ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

SIMULAÇÃO DE REDES DE PETRI EM AMBIENTE JAVA

Trabalho de Conclusão de Curso

Engenharia da Computação

César Augusto Lins de Oliveira
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, 04 de julho de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

SIMULAÇÃO DE REDES DE PETRI EM AMBIENTE JAVA

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

César Augusto Lins de Oliveira
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, 04 de julho de 2006

UNIVERSIDADE
DE PERNAMBUCO

César Augusto Lins de Oliveira

SIMULAÇÃO DE REDES DE PETRI EM AMBIENTE JAVA

Resumo

Apesar de bem estabelecidos e de serem utilizados com sucesso por diversas empresas há muitos anos, os métodos formais ainda enfrentam a resistência dos profissionais recém-formados que, na sua maioria, não são capazes de descobrir uma aplicação prática para os conhecimentos adquiridos durante sua formação e os evitam veementemente, imaginando que são complexos demais para serem utilizados no dia-a-dia fora da academia. As redes de Petri constituem uma técnica formal bastante poderosa, que é capaz de modelar sistemas assíncronos e concorrentes e fornecer diversas informações qualitativas e quantitativas sobre este, através de métodos analíticos e/ou de simulações. Apesar de sua grande aplicabilidade, as redes de Petri não são utilizadas tão bem quanto poderiam. Isto se deve, em parte, ao fato de as ferramentas serem em sua maioria difíceis de serem utilizadas e de apresentarem interfaces obsoletas. Isto dificulta que novos usuários venham a aplicá-las em seus projetos. A necessidade de novas ferramentas é uma realidade. O trabalho apresentado nesta monografia consistiu na implementação de um conjunto de simuladores de redes de Petri na linguagem *Java*. O objetivo desta implementação é oferecer a outros desenvolvedores a possibilidade de incluírem a capacidade de simular redes de Petri em suas próprias ferramentas. Para facilitar a sua utilização, os simuladores foram agrupados em uma biblioteca, na qual o desenvolvedor poderá encontrar também facilidades para realizar a comunicação entre sua aplicação e os simuladores. A biblioteca recebeu o nome de ***jPetriSim***, e foi desenvolvida a partir de uma pesquisa abrangente da teoria de redes de Petri, que será apresentada aqui.

Abstract

Although well established and apart from being used successfuly by many enterprises for many years, the formal methods still face the resistence of the newly graduated professionals which, in their majority, are not capable to find a practical application for the knowledge they acquired in their graduation and avoids them vehemently – thinking that they are too complex to be used in the day-by-day outside academy. Petri nets constitute a very powerful formal technique, which is capable to model concurrent and asynchronous systems and to supply many qualitative and quantitative information about them, through analytical methods and/or simulation. Apart from its very applicability, Petri nets are not being so well used as they could be. This is, in part, due to the fact that the available tools are, in majority, difficult to use and have obsolete user interfaces. This makes difficult to new users to start applying them on their projects. The demand for new tools is a reality. The work presented in this monograph consisted of the implementation of a set of Petri net simulators in the Java language. The objective of this implementation is to offer to other developers the possibility to include the capacity to simulate Petri nets in their own tools. To make it simple to use, the simulators set was packaged in a Java library, were can also be found facilities to help to perform the communication between application and the simulators. The library was called ***jPetriSim***, and was developed based in a research on the theory of Petri nets, which is presented in this monograph.

Sumário

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Tabela de Símbolos e Siglas

(Dispostos em ordem de aparição no texto)
INA – Integrated Net Analyser
PNML – Petri Net Markup Language
XML – eXtensible Markup Language
URI – Uniform Resource Identifier
DTD – Document Type Definition
PNTD – Petri Net Type Definition
UML – Unified Modelling Language
O(1) – Complexidade constante de algoritmos na notação Big-O
VTL – Velocity Template Language

Agradecimentos

No segundo semestre do ano de 2001, eu entrei nesta faculdade numa turma de quarenta alunos. Era a quarta turma de um curso que estava em seu início e sobre o qual nós tínhamos muitas dúvidas mas também bastante esperança. Observamos ao longo do tempo o crescimento do curso, que hoje se tornou um curso reconhecido e bastante valorizado, graças à dedicação incansável dos nossos professores. Passados cinco anos, agora no final do curso, posso dizer que valeu a pena tudo o que passamos para chegar até aqui.

Agradeço a meus pais, Sonia Maria Lins de Oliveira e José Cláudio de Oliveira, por toda a dedicação, amor e carinho. Graças e eles eu cheguei onde estou. Agradeço também a meu irmão José Cláudio de Oliveira Júnior por todo o companheirismo e amizade.

Agradeço aos meus amigos, que passaram por tudo junto comigo: Milena Rodrigues, Nívia Quental e Marcelo Nunes, que estiveram presentes desde o início e, em especial, a Cleyton Rodrigues, Laura Moraes e Petrônio Braga, sem os quais eu não poderia ter chegado aqui. Estes formaram a minha família nesta faculdade.

Agradeço ao meu orientador, Ricardo Massa Ferreira Lima, que sempre acreditou na minha capacidade e sempre me apoiou durante minha carreira acadêmica, ajudando no meu crescimento profissional e pessoal.

E agradeço principalmente a Deus, que colocou todas estas pessoas em meu caminho.

Obrigado a todos!

Capítulo 1

Introdução

O desafio de fazer com que a utilização de métodos formais seja mais difundida fora do ambiente acadêmico é enfrentado por todos os docentes que não querem que seus alunos deixem a faculdade com a impressão de que tais formalismos não passam de teorias acadêmicas que existem para tornar o curso “mais complicado”.

O termo “métodos formais” identifica um conjunto amplo de técnicas matemáticas utilizadas na Engenharia de Software para especificar sistemas com a maior exatidão possível, de forma que a sua implementação possa ser validada posteriormente. Eles também auxiliam o processo de implementação, evidenciando e ajudando a solucionar problemas que seriam difíceis de tratar sem o seu apoio [Hall90]. Os métodos formais são aplicáveis não apenas à especificação de sistemas de *software* mas também de sistemas físicos diversos [Bowen95].

Apesar de bem estabelecidos e de serem utilizados com sucesso por diversas empresas há muitos anos [Hall90][Bowen95], os métodos formais ainda enfrentam a resistência dos profissionais recém-formados que, na sua maioria, não são capazes de descobrir uma aplicação prática para os conhecimentos adquiridos durante sua formação e os evitam veementemente, imaginando que são complexos demais para serem utilizados no dia-a-dia fora da academia. Este temor faz com que os métodos formais sejam vistos como técnicas obscuras, utilizadas apenas por grandes indústrias que teriam a possibilidade de investir em pesquisa científica.

Este cenário, montado na mente de alguns profissionais, não corresponde à realidade. Existem técnicas formais bastante simples, que poderiam ser utilizadas para facilitar a realização de diversas tarefas na engenharia de software [Hall90]. Mas o temor ou a falta de conhecimento simplesmente não permitem que elas sejam utilizadas como deveriam [Bowen95].

Redes de Petri constituem uma dessas técnicas. Foram apresentadas em 1962 por Carl Adam Petri, em sua tese de doutorado, intitulada *Kommunikation mit Automaten* [Petri62], na Faculdade de Matemática e Física da Universidade de Darmstad, no país que naquela época ainda era conhecido como Alemanha Ocidental [Maciel96]. Trata-se de um modelo matemático que tem como característica mais destacável a possibilidade de descrever sistemas graficamente, demonstrando as interações entre as diferentes partes que os compõem, além de possuir mecanismos de análise poderosos, capazes de verificar propriedades e a corretude do sistema descrito. Além disso, a representação do sistema permite que o modelo seja simulado e o seu comportamento seja observado dinamicamente.

O modelo proposto por Petri chamou a atenção de vários pesquisadores e, ao longo dos anos, foi se tornando cada vez mais difundido. Diversas propostas de extensões ao modelo

original foram feitas, causando o surgimento de novos tipos de redes de Petri como as redes de Petri temporizadas [Zuberek91], as estocásticas [Ajmone96], entre outras [Maciel96].

As redes de Petri possuem grande poder de expressão e podem ser utilizadas para modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos. Têm sido aplicadas em variadas áreas, como na modelagem de sistemas de hardware/software, protocolos de comunicação e análise de desempenho [Murata89][Maciel96].

Na prática, o uso de redes de Petri é dependente de ferramentas de software, que são necessárias para auxiliar na modelagem do sistema e que são verdadeiramente essenciais para a execução das diversas análises que podem ser realizadas sobre o modelo. Grande quantidade de ferramentas foram desenvolvidas para este propósito (a página do *TGI group* [TGI06] contém uma vasta lista delas) e atualmente são muito usadas as ferramentas INA (*Integrated Net Analyser*) [Roch03], *TimeNet* [Zimm01] e *CPN Tools* (atualização da conhecida *Design/CPN*) [Beaudouin00]. Estas ferramentas costumam ser de difícil utilização para iniciantes, seja porque possuem interfaces ultrapassadas (a ferramenta INA executa apenas em console, por exemplo) ou porque possuem problemas de desempenho (como é o caso da *CPN Tools*, que possui problemas que são bem conhecidos entre pesquisadores da Universidade de Pernambuco e da Universidade Federal de Pernambuco).

Além disso, apesar da grande quantidade de ferramentas ser uma vantagem, ela também traz alguns problemas para os pesquisadores. O que ocorre é que em muitos casos uma única ferramenta não supre todas as necessidades da pesquisa, levando à necessidade da utilização simultânea de várias ferramentas, cada qual oferecendo o seu próprio conjunto de recursos. Como, em geral, estas ferramentas não utilizam o mesmo formato de representação para as redes de Petri, acaba sendo necessário recorrer a outras ferramentas que realizem a conversão entre formatos – talvez havendo a necessidade do próprio pesquisador implementá-las.

Recentemente, a comunidade de pesquisadores tem realizado um esforço na criação de um formato de representação padrão intercambiável entre ferramentas. Uma das propostas que se tornou bastante popular é a da linguagem PNML (*Petri Net Markup Language*) [Weber02]. Trata-se de um padrão baseado em XML (*eXtensible Markup Language*) [WG04] que tem como objetivo ser capaz de representar todos os tipos de redes de Petri existentes ou que venham a existir. Algumas ferramentas já foram desenvolvidas para suportar este formato, mas a padronização ainda está em seu início e não há, no momento, um formato universalmente aceito [Weber02]. Todos estes fatores são empecilhos que fazem com que as redes de Petri sejam menos utilizadas do que poderiam.

Para facilitar a difusão desta técnica, algumas pesquisas realizadas pelo Departamento de Sistemas Computacionais da Universidade de Pernambuco têm desenvolvido ferramentas que têm como objetivo facilitar o aprendizado e a utilização das redes de Petri de maneira geral, tais como o *EZPetri* [EZPetri03] e o *Petrilogic* [Petrilogic06], sendo este último uma iniciativa premiada pela IBM em seu *Faculty Awards 2005* [EIG05], tendo sido o único projeto brasileiro a receber tal reconhecimento naquele ano. Tal premiação demonstra que a criação de novas ferramentas para facilitar a utilização de métodos formais é bastante valiosa no cenário tecnológico mundial.

O presente trabalho é uma contribuição para esta iniciativa e se concentra em particular na questão da simulação de redes de Petri. O objetivo é fornecer uma biblioteca de simulação para o ambiente *Java* [Gosling05] ou, em outras palavras, fornecer uma implementação de simulação de redes de Petri de uma forma modular, para que possa ser utilizada no desenvolvimento de novas ferramentas.

Uma das possíveis aplicações desta biblioteca é a visualização da simulação em alto-nível. É sempre útil ao pesquisador observar certas propriedades do sistema observando uma representação em alto nível deste sistema, onde o modelo correspondente em redes de Petri não é

exibido. Na maioria dos casos, o pesquisador irá implementar esta interface, pois ela será dependente de características particulares de seu sistema e de seus modelos. Através da biblioteca, este pesquisador poderá facilmente adicionar também a capacidade de simulação à sua implementação, interpretando os eventos ocorridos na rede para o seu modelo em alto-nível.

A questão da simulação é importante pois, em muitos casos, a simulação é o único método capaz de extrair informações sobre um modelo. Isto ocorre, por exemplo, quando o espaço de estados de uma rede é muito grande para ser estudado analiticamente.

A biblioteca desenvolvida recebeu o nome de ***jPetriSim***. Esta monografia apresentará a base científica sobre a qual foram implementados os simuladores da biblioteca e descreverá o seu processo de implementação.

A estrutura da monografia foi dividida como se segue:

No Capítulo 2 são apresentados os conceitos fundamentais que definem as redes de Petri. No Capítulo 3, uma classe de redes de Petri que associa informações temporais é apresentada. São descritos os seus conceitos básicos e as mudanças que a consideração de tempos causa no comportamento dinâmico da rede. No Capítulo 4 é apresentada a rede de Petri colorida. Uma representação de alto nível bastante poderosa das redes de Petri. Serão também analisados alguns problemas que envolvem a simulação deste tipo de rede. No Capítulo 5 é apresentado o formato de descrição para redes de Petri baseado em XML, chamado de *Petri Net Markup Language* (PNML) e as extensões que foram criadas para representar as redes na biblioteca. No Capítulo 6 é descrita a arquitetura da biblioteca jPetriSim e os requisitos para integração com aplicações. No Capítulo 7 é descrita a implementação dos simuladores em Java. O Capítulo 8 apresenta conclusões e sugere alguns trabalhos futuros.

Capítulo 2

Introdução às Redes de Petri

Para uma correta compreensão deste trabalho, uma introdução conceitual se faz necessária. Este capítulo não tem como propósito cobrir toda a teoria que envolve as redes de Petri, pois esta é extensa e inclui grande quantidade de informação que não está no escopo desta monografia.

A característica mais destacável das redes de Petri é a sua facilidade em representar paralelismo e concorrência, e também as relações de causalidade entre as partes do sistema. Isto se deve ao fato de ser uma técnica capaz de modelar o estado do sistema num nível maior de granularidade do que o que é fornecido pela teoria convencional de autômatos. O estado de uma rede de Petri é capaz de expressar variadas condições que estejam ocorrendo simultaneamente em diversas partes do sistema e o modelo permite descrever a forma como estas condições afetam a transição entre estados. Isto permite uma decomposição do sistema em processos elementares e, ao mesmo tempo, permite que seja expressa a forma como estes processos afetam uns aos outros.

Este capítulo irá abordar o tipo de rede de Petri conhecido como rede *Place/Transition*.

2.1 Visão Geral

Os elementos que compõem uma rede de Petri são chamados de lugares, transições e arcos [Murata89]. A Figura 1 mostra a representação gráfica destes elementos. Lugares (Fig 1.a) são representados graficamente por círculos (ou elipses), transições (Fig. 1.b) por retângulos e arcos por setas que ligam um lugar a uma transição ou uma transição a um lugar (nunca um lugar a outro ou uma transição a outra) (Fig. 1.c,d).

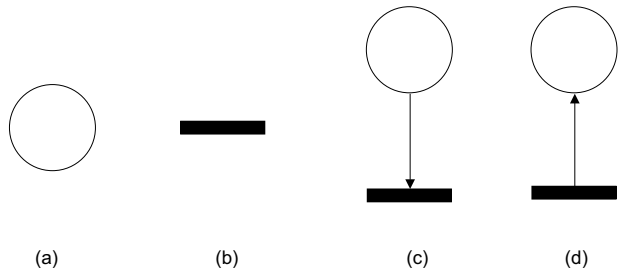


Figura 1. a) Lugar; b) Transição; c) Arco ligando um lugar a uma transição; d) Arco ligando uma transição a um lugar

Lugares podem ser marcados ou não. Um lugar marcado é representado por um ou mais pontos desenhados dentro do círculo que representa o lugar (Figura 2). Estes pontos são chamados de *tokens* (fichas). A quantidade de *tokens* que um lugar possui é o que define o seu *estado* e é chamada de *marcação* deste lugar. O conjunto dos lugares da rede de Petri forma o conjunto das variáveis de estado que representam o sistema modelado. Assim, as marcações de todos os lugares em um dado momento é uma característica importante da rede, representando o estado do sistema naquele momento. Este estado global, composto pelos estados de cada lugar, é chamado de *marcação da rede*.

Figura 2. a) Lugar com um *token* e b) com vários *tokens*

Os lugares são elementos passivos e definem o estado do sistema. As transições, por outro lado, são elementos ativos. Elas representam ações que podem ocorrer e que modificam o estado do sistema (marcação da rede). Os lugares que estão ligados a uma transição através de arcos são o que define quando uma transição pode ocorrer e como o estado é modificado após a sua ocorrência.

Um arco ligando um lugar a uma transição (conforme visto na Fig. 1.c) representa uma condição que deve ser verdadeira para que aquela transição ocorra. Chama-se a este lugar de *pré-condição* da transição [Maciel96]. Para cada arco que liga qualquer lugar a esta transição deve existir pelo menos um *token* naquele lugar. Se esta condição for satisfeita, diz-se que a transição está *habilitada* para ocorrer [Murata89]. Já um arco que liga a transição a um lugar (conforme Fig. 1.d) representa a condição que se torna verdadeira após a ocorrência da transição. Este lugar é denominado de *pós-condição* da transição [Maciel96]. Assim, quando esta transição ocorrer, deverá ser adicionado um *token* ao lugar que é sua pós-condição para cada arco que liga a transição a este lugar. Os *tokens* existentes nas pré-condições da transição e que a tornaram habilitada são removidos após sua ocorrência. Diz-se que eles foram *"consumidos"* pela transição. Todo este processo que acontece quando uma transição ocorre é chamado de *disparo* da transição.

Quando mais de um arco liga os mesmos elementos da rede, digamos *n* arcos, é convencionado representar todos por apenas um arco, acompanhado da notação numérica (rótulo) "*n*", indicando que se trata de um arco de *peso n*, ou seja, que tem o mesmo valor de *n* arcos [Murata89].

A tarefa de avaliar as transições habilitadas e de proceder os seus disparos, levando a rede a passar por uma seqüência de estados ao longo do tempo é a *simulação* da rede, também chamada de *"token game"*. A simulação é uma ferramenta importante para estudo do sistema pois, através dela, é possível compreender vários aspectos sobre o seu comportamento.

O modelo descrito até o momento constitui o tipo fundamental de rede de Petri, chamado de rede *Place/Transition*. Entretanto, outros tipos de rede surgiram ao longo da história das redes de Petri. Esses tipos oferecem maior poder de representatividade para a rede de Petri através da adição de tempo, probabilidades, funções matemáticas e outras variações. Particularmente, este trabalho se concentrará em três tipos de rede de Petri: a rede *Place/Transition*, descrita aqui, e as redes de Petri temporizada [Zuberek91][Ajmone95] e colorida [Jensen94], descritas nos capítulos posteriores.

A seguir, é fornecido um exemplo para ilustrar os conceitos básicos apresentados e na Seção 2.2 será dada a definição formal de redes de Petri.

Exemplo 2.1.1 A Figura 3 representa uma rede de Petri que modela a comunicação entre dois servidores. Quando o servidor *A* está pronto para enviar uma mensagem, um *token* é colocado no lugar "pronto_para_enviar". A transição "envia_msg" é então habilitada. O disparo desta transição gera um *token* para o lugar "transmissão", indicando que a transmissão de uma mensagem está ocorrendo. O lugar "pronto_para_receber" indica que o servidor *B* está esperando a chegada de uma mensagem. Quando este lugar e o lugar "transmissão" possuirem um *token* cada, a transição "recebe_msg" será habilitada. O disparo desta transição consumirá o *token* colocado no lugar "transmissão", indicando que a mensagem foi recebida e a transmissão foi concluída. Um *token* do lugar "pronto_para_receber" também é retirado e depois do disparo é colocado de volta, liberando o servidor *B* para a recepção de uma nova mensagem.

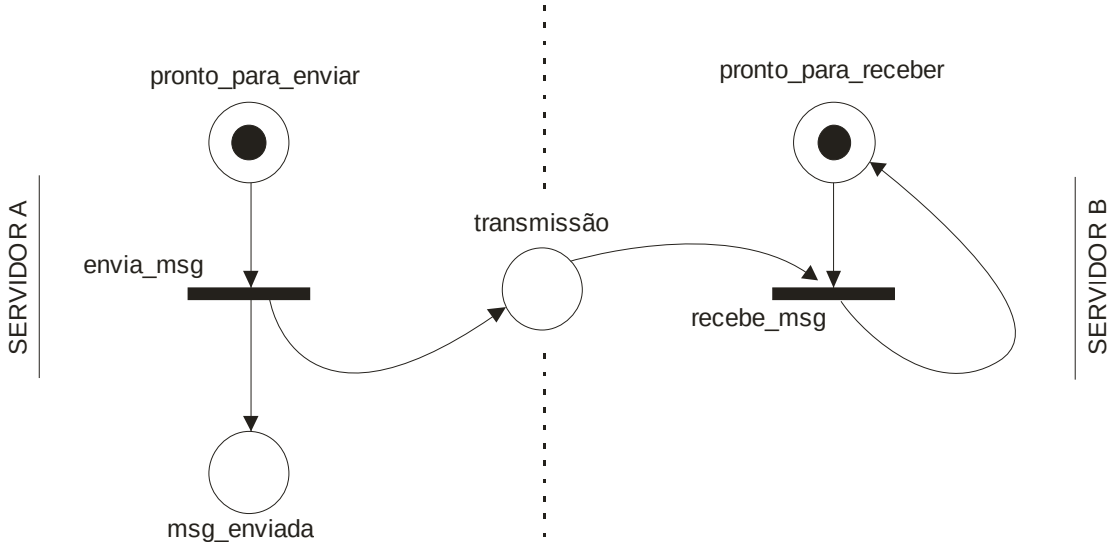


Figura 3. Comunicação entre dois servidores

2.2 O Modelo Matemático

Para evitar uma saturação de referências ao longo desta seção, fica explicitado que as definições aqui citadas foram compiladas a partir de [Maciel96], como base, e [Ajmone95], [Murata89] e [Zuberek91] como auxiliares.

O modelo das redes de Petri pode ser descrito de formas variadas. Existe uma abordagem que as define em termos da teoria de *bag* (extensão de conjunto que aceita repetição de elementos), outra que as define em termos de álgebra matricial e também uma baseada no conceito de relações. Aqui, será utilizada a representação matricial, que é definida como se segue:

DEFINIÇÃO 2.2.1 Uma **rede de Petri** é uma 4-tupla

$$R = (P, T, I, O) \tag{2.1}$$

Onde,

P é o conjunto finito de lugares;
T é o conjunto finito de transições;
$I : P \times T \rightarrow \mathbf{N}$ é a matriz de entrada (ou de *pré-condições*);
$O : P \times T \rightarrow \mathbf{N}$ é a matriz de saída (ou de *pós-condições*).

■

As matrizes de entrada e de saída contêm, em cada elemento a_{ij}, o peso do arco que liga o lugar p_i à transição t_j, se existir um arco, ou *zero*, caso não exista nenhum arco entre p_i e t_j. Os arcos presentes na matriz de entrada (I) são aqueles que partem do lugar e apontam para a transição correspondente, ou seja, são aqueles que **entram** nas transições. Os arcos na matriz de saída (O) são aqueles que partem da transição e apontam para o lugar, ou seja, **saem** das transições.

Exemplo 2.2.2 A estrutura correspondente à rede do Exemplo 2.1.1 é representada formalmente como a seguir:

$$R = (P, T, I, O)$$

$$P = \{\ pronto\_para\_enviar, transmissão, pronto\_para\_receber\ , msg\_enviada\ \}$$
$$T = \{\ envia\_msg, recebe\_msg\ \}$$

$$I = \begin{array}{c} \begin{matrix} envia\_msg & recebe\_msg \end{matrix} \\ \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{array}\right] \end{array} \begin{array}{l} pronto\_para\_enviar \\ transmissão \\ pronto\_para\_receber \\ msg\_enviada \end{array} \qquad O = \begin{array}{c} \begin{matrix} envia\_msg & recebe\_msg \end{matrix} \\ \left[\begin{array}{cc} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{array}\right] \end{array} \begin{array}{l} pronto\_para\_enviar \\ transmissão \\ pronto\_para\_receber \\ msg\_enviada \end{array}$$

Para representar a marcação da rede pode-se utilizar uma definição em termos de função ou em termos vetoriais. Dado que neste trabalho está sendo utilizada a representação matricial para redes de Petri, se torna mais coerente a utilização da representação vetorial de marcação, conforme a definição que se segue:

DEFINIÇÃO 2.2.2 Seja P o conjunto de lugares de uma rede R, o **vetor marcação** M é definido como

$$M = (\ m(p_1), m(p_2), ..., m(p_n)\ ) \qquad (2.2)$$

Onde,

$p_i \in P$;
$n = \#P$;
$m(p_i) \in \mathbf{N}$, corresponde à marcação do lugar p_i.

■

A associação de uma rede (estrutura) a uma marcação é chamada de *rede marcada*, e é definida a seguir:

DEFINIÇÃO 2.2.3 Uma **rede de Petri marcada** é um par

$$RM = (R, M_o) \tag{2.3}$$

Onde,

R é uma rede de Petri tal que $R = (P, T, I, O)$;
M_o é o vetor marcação que representa a marcação inicial da rede.

■

Exemplo 2.2.3 A marcação da rede apresentada no Exemplo 2.1.1 é caracterizada pelo vetor abaixo:

$$M_O = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} pronto\_para\_enviar \\ transmissão \\ pronto\_para\_receber \\ msg\_enviada \end{matrix}$$

2.3 Regras de Execução

Na Seção 2.1 foi apresentado de forma geral o processo de execução das redes de Petri elementares. Nesta seção são descritas as regras formais que regem o comportamento dinâmico de uma rede.

Considere para as definições ao longo desta seção uma rede de Petri dada por $R = (P, T, I, O)$.

DEFINIÇÃO 2.3.1 Uma transição t está **habilitada** (ou é **disparável**) em uma marcação M se

$$\forall p \in P, m(p) \geq I(p, t) \tag{2.4}$$

Denota-se a habilitação desta transição na marcação M por

$$M\ [t >$$

■

Note-se que uma transição que não possui nenhum lugar como pré-condição sempre satisfará esta condição e, portanto, sempre estará habilitada. Este tipo de transição é chamada de transição *fonte* (ou *source*) [Murata89].

Uma transição habilitada *pode* ser disparada, mas este evento *não é obrigatório*. O disparo de uma transição ocorre de forma não-determinística.

