Universidade do Estado do Rio de Janeiro
Instituto de Matemática e Estatística
Departamento de Informática e Ciência da Computação

ANALISADOR E SIMULADOR DE REDES DE PETRI

Autor: FELIPE GONÇALVES DE OLIVEIRA LINO

RIO DE JANEIRO
MAIO/2007

ANALISADOR E SIMULADOR DE REDES DE PETRI

Felipe Gonçalves de Oliveira Lino

Monografia submetida ao corpo docente do Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro - UERJ, como parte dos requisitos necessários à obtenção do grau de Bacharel em Informática e Tecnologia da Informação.

Banca Examinadora:

Prof^{o} : _____
Alexandre Sztajnberg - Orientador
Professor Adjunto IME/UERJ

Prof^{o} : _____
Paulo Eustáquio Duarte Pinto
Professor Adjunto IME/UERJ

Prof^{o} : _____
Eduardo Gonçalves Galúcio
Professor Assistente IME/UERJ

Rio de Janeiro, 16 de maio de 2007.

Ao meu Senhor Jesus Cristo.

AGRADECIMENTOS

Agradeço ao meu orientador, o professor Alexandre Sztajnberg, pela oportunidade de aprendizado durante a elaboração deste trabalho e pela experiência como monitor.

Agradeço também ao professor Paulo Eustáquio, que me auxiliou no desenvolvimento deste trabalho e me enriqueceu com seus conhecimentos em algoritmos.

Não posso esquecer também de agradecer ao professor Carlos Maziero (PUC-PR) que gentilmente cedeu o código fonte do ARP junto de sua documentação.

Com certeza minha família tem um papel importante em minha vida por isso registro aqui meus agradecimentos aos meus pais e irmãos.

Finalmente gostaria de agradecer ao apoio parcial da Faperj (APQ1 E-26/171.130/2005).

"Se, porém, algum de vós necessita de sabedoria, peça-a a Deus, que a todos dá liberalmente, e nada lhes impropera, e ser-lhe-á concedida." (Tiago 1:5)

RESUMO

As Redes de Petri têm aplicação na modelagem e verificação de várias características de sistemas operacionais como a concorrência e sincronização de processos, verificação de conflitos, compartilhamento de recursos, entre outros. Esta monografia descreve uma ferramenta que permite a descrição e a verificação de Redes de Petri com o suporte de uma interface gráfica. Esta ferramenta, baseada no ARP, foi desenvolvida em Java e emprega técnicas modernas de orientação a objetos e *design patterns*. Desta forma extensões a ferramentas, tais como, restrições temporais e a introdução de novas estratégias de verificação, podem ser "plugadas" com certa facilidade.

SUMÁRIO

Lista de Figuras

Lista de Siglas

API	Application Programming Interface
ARP	Analisador de Redes de Petri
ASCII	American Standard Code for Information Interchange
DOS	Disk Operating System
GUI	Graphical User Interface
J2SE	Java 2 Standard Edition
JDK	Java Development Kit
JRE	Java Runtime Environment
PNML	Petri Net Markup Language
RdP	Rede de Petri
SDK	Software Development Kit
UML	Unified Modeling Language
XML	Extensible Markup Language

Capítulo 1

Introdução

As Redes de Petri têm aplicação na modelagem e verificação de várias características de sistemas operacionais como a concorrência e sincronização de processos, verificação de conflitos, compartilhamento de recursos, entre outros. Esta monografia descreve uma ferramenta que permite a descrição e a verificação de Redes de Petri com o suporte de uma interface gráfica. Esta ferramenta, baseada no ARP, foi desenvolvida em Java e emprega técnicas modernas de orientação a objetos e design patterns. Desta forma extensões a ferramentas, tais como, restrições temporais e a introdução de novas estratégias de verificação, podem ser plugadas com certa facilidade.

O modelo básico da representação interna de uma rede e as verificações são baseadas no Analisador de Redes de Petri (ARP) [01]. Em nossa implementação o modelo de representação interna do ARP foi adaptado para beneficiar-se da orientação a objetos, e uma interface gráfica foi proposta para facilitar e aumentar a interatividade em todas as etapas de seu uso. O ARP foi usado como fonte de consulta através de seu código fonte aberto, nosso aplicativo não faz uso das funcionalidades do ARP. Procuramos incluir em nossa ferramenta as funcionalidades existentes no ARP.

1.1. Motivação

O formalismo matemático associado às Redes de Petri possibilita a verificação de propriedades dos sistemas modelados. A capacidade de verificarem-se a presença ou ausência dessas propriedades em uma rede modelada permite a análise do sistema-alvo modelado. Em nossa ferramenta, além do suporte à edição gráfica de Redes de Petri, várias destas propriedades podem ser verificadas.

Em [02] é mantido um repositório sobre Redes de Petri. Entre as informações mantidas, uma lista de ferramentas acadêmicas e comerciais é disponibilizada, juntamente com uma análise das características relevantes de cada uma delas.

Em nossa ferramenta procuramos aliar a interatividade, desde o editor até o "*token animation game*", com a capacidade de simulação passo a passo, geração da

árvore de alcançabilidade, e a verificação de várias características. Tais capacidades não estão presentes, juntas, em todas as ferramentas disponíveis. Por outro lado, no estado atual, nossa ferramenta ainda não contempla a simulação e análise de Redes de Petri com características temporais (embora o arcabouço necessário esteja preparado – veja Seção 4.4), estocásticas ou de predicados no disparo de transições. Além de oferecer maior interatividade através da interface gráfica, procuramos facilitar a extensão através dos mecanismos de extensão de orientação a objetos (herança) e Java (reflexão estrutural). As redes editadas em nossa ferramenta são persistidas em XML, o que facilita também a interoperabilidade com outras ferramentas para Redes de Petri.

