entendimento das falhas conceituais de especificação de uma aplicação.

Como trabalho futuro é sugerido a aplicação da arquitetura do visualizador para a representação de outros dispositivos adaptativos, bem como a proposta e o desenvolvimento de uma ferramenta gráfica para auxiliar os projetistas na realização do projeto de suas aplicações.

REFERÊNCIAS

[1] A.R. Camolesi, J. J. Neto "Modelagem Adaptativa de Aplicações Complexas," in 2004 *Anais XXX Conferencia Latinoamericana de Informática* Sep.

[2] W. L. Souza et al., "Design de Aplicações Multimídia Distribuídas (DAMD)," in *Anais do II Seminário Franco-Brasileiro em Sistemas Distribuídos Conf*, Fortaleza – CE, Novembro 1997

[3] A. R. Camolesi, J.J. Neto, "Representação Intermediária para Dispositivos Adaptativos Dirigidos por Regras," in *Proc 2004 3rd International Information and Telecommunication Technologies Symposium,* UFSCar, São Carlos, Brasil.

[4] Java Graph and Graph Drawing Project – OpenjGraph, avaible in http://openjgraph.sourceforge.net/

[5] C. A. Petri, "Kommunikation mit Automaten," Ph. D thesis, Institut für Instrumentelle Mathematik,Schriften, University of Bonn, Bonn, German, 1962.

[6] J. J. Neto, "Contribuições à metodologia de construção de compiladores," Tese de Livre Docência, Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 1993.

[7] J. J. Neto "Adaptive Rule-Driven Devices - General Formulation and Case Study," in *Proc. 2001 Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conf.,* Springer-Verlag, Vol.2494, pp. 234-250.

[8] H. Pistori, "Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações," Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 2003.

Wilson C. M. Gomes nasceu em Assis, São Paulo. Faz curso de graduação de Bacharelado em Ciência da Computação na Fundação Educacional do Município de Assis e participa do Programa de Iniciação Científica na mesma instituição.

Almir R. Camolesi nasceu em Cândido Mota, São Paulo, Brasil, em janeiro de 1972. Graduou-se em Tecnologia em Processamento de Dados na Fundação Educacional do Município de Assis, em 1992. Recebeu o título de mestre em Ciência da Computação pela Universidade Federal de São Carlos, em 2000, e doutorou-se em Engenharia de Computação e Sistemas Digitais pela Universidade de São Paulo, em 2007. Atualmente, é professor titular da Fundação Educacional do Município de Assis. Tem experiência na área de Ciência da Computação, com ênfase em Tecnologias Adaptativas, atuando principalmente nos seguintes temas: modelagem abstrata, metaferramentas, dispositivos adaptativos, ensino a distância, modelagem e desenvolvimento de aplicações distribuídas.