A definição formal do disparo de uma transição é dada por:

DEFINIÇÃO 2.3.2 O **disparo** de uma transição t em uma marcação M altera esta marcação, gerando uma nova marcação M' tal que

$$m'(p) = m(p) - I(p, t) + O(p, t), \forall p \in P \tag{2.5}$$

Quando o disparo de uma transição t em uma certa marcação M gera uma marcação M', denota-se isto por

$$M\ [t > M'$$

■

A Equação 2.5 corresponde ao processo de remoção de *tokens* de lugares que são pré-condições da transição (por isso a subtração do elemento da matriz de entrada) e a adição de *tokens* aos lugares que são pós-condiçãões da transição disparada (adição do elemento da matriz de saída).

Exemplo 2.3.1 Ainda continuando com o exemplo de comunicação entre servidores, demonstrado no Exemplo 2.1.1, vamos observar como ocorre o disparo da transição "envia_msg", do ponto de vista matemático. O disparo desta transição irá levar a rede da marcação M_o para uma marcação M.

$$M = \underset{M_O}{\begin{bmatrix}1\\0\\1\\0\end{bmatrix}} - \underset{I}{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}} + \underset{O}{\begin{bmatrix}0\\1\\0\\1\end{bmatrix}}$$

$$M = \begin{bmatrix}0\\1\\1\\1\end{bmatrix} \begin{matrix}\textit{pronto\_para\_enviar}\\ \textit{transmissão}\\ \textit{pronto\_para\_receber}\\ \textit{msg\_enviada}\end{matrix}$$

Transições que não possuem nenhum lugar como pós-condição, quando disparadas, apenas consomem marcas dos seus lugares de entrada. Tais transições recebem o nome de transições *de absorção* (ou *sink*) [Murata89].

Existe uma semântica de disparo na qual todas as transições habilitadas são disparadas ao mesmo tempo. Esta regra de disparo é chamada de *semântica de passo* [Reisig85]. Cada disparo de um conjunto de transições é chamado de disparo de um *passo*. No caso de transições em conflito (ver Seção 2.4), o conflito precisa ser resolvido antes do disparo, fazendo com que apenas uma das transições dispare, desabilitando a(s) conflitante(s).

As definições dadas até o momento constituem os aspectos fundamentais que regem a estrutura e o comportamento das redes de Petri. Apesar de terem sido apresentados apenas os conceitos referentes à rede de Petri *Place/Transition*, estes conceitos são essenciais e formam a base sobre a qual os outros modelos foram construídos. A teoria, entretanto, apresenta outros pontos que serão importantes para a compreensão dos modelos mais complexos, apresentados nos capítulos posteriores, e de aspectos relevantes relativos à questão da simulação, tema deste trabalho.

2.4 Conceitos Complementares

A partir das matrizes que definem a rede de Petri e das equações que regem o seu comportamento, diversas propriedades podem ser obtidas. Na verdade, isto é o que torna as redes de Petri uma técnica poderosa: possibilidade de análise matemática sobre os modelos.

Uma estrutura que auxilia o estudo de uma rede de Petri é a sua *matriz de incidência*, que é a diferença entre as matrizes de saída e entrada.

DEFINIÇÃO 2.4.1 Sejam *I* e *O* as matrizes de entrada e saída, respectivamente, de uma rede de Petri, a **matriz de incidência** desta rede é dada por

$$C = O - I \tag{2.6}$$

■

Esta matriz é útil no cálculo de diversas propriedades da rede porque representa a estrutura da rede de forma compacta. Mas para que isso seja verdadeiro é preciso que a rede não contenha *self-loops* [Murata89]. Um *self-loop* é um par (*p, t*) de lugar e transição onde este lugar é ao mesmo tempo pré-condição e pós-condição da transição. Uma rede que contenha *self-loops* é dita *impura* e sua matriz de inicidência não representa completamente a estrutura da rede.

DEFINIÇÃO 2.4.2 Uma rede $R = (P, T, I, O)$ é **pura** se, e somente se,

$$I(p, t) \cdot O(p, t) = 0, \forall (p,t) \in P \times T \tag{2.7}$$

■

A Definição 2.4.2 corresponde a dizer que a rede é *pura* quando não possui *self-loops*.

Uma rede impura pode ser refinada de forma que se torne uma rede pura através da inserção de pares *dummies* [Maciel96]. Um par *dummy* é composto por um lugar e uma transição ligados por um arco que são inseridos entre a transição e o lugar que pertencem a um *self-loop,* de maneira que o comportamento da rede seja mantido mas o *self-loop* seja eliminado. A Figura 4 demonstra um *self-loop* (Fig. 4.a) e um refinamento através de um par *dummy* que elimina o *self-loop* (Fig. 4.b).

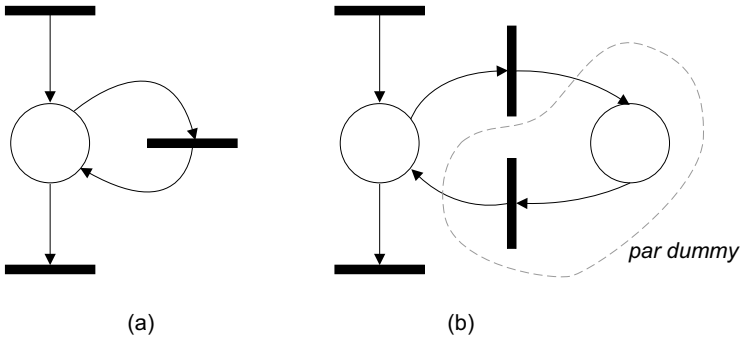


Figura 4. a) Exemplo de *self-loop*; b) Refinamento por par *dummy*.

Quando o comportamento de uma rede é estudado, é importante obter informações sobre que estados a rede pode alcançar e quais transições precisam ser disparadas para que aqueles estados possam ser alcançados. Assim, os conceitos de *seqüências disparáveis* e *alcançabilidade* são extremamente importantes.

Digamos que, em um certo estado, uma transição *t1* esteja habilitada. Se esta transição for disparada, levará o sistema a um novo estado em que uma outra transição *t2* pode estar habilitada. Quando isto ocorre, pode-se dizer que a seqüência *t1;t2* estava habilitada no primeiro estado. Assim, uma seqüência de transições $\sigma = t1;t2$ está habilitada se $M\ [t1 > M'$ e $M'\ [t2 > M''$. Denota-se então o disparo desta seqüência por $M\ [\sigma > M''$.

DEFINIÇÃO 2.4.3 Uma seqüência σ é uma **seqüência disparável** para uma marcação *M*, levando a rede para uma marcação *M''* (*M* [σ > *M''*) se, e somente se, ocorrer um dos casos

- σ é a seqüência vazia, tal que *M* [σ > *M''* $\Rightarrow$ *M''* = *M*; (2.8)
- $\sigma = \sigma';t$, tal que *M* [σ > *M'* e *M'* [*t* > *M''* (2.9)

■

A partir disto, podemos definir a *alcançabilidade* de uma marcação como:

DEFINIÇÃO 2.4.4 Uma marcação *M'* é **alcançável** a partir de uma outra marcação *M* se, e somente se, existe σ tal que *M* [σ > *M'*.

■

Se existir uma transição *t* na rede, tal que *M* [*t* > *M'*, diz-se que *M'* é *diretamente alcançável* a partir de *M*.

No estudo de redes de Petri é preciso, em diversos casos, gerar um grafo conhecido como *Grafo de Alcançabilidade (Reachability Graph)* ou *Grafo das Marcações Acessíveis* [Maciel96] da rede. Este é um grafo que possui como nodos o conjunto de todas as marcações alcançáveis a partir de uma marcação inicial. Um arco conecta um nodo *M* a qualquer outro nodo *M'* quando a marcação *M'* é diretamente alcançável a partir da marcação *M*. Estes arcos correspondem ao disparo de transições que levam a rede de cada marcação à outra.

Em uma certa marcação, pode acontecer de duas transições estarem habilitadas de uma forma tal que o disparo de qualquer uma delas leve a uma marcação na qual a outra não está mais habilitada. Isto ocorre quando os *tokens* consumidos pela transição que disparou eram os mesmos que habilitavam a outra transição. Diz-se, nesse caso, que as transições envolvidas se encontram em *conflito efetivo* (o *conflito estrutural*, é a definição dada ao caso em que duas transições compartilham um lugar como pré-condição). Esta é uma situação que merece atenção, pois pode implicar em fatos importantes relativos ao sistema modelado. Algumas discussões surgem quando a questão do conflito é abordada.

Em alguns casos, as pré-condições das transições podem possuir mais *tokens* do que aqueles necessários para que as transições sejam disparadas. Pode-se dizer que estas transições estão habilitadas "mais de uma vez" ou que o *grau de habilitação* [Ajmone95] delas é maior que um. Dessa forma, o disparo de uma transição não irá necessariamente impedir que a outra transição seja disparada logo em seguida. Esta é uma situação que merece atenção pois existem duas definições para o conflito efetivo. Segundo [Ajmone95], a definição de conflito efetivo engloba também este caso, afirmando que há conflito sempre que o disparo de uma transição diminui o grau de habilitação de outra. Em outra definição, apresentada em [Maciel96], a situação de conflito efetivo é apenas aquela em que uma das transições deixa de estar habilitada na marcação seguinte. A Figura 5 [Ajmone95] mostra um caso em que as duas definições discordam quanto a tratar-se ou não de um conflito efetivo.

Na rede representada na Figura 5, as transições *t1* e *t2* compartilham um lugar *p* como pré-condição e estão ambas habilitadas. Segundo [Maciel96, p. 37], *t1* e *t2* estão em conflito **se, e somente se, estiverem ambas habilitadas** e ***m(p) < I(p, t1) + I(p, t2)***. Na rede em questão (em que todos os pesos são iguais a 1), esta condição é falsa – a soma dos arcos conectando *p* a *t1* e *t2* não é menor que o número de *tokens* neste lugar. Logo, segundo a definição encontrada em [Maciel96], não há conflito efetivo nesta rede.

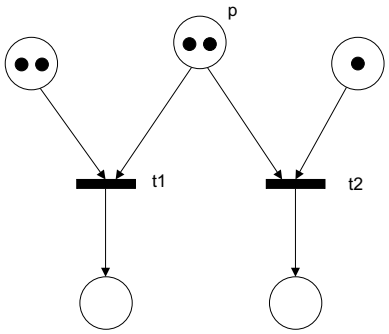


Figura 5. Situação de conflito efetivo, segundo [Ajmone95]

Por outro lado, a definição de [Ajmone95] afirma que o conflito efetivo ocorre quando **o grau de habilitação da transição é diminuído pelo disparo de outra**. Isto é verdadeiro para *t1* em relação ao disparo de *t2*. Logo, há conflito (embora o mesmo não seja verdadeiro para *t2* em relação ao disparo de *t1*. A esta situação [Ajmone95] dá o nome de *conflito assimétrico*).

A existência de duas definições homônimas mas referindo-se a situações diferentes na teoria das redes de Petri torna necessário indicar qual a definição escolhida, uma vez que a menção ao termo "conflito efetivo" não é suficiente. Neste trabalho, foi seguida a definição segundo [Ajmone95].

A seguir estão formalizadas as definições de grau de habilitação e conflito efetivo. Elas foram adaptadas a partir de [Ajmone95] de forma a manter a notação matricial para as redes de Petri.

DEFINIÇÃO 2.4.5 Chama-se **grau de habilitação** (*enabling degree*) de uma transição a função $ED : T \times (P \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$ tal que:

$\forall t \in T$, dada a marcação M,

$$ED(t, M) = k \text{ se, e somente se,} \quad (2.10)$$
$$\forall p \mid I(p, t) > 0, m(p) \geq k \cdot I(p, t) \text{ e}$$
$$\exists p \mid I(p, t) > 0 : m(p) < (k + 1) \cdot I(p, t).$$

■

Denota-se então por $ED(t, M)$ o grau da habilitação de t na marcação M.

DEFINIÇÃO 2.4.6 Duas transições *t1* e *t2* se encontram em **conflito efetivo** numa marcação *M* se, e somente se,

$$M\,[t1 > M' \text{ e}$$
$$ED(t2, M) > ED(t2, M') \quad (2.11)$$

■

O que significa que, sempre que o disparo de uma transição diminui o grau de habilitação de uma outra, estas transições estão em conflito efetivo (esta definição está de acordo com a bibliografia, porém há um equívoco na definição formal dada em [Ajmone95, p. 44], certamente devido a um erro tipográfico. A versão apresentada aqui é a correta.).

Há um caso especial de conflito que é de interesse particular, chamado de conflito de *escolha livre* (*free-choice*). Duas transições estão em conflito de escolha livre se estão em

conflito efetivo *simétrico* (qualquer uma delas que seja disparada irá causar diminuição no grau de habilitação da outra) e se são habilitadas pelas mesmas pré-condições [Ajmone95].

DEFINIÇÃO 2.4.7 Duas transições *t1* e *t2* estão em **conflito de escolha livre** numa marcação *M* se, e somente se, todas as condições forem satisfeitas:

- $\forall p \in P, I(p, t1) = I(p, t2)$;
- $M\,[t1 > M'$ e $ED(t2, M) > ED(t2, M')$; *e*
- $M\,[t2 > M''$ e $ED(t1, M) > ED(t1, M'')$ (2.12)

■

Existem duas sub-classes das redes de Petri que são definidas a partir de variações do conceito de escolha livre [Maciel96]. A primeira é chamada de rede de Petri *escolha livre*, e é aquela em que, se um lugar fizer parte da pré-condição de várias transições, este lugar será a única pré-condição destas transições (Figura 6).

A segunda sub-classe é chamada de rede de Petri *escolha livre estendida*, e é aquela em que, se dois lugares fizerem parte das pré-condições de uma mesma transição, então o conjunto de transições para as quais estes lugares são pré-condições deve ser igual (Figura 1).

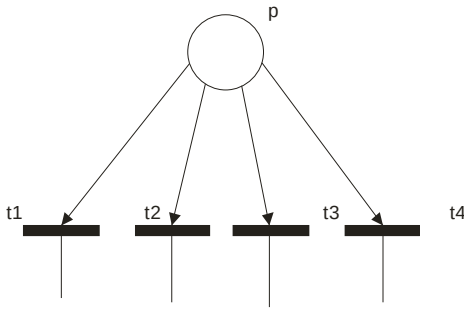


Figura 6. Exemplo de escolha livre

Um aspecto que muitas vezes está obscuro na bibliografia e que é de relevante importância é quanto aos pesos dos arcos que ligam os lugares e as transições conflitantes. Para manter a semântica da escolha livre nas duas sub-classes acima, os pesos dos arcos que ligam cada uma das transições a um mesmo lugar devem sempre ser iguais (Figura 8.a) [Zuberek91]. O motivo disto é que, uma vez que os pesos sejam diferentes, as pré-condições não serão mais as mesmas, levando as transições a estarem habilitadas em momentos diferentes. A Figura 8.b demonstra uma situação falsa de escolha livre, pois os pesos são diferentes.

Este requisito está formalizado na Definição 2.4.7.

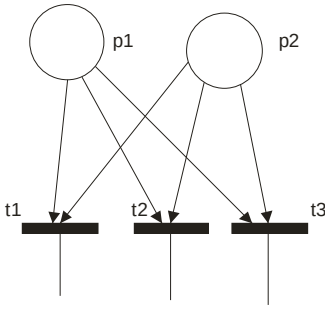


Figura 7. Exemplo de escolha livre estendida

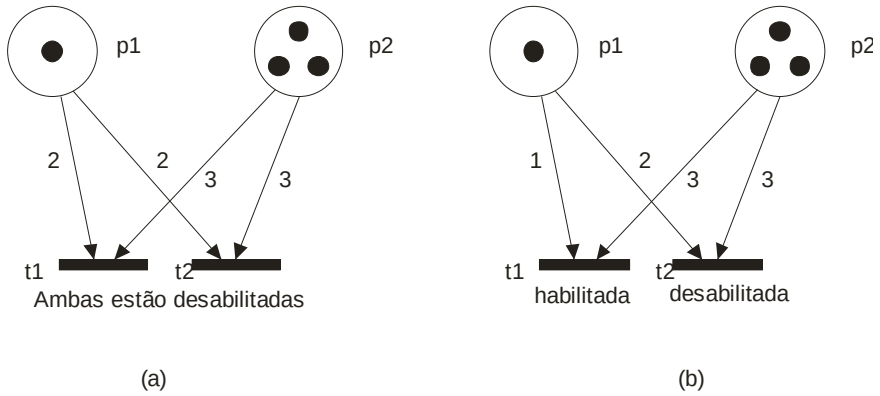


Figura 8. a) Escolha livre. Pesos são todos iguais. Ambas as transições estão desabilitadas; b) Ausência de escolha livre. Transições não são sempre habilitadas no mesmo momento.

Nos capítulos posteriores, duas extensões das redes de Petri serão estudadas: as redes de Petri temporizadas e as redes de Petri coloridas. A primeira é apresentada no Capítulo 3 e consiste em uma extensão do modelo aqui apresentado, acrescentando um novo atributo às transições relativo a questões temporais. Esta modificação simples cria uma série de novas possibilidades e problemas que serão apresentados no capítulo correspondente. No Capítulo 4 será apresentada a classe de redes de Petri que é computacionalmente mais poderosa. Através de notações adicionadas à rede, a rede de Petri colorida é capaz de simplificar enormemente a estrutura de um modelo. Será visto que as regras de execução são um pouco mais complexas para esta classe de redes de Petri e será discutida a abordagem utilizada para contornar alguns problemas existentes durante a implementação do simulador.

Capítulo 3

Redes de Petri Temporizadas

A descrição original das redes de Petri, correspondente à que foi apresentada no Capítulo 2, não faz nenhuma consideração sobre a duração que leva uma determinada ação para ser executada no sistema. Todas as transições entre estados ocorrem instantaneamente.

Devido a esta característica, as redes de Petri originalmente não eram capazes de fornecer nenhuma informação relacionada ao desempenho do sistema estudado. Nos primeiros anos após a publicação da tese de Petri, surgiram diversas questões quanto à aplicação desta técnica sob um outro ponto de vista, no qual os aspectos relacionados ao comportamento do sistema ao longo do tempo pudessem ser avaliados. Para que isto fosse possível, era necessário criar uma forma de representar os tempos despendidos pelo sistema na realização de suas atividades.

As primeiras propostas de redes de Petri com consideração de tempo surgiram em 1973 e em 1976 [Ajmone95], e foram aplicadas pelos seus autores na modelagem de uma arquitetura de computadores (no trabalho de Noe & Nutt) e de protocolos de comunicação (Merlin & Faber). Outros trabalhos surgiram ao longo dos anos, cada um propondo um modelo que melhor se enquadrasse às necessidades de sua aplicação [Ajmone95]. Atualmente existe um conjunto relativamente amplo de definições para redes de Petri temporizadas, que podem ser utilizadas dependendo da aplicação ou do suporte da ferramenta escolhida pelo projetista. Ao longo deste capítulo serão descritos os princípios básicos que regem as redes de Petri temporizadas, conforme apresentadas por [Ajmone95], e os conceitos relevantes para a implementação dos simuladores.

3.1 Relacionando Tempo e Transições

À primeira vista, pode parecer fácil relacionar tempo ao disparo de transições. Poder-se-ia, por exemplo, simplesmente adicionar uma notação a cada transição, informando quanto tempo é despendido pelo sistema na ocorrência (disparo) daquela transição. Quando uma certa transição disparasse, apenas se somaria o tempo despendido por ela a uma contagem de tempo total. Entretanto, quando observamos o processo de forma mais detalhada são levantadas diversas dúvidas.

Digamos que numa certa marcação uma transição chamada “muda_chave”, representando a mudança de chave em alguma máquina hipotética, esteja habilitada. Sabemos que quando a chave for modificada nesta máquina, o sistema demorará 3 segundos para mudar efetivamente de estado. A Figura 9 exemplifica esta situação. Observe que a transição nesta figura é representada

por uma caixa sem preenchimento. Esta é a representação comumente utilizada para indicar que trata-se de uma transição temporizada [Ajmone95].

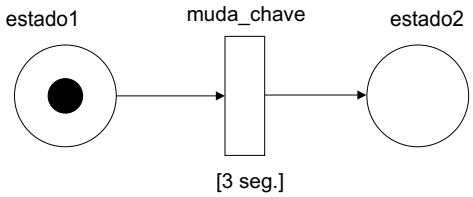


Figura 9. Transição "muda_chave" na máquina hipotética

Para executar a ação de mudar a chave com a duração de 3 segundos, poderíamos iniciar o disparo da transição no instante 0 segundo, removendo todos os *tokens* dos lugares que lhe são pré-condição, esperar um período de 3 segundos e, então, concluir o disparo no instante 3 segundos, adicionando os *tokens* às suas pós-condições (a Figura 10 mostra a execução desses três passos). Desta forma, pode-se representar claramente o tempo que o sistema leva para mudar de estado. Este modelo é bastante simples e é utilizado na prática em diversas situações. Mas nem sempre é possível representar o tempo desta forma.

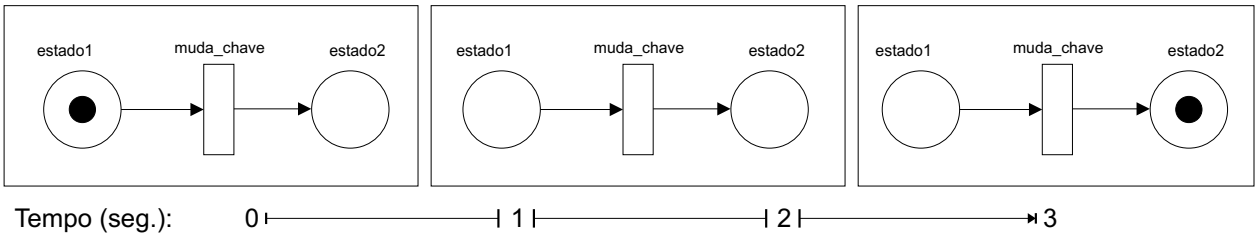


Figura 10. Disparo da transição "muda_chave". A) Início do disparo; b) *Tokens* foram removidos do lugar da pré-condição; c) *Tokens* são adicionados ao lugar da pós-condição

Imagine que o sistema modelado não seja mais uma máquina, mas agora uma loja de sapatos. Nesta loja existem diversos atendentes que recebem comissão por suas vendas. Sempre que chega um cliente, todos os funcionários competem para atendê-lo, pois quanto mais clientes um funcionário atender, maiores suas chances de receber uma boa comissão no final do mês. Quando um dos funcionários alcança o novo cliente, entretanto, os outros funcionários não têm mais nada a fazer, senão voltar a seus lugares para esperar que o próximo cliente chegue. Como isso pode ser representado por uma rede de Petri?

Poderíamos dizer que a ação de um funcionário ir até o cliente atendê-lo é uma transição, e esta ação dura um certo tempo para ser executada, dependendo do funcionário. Podemos dizer também que a chegada de um cliente na loja é a condição que faz com que os funcionários realizem essa ação, pois sem clientes os funcionários ficam ociosos em seus lugares. Portanto, a *chegada de um cliente* é modelada como um lugar, e acontece quando um *token* atinge este lugar. Também pode-se dizer que vários *tokens* neste lugar representam vários clientes chegando. Quando o funcionário alcança o cliente, o *token* é removido e um *token* é colocado em outro lugar, o qual representa que um *cliente está sendo atendido*. Este sistema está representado na Figura 11.

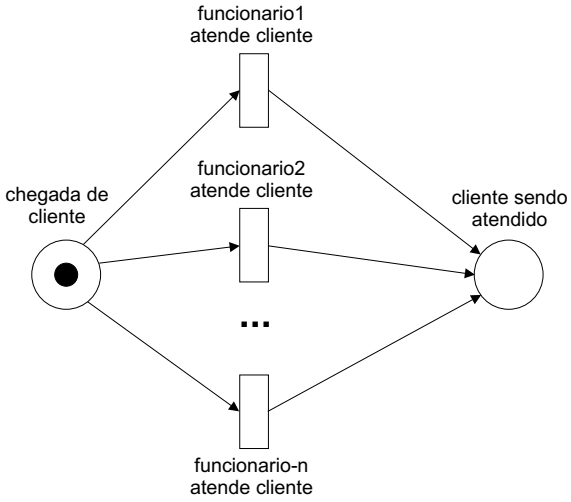


Figura 11. Rede de Petri temporizada representando atendimento numa loja de sapatos

Se um cliente chega à loja, todos os funcionários se dirigem para atendê-lo. Entretanto, quando o funcionário que concluiu esta tarefa mais rápido chegar ao cliente, todos os outros funcionários ficam inabilitados de o atenderem. Surge então a necessidade de representar casos em que atividades que estão em execução no sistema possam ser interrompidas por outras. Neste caso, as atividades do sistema são aqueles realizadas pelos funcionários. Note que neste modelo há uma diferença ao que foi utilizado para representar a máquina. Apesar de existirem tempos associados às tarefas, o disparo das transições não podem ocorrer da mesma forma como aconteceu naquele exemplo, pois isto não permitiria que as atividades ocorressem simultaneamente e que uma delas interrompesse as demais. Aqui, o tempo de execução das tarefas é contado a partir do momento em que a transição é habilitada e, uma vez que o tempo necessário para a conclusão da tarefa decorra, o disparo da transição correspondente ocorre de forma instantânea. Ou seja, a transição do funcionário vencedor consome o *token* do lugar "chegada de cliente" e todas as outras transições ficam desabilitadas. Esta transição irá então gerar imediatamente um *token* para o lugar "cliente sendo atendido", que é sua pós-condição. A semântica utilizada no exemplo anterior, portanto, não é mais aplicável para este exemplo.

Se continuássemos pensando em outros exemplos, encontraríamos diversas outras situações em que as escolhas para o mecanismo de funcionamento da rede de Petri com tempos associados a transições teriam grande impacto na sua representatividade. Situações em que duas transições disparam simultaneamente, em que o grau de habilitação de uma transição é maior que um, dentre outras. As definições encontradas nas seções que se seguem irão tratar destas questões sob o ponto de vista formal.

Também é essencial que a rede de Petri temporizada não perca a representatividade do sistema modelado quando a utilizamos sem considerar o tempo. Garantindo-se isto, as técnicas disponíveis para a análise de redes de Petri *Place/Transition* continuam podendo ser utilizadas para analisar o sistema modelado por redes de Petri temporizadas.

3.2 Conceitos

Estruturalmente, a única diferença entre uma rede de Petri temporizada e a rede de Petri *Place/Transition* é a notação de tempo associada às transições. Grandes diferenças, entretanto, surgem na dinâmica de execução da rede.

O comportamento dinâmico da rede de Petri temporizada será definido pelas regras escolhidas para o disparo das transições. A primeira opção que temos é quanto à *política de disparo* a ser adotada. Existem duas possibilidades [Tavares06]:

- **Disparo em três fases.** O tempo é consumido durante o disparo, ou seja, os *tokens* são removidos das pré-condições no instante em que a transição é habilitada. Após isso um intervalo de tempo é decorrido, correspondendo ao tempo associado à transição. Em seguida são adicionados os *tokens* às pós-condições, conforme o exemplo da máquina na seção anterior.
- **Disparo atômico.** O disparo é instantâneo, entretanto, quando a transição é habilitada, os *tokens* permanecem nos lugares das pré-condições pelo tempo de execução associado à tarefa. Só após esse tempo o disparo pode ocorrer. Isto corresponde ao modelo do exemplo da loja de sapatos.

Para a implementação dos simuladores, foi escolhida a política de disparo atômico, pois é mais geral e permite que processos possam ser interrompidos por outros quando eles estão concorrendo no tempo, uma característica que é de grande utilidade para representar diversos tipos de sistemas.

Para a representação do tempo, optou-se por utilizar o modelo em que é feita a associação não de um tempo fixo para cada transição, mas de um intervalo dentro do qual esta transição deve ocorrer [Merlin76][Tavares06]. O intervalo é especificado em termos de um tempo mínimo e um máximo e o disparo pode ocorrer em qualquer instante aleatoriamente escolhido dentro deste intervalo, com distribuição de probabilidade uniforme. Note que, se até o instante de tempo máximo a transição não fori disparada, ela *deve* ser disparada neste momento.

A seguir é dada a definição de uma rede de Petri temporizada. Em seguida, serão discutidos os aspectos que envolvem a sua execução (simulação).

DEFINIÇÃO 3.2.1 Uma **rede de Petri temporizada** é uma 5-tupla

$$R = (P, T, I, O, C) \tag{3.1}$$

Onde,

P é o conjunto finito de lugares;
T é o conjunto finito de transições;
$I : P \times T \rightarrow \mathbf{N}$ é a matriz de entrada (ou de *pré-condições*);
$O : P \times T \rightarrow \mathbf{N}$ é a matriz de saída (ou de *pós-condições*);
$C : T \rightarrow \mathbf{N} \times \mathbf{N}$ é uma função que associa a cada transição um intervalo de tempo (min, max), tal que $min >= 0$ e $max >= min$.

■

Os instantes indicados por *min* e *max* são relativos ao momento em que a transição foi habilitada. Por exemplo, um intervalo (1, 3) indica que a transição só poderá disparar após transcorrida 1 unidade de tempo a partir de sua habilitação e deve ser disparada no máximo até 3 unidades de tempo após sua habilitação.

Deve ainda ser considerada a memória histórica mantida pela rede sobre os tempos passados nas transições. A *política de memória* define o que acontece quando uma mudança de estado ocorre na rede. Três alternativas podem ser levadas em consideração [Ajmone95]:

- **Reamostragem** (*Resampling*). Após o disparo de uma transição todos os temporizadores são reiniciados. Os tempos de habilitação de todas as transições voltam a ser zero, nenhuma memória do passado é mantida.

- **Memória de Habilitação** (*Enabling Memory*). Após o disparo de uma transição, todas as transições que estavam habilitadas na marcação anterior e que permanecem habilitadas na nova marcação mantém os valores de seus temporizadores. A rede mantém a memória de quanto tempo cada transição permaneceu habilitada desde a última vez em que elas foram habilitadas. Apenas quando uma transição é desabilitada na nova marcação ou quando esta é disparada é que o seu tempo é perdido. Os tempos de habilitação são mantidos por *variáveis de memória de habilitação* associadas a cada transição.
- **Memória de Idade** (*Age Memory*). Os tempos não são reinicializados quando uma transição é desabilitada. Ou seja, se uma transição que estava habilitada perder a habilitação em uma certa marca, o seu temporizador é mantido com o mesmo valor e continuará a contagem assim que a transição for habilitada novamente em uma marcação futura. É associada uma *variável de memória de idade* a cada transição para manter estes valores.

A memória dos tempos de transições, junto com a marcação da rede e os estados de habilitação das transições formam o *estado global da rede temporizada*.

O simulador implementado para a biblioteca utiliza a abordagem de memória de habilitação. Para representar as variáveis de memória, é definido um *vetor memória de habilitação*, como se segue:

DEFINIÇÃO 3.2.2 Seja $T = \{ t_1, t_2, ..., t_n \}$ o conjunto de transições de uma rede de Petri temporizada, o **vetor memória de habilitação** é dado por:

$$H = (h(t_1), h(t_2), ..., h(t_n)) \qquad (3.2)$$

Onde,

$h(t_i) \in \mathbf{N}$ é a variável de memória de habilitação da transição t_i;

Se $t_i \in T$ não estiver habilitada, $h(t_i) = 0$.

■

A definição de memória de habilitação utilizada aqui considera que os temporizadores são incrementados ao longo do tempo e não decrementados, como ocorre em [Ajmone95]. Note que, uma vez que o estado de habilitação de uma transição também faz parte do estado global da rede, pode-se manter todas as transições em um único vetor de memória sem a possibilidade de equívoco quanto ao seu estado. Transições não-habilitadas simplesmente têm tempo de habilitação igual a zero.

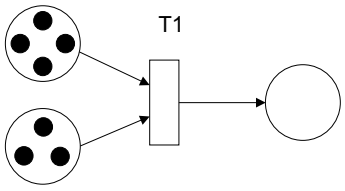


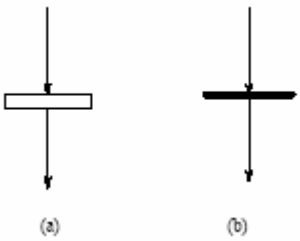
Figura 12. Transição com grau de habilitação igual a três

Mas o que acontecerá com o temporizador da transição se o grau de habilitação dela for maior do que um? A Figura 12 demonstra uma transição com grau de habilitação igual a três. Diferentes semânticas podem ser adotadas para tratar desta situação [Ajmone95]:

- **Servidor Único** (*Single-Server Semantics*). Os *tokens* são processados de forma serial. Após o primeiro disparo, o temporizador é reiniciado como se a transição tivesse sido habilitada novamente. Assim, se o retardo associado à transição for de um tempo fixo τ, e o grau de habilitação da transição for n, então será necessário $n \times \tau$ unidades de tempo para que todos os *tokens* sejam consumidos pela transição.
- **Servidores Infinitos** (*Infinite-Server Semantics*). Para cada conjunto de *tokens* habilitando a transição, um novo temporizador é criado para processar o retardo do disparo. Assim, uma transição com grau de habilitação igual a dois terá dois temporizadores correndo em paralelo. Se o grau de habilitação for n, haverá n temporizadores. A transição irá disparar sempre que um destes temporizadores atingir o tempo limite e o disparo não afeta os contadores dos temporizadores restantes.
- **Servidores Múltiplos** (*Multiple-Server Semantics*). É semelhante à semântica de servidores infinitos, mas aqui existe um limite para a quantidade de temporizadores executando em paralelo. Se o grau de habilitação da transição for menor ou igual a um certo limite k, então k temporizadores podem executar em paralelo. Se o grau de habilitação for aumentado além de k, não será criado nenhum novo temporizador para processar o tempo para o novo disparo até que o grau de habilitação tenha diminuído abaixo de k (após o disparo de alguma transição).

No simulador implementado, foi utilizada a semântica de servidor único.

Em muitos casos, existe a necessidade de representar transições entre estados que não consomem tempo para serem executadas. Isto pode ocorrer por diversos motivos [Ajmone95]. Muitas vezes a mudança de estado é complexa demais e precisa ser dividida em passos intermediários que não consomem tempo. Para dar suporte a essas situações, é possível adicionar à rede de Petri temporizada *transições imediatas*. Trata-se simplesmente de transições que disparam imediatamente quando são habilitadas. Uma representação gráfica utilizada para transições imediatas é um retângulo estreito preenchido na cor preta, tal como a que representa transições comuns em uma rede não-temporizada (Figura 13) [Ajmone95].



(Figura retirada de [Ajmone95])

Figura 13. (a) Transição temporizada; (b) Transição imediata

3.3 Conflitos

Quando introduzimos tempo à rede de Petri, uma atenção precisa ser dada à questão do conflito. No exemplo da loja de sapatos (Figura 11) as transições que representam os funcionários estão todas em conflito de escolha livre. Em redes não-temporizadas, a escolha de qual transição irá disparar é não-determinística. Entretanto, quando adicionamos tempo às transições, a resolução

do conflito pode se tornar dependente dos retardos associados às transições [Ajmone95]. A transição de menor tempo de retardo é a que irá disparar neste caso, desabilitando as transições conflitantes.

Quando o conflito é entre transições imediatas, a resolução de conflito pode ser determinística, utilizando-se uma política de prioridade [Ajmone95] ou de forma não-determinística, obedecendo a alguma distribuição de probabilidade.

É possível utilizar transições imediatas para separar a resolução de conflitos da especificação de tempo. Digamos que, na loja de sapatos, o gerente tenha observado que a competição entre os funcionários para alcançar o cliente que chega na loja estava gerando muitas desavenças entre eles. Para evitar isso e para evitar que os clientes sejam constrangidos pela competição entre os funcionários para atendê-lo, ele estipula uma nova regra para decidir quem atenderá o cliente recém-chegado. Os funcionários deverão fazer um sorteio entre eles à cada chegada de cliente. Aquele que for sorteado, irá atender o cliente.

A rede de Petri da Figura 14 modela esta nova condição através da inserção de transições imediatas. Quando um *token* atingir o lugar "chegada de cliente", todas as transições imediatas são habilitadas e a escolha de qual delas irá disparar é aleatória. Cada transição imediata representa que um dos funcionários foi sorteado. Após a resolução por sorteio, o funcionário se dirige até o cliente, o que consome uma quantidade de tempo. As transições temporizadas representam essa ação, como fora modelado anteriormente.

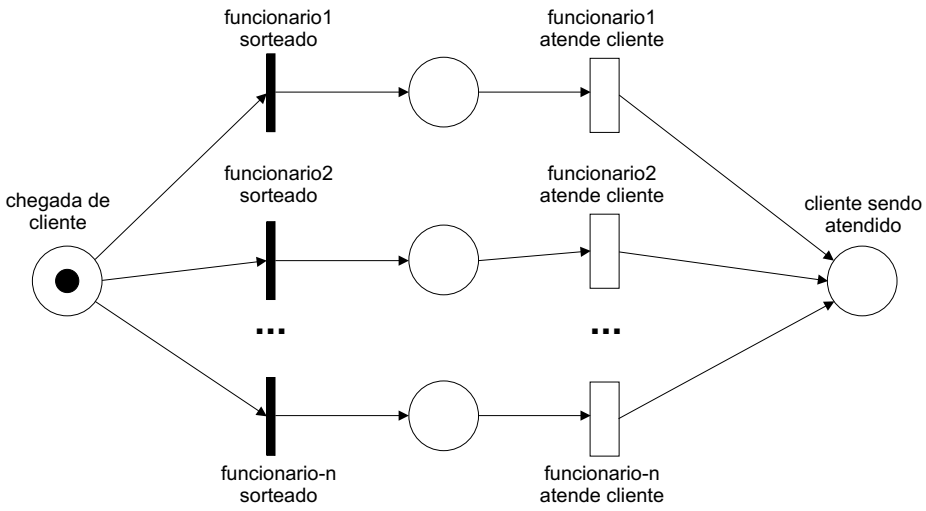


Figura 14. Rede de Petri com transições imediatas para modelar sorteio na loja de sapatos

Uma restrição ao uso de transições imediatas em uma rede temporizada é a de que a rede não pode apresentar estados nos quais uma seqüência infinita de transições imediatas esteja habilitada [Zuberek91].

3.4 Exemplo

A seguir, é apresentado um exemplo para ilustrar uma rede de Petri temporizada. Neste exemplo será utilizado o modo de disparo atômico, a política de memória de habilitação e a semântica de servidor único, conforme ocorre na implementação do simulador.

O exemplo ilustra uma parte de uma linha de produção. Os processos A e B produzem partes de um produto, que são reunidas pelo processo final C. O processo A representa um

processo de etiquetagem de embalagens, que é automatizado. O processo B representa uma tarefa manual, na qual funcionários verificam a qualidade do produto antes deste ser embalado. A transição “reune_materiais” é a entrada para o processo C, que reúne a embalagem proveniente do processo A ao produto proveniente do processo B. A Figura 15 demonstra a aparência desta rede.

A representação matemática desta rede é a que se segue:

$$R = (P, T, I, O, C)$$

$$P = \{ embalagem\_padrao, embalagem\_etiquetada, produto\_pronto, produto\_verificado, pronto\_para\_embalar, produto\_embalado \}$$

$$T = \{ etiquetagem, verificação, reune\_materiais, embala\_produto \}$$

$$I = \begin{array}{c} \begin{matrix} etiquetagem & verificacao & reune\_materiais & embala\_produto \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} embalagem\_padrao \\ embalagem\_etiquetada \\ produto\_pronto \\ produto\_verificado \\ pronto\_para\_embalar \\ produto\_embalado \end{array}$$

$$O = \begin{array}{c} \begin{matrix} etiquetagem & verificacao & reune\_materiais & embala\_produto \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{l} embalagem\_padrao \\ embalagem\_etiquetada \\ produto\_pronto \\ produto\_verificado \\ pronto\_para\_embalar \\ produto\_embalado \end{array}$$

$$C = \{ (etiquetagem, (3, 4)), (verificacao, (5, 10)), (embala\_produto, (1, 2)) \}$$

A marcação, conforme apresentada na Figura 15, é dada por:

$$M = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{array}{l} embalagem\_padrao \\ embalagem\_etiquetada \\ produto\_pronto \\ produto\_verificado \\ pronto\_para\_embalar \\ produto\_embalado \end{array}$$

Note que sob esta marcação todas as transições estão habilitadas. Como a transição “reune_materiais” é imediata, ela será a primeira a disparar.

Uma vez que a transição imediata tenha disparado e se tornado desabilitada, os temporizadores das transições restantes são iniciados (assumindo que todas as transições foram habilitadas no mesmo instante).

Observando-se as restrições de tempo das transições, pode-se verificar que a transição “embala_produto” será a primeira a disparar, pois o seu tempo limite é de 2 unidades de tempo, o que é menor que os tempos mínimos das outras transições. A próxima a disparar será a transição “etiquetagem”, uma vez que seu tempo limite é inferior ao tempo mínimo da transição “verificacao”.

O disparo da transição “verificacao" tem um intervalo mais amplo pois trata-se de uma atividade humana e, portanto, mais propensa a variações. Pode-se notar que neste intervalo de 5 unidades de tempo, a transição “etiquetagem” pode disparar novamente. O que ocorrerá é que certamente haverá um acúmulo de *tokens* no lugar “embalagem_etiquetada”, devido ao Processo A ser mais rápido que o Processo B e em vista do fato de que o Processo C depende de ambos os anteriores para prosseguir. É possível verificar facilmente que o Processo B constitui o gargalo do sistema.

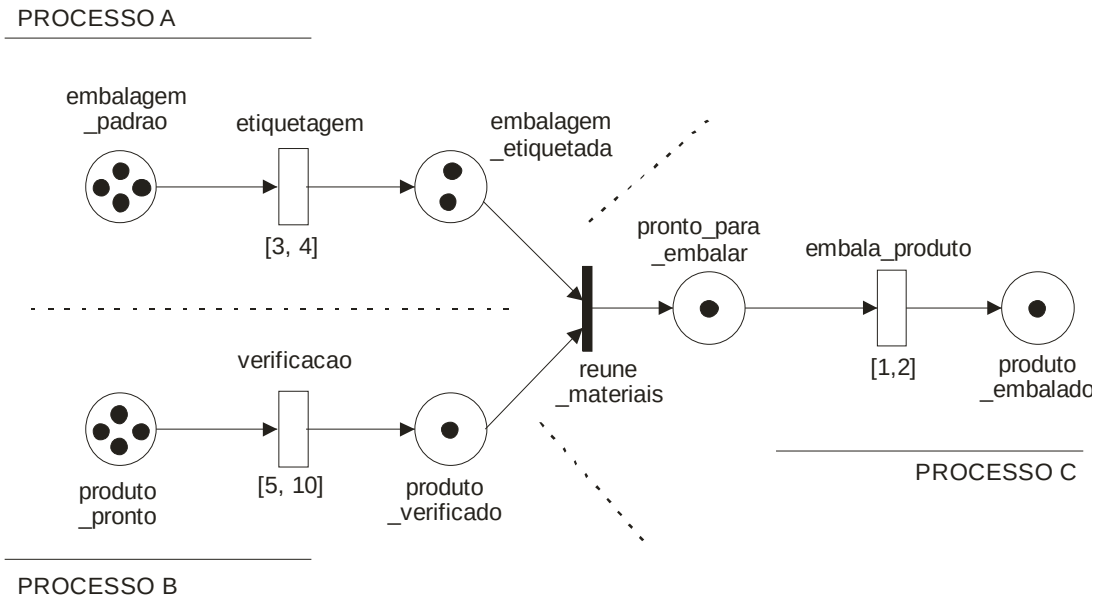


Figura 15. Exemplo de linha de produção modelada com redes temporizadas.

Através da simulação de redes de Petri temporizadas, é possível verificar este tipo de situação em sistemas e diversas outras que não poderiam ser verificadas com a utilização de redes *Place/Transition* Este tipo de estudo é realizado na área de Análise de Desempenho, na qual as redes de Petri têm sido aplicadas com enorme sucesso.

Capítulo 4

Redes de Petri Coloridas

As redes de Petri coloridas surgiram da necessidade em representar sistemas muito grandes e complexos, que são encontrados em aplicações industriais reais. Utilizando-se redes de Petri ordinárias, o tamanho destes sistemas se tornava um grande complicador para sua modelagem e estudo [Maciel96][Jensen94]. A idéia por traz das redes de Petri coloridas é unir a capacidade de representar sincronização e concorrência das redes de Petri com o poder expressivo das linguagens de programação. Através dessa união, sistemas cujo estudo anteriormente era impraticável tornaram-se passíveis de estudo.

Uma rede de Petri colorida é uma rede de Petri na qual os *tokens* possuem um tipo e um valor associados, permitindo que eles possam ser diferenciados entre si. Desta forma, um lugar na rede de Petri passou a representar não apenas uma certa condição, mas toda uma classe de situações que podem se apresentar de diferentes formas, de acordo com os valores dos *tokens* presentes em sua marcação. Estes valores são então utilizados em expressões que são calculadas durante a avaliação de transições, bem como durante o disparo destas.

No início, a diferenciação dos *tokens* era feita por cores, daí o nome de rede de Petri colorida. Atualmente são utilizadas tipos de dados estruturados, permitindo que operações bastante complexas sejam representadas na rede [Maciel96].

Neste capítulo serão apresentadas as regras gerais que regem o comportamento das redes de Petri coloridas. Aspectos relacionados à implementação do seu simulador em Java serão tratados em capítulos posteriores, quando for mais oportuno. O objetivo deste capítulo é expor os conceitos relacionados às redes coloridas que serão importantes para a compreensão dos processos e problemas que envolvem a sua simulação.

4.1 Visão Geral

As redes de Petri coloridas são compostas por três partes [Maciel96]:

- estrutura;
- declarações;
- inscrições.

A *estrutura* da rede colorida corresponde à estrutura de uma rede de Petri ordinária (*Place/Transition*), sendo formada pelos lugares, transições e arcos. Cada lugar possui um tipo de

dado associado. Esse tipo indica o conjunto de valores que os *tokens* contidos neste lugar podem representar. O tipo de um lugar também é chamado de seu *conjunto de cores* (*color set*).

As *declarações* definem os tipos e as variáveis que serão utilizados na rede. Esta definição é dada utilizando-se uma linguagem matemática ou de programação.

As *inscrições* são anotações associadas aos elementos da rede. Os tipos de inscrições são diferentes para lugares, transições e arcos [Maciel96]:

- **Inscrições de lugares.** Os lugares possuem três tipos de inscrições: *nome*, *tipo* e *expressão de inicialização*. O *nome* não possui valor semântico, sendo apenas um elemento que facilita a sua identificação. O *tipo* indica o conjunto de possíveis valores associado ao lugar. O lugar só poderá conter *tokens* com valores deste tipo. A *expressão de inicialização*, por sua vez, é uma expressão escrita na linguagem de representação adotada. Esta expressão indica a marcação inicial a ser atribuída ao lugar.
- **Inscrições de transições.** As transições possuem dois tipos de inscrições: *nome*, que também não possui significado formal, e *expressão guarda*. O papel da *expressão guarda* será compreendido nas discussões posteriores.
- **Inscrições de arcos.** Os arcos possuem como inscrição apenas uma *expressão*. Essa expressão substitui o *peso* que existia nas redes anteriores. Seu papel é semelhante: indica o conjunto de *tokens* que devem ser retirados ou adicionados aos lugares adjacentes à este arco no caso do disparo da transição. Como agora os *tokens* armazenam valores, torna-se necessário o uso de uma expressão mais complexa.

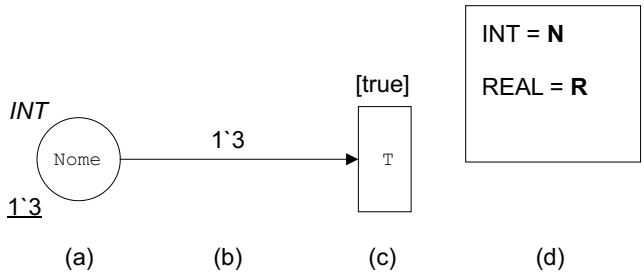


Figura 16. Elementos de uma rede colorida.

A Figura 16 ilustra esses elementos em uma rede colorida simples. As *declarações* são comumente escritas dentro de uma caixa de texto posicionada próximo à rede (Figura 16.d). Para facilitar a visualização, as *inscrições* são escritas de forma diferenciada. Aquelas que representam *nomes* são escritas utilizando-se fonte uniformemente espaçada (como a tipografia das máquinas datilográficas mecânicas). Para os lugares, os *tipos* são escritos em itálico e as expressões de inicialização são sublinhadas (Figura 16.a). As *expressões guarda* das transições ficam entre colchetes. Observe que as transições são representadas por retângulos não-preenchidos (Figura 16.c). As *expressões dos arcos* (Figura 16.b) não recebem formatação especial.

A inscrição “1´3” da inicialização do lugar rotulado por “Nome” representa um *bag* que contém um único elemento de valor 3.

O comportamento dinâmico da rede de Petri colorida mantém o mesmo princípio original das redes de Petri, mas é bem mais complexo. A Tabela 1 compara as redes de Petri *Place/Transition* às redes coloridas.

No exemplo demonstrado na Figura 16, as expressões dos arcos não possuem variáveis. São apenas expressões constantes. Nas redes coloridas, entretando, este é o tipo menos comum de expressão. Em geral, o conjunto de arcos que entram ou saem de uma transição compartilham variáveis livres entre si e suas expressões são definidas em função destas. No momento da

avaliação de uma transição, as variáveis livres das expressões precisam ser associadas a valores. Essa associação é chamada de *ligação* (*binding*) de variáveis [Maciel96][Jensen94].

Tabela 1. Comparação entre rede *Place/Transition* e colorida

	Place/Transition	**Colorida**
Habilitação da transição	Lugares da pré-condição devem conter pelo menos a quantidade de *tokens* definida pela soma dos pesos dos arcos.	Lugares da pré-condição devem conter o conjunto de *tokens* formado pela soma dos resultados das expressões dos arcos.
Guarda da transição	Não há.	A transição é habilitada apenas se a guarda definida pela expressão de guarda for satisfeita.
Disparo da transição	Quantidade de *tokens* removidos e adicionados é definida pelos pesos dos arcos.	*Tokens* que são removidos e adicionados são aqueles com valores resultantes das expressões dos arcos.

A Figura 17 exemplifica uma situação em que há variáveis livres.

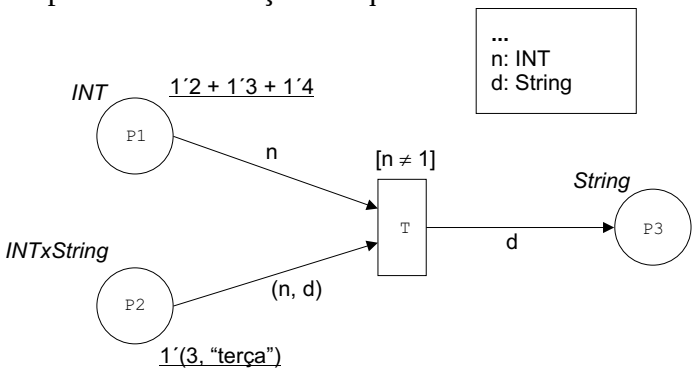


Figura 17. Rede de Petri colorida com variáveis livres nas expressões dos arcos.

Conforme pode ser observado na Figura 17, o lugar *P1* possui tipo inteiro e contém 3 *tokens*, com os valores 2, 3 e 4. O tipo do lugar *P2* é um produto cartesiano de inteiro e string. Ele possui apenas um *token* com a tupla (3, "terça"). As variáveis *n* e *d* nas expressões são variáveis livres. Para que possamos avaliar a condição da transição, precisamos atribuir valores a essas variáveis. Esta atribuição é chamada de *ligação*. Precisam procurar valores para *n* e *d* tais que tornem a transição *T* habilitada. Para a marcação inicial desta rede, a única ligação que permite isto é <*n* = 3, *d* = "terça">. Pode-se notar que com esta ligação, existem *tokens* nas pré-condições com valores iguais aos valores resultantes da avaliação das expressões dos arcos de entrada da transição (neste caso o valor 3 para *P1* e o valor (3, "terça") para *P2*. Além disso, a expressão guarda da transição é satisfeita, pois $n \neq 1$. Logo, a transição está habilitada na ligação fornecida. Diz-se que *a ligação <n = 3, d = "terça"> habilita a transição T.*

O disparo da transição *T* sob esta ligação irá fazer com que o *token* de valor 3 seja removido de *P1* e o *token* de valor (3, "terça") seja removido de *P2* e então, conforme a expressão do arco de saída, um *token* de valor "terça" seja adicionado ao lugar *P3*.

Caso existissem outras ligações para as variáveis *n* e *d* que permitissem o disparo da transição *T*, qualquer uma delas poderia ser escolhida, de forma não-determinística, para o disparo da transição.

Apesar do exemplo da Figura 17 ser simples, a tarefa de atribuir valores às variáveis livres é mais complexa do que parece. Isso será discutido na Seção 4.3.

4.2 Conceitos

A seguir são apresentados os conceitos formais das redes de Petri coloridas. As definições apresentadas levam em consideração que os *tokens* não possuem apenas "cores", mas representam valores em alguma linguagem de notação de alto-nível. A definição fornecida por [Jensen94] apresenta esta linguagem como um conjunto de tipos $\sum$ ao qual pertecem todos os possíveis valores, operadores e expressões da linguagem. Esta definição é bastante geral e permite que inúmeras linguagens sejam mapeadas na definição fornecida. O autor apresenta também um conjunto de funções auxiliares com as quais define a rede de Petri colorida. A definição de rede colorida fornecida aqui é uma adaptação das apresentadas em [Jensen94] e em [Maciel96].

Para simplificar a definição das redes de Petri coloridas, uma definição não-matricial das redes de Petri é utilizada [Maciel96]:

DEFINIÇÃO 4.2.1 Uma rede de Petri é uma tripla

$$R = (P, T, A)$$

Onde,

P é um conjunto não-vazio, finito de lugares;
T é um conjunto não-vazio, finito de transições;
$A \subseteq ((P \times T) \cup (T \times P))$ é o conjunto de arcos.

■

A definição de linguagem de notação que será utilizada aqui é uma simplificação que permite uma visualização mais clara do papel da linguagem na prática ao mapear expressões em tipos e em valores constantes.

DEFINIÇÃO 4.2.2 Uma **linguagem de notação** é uma 6–tupla

$$\Lambda = (Exp, Cons, \daleth, Type, Par, Eval)$$

Onde,

Exp é o conjunto de todas as expressões possíveis na linguagem;
Cons é o conjunto de todas as constantes presentes na linguagem;
$\daleth$ é o conjunto de todos os tipos de dados da linguagem;
$Type : Cons \rightarrow \daleth$ é a função que associa a cada constante $c \in Cons$ um tipo $t \in \daleth$;
$Par : Exp \rightarrow \{ v1, v2, ..., vn \}$ é a função de parâmetros, que associa a cada expressão um conjunto de variáveis que corresponde aos parâmetros daquela expressão;
$Eval : Exp \rightarrow Cons$ é a função de avaliação de expressões, que associa a cada expressão o valor do resultado de sua avaliação;

■

Esta definição está contida na definição de [Jensen94], uma vez que pode-se considerar os conjuntos *Exp* e *Cons* como sendo conjuntos de elementos que pertencem ao conjunto de tipos $\sum$ definido pelo autor para a linguagem. Desta forma, a definição apresentada aqui pode ser considerada apenas mais restritiva. Na prática, dificilmente será criado um modelo no qual os *tokens* não contenham valores constantes e, como conseqüência, as expressões utilizadas na rede sejam sempre avaliadas em valores constantes. A definição fornecida aqui tira proveito destas questões práticas.

Uma crítica à definição de [Jensen94] é a existência de uma função universal *Type* que pode mapear uma variável qualquer *v* em um *tipo* na linguagem. Na verdade, as variáveis são entidades particulares de cada modelo, sendo declaradas na declaração da rede correspondente. Por este motivo, na definição para redes coloridas apresentada aqui será incluído o elemento *declaração da rede*, permitindo que variáveis sejam criadas e tipos sejam associados a elas, tendo esta associação escopo apenas dentro da definição do modelo.

DEFINIÇÃO 4.2.3 Uma **rede de Petri colorida** é uma 8-tupla

$$RC = (R, \sum, \Lambda, C, G, E, In, D)$$

Onde,

$R = (P, T, A)$ é uma rede de Petri;

$\sum$ é o conjunto finito não-vazio de *cores*;

$\Lambda = (Exp, Cons, \digamma, Type, Par, Eval)$ representa a linguagem de notação da rede;

$C : P \rightarrow \sum$ é a função que associa cada lugar a uma *cor*;

$G : T \rightarrow Exp$ é a função de guarda, que associa a cada transição uma expressão *exp*, tal que $Eval(exp)$ pertence a um tipo booleano da linguagem, para todo $t \in T$;

$E : A \rightarrow Exp$ é a função que associa a cada arco da rede uma expressão *exp* tal que, para todo arco (p, t) ou $(t, p) \in A$, $Eval(exp)$ corresponde a um *bag* de elementos com tipo igual a $D(C(p))$;

$In : P \rightarrow Exp$ é a função de inicialização, que associa a cada lugar uma expressão de inicialização *exp*, tal que $Eval(exp)$ corresponde a um *bag* de elementos com tipo igual a $D(C(p))$ e $Par(exp) = \emptyset$, ou seja, não há variáveis na expressão;

$D : (\sum \rightarrow \digamma) \cup (V \rightarrow \sum)$ é a função de declaração da rede, que contém a definição de cores e variáveis e associa a cada *cor* um tipo na linguagem e a cada variável $v \in V$ uma *cor*.

■

O leitor que consultar [Maciel96] irá perceber que a definição apresentada pelo autor é a mesma apresentada aqui, com a adaptação da formalização da linguagem de notação. Na verdade, o autor utiliza funções auxiliares (*Tipo* e *Var*) que podem ser facilmente mapeadas nas funções fornecidas na Definição 4.2.2 e na Definição 4.2.3. A correspondência a seguir demonstra este mapeamento:

- $D(\ Tipo(expr)\ ) = Type(Eval(expr))$
- $Tipo(v) = D(v)$
- $Var(expr) = Par(expr)$

Uma marcação nas redes de Petri coloridas atribui não simplesmente um número de *tokens* aos lugares mas um *bag* dos valores que os *tokens* contidos em um lugar representam. Um *bag* é uma extensão de conjuntos que admite que elementos se repitam dentro do conjunto. A utilização de *bags* é necessária pois um lugar pode conter diversos *tokens* com os mesmos valores.

DEFINIÇÃO 4.2.4 A **marcação** ou **distribuição de marcas** de uma rede de Petri colorida é uma função $M : P \rightarrow bag\ D(\sum)$, onde $D(\sum)$ é o conjunto de tipos correspondentes ao conjunto $\sum$ de *cores* da rede, conforme a declaração D da rede.

■

Cada lugar pode receber como marcação apenas um *bag* de elementos do tipo associado à sua cor.

A avaliação das transições nas redes coloridas é um processo mais complexo do que os das classes de redes anteriormente apresentadas. No momento da avaliação das transições, é necessário fazer a ligação de valores às variáveis livres associadas à transição. Estas variáveis são aquelas presentes na expressão de guarda da transição e aquelas presentes em todos os arcos que se conectam a esta transição [Maciel96]. Denota-se o conjunto das variáveis associadas a uma transição t por $Var(t)$. Uma variável v pertence a $Var(t)$ se $v \in Par(G(t))$ ou se $v \in Par(E(a))$, $\forall a \in A^t$, onde $A^t \subseteq A$ é o conjunto de todos os arcos que chegam ou partem da transição t.

Quando ligamos todas as variáveis livres de uma transição à valores, obtemos um conjunto de ligações, dado por [Maciel96]:

DEFINIÇÃO 4.2.5 Um **conjunto de ligações** de uma transição t é denotado por

$$B(t) = < v_1 = c_1, ..., v_n = c_n >$$

Onde,

$v_i \in Var(t)$;
c_i é o valor ligado à variável v_i.

■

A avaliação de uma expressão *exp* onde as variáveis livres foram ligadas de acordo com o conjunto de ligações B é denotado por $Eval(exp)< B(t) >$.

Uma vez que foi obtido um conjunto de ligações para uma transição, podemos verificar se ela está ou não habilitada.

DEFINIÇÃO 4.2.6 Uma transição t está habilitada numa marcação M se, e somente se, existe um conjunto de ligações $B(t)$ tal que $Eval(\ G(t)\ )<B(t)>$ = true e $Eval(\ E(p, t)\ )<B(t)> \subseteq M(p)$, $\forall p \in P \mid (p, t) \in A$.

■

O par $(t, B(t))$ é chamado de *passo* [Maciel96] ou **elemento de ligação** (*binding element*) [Jensen94]. Se numa certa marcação uma transição t está habilitada sob uma ligação $B(t)$, então diz-se que *o elemento de ligação (t, B(t)) está habilitado* naquela marcação. Este elemento de ligação pode *ocorrer*, levando a rede a uma nova marcação.

A escolha de qual elemento de ligação irá ocorrer em uma certa marcação é não-determinística. A ocorrência de um elemento é definida como:

DEFINIÇÃO 4.2.7 A **ocorrência de um elemento de ligação** ou **ocorrência de um passo** (*t*, *B(t)*) em uma marcação *M* leva a rede a uma marcação *M'* tal que

$$M'(p) = M(p) - \sum_{(p, t) \in A} Eval(\, E(p, t)\,)\langle B(t)\rangle + \sum_{(t, p) \in A} Eval(\, E(t, p)\,)\langle B(t)\rangle, \forall p \in P \quad (4.1)$$

■

A soma e a subtração ocorrem da forma convencional já definida pela teoria de *bags*.

4.3 Ligação de Variáveis

Pode-se notar que as redes de Petri coloridas são muito mais complexas que as outras classes de redes de Petri apresentadas anteriormente. Esta complexidade se reflete na forma de simulação dessas redes.

A simulação das redes coloridas incluem duas tarefas que não estão diretamente ligadas ao modelo da rede de Petri, mas sim à sua linguagem de programação:

- **Interpretação das notações**
- **Ligação de variáveis livres**

A interpretação das notações é necessária para a realização dos cálculos das expressões, assim como também para extrair o conjunto de variáveis livres presentes nestas e para fazer a verificação de tipos de lugares.

Uma vez que as variáveis livres de todas as expressões envolvendo uma transição tenham sido encontradas, o simulador precisa atribuir-lhes valores, realizando as ligações. Se existirem valores tais que, atribuídos às variáveis livres, permitam que a transição seja habilitada, então o simulador precisa encontrá-las.

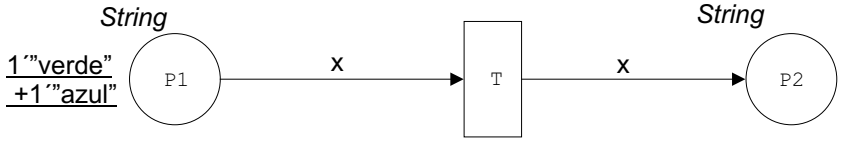


Figura 18. Caso simples de ligação

Esta tarefa em alguns casos pode ser simples. Na Figura 18, observa-se que basta ao simulador atribuir à variável *x* um dos valores correspondentes aos *tokens* no lugar *P1*, que são "verde" e "azul". A expressão equivale à função *identidade* e, portanto, qualquer valor escolhido para *x* irá retornar ele mesmo como resultado da avaliação da expressão. Esta, entretanto, é a situação mais simples.

A Figura 19 representa um caso bem mais complexo. Uma função *toWeekDay* é utilizada para converter um número inteiro no intervalo de 1 a 7 em uma *string*, correspondente ao nome do dia da semana ("domingo" para 1, "segunda" para 2 e assim por diante). Um certo evento, representado por uma *string*, é associado a um número no lugar *Eventos*, formando um par (evento, número). As variáveis *e* e *n* da expressão (*e*, *n*) precisam, portanto, ser ligadas a um

evento e um número, respectivamente. No lugar *DiaLivre*, os *tokens* são *strings* que representam os dias da semana. A expressão para o arco ligado a este lugar utiliza a função *toWeekDay* para remover o *token* que representa o dia correspondente ao valor da variável *n*.

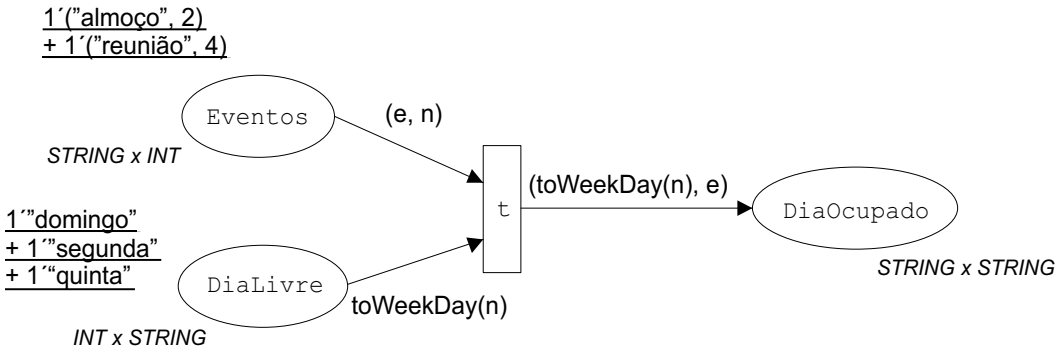


Figura 19. Exemplo de ligação complexa

Uma vez que o simulador tenha verificado a existência das duas variáveis livres, *e* e *n*, como ele poderá encontrar os valores corretos para que a transição *t* seja habilitada (que seriam *e* = “almoço” e *n* = 2)?

Uma abordagem possível é testar todos os valores dentro dos conjuntos a que pertencem as variáveis (seus tipos). Por exemplo, no caso da variável *n*, testar todos os números inteiros possíveis. Mas, uma vez que os valores possíveis para as variáveis estão compreendidos em conjuntos infinitos (inteiros e *strings*), esta abordagem se torna impraticável (no caso exemplificado, os conjuntos poderiam ser restringidos, pois sabe-se que apenas inteiros de 1 a 7 são usados, e que os dias da semana também correspondem a apenas sete *strings* possíveis. Entretanto, em outras situações isto poderia não ocorrer). É isto o que é feito, por exemplo, pelas ferramentas *Design/CPN* [Meta93] e sua sucessora *CPN Tools*. Para conjuntos de tamanho limitado (no máximo 100 elementos), a ferramenta testa todas as possibilidades. Para conjuntos maiores, a ferramenta não permite que expressões complexas sejam utilizadas, emitindo uma mensagem de erro informando que “a ligação é muito complexa”.

Esta abordagem não é eficiente e restringe o modelo que pode ser simulado.

Durante a implementação dos simuladores neste trabalho, várias alternativas foram cogitadas, mas todas apresentavam falhas e foram descartadas. Além disso, qualquer que fosse o algoritmo utilizado para resolver este problema, havia a necessidade de implementar um analisador sintático completo para a linguagem *Java*, que foi a linguagem de notação escolhida para a biblioteca. Isto faria com que o trabalho tomasse dimensões muito maiores do que o escopo proposto.

A solução escolhida foi deixar esta tarefa para o usuário. O usuário deverá fornecer, na definição da rede, o código responsável pela ligação. Como vantagem ele terá um código específico e otimizado para o seu problema. A desvantagem é que isto aumentará o seu trabalho ao criar o modelo, tendo que se preocupar com detalhes de programação.

Para minimizar isso, as situações mais comuns encontradas poderiam receber uma implementação genérica no simulador, cabendo ao usuário apenas identificar a situação e informa-la ao simulador. Um exemplo é o caso da ligação trivial, ilustrado na rede da Figura 18.

Esta solução simplifica a implementação do simulador pois dispensa a implementação de um analisador sintático, ou qualquer fase de interpretação da linguagem por parte do simulador. Nos Capítulo 5 e 7 será explicado como isto acontece.

No futuro, a implementação de um analisador sintático pode fornecer meios para que outras soluções mais eficientes possam ser implementadas.

Capítulo 5

Formatos de Representação

No ano 2000, durante a Conferência Internacional sobre Aplicação e Teoria das Redes de Petri (*International Conference on Application and Theory of Petri Nets*) em Aarhus, Dinamarca, se iniciou um esforço em padronizar o formato de representação das redes de Petri. O motivo desta iniciativa era a grande diversidade de ferramentas disponíveis para redes de Petri e a total ausência de um formato intercambiável que pudesse ser compreendido por elas. Como cada ferramenta apresenta seu próprio conjunto de funcionalidades, a possibilidade de utilizar um único formato de representação que pudesse ser compreendido pelas principais delas era desejada pela comunidade de pesquisadores [Billington03].

Durante esta conferência, diversas propostas de formatos baseados em XML (*eXtensible Markup Language*) [WG04] foram apresentadas. Dentre elas, um formato que recebeu o nome de *Petri Net Markup Language* (PNML). Este formato foi utilizado como o padrão compreendido pela biblioteca implementada.

Este capítulo apresentará as características básicas da linguagem PNML, assim como as extensões que foram adicionadas para representar as redes temporizada e colorida na biblioteca. A rede colorida será discutida mais extensamente, pois é aquela que apresenta mais adições em relação ao PNML convencional, incluindo-se aí a linguagem de notação.

5.1 Características Básicas da PNML

Três princípios foram levados em consideração na definição do padrão PNML pelos seus autores [Weber02]:

- **Legibilidade**. O formato seria humanamente legível e editável por um editor de textos convencional.
- **Universalidade**. O formato não deveria excluir nenhuma classe de redes de Petri, sendo possível representar qualquer versão com qualquer extensão desenvolvida.
- **Mutualidade**. O formato deveria permitir que o maior número de informações sobre a rede fossem extraídos, mesmo nos casos em que o tipo da rede não é conhecido.

O quesito da legibilidade foi atendido a partir do uso de XML como base. A linguagem XML é uma linguagem de marcação para definição de documentos estruturados. Ela é simples, seu formato é baseado em texto, não havendo, portanto, necessidade de processamento binário

para definição do conteúdo de um documento. Além de poder ser utilizada com facilidade para descrever as redes de Petri, a linguagem XML é amplamente difundida e possui um número incontável de recursos, dentre ferramentas e especificações, para suportar a sua utilização.

A universalidade foi garantida com a inclusão de *tags* especiais que são utilizadas para adicionar informações extras à cada objeto da rede. A definição do PNML permite que estas *tags* sejam personalizadas para representar informações específicas do tipo de rede do usuário.

A mutualidade é obtida pela utilização de um conjunto padrão de *tags* e atributos destinados a representação das informações que são comuns a todas as redes de Petri, como os conjuntos de lugares, transições e arcos. Desta maneira, mesmo que o tipo da rede seja desconhecido, todas as informações que seguem o padrão ainda poderão ser extraídas do XML.

A seguir serão apresentados os elementos básicos que compõem um **arquivo PNML**.

5.1.1 Rede

Um único arquivo PNML pode conter a descrição de várias redes de Petri. Cada rede de Petri possui um identificador único dentro do arquivo e um tipo como atributos. O tipo fornecido como atributo deve ser um URI (*Uniform Resource Identifier*) para o arquivo de definição do tipo correspondente (ver Sec. 5.1.7). Através desta URI o arquivo pode ser verificado corretamente, seguindo-se as instruções definidas para aquele tipo específico.

A Listagem 1 exemplifica a estrutura de um PNML com duas redes, chamadas de “net1” e “net2”.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<pnml>
 <net id=”net1”
   type=”http://www2.informatik.hu-berlin.de/top/pnml/pntd/ptNetb.pntd”>

 </net>
 <net id=”net2”
   type=” http://www2.informatik.hu-berlin.de/top/pnml/pntd/ptNetb.pntd”>

 </net>
</pnml>
```

Listagem 1. Duas redes representadas em PNML.

As *tags* `<pnml>` e `</pnml>` delimitam o conteúdo do arquivo PNML. Cada rede dentro do arquivo é delimitada pelas *tags* `<net>` e `</net>`.

Uma rede contém vários objetos que representam a sua estrutura: seus lugares, transições, arcos e quaisquer outros objetos relacionados a algum tipo específico de rede de Petri. Cada objeto possui um identificador único que o referencia dentro da rede.

5.1.2 Lugares

Um lugar é definido em PNML pela *tag* `<place>`, que pode estar em qualquer lugar dentro da definição da rede. A Listagem 2 demonstra uma declaração de lugar típica. Este lugar tem “p1” como seu identificador, está disposto na posição (x=10, y=10) do ambiente gráfico, possui um rótulo “Place1” colocado na posição (x=5, y=8) e possui um token como marcação inicial.

As informações gráficas de qualquer objeto em PNML são fornecidas pelo elemento `graphics`. Seu conteúdo varia de acordo com o objeto em que está contido. Em geral podem ser definidos posição, tamanho e cores. Na Listagem 2 é definida a posição do lugar através da *tag* `<position>`. Existem outros atributos gráficos que podem ser utilizados, o leitor pode consultar a

especificação da linguagem, encontrada em [Weber06], para obter uma lista completa destes e outros elementos que não serão apresentados nesta monografia.

O nome é também uma informação comum a todos os objetos. Ele é utilizado para definir rótulos, que podem ser apresentados graficamente. Este rótulo não precisa necessariamente corresponder ao identificador do objeto. Uma vez que ele também é um objeto gráfico, tem seus próprios atributos gráficos definidos pela *tag* `<graphics>`.

```
<place id="p1">
 <graphics>
        <position x="10" y="10" />
 </graphics>
 <name>
        <text>Place1</text>
        <graphics>
                <offset x="-5" y="-2" />
        </graphics>
 </name>
 <initialMarking>
        <text>1</text>
 </initialMarking>
</place>
```

Listagem 2. Exemplo de descrição de lugar.

A marcação inicial do lugar é definida com a utilização da *tag* `<initialMarking>`. Visto como esta marcação pode ser uma expressão de uma rede colorida, ela é definida como um elemento textual.

5.1.3 Transições

As transições são identificadas pela *tag* `<transition>`. A Listagem 3 demonstra a definição de uma transição acompanhada de um rótulo "t".

```
<transition id="t1">
 <graphics>
        <position x="30" y="10" />
        <dimension x="8" y="2" />
 </graphics>
 <name>
        <text>t</text>
        <graphics>
                <offset x="-5" y="-2" />
        </graphics>
 </name>
</transition>
```

Listagem 3. Exemplo de descrição de transição.

5.1.4 Arcos

Os arcos são definidos através de um identificador e dos seus elementos de origem e destino, utilizando-se a *tag* `<arc>`. A Listagem 4 mostra um exemplo de arco.

Pode-se definir um conjunto de pontos pelos quais o desenho do arco deve passar. Na Listagem 4 pode-se observar que no elemento `graphics` foram definidos dois pontos. Estes são pontos que se deseja que a linha do arco intersecte.

Outro atributo importante de um arco é a sua inscrição, que pode definir o seu peso (como em redes *Place/Transition*) ou sua expressão (como nas redes coloridas). Para oferecer maior flexibilidade, a inscrição é definida como um elemento de texto, utilizando-se por isso a *tag* `text`.

```
<arc id="a1" source="p1" target="t1">
 <graphics>
        <position x="20" y="10" />
        <position x="25" y="10" />
 </graphics>
 <inscription>
        <text>2</text>
        <graphics>
                <offset x="5" y="-2" />
        </graphics>
 </inscription>
</arc>
```

Listagem 4. Exemplo de descrição de arco

5.1.5 A Tag toolspecific

Para representar informações especiais, que são dependentes de uma certa ferramenta, é utilizada a *tag* `<toolspecific>`. Seu papel é delimitar dentro da definição da rede as especificações que são destinadas a apenas uma ferramenta, pois somenta ela pode compreender o seu significado. O conteúdo de um elemento `toolspecific` é definido pela ferramenta correspondente, conforme suas necessidades. Um exemplo de conteúdo específico é apresentado na Listagem 5.

```
<transition id="t1">
 <name>
        <text>checkup</text>
        <graphics>
                <offset x="-5" y="-2" />
        </graphics>
 </name>
 <toolspecific tool="example" version="1.0">
        <action type="human" cost="3" />
 </toolspecific>
</transition>
```

Listagem 5. Exemplo de utilização da *tag* toolspecific

No exemplo da Listagem 5, uma ferramenta hipotética chamada “example” associa a uma transição informações sobre a ação que ela representa. Neste caso, a transição representa uma ação executada por um agente humano, e que tem um certo custo associado igual a 3. Observe que esta informação é associada à rede de Petri mas não é de interesse de outras ferramentas. Apenas a ferramenta “example” pode compreender o seu significado.

O exemplo demonstra uma situação em que a informação pode não ser relevante para a correta interpretação da rede de Petri. Entretanto, em certos casos a informação pode ser essencial para o modelo representado, mas, uma vez que apenas uma ferramenta é capaz de lidar com ela, não há uma convenção a ser seguida. Nestes casos a *tag* `toolspecific` também precisa ser utilizada.

Na biblioteca implementada, tanto as informações de intervalos de tempo para as redes de Petri temporizadas quanto as notações das redes de Petri coloridas foram representadas com a utilização de elementos `toolspecific`.

5.1.6 Módulos e Páginas

Uma questão importante que foi levada em consideração pelos autores da linguagem foi quanto a estruturação de redes muito grandes. Diversas ferramentas possuem alternativas para facilitar a visualização destas redes. Um exemplo é a divisão da rede em *páginas*, que fornece uma nova dimensão para a disposição dos objetos (além de estarem dispostos horizontal e verticalmente, os objetos podem ser distribuídos em páginas). Este mecanismo é bastante utilizado em diversas ferramentas. A linguagem PNML provê suporte tanto a esta alternativa quanto a outra menos comum: a definição e instancação de módulos. Utilizando o conceito de módulos, uma rede pode ser definida como um objeto de alto-nível, que pode ser instanciado uma ou várias vezes dentro de outra rede.

A biblioteca implementada não suporta a utilização de módulos ou páginas. O leitor interessado neste assunto pode consultar [Weber02] e [Billington03] para mais informações.

5.1.7 Petri Net Type Definition (PNTD)

A verificação sintática da estrutura de um arquivo XML é feita através do uso de documentos de definição, como *Document Type Definitions* (DTD) e XML *schemas*. Ambos têm o mesmo princípio, que é fornecer uma descrição dos elementos que podem estar presentes em um arquivo XML e de seus formatos. Através deles um analisador sintático pode verificar se um certo arquivo XML está estruturalmente bem formado de acordo com as regras definidas.

A linguagem PNML permite a utilização de *schemas* para descrever a sintaxe correta para cada tipo de rede de Petri. Estes *schemas* são chamados de *Petri Net Type Definitions* (PNTD).

A Listagem 6 demonstra um arquivo PNTD que descreve o tipo de rede *Place/Transition*. A linguagem utilizada é *RELAX NG* [Oasis01].

```
<grammar ns="http://www.example.org/pnml"
      xmlns="http://relaxng.org/ns/structure/1.0"
      xmlns:conv="http://www.informatik.hu-berlin.de/top/pnml/conv">

 <include href="http://www.informatik.hu-berlin.de/top/pnml/pnml.rng"/>
 <include href="http://www.informatik.hu-berlin.de/top/pnml/conv.rng"/>
 <define name="NetType" combine="replace">
      <text>http://www.example.org/pnml/PTNet</text>
 </define>
 <define name="Net" combine="interleave">
       <optional><ref name="conv:Name"/></optional>
 </define>
 <define name="Place" combine="interleave">
      <interleave>
       <optional><ref name="conv:PTMarking"/></optional>
       <optional><ref name="conv:Name"/></optional>
      </interleave>
 </define>
 <define name="Arc" combine="interleave">
      <optional><ref name="conv:PTArcInscription"/></optional>
 </define>
</grammar>
```

Listagem 6. Exemplo de arquivo PNTD para uma rede *Place/Transition*

Apesar de o padrão PNML já ser bastante difundido, novas PNTDs ainda não foram publicadas na página oficial da linguagem [Weber06]. Apenas as redes *Place/Transition* foram especificadas. Devido a esta ausência de um formato padrão para representar as redes

temporizada e colorida, optou-se por utilizar o modelo *Place/Transition* nesta biblioteca, que já possui especificação publicada, extendido pela utilização de elementos *toolspecific*.

5.1.8 Um Exemplo de PNML

A Listagem 7 contém a descrição em PNML da rede apresentada na Figura 20. As informações gráficas não estão presentes para evitar que o exemplo se tornasse extenso demais. Os rótulos associados aos lugares e transições são diferentes dos seus identificadores propositalmente, para demonstrar que são totalmente independentes um do outro.

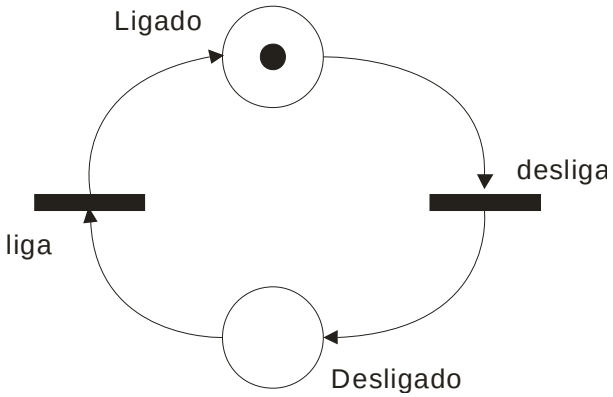


Figura 20. Aparência da rede que será descrita em PNML

```
<pnml>
  <net id="exemplo">
        <place id="p1">
              <name><text>Ligado</text></name>
              <initialMarking><text>1</text></initialMarking>
        </place>
        <place id="p2">
              <name><text>Desligado</text></name>
        </place>
        <transition id="t1">
              <name><text>desliga</text></name>
        </transition>
        <transition id="t2">
              <name><text>liga</text></name>
        </transition>
        <arc id="a11" source="p1" target="t1" />
        <arc id="a12" source="t1" target="p2" />
        <arc id="a22" source="p2" target="t2" />
        <arc id="a21" source="t2" target="p1" />
  </net>
</pnml>
```

Listagem 7. Descrição em PNML da rede exemplificada

5.2 Representação das Redes Temporizadas

A biblioteca *jPetriSim* compreende as informações temporais a partir de um elemento *toolspecific* adicionado às transições, que deverá conter os valores mínimo e máximo do intervalo de disparo destas.

O formato deve seguir o seguinte padrão:

```
<toolspecific tool="jpetrisim" version="1.0">
       <time  min="VALOR_MIN" max="VALOR_MAX" />
</toolspecific>
```

Onde *VALOR_MIN* e *VALOR_MAX* devem ser substituídos pelos tempos mínimo e máximo de disparo da transição.

Esta é a única informação adicional necessária para que o simulador possa processar uma transição temporizada. Quando os valores mínimo e máximo forem iguais a zero, o simulador interpretará a transição como uma transição imediata. Como representação do tempo infinito, o valor -1 é aceito.

A utilização do elemento *toolspecific* é necessária pois não foi criada uma PNTD para especificar o padrão utilizado. Caso o tipo fosse definido por uma PNTD, não seria necessária a utilização desta *tag*. O mesmo ocorre com as redes coloridas, como será mostrado na Seção 5.3.

5.3 CPNJava

As redes de Petri coloridas com Java anotado foram batizadas de *CPNJava*.

Para que a linguagem Java pudesse ser utilizada como notação, uma série de regras foi definida. Não apenas o formato do PNML, mas também a maneira de codificar cada expressão, de forma a garantir o uso dos variados recursos disponíveis para a linguagem Java, mas mantendo as limitações impostas pelo modelo das redes de Petri coloridas.

5.3.1 Declarações

A seção de declarações da rede de Petri colorida é uma propriedade da rede como um todo. Nela poderão ser declaradas as cores, variáveis e também métodos que poderão ser utilizados nas expressões ao longo da rede.

A notação de declaração é expressa em um elemento `enviroment`. O formato de sua declaração deve obedecer o padrão a seguir:

```
<net id="ID"
  type=" http://www2.informatik.hu-berlin.de/top/pnml/pntd/ptNetb.pntd">

 <toolspecific tool="jpetrisim" version="1.0">
  <enviroment>
       <declarations>

        DECLARAÇÕES

       </declarations>
       <initialization>

        CÓDIGO DE INICIALIZAÇÃO

       </initialization>
  </enviroment>
 </toolspecific>

...

</net>
```

Nas redes *CPNJava*, as cores são classes. Assim, qualquer classe da API Java pode ser definida como cor de um lugar. Caso seja necessária a criação de estruturas para representar um certo tipo de dado não presente na API, o usuário poderá definí-las como subclasses, no espaço de declarações do elemento `enviroment`, compreendido entre as *tags* `<declarations>` e

`</declarations>`. Neste espaço também poderão ser definidos métodos e variáveis globais. Nas redes coloridas apresentadas por [Jensen94], é preciso que sejam declaradas as variáveis que serão utilizadas nas expressões dos arcos e transições. No caso da *CPNJava*, estas variáveis não serão declaradas neste momento. Aqui, entretanto, podem ser declaradas variáveis com valores inicializados (ou instanciados), de forma que possam ser utilizadas como constantes ao longo da rede, tornando mais clara a leitura do código.

A inicialização é opcional, e poderá conter qualquer código necessário para inicializar as variáveis declaradas na seção de declarações. Isto é necessário pois, em *Java*, a inicialização de algumas variáveis pode ser complexa demais para ser feita diretamente na sua declaração (por exemplo, para preencher um objeto `Hashtable` com valores será preciso várias linhas de código). Desta forma, o usuário tem a disposição um espaço para realizar operações adicionais de inicialização da rede. Este código será executado antes do início da simulação.

5.3.2 Lugares

Os lugares possuem duas informações associadas: cor e expressão de inicialização.

As expressões de inicialização dos lugares são definidas no elemento `initialMarking`, seguindo a convenção do PNML. A cor é definida por um elemento `colorSet`, inserido como *toolspecific*.

```
<place id="ID">
 <toolspecific tool="jpetrisim" version="1.0">
       <colorSet> CLASSE ASSOCIADA AO LUGAR </colorSet>
 </toolspecific>
 <initialMarking>
       <text>
             CÓDIGO DA EXPRESSÂO DE INICIALIZAÇÃO
       </text>
 </initialMarking>
</place>
```

A classe definida como cor do lugar deve ser expressa com o nome completo da classe escolhida (por exemplo, `java.util.Vector`). Em especial, o pacote `java.util` é incluído ao simulador, pois suas classes são bastante utilizadas. Desta forma, a classe `Vector` e outras classes do mesmo pacote podem ser utilizadas sem a necessidade de expressar seu nome completo.

O código da expressão de inicialização deve preencher com os valores dos *tokens* o conteúdo da variável especial "`$`", que contém uma instância da classe `br.upe.dsc.jpetrisim.Multiset`. Esta classe implementa a interface `Collection` e representa *bags* na biblioteca. Ela possui métodos especiais para serem utilizados pelo simulador. Os elementos contidos neste `Multiset` devem ser apenas instâncias da classe definida como conjunto de cores do lugar correspondente.

A Figura 21 demonstra um lugar do tipo `java.util.Vector` inicializado com um vetor de um elemento ([2]) e um vetor de três elementos ([3, 4, 5]). A Listagem 8 apresenta a descrição em PNML correspondente a este lugar. Observe que uma variável *v* é utilizada para instanciar os vetores. Esta variável deve ser declarada na seção de declarações da rede, nunca dentro da expressão do lugar.

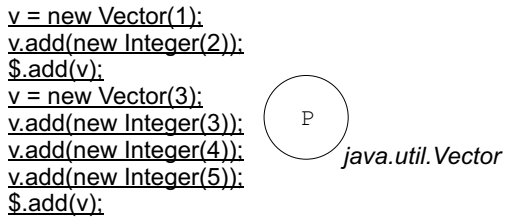


Figura 21. Exemplo de lugar numa rede *CPN Java*

```
<place id="p">
 <toolspecific tool="jpetrisim" version="1.0">
       <colorSet> java.util.Vector </colorSet>
 </toolspecific>
 <initialMarking>
       <text>
              v = new Vector(1);
              v.add(new Integer(2));
              $.add(v);

              v = new Vector(3);
              v.add(new Integer(3));
              v.add(new Integer(4));
              v.add(new Integer(5));
              $.add(v);
       </text>
 </initialMarking>
</place>
```

Listagem 8. Exemplo de representação do lugar correspondente à Figura 21.

5.3.3 Arcos

As expressões dos arcos nas redes coloridas devem retornar um *bag* com valores do mesmo tipo do lugar adjacente ao arco. No caso da rede *CPNJava*, o código do arco deve preencher o `Multiset` da variável especial "`$`" com os valores correspondentes, sendo instâncias da classe definida como conjunto de cores do lugar. É equivalente ao que ocorre com as expressões de inicialização dos lugares.

As variáveis livres não precisam ser declaradas. Quaisquer outras variáveis temporárias no código devem ser declaradas antes de utilizadas. A restrição é que não é permitido que dois arcos ligados a uma mesma transição declararem variáveis com o mesmo nome em seus códigos.

O código do arco deve ser escrito no elemento `inscription`, conforme a convenção padrão do PNML:

```
<arc id="ID" source="SOURCE_ID" target="TARGET_ID">
 <inscription>
       <text>

              CÓDIGO DO ARCO

       </text>
 </inscription>
</arc>
```

O exemplo a seguir demonstra dois arcos utilizando uma variável livre *x*. Um deles tem por objetivo remover um inteiro do lugar de origem. O outro adiciona ao lugar de destino uma instância de uma certa classe `Point` representando o ponto (*x*, *x*). Esta classe pode ser uma subclasse definida na seção de declarações da rede. A rede está representada na Figura 1 e a descrição correspondente é dada na Listagem 9.

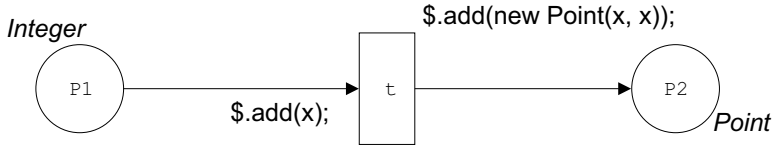


Figura 22. Exemplos de arco numa rede *CPN Java*

```
<arc id="a1" source="p1" target="t">
 <inscription>
        <text>
                $.add(x);
        </text>
 </inscription>
</arc>
<arc id="a2" source="t" target="p2">
 <inscription>
        <text>
                $.add(new Point(x, x));
        </text>
 </inscription>
</arc>
```

Listagem 9. Exemplos de arcos com variáveis livres.

5.3.4 Transições e Ligações de Variáveis

As transições são os elementos que possuem o maior número de informações associadas. Isto ocorre porque elas serão responsáveis por conter não apenas os códigos das expressões guarda mas também os códigos de declaração e ligação das variáveis livres.

A seguir é apresentada a estrutura de uma transição:

```
<transition id="ID">
 <toolspecific tool="jpetrisim" version="1.0">
        <guard>

                CÓDIGO DE GUARDA

        </guard>

        <variables>

                <var name="VAR" class="FULL_CLASSNAME" />
                ...

        </variables>

        <binding>

                CÓDIGO DAS LIGAÇÕES

        </binding>
 </toolspecific>
</transition>
```

As expressões guarda no *CPNJava* são códigos que devem atribuir à variável especial `$guard` um valor booleano. Se verdadeiro, indica que a transição poderá ser habilitada.

A declaração de variáveis livres deve conter a declaração de todas as variáveis não declaradas presentes nas expressões dos arcos ligados a esta transição. As classes devem ser especificadas utilizando-se o nome completo.

No código das ligações, o usuário é responsável por prover todos os elementos de ligação que puderem ser habilitados (ver Seção 4.2). Existem duas variáveis especiais que devem ser utilizadas por ele:

- **`br.upe.dsc.jpetrisim.BindingSet`** `thisBinding`. Guarda um conjunto de ligação (conjunto de variáveis e respectivos valores).
- **`br.upe.dsc.jpetrisim.Multiset`** `allEnabledBindings`. Deve ser preenchido com todos os conjuntos de ligação que habilitam a transição (instâncias de `BindingSet`).

Para calcular as ligações, será necessário conhecer os *tokens* presentes nos lugares adjacentes à cada arco. Esta informação pode ser obtida através de um método interno do simulador:

- **`Multiset`** `getMark(`**`String`** `placeId)`. Retorna o *Multiset* correspondente à marcação do lugar identificado por *placeId* ou `null`, caso o lugar não possua *tokens*.

O código do usuário deve preferencialmente atribuir às variáveis livres apenas valores tais que a guarda seja satisfeita. Se, por algum motivo, o usuário não puder prover apenas ligações que satisfaçam a guarda, ele poderá fornecer todas as ligações que puder calcular. Após a execução do código do usuário, o simulador verificará a condição de guarda nos elementos de ligação que foram selecionados por ele. Ele faz isso executando o código da guarda para verificar cada ligação fornecida. Aqueles elementos que não satisfizerem a guarda serão removidos do conjunto. Se nenhum elemento de ligação restar após este processamento, a transição não será habilitada.

O exemplo a seguir demonstra a definição de uma transição com uma ligação relativamente complexa. A Figura 23 apresenta a aparência da rede na qual esta transição está inserida. O lugar “Ponto” representa pontos em um certo gráfico. O lugar “Abscissa” representa as abscissas dos pontos deste gráfico e o ponto “Ordenada” as ordenadas. As expressões na transição `t` fazem com que, ao ocorrer, para cada ponto retirado do lugar “Ponto” a abscissa correspondente seja removida do lugar “Abscissa” e a a ordenada correspondente seja gerada para o lugar “Ordenada”.

Uma vez que as variáveis *x* e *y* são livres, é preciso fornecer o código de ligação para elas. Neste código o usuário deve fornecer todas as ligações possíveis.

A descrição em PNML correspondente à definição da transição `t` é fornecida na Listagem 10.

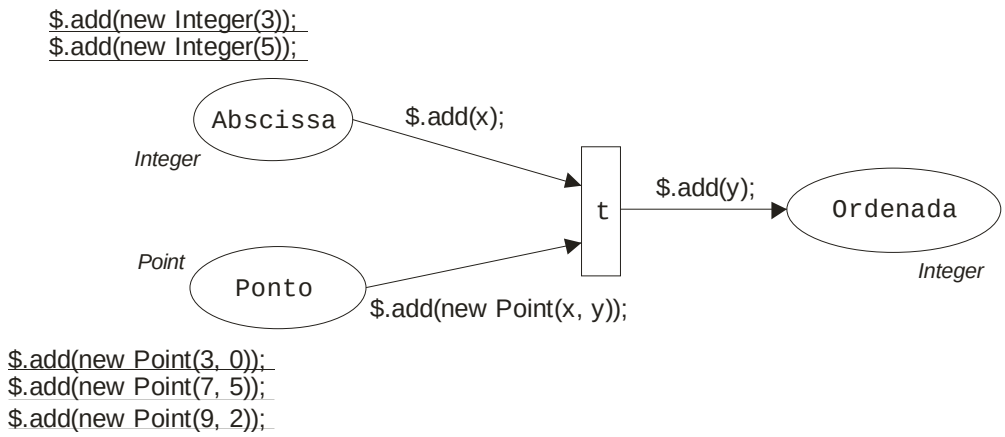


Figura 23. Exemplo de transição numa rede *CPNJava*

O algoritmo utilizado para obter os elementos de ligação é simples. Os *tokens* do lugar "Ponto" são obtidos como uma lista. Sabe-se que a restrição para que os *tokens* deste lugar possam ser selecionados é que o elemento *x* do ponto esteja presente no lugar "Abscissa". Portanto, simplesmente verifica-se para cada ponto se o valor *x* deste ponto está contido no *Multiset* correspondente à marcação do lugar "Abscissa". Se estiver, um novo *BindingSet* é criado, contendo as ligações para as variáveis *x* e *y*, que terão os valores correspondentes ao *x* e o *y* do ponto em questão. Este *BindingSet* criado é adicionado à lista de todos os elementos habilitados desta transição (`allEnabledBindings`).

```
<transition id="t">
 <toolspecific tool="jpetrisim" version="1.0">
        <guard>
                $guard = true;
        </guard>

        <variables>
                <var name="x" class="Integer" />
                <var name="y" class="Integer" />
        </variables>

        <binding>
                //obtém as marcações dos lugares
                Multiset abscissaMS = getMark("Abscissa");
                Multiset pontoMS = getMark("Ordenada");

                //obtém os valores dos tokens do lugar Ponto
                List l = pontoMS.getAsList();
                Iterator i = l.iterator();

                while (e.hasNext())
                {
                        //obtém o ponto e as coordenadas X e Y
                        Point p = (Point) e.next();
                        Integer x = new Integer(p.getX());
                        Integer y = new Integer(p.getY());

                        //se o valor de X estiver presente no lugar
                        // Abscissas, então os elementos X e Y fazem
                        // parte das ligações que habilitam t
                        if (abscissaMS.contains(x))
                        {
                                thisBinding = new BindingSet();
                                thisBinding.add("x", x);
                                thisBinding.add("y", y);
                                allEnabledBindings.add(thisBinding);
                        }
                }
        </binding>

 </toolspecific>
</transition>
```

Listagem 10. Exemplo de transição descrita em PNML

Capítulo 6

Arquitetura da Biblioteca

A biblioteca ***jPetriSim*** foi desenvolvida com o objetivo de permitir que desenvolvedores possam incluir a capacidade de simulação de redes de Petri em suas ferramentas com facilidade.

A parte central da biblioteca contém os motores de simulação, responsáveis por executar as simulações das redes de Petri *Place/Transition*, temporizada e colorida. A rede colorida funciona de uma forma um pouco diferente, pois não existe um único motor de simulação, mas sim um *gerador de motores de simulação*, que cria um código-fonte do simulador correspondente a uma rede *CPNJava* que lhe é fornecida. Este código pode ser compilado pelo usuário e utilizado junto com a biblioteca na sua aplicação. O conjunto de motores de simulação e o gerador para redes coloridas juntos são chamados de **núcleo de simulação**. Além do núcleo de simulação, a biblioteca contém classes adicionais, responsáveis pela comunicação entre ela e os aplicativos externos.

Este capítulo descreve a estrutura da biblioteca e suas principais classes. A implementação dos simuladores será apresentada no Capítulo 7.

6.1 Visão Geral

A arquitetura da biblioteca pode ser dividida em quatro camadas:

- camada de entrada;
- camada de modelo;
- camada de simulação;
- camada de comunicação.

A Camada de Entrada é a responsável por receber os arquivos PNML e processá-los para obter o modelo da rede de Petri correspondente. Este modelo é instanciado a partir das classes disponíveis na Camada de Modelo. A Camada de Simulação é aquela que contém as classes do núcleo de simulação da biblioteca. Por fim, a Camada de Comunicação é a que contém as interfaces de conexão entre a biblioteca e o ambiente externo.

A Figura 24 apresenta os principais componentes da biblioteca dispostos nestas quatro camadas.

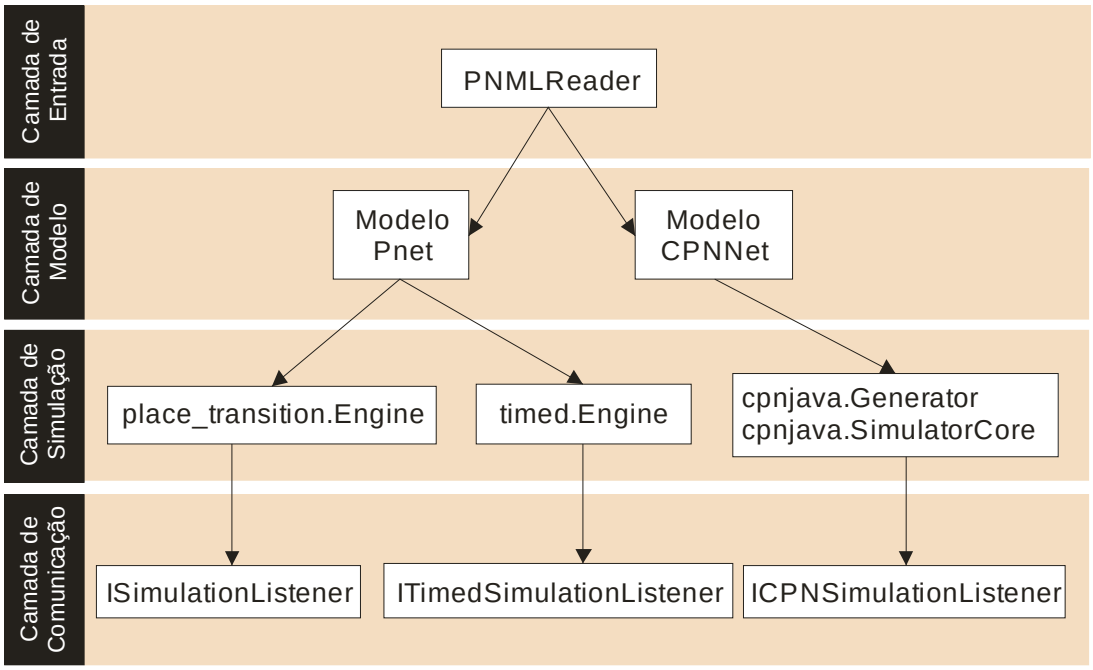


Figura 24. Visão em camadas da biblioteca ***jPetriSim*** e seus principais componentes

Cada um dos componentes ilustrados na Figura 24 será descrito na Seção 6.2, sobre pacotes e classes. As setas indicam, informalmente, o fluxo de informação dentro da biblioteca.

A biblioteca ***jPetriSim*** pode ser inserida em uma aplicação *java* facilmente. Tudo o que o usuário precisa fazer é criar uma classe que implemente uma das interfaces disponibilizadas na Camada de Comunicação, conforme o tipo de rede que deseja simular. Quando desejar realizar a simulação, ele deve instanciar o simulador correspondente, carregar a rede e registrar a sua classe como ouvidora (*listener*) da simulação. A classe ouvidora tem o poder de acompanhar e controlar o andamento da simulação. A Figura 25 ilustra a conexão entre a biblioteca e uma aplicação.

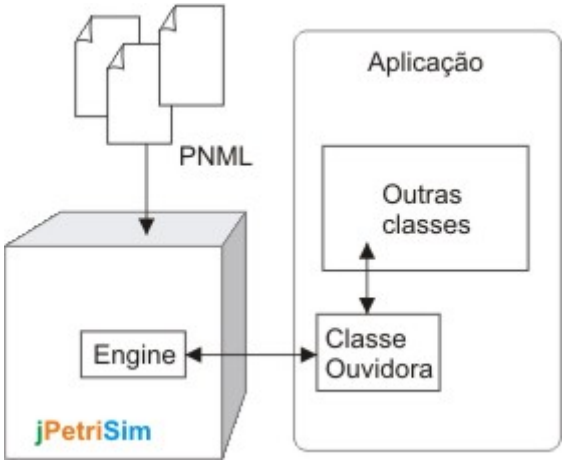


Figura 25. Conexão entre a biblioteca e uma aplicação

6.2 Pacotes e Classes

A biblioteca está contida no pacote **br.upe.dsc.jpetrisim**. A estrutura interna de pacotes é:

- **`br.upe.dsc.jpetrisim.model.petrinet`** – Contém as classes correspondentes à estrutura de dados que representa as redes *Place/Transition* e temporizada.
- **`br.upe.dsc.jpetrisim.model.cpnnet`** – Contém as classes correspondentes à representação da rede de Petri *CPNJava*.
- **`br.upe.dsc.jpetrisim.place_transition`** – Contém o motor de simulação das redes *Place/Transition* e a interface de comunicação correspondente (*listener*).
- **`br.upe.dsc.jpetrisim.timed`** – Contém o motor de simulação das redes temporizadas e as interfaces de comunicação correspondentes.
- **`br.upe.dsc.jpetrisim.cpnjava`** – Contém o gerador de simulação para redes coloridas *CPNJava*, a classe base para os simuladores e classes auxiliares.
- **`br.upe.dsc.jpetrisim.pnml`** – Contém as classes responsáveis pelo processamento de arquivos PNML.
- **`br.upe.dsc.jpetrisim.logics`** – Contém classes auxiliares que modularizam a verificação de condições de falha de execução.
- **`br.upe.dsc.jpetrisim.exception`** – Contém classes relativas a exceções da biblioteca.

A seguir, são descritas as principais classes destes pacotes. Elas estão classificadas em três categorias: Modelo, Núcleo de Simulação e Auxiliares.

6.2.1 Modelo

Este pacote contém as classes que representam as redes de Petri. As redes de Petri *Place/Transition* e temporizada são representadas pelo mesmo conjunto de classes, contidas no pacote `petrinet`. O simulador *Place/Transition* simplesmente ignora as informações temporais que estão presentes no modelo. Quatro classes descrevem uma rede de Petri, conforme pode ser observado no diagrama UML da Figura 26.

As classes correspondentes ao modelo das redes coloridas estão contidas no pacote `cpnnet`. A Figura 27 exibe o diagrama UML destas classes.

É importante ressaltar que estas estruturas não são manipuladas pela biblioteca durante a simulação. Elas são instanciadas a partir do processamento do arquivo PNML e apenas são utilizadas pelos simuladores antes da simulação para o preenchimento de suas tabelas internas. Após isso, não são mais utilizadas. O usuário pode utilizá-las em sua aplicação como seu formato de representação interno, se assim desejar.

Como será visto no Capítulo 7, os simuladores representam internamente as redes de Petri de uma forma mais otimizada.

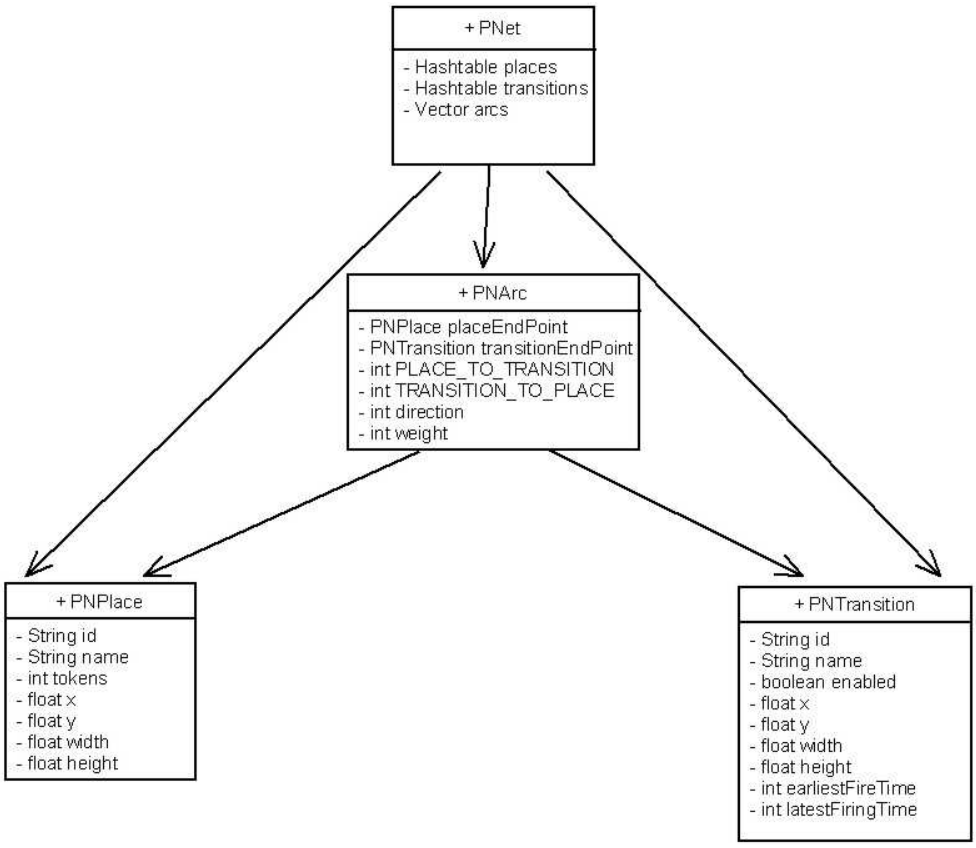


Figura 26. Diagrama UML das classes para redes *Place/Transition* e temporizada

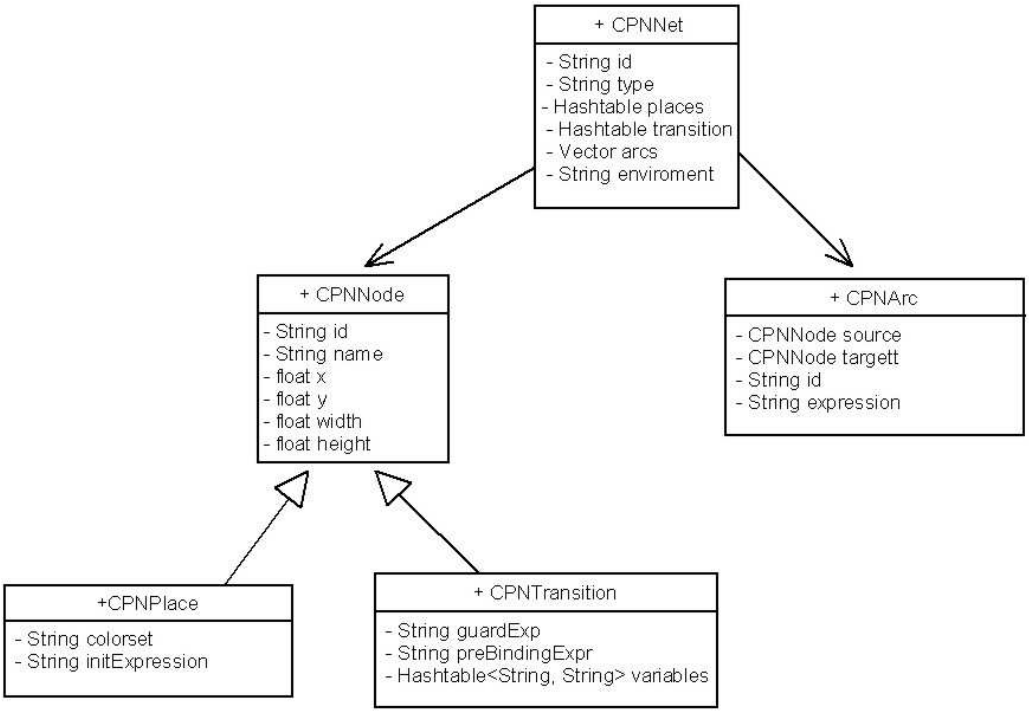


Figura 27. Diagrama UML das classes do modelo de rede colorida

6.2.2 Simuladores

A biblioteca contém dois motores de simulação prontos: O simulador de redes *Place/Transition*, que é implementado na classe `br.upe.dsc.jpetrisim.place_transition.`**`Engine`**, e o simulador de redes temporizadas, que é implementado pela classe `br.upe.dsc.jpetrisim.timed.`**`Engine`**. Cada uma delas provê uma interface que deve ser implementada pelas classes ouvidoras. Essas interfaces estão declaradas em `br.upe.dsc.jpetrisim.place_transition.`**`ISimulationListener`**, para redes *Place/Transition*, e em `br.upe.dsc.jpetrisim.timed.`**`ITimedSimulationListener`**, para as redes temporizadas.

Para as redes coloridas não existe um simulador geral, mas um gerador de código. Este gerador cria o código para um motor de simulação específico para a rede que deve ser simulada. O motivo da opção por criar um gerador é que cada arco da rede colorida contém expressões que precisam ser calculadas durante a simulação. Caso o simulador fosse geral, seria necessário que ele mantivesse todas as expressões para a rede simulada na memória e que fizesse a análise sintática das mesmas durante a execução da simulação para interpretá-las. Isto tornaria a implementação muito mais difícil e o desempenho do simulador provavelmente muito abaixo do esperado. Através da geração, é possível otimizar a simulação e é eliminada a necessidade de análise sintática das expressões durante a simulação, tornando assim o simulador infinitamente mais eficiente.

As redes de Petri coloridas *CPNJava* possuem uma implementação mais complexa. As seguintes classes estão contidas no pacote `br.upe.dsc.jpetrisim.cpnjava`:

- **`Generator`** – Classe que implementa o gerador de motor de simulação.
- **`SimulatorCore`** – Classe base de simulação de redes *CPNJava*. Esta classe implementa a maior parte do motor de simulação das redes coloridas. Entretanto, trata-se de uma classe abstrata e não pode ser instanciada. O papel do gerador de simulação é criar uma classe que estende de `SimulatorCore` e que implemente os métodos necessários para complementá-la, de acordo com o modelo de rede passado como parâmetro (Figura 28).
- **`Binding`** – Classe que representa uma ligação de variável.
- **`BindingSet`** – Classe que representa um conjunto de ligações.
- **`BindingElementPair`** – Representa um par (*t, B(t)*), chamado de elemento de ligação, que contém uma transição *t* e um conjunto de ligações *B(t)*. Durante a simulação, a classe ouvidora é responsável por selecionar um elemento de ligação habilitado (*binding element*) para ocorrer.
- **`Multiset`** – Classe que implementa as funcionalidades de um *bag* e que contém métodos para lidar com *bags* de *tokens*.

A geração dos simuladores *CPNJava* é feita utilizando-se um arquivo de *template*, que contém a estrutura básica da classe que será gerada. A geração do arquivo é realizada baseada no *template*, através de uma biblioteca chamada *Velocity* [Apache05], desenvolvida pelo projeto *Apache Jakarta*.

A Figura 28 ilustra o processo de geração dos simuladores *CPNJava*. Após a interpretação do arquivo PNML (1), o modelo CPNNet que representa a rede é carregado pelo `Generator` (2). Este, então, executa o *Velocity*, fornecendo as informações necessárias para a complementação do arquivo de *template*. O *Velocity*, finalmente, interpreta o *template* e gera o arquivo-fonte *Java* que contém a implementação da classe do simulador (3). Esta classe é nomeada seguindo o padrão `Net_`*`[Identificador da Rede]`*`_Simulator`.

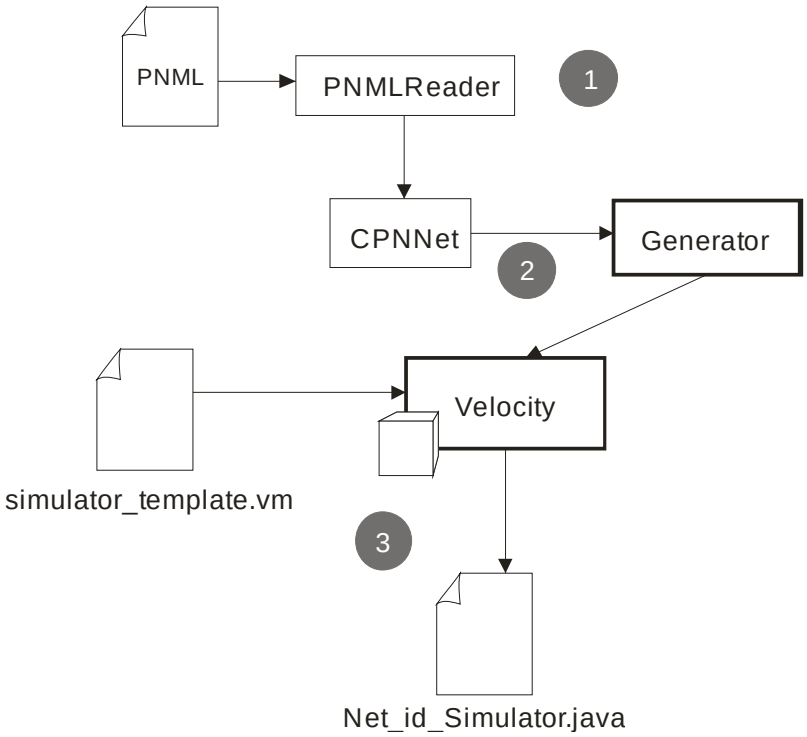


Figura 28. Processo de geração do simulador *CPNJava*

A classe gerada estende da classe `br.upe.dsc.jpetrisim.cpnjava.SimulatorCore`. No Capítulo 7 será descrito o conteúdo da classe gerada.

6.2.3 Classes Auxiliares

Dentre as classes auxiliares, a de maior importância é a classe `br.upe.dsc.jpetrisim.util.PNMLReader`. Esta é a classe responsável pela interpretação de arquivos PNML. Ela oferece os seguintes métodos estáticos:

- `PNet` **`readPetriNet`**`(String filePath)` – Processa o arquivo definido e gera a instância da rede *Place/Transition* correspondente (ignora elementos temporais).
- `PNet` **`readTimedPetriNet`**`(String filePath)` – Processa o arquivo definido e gera a instância da rede temporizada correspondente (contendo as informações temporais).
- `CPNNet` **`readCPNNet`**`(String filePath)` – Processa o arquivo definido e gera a instância da rede colorida *CPNJava*.

A biblioteca não impõe que a classe `PNMLReader` seja utilizada pelo usuário, uma vez que os simuladores carregam seus dados diretamente das instâncias, independentes de arquivos PNML. Se, por algum motivo, o usuário preferir outra forma de instanciar as redes, sem utilizar a classe `PNMLReader`, poderá fazê-lo. Se, por outro lado, ele não possuir uma forma própria de interpretar arquivos PNML, encontrará suporte para esta tarefa na biblioteca.

6.3 Protocolo de Comunicação

As interfaces da Camada de Comunicação definem todos os requisitos para comunicação com o simulador. O processo de comunicação é simples: a partir do momento em que a simulação é iniciada, a cada evento que ocorrer – habilitação de transição, desabilitação ou mudança de marcações – a aplicação é avisada de que algo ocorreu e solicitada a informar que ação deseja tomar. A simulação fica parada até que uma resposta da aplicação seja recebida. Esta resposta pode ser uma ordem para disparar uma certa transição, ou a escolha de um elemento de ligação para ocorrer (em redes coloridas) ou então uma ordem para finalizar a simulação.

A seguir, as três interfaces serão descritas e também as opções que a aplicação possui em cada evento.

6.3.1 ISimulationListener

A comunicação com o simulador de redes *Place/Transition* é realizada através da interface `ISimulationListener` (Interface de Ouvidor de Simulação). Os seguintes métodos são necessários para a implementação desta interface:

- `void` **`enableTransition`**`(String tid)` – Informa à aplicação que a transição identificada por *tid* foi habilitada.
- `void` **`disableTransition`**`(String tid)` – Informa à aplicação que a transição identificada por *tid* deixou de estar habilitada.
- `void` **`setTokens`**`(String pid, int n)` – Informa que o número de *tokens* do lugar identificado por *pid* mudou, e que o número atual de *tokens* neste lugar é *n*.
- `String` **`fireChoice`**`(Hashtable enabled)` – Indica que todas as transições habilitadas na marcação atual já foram verificadas, e pede que a aplicação selecione uma para ser disparada. A informação sobre as transições está contida na tabela *hashing enabled* (instância de *Hashtable*). As chaves na tabela são os identificadores das transições habilitadas e o valor associado a cada chave é um inteiro indicando o passo da simulação em que esta transição foi habilitada. O passo é uma variável que é incrementada sempre que um disparo de transição é executado. Neste momento, o usuário pode escolher parar a simulação, enviando `null` como resposta.
- `void` **`finished`**`()`- Informa que não há mais transições habilitadas na rede e, portanto, a simulação foi finalizada.
- `void` **`error`**`(String msg)` – Envia uma mensagem de erro à aplicação.

A interface `ISimulationListener` constitui a interface mais simples da biblioteca.

6.3.2 ITimedSimulationListener

Esta interface é a que permite a comunicação com o simulador de redes temporizadas. A interface `ITimedSimulationListener` (Interface de Ouvidor de Simulação Temporizada) estende a interface `ISimulationListener`, adicionando-lhe métodos específicos para tratar os eventos temporais da simulação.

Para compreensão desta interface, deve-se tomar conhecimento do método de avanço de tempo do simulador. O método de manipulação de tempo do simulador temporizado é direcionado a eventos. Isto significa que o tempo não progride de forma incremental durante a simulação, mas é avançado arbitrariamente para os instantes de ocorrência dos próximos eventos programados. Se uma aplicação que utilize o simulador necessitar de progresso de tempo

incremental, por exemplo, para exibir a simulação em uma animação gráfica, ela deve controlar a simulação para realizar a sincronização entre o simulador e a aplicação. Isto pode ser feito, por exemplo, fazendo com que o simulador fique congelado após cada avanço arbitrário e espere até que a aplicação tenha alcançado o tempo do simulador. Os detalhes sobre a implementação do simulador serão descritos no Capítulo 7.

A interface contém os seguintes métodos definidos (alguns métodos da interface `ISimulationListener` estão repetidos aqui):

- `void` **`enableTransition`**`(String tid)` – Informa à aplicação que a transição identificada por *tid* foi habilitada e que o seu tempo mínimo para disparo já foi alcançado, ou seja, ela está em condições de ser disparada.
- `void` **`disableTransition`**`(String tid)` – Informa à aplicação que a transição identificada por *tid* deixou de estar habilitada.
- `String` **`fireChoice`**`(Hashtable enabled)` – Indica que todas as transições habilitadas na marcação atual já foram verificadas e pede que a aplicação selecione a que deve ser disparada. No simulador de redes temporizadas, é possível utilizar a semântica de passo a qualquer momento. Para isso, basta que seja utilizada a palavra-chave “#step”. Recebendo “#step” ao invés do identificador de uma transição, o simulador irá disparar todas as transições possíveis, seguindo a *semântica de passo*. Além da opção de passo, o usuário tem a opção de realizar um avanço de tempo, através da palvra-chave “#clock:K”, onde ele pode substituir ‘K’ por um inteiro positivo, representando o número de unidades de tempo que o simulador deve avançar. O simulador irá checar se não há nenhuma transição com tempo limite de disparo inferior a este avanço de tempo. Se houver, o avanço é realizado apenas até o tempo limite desta transição. A aplicação irá receber uma mensagem clock (ver a seguir), com a informação do tempo atual do clock, após o avanço.
- `void` **`clock`**`(int currentClock)` – Informa à aplicação que houve uma mudança no tempo de simulação para o instante definido por *currentClock*. Neste momento, se a aplicação necessitar de um tempo incremental, ela poderá parar a simulação enquanto faz o seu próprio tempo progredir até alcançar o instante equivalente ao tempo atual de simulação. Após isso, basta retornar deste método, permitindo que a simulação prossiga normalmente.
- `void` **`scheduledEnable`**`(String tid, int clockCycle)` – Informa à aplicação que a transição identificada por *tid* se tornou habilitada e poderá ser disparada a partir do instante *clockCycle* do tempo de simulação.
- `void` **`scheduledFire`**`(String tid, int clockCycle)` – Informa que a transição identificada por *tid* deve ser disparada no máximo até o instante *clockCycle* do tempo de simulação.
- `void` **`unscheduled`**`(String tid)` – Informa que a transição identificada por *tid* não está mais programada, ou seja, não está mais sendo temporizada pelo simulador. Isto pode ocorrer quando a transição *tid* for desabilitada ou quando a mesma for disparada.
- `void` **`unsolvableConflict`**`(Vector transitionList)` – Ocorre quando a semântica de passo é utilizada para realizar o disparo e existem transições em conflito. Neste caso, nenhuma transição é disparada e o usuário será requisitado a escolher novamente uma transição para o disparo.

6.3.3 ICPNSimulationListener

A interface de comunicação com o simulador de rede colorida é simples, mas seu funcionamento se torna mais complexo pois existe a necessidade de um processamento mais elaborado por parte da aplicação.

Os métodos definidos são os seguintes:

- `void` **`markChanged`**`(String pid, Multiset mark)` – Informa que a marcação do lugar *pid* foi modificada e que a nova marcação é definida pelo `Multiset` *mark*.
- `BindingElementPair` **`occurChoice`**`(Hashtable enabledSet)` – Pede à aplicação que selecione um dos elementos de ligação habilitados para ocorrer. A tabela *hashing* fornecida como parâmetro apresenta como chaves os identificadores das transições habilitadas e seus correspondentes valores são `Multiset`'s, cujos conteúdos são todos os conjuntos de ligação habilitados para aquela transição, representados por instâncias da classe `BindingSet`. A aplicação deve responder fornecendoo uma instância de `BindingElementPair`, que deverá conter o identificador da transição que deve ocorrer e o conjunto de ligações a ser utilizado.
- `void` **`message`**`(String msg)` – Este método não é chamado pelo simulador, mas é um recurso que pode ser utilizado nas notações da rede, para enviar uma mensagem à aplicação. Isto permite que o usuário possa processar eventos específicos para o seu modelo. Na Seção 6.4 há uma lista dos métodos que o simulador disponibiliza para serem utilizados nas notações da rede.
- `void` **`finished`**`()` – Informa que não há mais transições habilitadas e, portanto, a simulação está concluída.

6.4 Comandos Especiais para CPNJava

Algumas variáveis e métodos especiais estão disponíveis para serem utilizados pelo usuário nas notações das redes. Elas dão acesso a estruturas internas do simulador, que podem ser necessárias durante a codificação de alguns métodos mais complexos, como a ligação de variáveis. Estas variáveis e métodos estão listadas a seguir.

- `void` **`listener.message`**`(String msg)` – Envia uma mensagem à classe ouvidora.
- `Multiset` **`getMark`**`(String pid)` – Retorna a marcação do lugar *pid* ou `null`, se o lugar não possuir marcação. A classe `Multiset` oferece diversos recursos desenvolvidos especificamente para auxiliar o usuário durante a simulação.

A idéia é disponibilizar mais métodos futuramente, fornecendo formas de interação bastante poderosas. Uma possibilidade é permitir que simuladores troquem mensagens entre si, permitindo a sincronização de simulações em paralelo. Esta abordagem foi deixada de fora do escopo deste trabalho, mas é uma proposta para trabalhos futuros, como será comentado no Capítulo 8.

Capítulo 7

Implementação dos Simuladores

Este capítulo descreverá a implementação dos simuladores de redes de Petri da biblioteca, que constituem o seu núcleo de simulação. Isto inclui os motores de simulação de redes *Place/Transition* e temporizadas, assim como também o gerador de simulador para redes coloridas *CPNJava* e o funcionamento dos simuladores gerados por este.

É preciso que o leitor tenha compreendido os conceitos apresentados nos capítulos anteriores para um melhor entendimento deste capítulo.

7.1 Representação Interna das Redes de Petri

Para que os simuladores pudessem apresentar um desempenho satisfatório, as redes de Petri foram representadas internamente da forma mais otimizada possível. Os seguintes critérios foram seguidos na criação do modelo de representação:

- **Não-redundância**. Informações redundantes na representação foram excluídas ao máximo. Desta forma o modelo poderia ser compacto, evitando o desperdício de memória.
- **Acesso rápido**. Os simuladores precisam obter rapidamente as informações que procuram. O modelo elimina ao máximo a necessidade de buscas ao longo da estrutura.
- **Relevância**. Apenas as informações necessárias para a simulação são mantidas. Assim, informações gráficas foram totalmente eliminadas. Além disso, as informações mantidas a cada instante são apenas as essenciais. As informações são eliminadas assim que não se tornem mais úteis.

O modelo foi desenvolvido da seguinte forma:

Tabelas *Hashing*. Todas as informações estão contidas em tabelas. Estas tabelas são implementadas como tabelas *hashing*. As tabelas *hashing* são estruturas de dados que têm alto desempenho na realização de buscas, pois utilizam um algoritmo de codificação que permite que os dados sejam encontrados imediatamente, independente da quantidade de dados armazenados (algoritmo de complexidade constante ou $O(1)$).

Tabelas de Entrada e Saída. Poder-se-ia imaginar que as informações relevantes de uma rede de Petri que precisam ser mantidas pelo simulador são os seus lugares, transições e arcos. Na verdade, estas informações apresentam excessiva redundância sob o ponto de vista da simulação. Vamos considerar inicialmente o caso da rede *Place/Transition*. Ocorre que a única informação que descreve uma rede de Petri *Place/Transition*, não levando em consideração a sua marcação, é o seu conjunto de arcos. A partir de seu conjunto de arcos, toda a estrutura da rede pode ser recuperada. A única ressalva é que este conjunto precisa estar separado em dois conjuntos menores: um contendo apenas os arcos que têm como origem lugares e outro contendo os que têm como origem transições.

Digamos que possuímos apenas o conjunto de arcos para certa rede e queremos obter a rede completa. Os arcos são um conjunto de pares (origem, destino). Como dito, este conjunto precisa estar divido em conjunto de arcos que partem de lugares e conjunto de arcos que partem de transições. Para que a rede seja obtida, basta seguir os passos a seguir:

1) Para cada arco no conjunto que parte de lugares, o primeiro elemento do par é adicionado ao conjunto de lugares da rede e o segundo elemento ao seu conjunto de transições;
2) Para cada arco no conjunto que parte de transições, se o primeiro elemento do par ainda não estiver no conjunto de transições da rede, ele é adicionado; se o segundo elemento ainda não fizer parte do conjunto de lugares, ele é adicionado;
3) Todos os arcos são adicionados ao conjunto de arcos da rede.

Como se pode facilmente verificar, a rede pode ser recuperada a partir do seu conjunto de arcos, desde que haja a separação entre arcos que partem de lugares e de transições. Os únicos elementos que não podem ser recuperados são aqueles que não possuem nenhum arco partindo dele ou apontando para ele. Entretanto, estes elementos estão desconectados da rede e não possuem função alguma, logo podem ser ignorados com segurança.

Mesmo quando se possui informações associadas aos arcos, por exemplo, pesos ou expressões, pode-se dizer que estas informações fazem parte dos arcos. Assim, a regra continua válida. Posteriormente será visto como foram tratadas as redes temporizada e colorida.

Assim, nos simuladores apresentados, **apenas os arcos são registrados para representar a estrutura da rede**. Eles são dispostos em tabelas *hashing* que associam cada arco ao seu peso. Duas tabelas são preenchidas, uma para os arcos que entram em uma transição (tabela de entrada) e outra para os que partem da transição (tabela de saída). Por questões de otimização, as tabelas foram estruturadas hierarquicamente. Ao invés de cada tabela possuir como chave um par (origem, destino), elas são estruturadas de forma que seja possível obter todos os arcos para uma dada transição de forma rápida. A 0 exemplifica a estrutura comum às tabelas de entrada e de saída. A tabela de entrada é composta por uma tabela principal e várias tabelas secundárias. O mesmo ocorre com a tabela de saída, sendo também organizada em uma principal e várias secundárias. A tabela principal de cada uma contém como chaves os identificadores das transições. A cada chave está associada uma tabela secundária, que contém como suas chaves os identificadores dos lugares que se conectam à transição relacionada na tabela principal através de um arco. Os valores nesta tabela secundária são os pesos dos arcos.

Esta estrutura é mais indicada, pois, em geral, os simuladores precisam obter todos os arcos para uma determinada transição e, depois, obter o peso de acordo com cada lugar de entrada ou de saída. Desta forma, cada uma destas buscas é realizada em tempo constante. Note que apenas fazem parte das tabelas os elementos associados a algum arco. Por exemplo, se não houver nenhum arco entrando em uma certa transição, esta transição não estará presente na tabela

principal de entrada. Assim também como, se um lugar não fizer parte da pré-condição de uma transição, este lugar não estará na tabela secundária correspondente àquela transição. O mesmo é válido para as tabelas de saída.

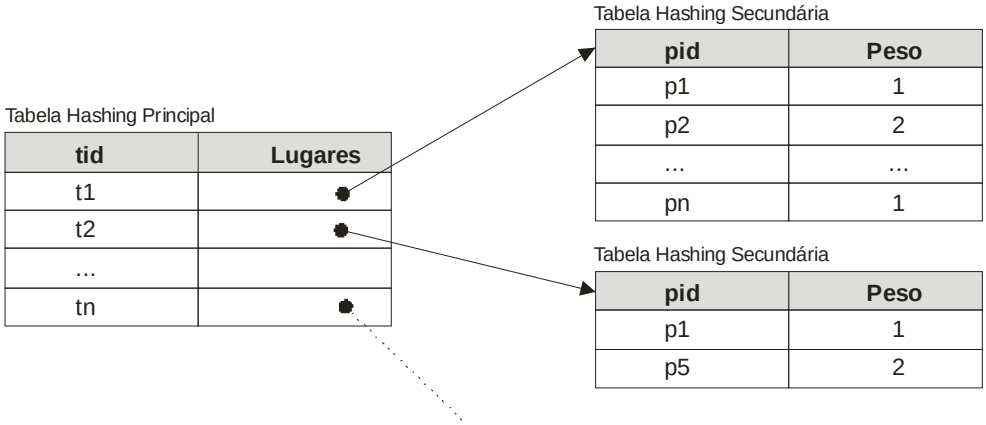


Figura 29. Hierarquia em que foram estruturadas as tabelas

O simulador *Place/Transition* mantém uma tabela extra, que associa lugares à lista de suas transições de saída. Apesar de redundante (uma vez que é uma repetição da tabela de entrada, apenas organizada de outra forma), ela é necessária para otimizar alguns métodos da implementação (que necessitam da informação estruturada nesta ordem).

Para as redes temporizadas, algumas novas tabelas foram necessárias. Duas surgem da necessidade de manter as informações temporais. Uma delas guarda os tempos mínimos dos intervalos de tempo associados a cada transição, a outra guarda os tempos máximos. Nestas tabelas, o tempo infinito é representado pelo valor -1. Tempos nulos (iguais a zero) não são armazenados. Se o simulador não encontrar a informação temporal para uma certa transição em alguma das tabelas, ele assumirá que o tempo para ela é igual a zero. Desta forma, manter informação para transições com tempo nulo passa a ser redundante.

Outra mudança é que, por questões de otimização, as tabelas de entrada e saída foram expandidas e passaram a conter informações redundantes. Isto ocorre porque, no simulador para redes de Petri temporizadas, é necessário tanto realizar a busca por todos os arcos para uma transição quanto por arcos para um certo lugar. Assim, as tabelas principais passaram a conter como chaves tanto lugares quanto transições. Isto permite que as duas buscas sejam feitas em tempos constantes. A desvantagem é que o dobro da informação precisa ser guardada (pois todo arco que parte de uma transição aponta para um lugar e todo arco que entra em um lugar parte de uma transição, ou seja, a mesma informação está repetida nas tabelas de entrada e de saída).

Para as redes de Petri coloridas se poderia esperar um número ainda maior de tabelas, por se tratar de uma estrutura mais complexa, mas na verdade, devido ao processo de geração de simuladores, seus simuladores contêm até menos tabelas que o simulador de redes temporizadas. Isto porque a maior parte das informações fica no próprio código que é gerado, dispensando o seu armazenamento em estruturas na memória.

Estes detalhes serão descritos quando tratarmos da implementação de cada simulador individualmente.

Tabelas de Marcação e Habilitação. As informações das tabelas de entrada e saída são carregadas uma vez apenas (quando a rede é lida) e são necessárias durante todo o processo de simulação. As informações sobre marcação da rede e estado das transições, por outro lado, se modificam constantemente durante o processo de simulação. Para manter estas tabelas o mais

compactas possível, o critério de relevância foi aplicado. Por exemplo, para mantermos o estado de marcação da rede, é preciso manter na memória a marcação de todos os lugares? A resposta é não. Na verdade, **apenas os lugares que possuem *tokens* precisam ser registrados**. Os lugares que não possuem nenhuma marca podem ser eliminados da tabela de marcações. No momento em que o simulador precisar consultar a marcação de um lugar, se o lugar não for encontrado na tabela de marcações, ele simplesmente deduz que o lugar procurado não possui *token*. Como, em geral, a minoria dos lugares possuem *tokens* em um certo instante, grande quantidade de memória é liberada com a eliminação destes lugares da tabela. O mesmo ocorre com a tabela de transições habilitadas. **Apenas as transições que estão habilitadas são mantidas na memória**.

Além de eliminar-se da tabela de marcação os lugares que não possuem *tokens*, estes lugares são também ignorados no processo de avaliação das transições – que calcula se cada transição está habilitada ou não. Quando o simulador procura por transições a serem habilitadas, ele **procura apenas por aquelas que possuem alguma marcação em suas pré-condições**. Primeiro, a lista de lugares marcados é verificada e, então, as transições que estão ligadas a estes lugares são avaliadas. As transições que não possuirem *tokens* em alguma de suas pré-condições nunca serão habilitadas (exceto no caso de transições *source*, que são habilitadas no início da simulação e permanecem habilitadas por toda a execução).

Outras tabelas temporárias são mantidas nos simuladores. Elas seguem o mesmo princípio das tabelas de marcação e habilitação.

```
// Tabela de marcações -------------
// chave: pid, valor: nº de tokens no lugar
Hashtable tokens = new Hashtable(); // p -> N

// Tabela de habilitação -----------
// chave: tid, valor: passo em que foi habilitada
Hashtable enabled = new Hashtable();

// Tabelas de Entrada e Saída ------
// chave: tid, valor: tabela secundária com: chave: pid, valor: peso do arco
Hashtable inputs = new Hashtable(); // t -> i(p) : Hashtable p -> N
// chave: pid, valor: tabela secundária com: chave: tid, valor: peso do arco
Hashtable inverseInputs = new Hashtable();//p -> i(t) : Hashtable t -> N
// chave: tid, valor: tabela secundária com: chave: pid, valor: peso do arco
Hashtable outputs = new Hashtable(); // t -> o(p) : Hashtable p -> N
```

Listagem 11. Declaração das tabelas internas do simulador

7.2 Motor de Simulação *Place/Transition*

O algoritmo de simulação das redes *Place/Transition* pode ser descrito como a seqüência de passos a seguir:

1) analisar as transições sob a marcação atual;
2) habilitar aquelas que devem ser habilitadas e desabilitar as demais;
3) se não houver transições habilitadas, encerrar a simulação aqui;
4) perguntar ao usuário que transição deve ser disparada;

5) disparar a transição escolhida, fazendo as mudanças das marcações dos lugares;
6) voltar ao passo 1.

O simulador foi implementado na classe **`Engine`** do pacote `br.upe.dsc.jpetrisim.place_transition`. A Listagem 11 mostra a definição das tabelas internas nesta classe. As tabelas foram implementadas através da classe `Hashtable` da API Java, que implementa o algoritmo de tabelas *hashing*.

O modelo da rede de Petri é carregado pelo método `load(PNet pnet)` da classe `Engine`. Neste método, a estrutura é varrida e as tabelas são preenchidas.

O trecho apresentado na Listagem 12 mostra o preenchimento das tabelas principais a partir dos arcos encontrados na estrutura da rede. Antes deste trecho, as tabelas foram instanciadas e foram inseridas chaves correspondentes aos identificadores dos lugares e transições encontrados na rede. O código referente ao tratamento de erros foi omitido, para maior clareza.

```
String thisTrans = thisArc.getTransitionEndPoint();
String thisPlace = thisArc.getPlaceEndPoint();

if (thisArc.getDirection() == PNArc.PLACE_TO_TRANSITION)
{
	//Adiciona o arco à tabela de entrada da transição
	((Hashtable) inputs.get(thisTrans.getId())).put(
			thisPlace.getId(),
			new Integer(thisArc.getWeight()));

	//Adiciona o arco à tabela redundante
	((Hashtable) inverseInputs.get(thisPlace.getId())).put(
			thisTrans.getId(),
			new Integer(thisArc.getWeight()));
}
else
{
	//Adiciona o arco à tabela de saída da transição
	((Hashtable) outputs.get(thisTrans.getId())).put(
			thisPlace.getId(),
			new Integer(thisArc.getWeight()));
}
```

Listagem 12. Preenchimento das principais tabelas internas.

O laço principal de execução do simulador verifica a existência de transições habilitadas e pede que o usuário escolha uma delas para que seja disparada. Quando não houver mais nenhuma transição habilitada, a simulação é encerrada. A Listagem 13 contém o trecho de código que realiza esta tarefa.

O método `fire`, que realiza o disparo da transição, é responsável por fazer as modificações necessárias na marcação da rede. Aqui mais uma vez o critério de relevância é utilizado. Uma vez que todas as transições habilitadas na marcação atual foram calculadas, após o disparo, apenas duas situações podem ocorrer:

- Algumas transições são desabilitadas devido ao lugar que perdeu *tokens* durante o disparo;
- Algumas transições são habilitadas devido ao lugar que recebeu *tokens* durante o disparo.

```
//Obtém a lista de lugares marcados
listOfPlaces = tokens.keys();

while (listOfPlaces.hasMoreElements())
{
	thisPlace = (String) listOfPlaces.nextElement();
	//Avalia todas as transições conectadas a este lugar
	evalTransitionsAfterPlace(thisPlace);
}

//adiciona transições source à tabela (sempre habilitadas)
Iterator sources = sourceTransitions.iterator();
while (sources.hasNext())
{
	enabled.put(sources.next(), new Integer(0));
}

//Todas as transições habilitadas na marcação inicial
//  já foram calculadas

String choosenTransition;
//Laço principal -----
while (!enabled.isEmpty()) //enquanto há transições habilitadas
{
	step++;
	//classe ouvidora escolhe qual deverá disparar
	choosenTransition = listener.fireChoice(enabled);

	if (choosenTransition != null)
		fire(choosenTransition); //dispara a transição escolhida
	else
		break; //null encerra simulação
}

//Fim da simulação
listener.finished();
```

Listagem 13. Laço principal do simulador

As transições que não se relacionam com os lugares que tiveram suas marcações alteradas não precisam ser avaliadas, pois os seus estados permanecerão os mesmos. Por este motivo, o método `fire`, além de alterar as marcações, solicita que as transições conectadas aos lugares que foram modificados sejam reavaliadas. Assim, **o estado da rede é reavaliado de uma forma otimizada, considerando-se apenas as transições cujo estado verdadeiramente possa ter sido alterado**.

Note que **transições fonte (*source*) nunca são desabilitadas**, pois não possuem lugares como pré-condições. Uma vez que tenham sido colocadas na tabela de habilitação no início da simulação, elas não mais serão removidas. O simulador identifica as transições fonte após o preenchimento das tabelas, verificando as transições presentes na tabela de saída que não estão presentes na tabela de entrada (significando que não há arcos tendo estas transições como alvo).

O trecho principal do método fire é mostrado na Listagem 14.

```
i = (Hashtable) inputs.get(tid);
o = (Hashtable) outputs.get(tid);

listOfPlaces = i.keys();      //pré-condições
while (listOfPlaces.hasMoreElements())
{
        thisPlace = (String) listOfPlaces.nextElement();
        //remove tokens conforme peso do arco
        int arcWeight = ((Integer) i.get(thisPlace)).intValue();
        setTokens(thisPlace,
         getTokens(thisPlace) – arcWeight);

        evalTransitionsAfterPlace(thisPlace);
}

listOfPlaces = o.keys();      //pós-condições
while (listOfPlaces.hasMoreElements())
{
        thisPlace = (String) listOfPlaces.nextElement();
        //adiciona tokens conforme o peso do arco
        int arcWeight = ((Integer) o.get(thisPlace)).intValue()
        setTokens(thisPlace,
         getTokens(thisPlace) + arcWeight);

        evalTransitionsAfterPlace(thisPlace);
}
```

Listagem 14. Trecho principal do método de disparo

O método poderia ser otimizado, fazendo com que as transições fossem avaliadas apenas no final do disparo. Isto evitaria que a mesma transição fosse avaliada várias vezes. Esta modificação poderá ser implementada futuramente.

```
boolean haveAllTokens = true;
i = (Hashtable) inputs.get(tid);
listOfPlaces = i.keys();

while (listOfPlaces.hasMoreElements())
{
        thisPlace = (String) listOfPlaces.nextElement();
        haveAllTokens = haveAllTokens &&
         (getTokens(thisPlace) >= i.get(thisPlace));
}
return haveAllTokens;
```

Listagem 15. Código que verifica se uma transição está habilitada

A avaliação de uma transição é feita pelo método `canEnable(String tid)`. Este método simplesmente obtém a tabela secundária que está ligada a esta chave na tabela de entrada e verifica se a marcação de cada lugar que esta tabela contém é maior ou igual ao peso do arco correspondente. Se isto for verdadeiro para todos os lugares da tabela, então a transição está habilitada. O trecho principal deste método está na Listagem 15.

7.3 Motor de Simulação de Redes Temporizadas

A simulação das redes temporizadas tem como base os mesmos princípios do simulador *Place/Transition*, mas para tratar as questões temporais um algoritmo mais elaborado precisou ser utilizado.

O primeiro ponto a ser observado está relacionado ao método de avanço do tempo. A opção utilizada foi de não realizar o avanço de tempo incremental, mas sim um avanço direcionado a eventos. O motivo disto é que se o tempo incremental fosse utilizado, a computação se tornaria bem mais complexa. Imagine que o simulador possua em um certo momento uma rede com dez transições habilitadas. Cada uma delas possui um certo tempo mínimo de habilitação após o qual elas podem disparar. Digamos que estas dez transições tenham tempos mínimos iguais a 10. Se o avanço incremental do tempo for utilizado pelo simulador, será necessário que, a cada incremento de tempo as dez transições sejam analisadas a fim de se saber se alguma delas está apta a disparar. Uma vez que todas possuem tempos mínimos iguais a 10, será necessário para o simulador fazer esta verificação dez vezes, o que faz com que $10 \times 10 = 100$ verificações sejam realizadas no total. Ele fará isso simplesmente para saber que, após 10 unidades de tempo, as dez transições estarão habilitadas.

Utilizando-se o avanço de tempo orientado a eventos, o simulador saberá que não há nada a ser feito nas dez unidades de tempo entre o momento em que as transições foram habilitadas e o momento em que a primeira delas irá disparar. Ele simplesmente verifica qual é o menor tempo para que qualquer uma das transições possa ser disparada e salta para aquele tempo. Para isso, ele precisa apenas verificar o menor dos tempos das dez transições. Ou seja, fará 10 verificações apenas no total.

Como já foi explicado brevemente no Capítulo 6, o usuário que necessite de um avanço de tempo incremental em sua aplicação pode bloquear o progresso da simulação (quando esta faz um avanço do tempo) até que a sua aplicação e o simulador estejam sincronizados novamente.

O simulador de redes temporizadas é implementado pela classe `Engine` do pacote `br.upe.dsc.jpetrisim.timed`. O algoritmo básico pode ser resumido na seguinte seqüência de passos:

1) analisar as transições sob a marcação atual;
2) as transições habilitadas são programadas para se tornarem disparáveis no tempo mínimo de cada uma;
3) as transições disparáveis no tempo atual são verificadas;
4) perguntar ao usuário qual das transições deve ser disparada;
5) disparar a transição escolhida, fazendo as mudanças das marcações dos lugares;
6) enquanto houver transições disparáveis no tempo atual, voltar ao passo 4;
7) avança o tempo para o menor tempo programado para que uma transição se torne disparável;

8) analisa as transições sob a marcação atual;
9) as transições habilitadas são programadas para se tornarem disparáveis nos seus tempos mínimos;
10) se não houver transições disparáveis ou programadas, encerrar a simulação;
11) senão volta para o passo 3.

O usuário tem à disposição três comandos que podem ser fornecidos ao invés do identificador de uma transição no momento em que é solicitado a escolher uma transição para disparar.

null – Enviando uma *String* nula, a simulação é encerrada;

"#step" – Enviando a *String* "#step", o máximo possível de transições são disparadas simultaneamente, de acordo com a semântica de passo;

"#clock:*X*" – O usuário pode enviar esta *String*, substituindo *X* por um valor inteiro maior que zero. Este comando irá fazer com que o clock seja avançado *X* unidades de tempo, sem disparar nenhuma transição. Se houver alguma transição com tempo limite de disparo no instante atual, o tempo não é avançado. Se houver uma transição com limite de disparo menor que o tempo atual + *X*, o tempo é avançado até o tempo de limite desta transição. Em qualquer caso, o simulador envia uma mensagem `clock` para a aplicação informando o tempo atual após o processamento deste comando.

O disparo de um passo é uma das tarefas mais complexas realizadas pelo simulador, pois ele precisa resolver todos os conflitos, considerando ainda os tempos limites das transições envolvidas.

Devido ao código do simulador temporizado ser cerca de 120% maior que o simulador *Place/Transition* e mais complexo, uma explicação detalhada do código aqui se torna impraticável. Entretanto, os principais algoritmos seguidos na implementação serão apresentados.

O simulador possui três tabelas para armazenar o estado das transições.

- *fireable* – Armazena as transições que são disparáveis no momento atual e os seus graus de habilitação respectivos;
- *scheduledFireable* – Armazena as transições habilitadas e os tempos em que devem se tornar disparáveis;
- *scheduledDeadline* – Armazena as transições habilitadas e os seus limites de disparo.

Os tempos são relativos ao início da simulação. Uma variável *clock* armazena o tempo atual da simulação.

Quando uma transição é habilitada, se ela possuir um tempo mínimo *t* de disparo maior que zero, então ela é programada para disparar no tempo *clock* + *t*, sendo adicionada à tabela *scheduledFireable*. Senão, ela se torna disparável imediatamente e é adicionada à tabela *fireable*. Ela também é adicionada à tabela *scheduledDeadline* junto com o tempo limite em que deve ser disparada.

Quando os tempos mínimo e máximo de uma transição são iguais a zero, ela se comporta como uma transição imediata.

A Listagem 16 apresenta o trecho do código que verifica o estado de uma transição. Algumas partes que não são relevantes no momento foram omitidas para facilitar a visualização.

```
enabledFactor = calculateEnabledFactor(thisTrans);
if (enabledFactor >= 1.0f)
{
        //testa restrições de tempo
        if (reachedTimeToFire(thisTrans))
        {
                //a transição pode disparar
                setFireable(thisTrans, enabledFactor);
        }
        else    //programa para se tornar disparável depois
          if (!isScheduled(thisTrans)) schedule(thisTrans);
}
else
{
        disable(thisTrans); //a transição está desabilitada
        unschedule(thisTrans); //remove qualquer programação para ela
}
```

Listagem 16. Avaliação das transições temporizadas

O grau de habilitação, calculado pelo método `calculateEnabledFactor`, é necessário para a resolução de conflitos na execução de um passo. Como será descrito mais adiante.

O disparo das transições ocorre da mesma forma que no simulador *Place/Transition*, com a diferença de que algumas otimizações foram implementadas. Estas otimizações podem também ser incorporadas ao simulador *Place/Transition* futuramente.

Nas redes temporizadas, o disparo de uma transição não consome tempo. Logo, o tempo de simulação não se modifica após o disparo de uma transição. Isto limita os eventos que podem ocorrer após esse disparo às seguintes opções:

- uma transição é habilitada, se tornando imediatamente disparável ou então sendo programada para disparar posteriormente;
- uma transição é desabilitada, podendo ser removida das tabelas de transições disparáveis ou programadas.

Estes eventos são tratados pelo trecho de código apresentado na Listagem 16.

Quando ocorre um avanço de tempo ordenado pelo usuário (através do comando "#clock:*X*"), apenas um evento pode ocorrer: o de uma transição se tornar disparável, devido ao seu tempo programado ter sido atingido. Como foi dito, o simulador não permite que o tempo seja avançado além do tempo limite de alguma transição que esteja habilitada.

O algoritmo precisa seguir a política de **memória de habilitação** e a semântica de **servidor único**. Pode-se confirmar que a política de memória de habilitação é obedecida pelos seguintes fatos:

- após um disparo, transições que não tiveram suas pré-condições modificadas não têm sua programação alterada, ou seja, seus contadores de tempo permanecem;

- transições que tiveram suas pré-condições alteradas são modificadas apenas se forem desabilitadas. Aquelas que permanecem habilitadas não têm sua programação alterada (conforme observa-se na linha marcada (linha 11) da Listagem 16).

A semântica de servidor único é garantida pelo fato de que as transições são programadas apenas no momento em que passam do estado de não-habilitadas para habilitadas. O aumento no grau de habilitação de uma transição não altera sua programação, apenas após o disparo seu estado será reavaliado e ela poderá ser habilitada novamente. Além disso, apenas uma entrada para cada transição é permitida na tabela de eventos programados. O que corresponde à existência de um único servidor no sistema.

O processo mais complexo da simulação ocorre quando o usuário ordena a execução de um passo. O simulador precisa encontrar e resolver todos os conflitos efetivos e, somente então, disparar as transições.

O algoritmo do passo é descrito a seguir:

1) para cada lugar que habilita uma transição, checar se este lugar habilita mais de uma transição. Se verdadeiro, este lugar será adicionado à tabela de possíveis conflitos;
2) para cada lugar em possível conflito, calcula o somatório de todos os pesos dos arcos que partem deste lugar. Se o somatório dos pesos for igual ou inferior ao número de *tokens* presentes no lugar, o lugar não estará em conflito;
3) para cada lugar em conflito:
   se o lugar habilita transições com tempo limite alcançado (serão chamadas de transições urgentes), vai para o passo 4.
   se o lugar não habilita transições urgentes, vai para o passo 6.
4) a primeira transição urgente da lista será selecionada e os *tokens* para o seu disparo são garantidos. Os graus de habilitação das outras transições são recalculados com base no número de *tokens* restantes. Transições com grau menor que 1 são desabilitadas;
5) enquanto houver transições urgentes habilitadas, garantir os *tokens* para a próxima transição e recalcular os graus de habilitação das restantes;
6) enquanto ainda houver transições habilitadas, selecionar aleatoriamente uma das transições para disparar e garantir os seus *tokens*, recalculando a habilitação das restantes;
7) disparar todas as transições que tiveram seus *tokens* garantidos.

Os passos 1 e 2 realizam a busca por conflitos. Os passos 3 a 6 constituem a resolução destes conflitos. As transições que atingiram seu tempo limite de disparo concorrem entre si, mas têm prioridade sobre as transições restantes.

7.4 Simulação das Redes CPNJava

Para que se possa compreender o funcionamento da simulação das redes *CPNJava* é necessário que seja explicado o processo de geração, que é a estratégia que permite que as redes com código *Java* anotado sejam simuladas.

O objetivo da geração é reunir todos os trechos de código *Java* presentes na rede para formar uma nova classe *Java*, organizando estes códigos de tal forma que eles sejam executados no processo de simulação para realizar as tarefas desejadas pelo projetista da rede. Um esqueleto

de simulação foi desenvolvido, no qual foram implementadas as funcionalidades básicas do processo de simulação. Este esqueleto é então preenchido com os trechos de código retirados da rede. A estrutura da rede também é analisada para a montagem do simulador final.

A idéia por trás da geração é simples. Ela tem como base o fato de que os códigos anotados precisam ser executados em apenas três momentos:

- durante a inicialização de um lugar;
- durante a avaliação de uma transição;
- durante o disparo de uma transição.

Independentemente da rede que se tenha, a execução destes códigos segue sempre o mesmo padrão, conforme a semântica das redes de Petri coloridas. Isso, com o auxílio do fato de o código da ligação de variáveis ser implementado pelo usuário, torna possível a montagem de um programa simulador sem nenhum conhecimento do conteúdo dos códigos presentes na rede.

Caso a ligação de variáveis fosse deixada para o simulador, um *parser* para *Java* precisaria ser construído para a interpretação das notações da rede. Isto é uma possibilidade que pode ser implementada em trabalhos futuros.

7.4.1 SimulatorCore

A seguir, será explicada a classe que contém o código fundamental da simulação, chamada `SimulatorCore`. Trata-se de uma classe abstrata que serve como base para o funcionamento dos simuladores que são gerados.

A classe `SimulatorCore`, contida no pacote `br.upe.dsc.jpetrisim.cpnjava`, é bastante semelhante ao simulador *Place/Transition*. O algoritmo principal é o mesmo, com a única diferença de que o usuário precisa selecionar um elemento de ligação para ocorrer, ao invés de uma transição. Funcionalmente, entretanto, os dois são praticamente iguais.

A diferença que torna a classe `SimulatorCore` capaz de simular redes coloridas está nos métodos de avaliação e de ocorrência das transições. A Listagem 17 apresenta o método `eval`, responsável por verificar se uma transição está habilitada sob alguma ligação.

```
protected void eval(String tid)
{
	Multiset allEnabledBindings;
	// obtém o nome do método de ligação
	sbuffer.replace(0, sbuffer.length(), "");
	sbuffer.append(tid).append("_getBindings");
	// invoca o método de ligação
	allEnabledBindings = (Multiset) this.getClass()
	     .getMethod(sbuffer.toString(), null).invoke(this, null);
	// verifica se ligações foram encontradas
	if ((allEnabledBindings != null) && !(allEnabledBindings.isEmpty()))
	{	//se sim, habilita a transição
		enabledSet.put(tid, allEnabledBindings);
	}
	else
		enabledSet.remove(tid);
}
```

Listagem 17. Método de avaliação de transições na rede colorida

O método invocado para obtenção das ligações é criado pelo gerador *CPNJava* a partir do código de ligação fornecido pelo usuário para a transição. Para cada transição, dois métodos são gerados na classe do simulador:

- `Multiset` *`TID`*`_getBindings()` – conterá o código que executa a ligação das variáveis livres da transição;
- `void` *`TID`*`_occur(BindingElementPair bindingElement)` – conterá o código de disparo da transição.

Para cada transição um método específico é criado, substituindo-se *TID* pelo identificador da transição correspondente.

O método de ocorrência da transição na classe `SimulatorCore` simplesmente invoca o método correspondente à transição fornecida, conforme pode ser visto na Listagem 18.

```
protected void occur(BindingElementPair bindingElement)
{
        sbuffer.replace(0, sbuffer.length(), "");
        sbuffer.append(bindingElement.getTID()).append("_occur");

        BindingSet[] binding  = { bindingElement.getBindingSet() };
        Class[] argTypes =  { binding[0].getClass() };

        this.getClass()
                .getMethod(sbuffer.toString(), argTypes)
                .invoke(this, binding);
}
```

Listagem 18. Ocorrência de um elemento de ligação

O método *`TID`*`_occur` da transição é invocado e o conjunto de ligações é repassado como parâmetro. Estes métodos, obviamente, não estão presentes na classe `SimulatorCore`. O trecho final da Listagem 18 é o responsável por obter a referência para a instância da classe-filha e invocar o método correspondente.

7.4.2 Generator

A geração do simulador é realizada pela classe `Generator`, do pacote `br.upe.dsc.jpetrisim.cpnjava`. Uma vez que o modelo da rede tenha sido carregado, o `Generator` realiza uma varredura, coletando todas as informações necessárias e preenchendo as suas tabelas internas.

Tendo toda a estrutura da rede armazenada internamente, o interpretador da biblioteca *Velocity* é executado para processar o arquivo de *template*. O funcionamento do *Velocity* é simples: ele é invocado a partir de uma aplicação para processar um arquivo de *template*, que contém códigos em uma linguagem própria do *Velocity*, chamada VTL (*Velocity Template Language*). Para que o processamento seja realizado, a aplicação fornece ao interpretador *Velocity* o acesso às suas próprias variáveis internas, registrando-as em um *contexto* de execução.

Uma vez que a aplicação tenha criado este contexto, o *Velocity* terá acesso a todas as variáveis registradas, podendo consultá-las durante o processamento do arquivo de *template*. Ao final do processamento, é gerado o arquivo interpretado.

A Listagem 19 demonstra o código que cria o contexto e invoca o *Velocity* para geração do arquivo *Java* do simulador. O código referente ao tratamento de exceções foi removido para facilitar a visualização.

```
//inicialização
Velocity.init();

// criação do contexto
VelocityContext context = new VelocityContext();
context.put("netId", thisNetId);
context.put("initmarks", initmarks);
context.put("inputs", inputs);
context.put("outputs", outputs);
context.put("transitions", transitions);
context.put("prebindings", prebindings);
context.put("guards", guards);
context.put("enviroment", enviroment);
context.put("out", System.out);

// carregando o template do arquivo 'simulator.vm'
Template template = null;
template = Velocity.getTemplate("simulator.vm");

// cria o arquivo de saída 'Net_[ID]_Simulator.java'
FileWriter fileWriter =
               new FileWriter("Net_" + thisNetId + "_Simulator.java");

//executa o Velocity
template.merge(context, fileWriter);

// fecha o arquivo
fileWriter.flush();
fileWriter.close();
```

Listagem 19. Invocando o *Velocity* para geração do simulador

As variáveis adicionadas ao contexto do *Velocity* pelo gerador são suas tabelas internas, que já foram carregadas com os dados da rede a ser processada (estrutura e anotações).

Todo o restante do processo está contido no código do *template* "simulator.vm". A Figura 30 mostra a estrutura deste arquivo. Em destaque, são exibidos os trechos que se tornarão parte do arquivo *Java* gerado. As outras partes do código correspondem aos comandos escritos em VTL, que são iniciados por uma palavra-chave, cujo primeiro caractere é sempre '#'.

Podem ser encontradas diversas variáveis ao longo do arquivo. Todas as variáveis têm um nome iniciado pelo caractere '$'. Na Figura 30, elas são apresentadas na cor verde. Estas variáveis podem ser locais, criadas dentro do próprio *template*, ou podem ser provenientes do contexto criado pela aplicação. Todos os métodos dos objetos presentes no contexto podem ser utilizados, mas o *Velocity* dá suporte completo a apenas algumas estruturas de dados. Felizmente, estas estruturas são todas as mais comuns. Quando uma variável é encontrada fora de um

comando, o *Velocity* converte a variável para uma *String*, utilizando o método `toString()` do objeto. A *String* é, então, copiada literalmente para o local onde está a variável, substituindo suas ocorrências no arquivo gerado.

```
#foreach ($trans in $transitions)
    /** Ligação da transição $trans **/

    private Multiset ${trans}_getBindings()
    {
        Multiset allEnabledBindings = null;
        BindingSet thisBinding = new BindingSet();

        String thisTrans = "${trans}";

        //Lista de lugares na pré-condição
        String[] inputPlaces =
            new String[] {
#set($inplaces = $inputs.get($trans))
#set($inplacesSet = $inplaces.keySet())
#set($beforeLast = $inplacesSet.size() - 2)
    #foreach ($el in $inplacesSet)
    #if ($velocityCount <= $beforeLast)
            "$el" ,
    #else
            "$el"
    #end
    #end
            };
        #set($prebind = false)
        #set($prebind = $prebindings.get($trans))
        #if ($prebind)
        $prebind        ## Expressão da ligação
        #end

        return allEnabledBindings;
    }
```

Figura 30. Estrutura de um arquivo de *template* do *Velocity*

O código implementado no *template* utilizando a VTL é o responsável por criar os métodos da classe que será gerada. O algoritmo seguido pode ser descrito como abaixo:

I - **Conteúdo da classe**

1) Um construtor é criado para inicializar as tabelas internas do simulador;

2) As variáveis e métodos contidos na declaração da rede (*enviroment*) são adicionados à classe do simulador;
3) Um método `init` é criado, contendo o código de inicialização fornecido na rede;
4) Para cada transição, um método *`TID`*`_getBindings` e *`TID`*`_occur` são criados, onde *TID* é o identificador da transição.

II - **Conteúdo do construtor**

1) Para cada lugar que possui uma marcação inicial não vazia, o seu código de inicialização é copiado para o construtor para que seja executado durante a instanciação da classe, permitindo que a tabela de marcações seja preenchida com a marcação inicial.
2) Para cada entrada presente na tabela de entradas do contexto, um trecho de código que adiciona uma entrada igual na tabela de entradas do simulador é criado;
3) O mesmo procedimento do passo 2 é realizado para a tabela de saídas;
4) O método `init` é invocado pelo construtor.

III - **Conteúdo do método *getBindings***

1) O código de ligação fornecido pelo usuário para a transição é copiado para o método `getBindings` correspondente;
2) O código da guarda é copiado no código de verificação da guarda.

IV – **Conteúdo do método *occur***

1) É criada a declaração das variáveis livres definidas pelo usuário;
2) É criado o código que atribui as ligações às variáveis correspondentes;
3) Para cada lugar de entrada, o código do arco correspondente é copiado para o método para realizar a atribuição à variável “$”, para que os *tokens* atribuídos a esta variável sejam removidos do lugar;
4) Para cada lugar de saída, o código do arco correspondente é copiado para o método para realizar a atribuição à variável “$”, para que os *tokens* atribuídos a esta variável sejam adicionados ao lugar;
5) O método é finalizado com a verificação de estados das transições.

Para que o processo possa ser melhor compreendido, um exemplo simples de geração será descrito.

7.4.3 Exemplo de Geração de um Simulador

A Figura 31 ilustra a rede cujo simulador será gerado. A rede contém uma transição e dois lugares. Neste exemplo, é definida uma classe `Number` para ser utilizada como cor dos lugares, mas a própria classe `Integer` poderia ser utilizada. A definição de uma nova classe tem caráter meramente ilustrativo neste exemplo.

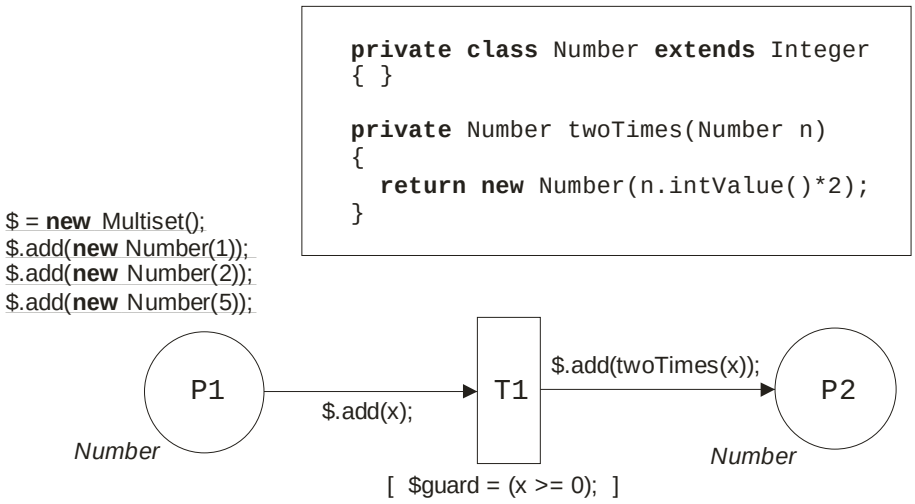


Figura 31. Rede para a qual será gerado o simulador

A representação em PNML para esta rede é apresentada na Listagem 20. Foi dado à rede o identificador “sample”. As informações gráficas foram suprimidas. É importante notar que a utilização de um editor de XML é recomendada para a escrita do PNML, pois certos caracteres presentes no código das anotações podem não ser permitidos literalmente no XML, necessitando de sua substituição por códigos especiais da linguagem.

O simulador gerado será escrito no arquivo *Net_sample_Simulator.java*. O código deste arquivo é apresentado no Anexo A.

7.5 Exemplo de Aplicação

Para ilustrar a integração da biblioteca a uma aplicação, foi desenvolvida uma aplicação simples que carrega um arquivo PNML e exibe o progresso da simulação, requisitando respostas do usuário quando necessário através do console. O código desta aplicação é apresentado no Anexo B.

```
<pnml>
        <net id="sample" type="coloured">
         <toolspecific tool="jpetrisim" version="1.0">
         <enviroment>
               private class Number extends Integer { }
               private Number twoTimes(Number n)
               {
                      return new Number(n.intValue()*2);
               }
         </enviroment>
         </toolspecific>
          <place id="P1">
               <toolspecific tool="jpetrisim" version="1.0">
                      <colorSet>Number</colorSet>
               </toolspecific>
               <initialMarking><text>
                      $ = new Multiset();
                      $.add(new Number(1));
                      $.add(new Number(2));
                      $.add(new Number(5));
               </text></initialMarking>
         </place>
         <place id="P2">
               <toolspecific tool="jpetrisim" version="1.0">
                      <colorSet>Number</colorSet>
               </toolspecific>
         </place>
         <transition id="T1">
               <toolspecific tool="jpetrisim" version="1.0">
               <guard>
                      $guard = (x &gt;= 0);
               </guard>
               <binding>
               //tipo de ligação já implementado pela classe Multiset
                      Multiset defaultBindings =
                                    getMark("P1").getDefaultBindings("x");
                      allEnabledBindings.add(defaultBindings);
               </binding>
               <variables>
                <var name="x" class="Number" />
               </variables>
               </toolspecific>
         </transition>
         <arc id="a1" source="P1" target="T1">
               <inscription><text>$.add(x);</text></inscription>
         </arc>
         <arc id="a2" source="T1" target="P2">
               <inscription><text>$.add(twoTimes(x));</text></inscription>
         </arc>
        </net>
</pnml>
```

Listagem 20. Descrição em PNML da rede "sample"

Capítulo 8

Conclusões e Trabalhos Futuros

Quando este trabalho foi proposto, tinha-se em mente o desenvolvimento de simuladores na linguagem *Java* para utilização em projetos próprios. A carência de ferramentas de simulação que suprissem todas as necessidades que eram encontradas na época levaram à proposta da implementação da simulação, para que esta fosse incorporada aos sistemas implementados.

Logo, porém, percebeu-se que os simuladores poderiam ser utilizados não apenas para estes sistemas, mas poderiam servir para solucionar os problemas de outras pessoas que também necessitassem simular seus modelos em uma aplicação particular e não estivessem encontrando suporte nas ferramentas atualmente disponíveis.

O resultado foi a implementação da biblioteca ***jPetriSim***. O seu objetivo foi tornar todos os componentes implementados para a simulação de redes de Petri disponíveis para aqueles que necessitassem integrar este recurso às suas próprias ferramentas. A biblioteca conta com um conjunto de simuladores e diversas interfaces através das quais as aplicações podem acompanhar e controlar o progresso da simulação. Três classes de rede de Petri foram contempladas neste trabalho: as redes *Place/Transition*, as redes temporizadas e as redes coloridas. Para as redes de Petri coloridas, um novo tipo de rede foi proposto, utilizando-se a linguagem *Java* como linguagem de notação da rede. Estas redes foram chamadas de *CPNJava*. A linguagem *Java* traz novas possibilidades para os pesquisadores, pois eles passam a ter à sua disposição, durante a implementação dos modelos, todos os recursos que são vastamente fornecidos para *Java* pelas comunidades de desenvolvimento de *software*.

Ao longo do trabalho uma pesquisa extensa foi realizada sobre toda a teoria das redes de Petri e os aspectos comportamentais necessários para a compreensão do seu processo de simulação. Todo o conteúdo apresentado nesta monografia serviu como base na implementação da biblioteca, garantindo-lhe o rigor científico que é necessário quando métodos formais são tratados. O levantamento teórico realizado não apenas apresenta os conceitos fundamentais das teorias de redes de Petri como também esclarece diversos pontos que muitas vezes são apresentados de forma obscura. Isto porque, para uma correta implementação, cada conceito precisou ser analisado em detalhes.

8.1 Propostas Para Trabalhos Futuros

O produto implementado neste trabalho é apenas um protótipo. Diversas melhorias precisam ser realizadas para que ele possa ser utilizado com maior segurança por outros desenvolvedores. Dentre estas melhorias podem ser citadas:

- tratamento de erros mais eficiente;
- melhor suporte para as redes *CPNJava*;
- novas possibilidades de interação entre aplicação e biblioteca.

Para que as redes *CPNJava* possam ser melhor utilizadas na prática, a implementação de um algoritmo de *ligação de variáveis* é desejável. Para isso também é necessário que seja implementado um *parser* para realizar a interpretação do código anotado à rede.

Dentre as possibilidades de interação, pode-se citar a possibilidade de modificar as marcações arbitrariamente durante a execução da simulação, algo que é muito útil para o estudo do modelo em alguns casos.

Uma outra proposta bastante interessante é a possibilidade de executar diversos simuladores em parelelo, interagindo entre si. Estes simuladores poderiam simular pequenos modelos que fizessem parte de um sistema maior e trocariam mensagens entre si para realizar a sincronização. Isto já pode ser realizado na implementação atual da biblioteca, mas a definição de um conjunto de interfaces e classes auxiliares para dar suporte específico a esta tarefa seria de grande utilidade. Também é importante que seja realizado um estudo para que sejam levantadas as vantagens e desvantagens de se utilizar a simulação em paralelo.

A biblioteca ***jPetriSim*** está disponível livremente na web, e pode ser encontrada no endereço http://www.dsc.upe.br/~calo/jpetrisim.

Bibliografia

[Ajmone95] AJMONE-MARSAN, M. et al. **Modelling with Generalized Stochastic Petri Nets**. [S. l.]: John Wiley And Sons, 1995. 316 p. (Wiley Series in Parallel Computing). Disponível em: <http://www.di.unito.it/~greatspn/GSPN-Wiley/>. Acesso em: 15 maio 2006.

[Apache05] THE APACHE JAKARTA PROJECT. **The Velocity User Guide**. The Apache Software Foundation, 2005. Disponível em: <http://jakarta.apache.org/velocity/docs/user-guide.html>. Acesso em: 15 maio 2006.

[Billington03] BILLINGTON, Jonathan et al. The Petri Net Markup Language: Concepts, Technology and Tools. In: International Conference on Application and Theory of Petri Nets, 24., 2003. Eindhoven. **Proceedings…** The Netherlands: Lecture Notes in Computer Science, Springer-Verlag, 2003. v. 2679, p. 483 - 505.

[Bowen95] BOWEN, J. P.; HINCHEY, M. G. **Seven More Myths of Formal Methods: Dispelling Industrial Prejudices**. IEEE Software, [S. l.], v. 12, n. 4, p. 34-41, July 1995. Disponível em: <http://citeseer.ist.psu.edu/13692.html>. Acesso em: 15 maio 06.

[Beaudouin00] BEAUDOUIN-LAFON, M. et al. CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. In: International Conference on Application and Theory of Petri Nets, 22., 2001. **Proceedings…** Newcastle upon Tyne, UK: Lecture Notes in Computer Science, Springer-Verlag, 2001, v. 2075, p. 71-80.

[EIG05] IBM INC. **2005 Award Recipients**. Disponível em: <http://www-304.ibm.com/jct09002c/us/en/university/scholars/products/eclipse/eig_2005.html> . Acesso em: 15 maio 06.

[EZPetri03] ARCOVERDE JR., Adilson. **EZPetri - Um Ambiente para integração de linguagens de descrição de redes de Petri**. 2004. Monografia (Graduação) – Curso de Engenharia da Computação, Departamento de Sistemas Computacionais, Universidade de Pernambuco, Recife, 2004.

[Gosling05] GOSLING, James et al. **The Java Language Specification**. 3. ed. Califórnia: Sun Microsystems Inc, 2005. (The Java Series). Disponível em: <http://java.sun.com/docs/books/jls/index.html>. Acesso em: 15 maio 2006.

[Hall90] HALL, Anthony. **Seven Myths of Formal Methods**. IEEE Software, [S. l.], v. 7, p.11-19, September 1990. Disponível em: <http://www.cs.virginia.edu/~jck/cs651/papers/seven.myths.pdf>. Acesso em: 15 maio 2006.

[Jensen98] JENSEN, Kurt. An Introduction to the Practical Use of Coloured Petri Nets. In: Advances in Petri Nets, Lecture Notes in Computer Science. **Proceedings…** Heidelberg: Springer, 1998, v. 1492, p. 237-292.

[Jensen94] JENSEN, Kurt. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In: A Decade of Concurrency, Reflections and Perspecitves. **Proceedings...** Lecture Notes in Computer Science, Noordwijkerhout, The Netherlands: Springer-Verlag, 1994, v. 803, p. 230-272.

[Maciel96] MACIEL, Paulo R. M.; LINS, R. D.; CUNHA, P. R. F. **Introdução às Redes de Petri e Aplicações**. Campinas: X Escola de Computação, 1996. 183 p.

[Merlin76] MERLIN, P. **A Methodology for the Design and Implementation of Communication Protocols.** IEEE Transactions on Communications, v. 24, p. 614-621, 1976.

[Meta93] META SOFTWARE. **Design/CPN Reference Manual for X-Windows Version 2.0**. Cambridge: Meta Software, 1993.

[Murata89] MURATA, Tadao. **Petri Nets: Properties, Analysis and Applications**. In: Proceedings of the IEEE, v. 77, n. 4, 1989.

[Petri62] PETRI, Carl A. **Kommunikation mit Automaten**, 1962. Tese (Doutorado) - University of Bonn, Bonn, West Germany, 1962.

[Petrilogic06] DEPTO. DE SISTEMAS COMPUTACIONAIS. **Petrilogic Home Page**. Recife, 2006. Disponível em: <http://www.dsc.upe.br/~petrilogic>. Acesso em: 16 maio 2006.

[Oasis01] OASIS COMMITTEE. **RELAX NG Specification**. The Organization for the Advancement of Structured Information Standards, 2001. Disponível em: <http://www.relaxng.org/spec-20011203.html>. Acesso em: 3 maio 2006.

[Reisig85] REISIG, W. **Petri Nets: An Introduction**. EATCS Monographs on Theoretical Computer Science, New Castle: Springer-Verlag, 1985, v. 4.

[Roch03] ROCH, Stephan; STARKE, P. H. **INA: Integrated Net Analyser Version 2.2 Manual**. Disponível em: <http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/manual.ps>. Acesso em: 16 maio 06.

[Tavares06] TAVARES, Eduardo A. G. **A Time Petri Net Based Approach for Software Synthesis in Hard Real-Time Embedded Systems with Multiple Processors**. 2006. 131 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2006.

[TGI06] TGI GROUP. **Petri Nets World: Online Services for the International Petri Nets Community**. Hamburg, Alemanha. Disponível em: <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>. Acesso em: 16 maio 2006.

[WG04] W3C XML WORKING GROUP. **Extensible Markup Language (XML) 1.1, W3C Recommendation 04 February 2004**. Disponível em: <http://www.w3.org/TR/2004/REC-xml11-20040204/>. Acesso em: 15 maio 2006.

[Weber02] WEBER, M.; KINDLER, E. The Petri Net Markup Language. In: Petri Net Technology for Communication Based Systems, Lecture Notes in Computer Science. **Proceedings…** Springer, 2002, v. 2472, p. 124-144.

[Weber06] WEBER, M. **Petri Net Markup Language Home Page**. Disponível em: <http://www2.informatik.hu-berlin.de/top/pnml/about.html>. Acesso em: 16 maio 2006.

[Zimm01] ZIMMERMANN, Armin. **TimeNet 3.0 User Manual**. Berlim: Performance Evaluation Group, 2001. Disponível em: <http://pdv.cs.tu-berlin.de/~timenet/TimeNET-UserManual30.pdf.gz>. Acesso em: 15 maio 2006.

[Zuberek91] ZUBEREK, W. M. **Timed Petri Nets: Definitions, Properties and Applications**. Microeletronics and Reliability, v. 31, n. 4, p. 627-644, 1991.

Anexo A

Exemplo de Simulador Gerado pela Biblioteca

A rede para a qual este simulador foi gerado está descrita na Seção 7.4.2. O código gerado corresponde ao arquivo `Net_sample_Simulator.java` e é apresentado abaixo.

```
import java.util.*;
import br.upe.dsc.cpnjava.*;
import br.upe.dsc.cpnjava.cpn_library.*;


/**********************************
*                                 *
*   Coloured Petri Net Simulation   *
*   Generated by CPN/Java Engine    *
*                                 *
**********************************/
public class Net_sample_Simulator extends SimulatorCore
{

	//Internal variables
	Multiset $;

	//------------- ENVIROMENT ---------------
		private class Number extends Integer { }

		private Number twoTimes(Number n)
		{
			return new Number(n.intValue()*2);
		}


	//------------- INITIALIZATION ---------------
	public Net_sample_Simulator()
	{
		//----- INITMARKS ----------

		//---- place P1
```

```
            $ = new Multiset();
            $.add(new Number(1));
            $.add(new Number(2));
            $.add(new Number(5));

      //register the initmark
      marks.put("P1", $);


      //---- INPUT FUNCTION ------
      Vector sourceCollection;

      sourceCollection = new Vector(15, 5);
      inputs.put("T1", sourceCollection);


      sourceCollection.add("P1");

      sourceCollection = new Vector(15, 5);
      inputs.put("P1", sourceCollection);


      sourceCollection.add("T1");


      //---- OUTPUT FUNCTION -----
      Vector targetCollection;

      targetCollection = new Vector(15, 5);
      outputs.put("P1", targetCollection);


      targetCollection.add("T1");


      //init() not provided

}


//------------- TRANSITIONS ---------------

/** T1 **/
private Multiset T1_getBindings()
{

      Multiset allEnabledBindings = null;
      BindingSet thisBinding = new BindingSet();

      Boolean $guard = false;

      String thisTrans = "T1";

      String[] inputPlaces =
            new String[] {

            "P1"

            };



            //tipo de ligação já implementado pela classe Multiset
            Multiset defaultBindings =
                  getMark("P1")
```

```
                        .getDefaultBindings("x");
                allEnabledBindings.add(defaultBindings);



                //verifies the guard-------------
                List $enabledBindingList = allEnabledBindings.getAsList();

                Iterator enabledBindingIterator$ =
                            enabledBindingList$.iterator();

                while (enabledBindingIterator$ .hasNext())
                {

                  BindingSet binds$ = (BindingSet) enabledBindingIterator$ .next();

                //generated variable declaration
                  Integer x = (Integer) binds$ .get("x").getValue();
                  Integer y = (Integer) binds$ .get("y").getValue();


                //user's guard
                  $guard = (x >= 0);


                  if (!$guard) allEnabledBindings.remove(binds$);

                }

        return allEnabledBindings;
}



private void T1_occur(BindingSet $bindings)
{
        Set transList$ = new HashSet();
        Vector pList$;
        Iterator i$;

        //variables
        Number x = (Number) $bindings.get("x").getValue();

        //Remove tokens

        $ .clear();
        $.add(x);
        removeTokens("P1", $);

        pList$ = (Vector) outputs.get("P1");

        if (pList$ != null)
        {
         i$ = pList$ .iterator();
         while (i$ .hasNext())
         {
           transList$ .add(i$.next());
         }
        }

        //Add tokens

        $ .clear();
        $.add(twoTimes(x));
        addTokens("P2", $);
```

```
            pList$ = (Vector) outputs.get("P2");
            if (pList$ != null)
            {
             i$ = pList$ .iterator();
             while (i$ .hasNext())
             {
               transList$. add(i$.next());
             }
            }

            //evaluate modified transitions
            try
            {
                  evalTransitionSet(transList$);
            }
            catch (Exception e)
            { e.printStackTrace(); }
      }

}
```

Anexo B

Exemplo de Aplicação

O código fornecido a seguir implementa uma aplicação que utiliza o simulador de redes temporizadas da biblioteca ***jPetriSim***. A aplicação funciona em modo console e informa ao usuário os eventos que estão ocorrendo na simulação, requisitando também uma resposta quando necessário.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

import br.upe.dsc.jpetrisim.pnml.PNMLReader;
import br.upe.dsc.jpetrisim.timed.Engine;
import br.upe.dsc.jpetrisim.timed.ITimeSimulationListener;

public class TestEngine
	implements ITimeSimulationListener
{

	Engine e;
	PNet testNet;

	/**
	* Receives as argument the name of the PNML
	* file containing the net to be simulated.
	*/
	static public void main(String[] args)
	{
		TestEngine test = new TestEngine();

		test.execute(args[0]);
	}

	public TestEngine()
	{
		e = new Engine(this);
```

```
}

public void execute(String fileName)
{
	try
	{
		//read and loads the pnml file
		e.load(PNMLReader.read(fileName));

		System.out.println("\nSimulation started...");
		System.out.println("(Type \"quit\" to quit!)\n");
		e.simulate();
	}
	catch (Exception e) { e.printStackTrace(); }
}

//Notices the event of enabling a transition called tid
public void enableTransition(String tid)
{
	System.out.println(" Enabled: "+tid);
}

//Notices the event of disabling a transition called tid
public void disableTransition(String tid)
{
	System.out.println(" Disabled: "+tid);
}

//Notices the event of modification of the number of tokens of a place
public void setTokens(String pid, int n)
{
	System.out.println(" Marking ("+pid+","+n+")");
}

/**Ask for the listener what enabled transition it wants to be fired
*    The enabled param is a Hashtable that contains entries
*          String tid -> int step
*  where tid is the name of an enabled transition and
*         step is the step of iteration when it was enabled
*/
public String fireChoice(Hashtable enabled)
{
	BufferedReader in
	= new BufferedReader(new InputStreamReader(System.in));
	String input = "";
	String defaultChoice = "";

	Enumeration i = enabled.keys();

	System.out.print("\nChoose a transition to fire: [");
	while (i.hasMoreElements())
	{
		defaultChoice = (String) i.nextElement();
		System.out.print(defaultChoice+" ");
	}
	System.out.println("]");
```

```
        System.out.print(">");
        try
        {
                input = in.readLine();
        }
        catch(IOException ioe) { ioe.printStackTrace(); System.exit(1); }

        //verifies the user input
        if (input.equals("")) input = defaultChoice;
        if (input.equals("quit")) input = null;

        //returns the input to the simulator
        return input;
}

//Notices the end of the simulation
public void finished()
{
        System.out.println("\nSimulation finished.");
}

//Notices that an error occurred in the Engine
public void error(String msg)
{
        System.out.println("An error occurred: "+msg);
}

//Notices that an unsolvable conflict was reached
public void unsolvableConflict(Vector transitionList) {
        System.out.println("The simulator could not solve the conflict:");
        System.out.println("Transitions:");
        for (int i = 0; i < transitionList.size(); i++)
        {
                System.out.println(" ["+transitionList.elementAt(i)+"]");
        }
        System.out.println("No transition was fired. Please, select one of
                        the above transitions to solve the conflict.");

}

//Notices the advance of the clock time
public void clock(int currentClock)
{
        System.out.println("clock advanced to ["+currentClock+"]");
}

/**
 * A transition have been scheduled to enable
 * in a later clockCycle
 *
 * @param tid
 * @param clockCycle
 */
public void scheduledEnable(String tid, int clockCycle)
{
```

```
		System.out.println(" scheduled ["+tid+": "+clockCycle+"]");
	}

	/**
	 * A transition have been scheduled to fire
	 * in a later clockCycle
	 * @param tid
	 * @param clockCycle
	 */
	public void scheduledFire(String tid, int clockCycle)
	{
		System.out.println(" deadline ["+tid+": "+clockCycle+"]");
	}

	/**
	 * A transition have been unscheduled at all
	 * @param tid
	 */
	public void unscheduled(String tid)
	{
		System.out.println(" unscheduled ["+tid+"]");
	}
}
```