O texto está organizado da seguinte forma: no Capítulo 2 falamos sobre redes de Petri e os conceitos envolvidos; o Capítulo 3 descreve trabalhos relacionados, com destaque para o ARP; nossa implementação é tratada no Capítulo 4; e nossas considerações finais são feitas no Capítulo 5. Incluímos ainda o Apêndice A com o manual da ferramenta e o Apêndice B descrevendo as classes auxiliares.

Capítulo 2

Redes de Petri

2.1 Conceitos Básicos

Através do trabalho de Carl Adam Petri [03], surgiram as Redes de Petri (RdP) que trata se de um grafo bipartite, composto por três peças chaves: lugares, transições e arcos, que inicialmente teve sua aplicação no estudo de autômatos [04]. Sua aplicação se estendeu para modelagem de sincronização de processos, concorrência, conflitos, partilha de recursos, entre outros, devido a potencialidade de modelagem das RdP.

Os lugares são representados no grafo sob a forma de círculos e possuem fichas. No desenho as fichas são representadas por valores inteiros escritos dentro do círculo ou tem em seu interior pequenos círculos preenchidos. A distribuição de fichas nos lugares, chamado de marcação, é o que caracteriza o estado da rede. São classificados como de entrada ou saída. Os de entrada são aqueles ligados a uma transição por um arco de entrada, enquanto os de saída são aqueles ligados por arcos de saída.

As transições são representadas sob a forma de retângulos. São os responsáveis por alterar o estado da rede. Nas RdP com temporização as transições possuem atributos de tempo.

Os arcos são representados sob a forma de setas direcionadas que ligam os lugares às transições. Vale observar que os arcos não fazem ligações entre lugares nem entre transições. Podem ser classificados como sendo de entrada ou saída. Os de entrada são os que têm origem em um lugar e apontam para uma transição, enquanto os de saída, são os que têm origem em uma transição e apontam para um lugar. Possuem como atributo o peso. Por *default* os arcos possuem peso unitário. O peso pode ser qualquer valor inteiro maior que um. No desenho o peso é representado por um valor inteiro escrito ao lado do arco. Quando o arco não possui nenhum valor escrito ao seu lado, ou sobre ele, assume-se que o valor do peso é um.

As RdP modelam basicamente dois aspectos de um sistema: eventos e

condições, bem como as relações entre eles. Através dessas características é possível observar certas condições em cada estado do sistema. Estas por sua vez, podem possibilitar a ocorrência de eventos que podem alterar o estado do sistema.

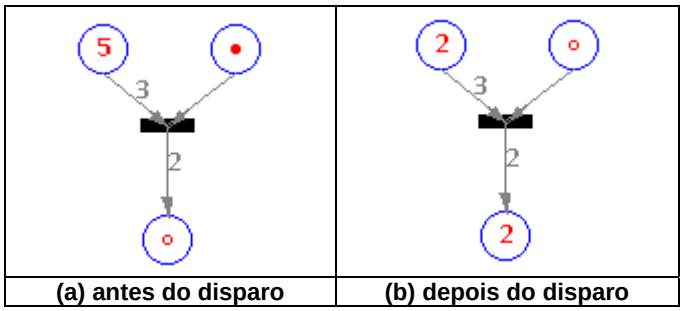
É muito comum, na modelagem com RdP associar os lugares às condições ou estados, e as transições aos eventos ou ações. Porém, essa abordagem não é obrigatória. Seria também possível associar ações e eventos aos lugares, de uma maneira dual ou mesmo associá-los tanto a lugares como a transições, obtendo um sistema misto [05].

2.2 Regras de Execução

Como já foi dito, as Redes de Petri possuem estado e a ocorrência de eventos altera o estado do sistema. Explicaremos sob quais condições uma ação pode ser executada sobre a rede.

Uma ação pode ser executada quando existem transições habilitadas para disparo. Uma transição pode disparar quando todos os seus lugares de entrada possuírem fichas em quantidade superior ou igual ao peso do arco que o liga até a transição.

Ao disparar uma transição a marcação dos lugares sofre modificação da seguinte maneira: para todo lugar de entrada da transição disparada é removida a quantidade de fichas equivalente ao peso do arco de entrada, enquanto para todo lugar de saída é adicionada a quantidade de fichas equivalente ao peso do arco de saída.



(a) antes do disparo | (b) depois do disparo

Figura 2.1 Disparo de Transições

A Figura 2.1 mostra no lado esquerdo a rede antes do disparo da transição, no lado direito após o disparo. Antes do disparo um dos lugares de entrada possui cinco fichas enquanto o outro possui uma ficha (círculo pequeno preenchido). O lugar de saída não possui fichas (círculo pequeno vazio). Após o disparo um lugar

de entrada fica com duas fichas e o outro com nenhuma ficha. O lugar de saída com duas fichas.

2.3 Propriedades Básicas

O formalismo matemático possibilita a análise precisa dos modelos de sistemas, com verificação de propriedades inerentes, como relações de precedência entre eventos, sincronização e bloqueio. Algumas propriedades são:

1. Uma rede é *limitada* se, para todas as marcações acessíveis, a partir de uma marcação inicial, o número de fichas em qualquer lugar da rede não exceder *K* (inteiro).
2. Uma rede é *segura* se todos os seus lugares são seguros. Um lugar é seguro se seu número de fichas não exceder a *K*=1. Segurança é um caso especial da propriedade de limitação.
3. Uma rede é *reinicializável* ou *própria* se, para todas as marcações possíveis da rede, existir uma seqüência de disparos que leve a rede à marcação inicial.
4. Uma rede é *conservativa* se o número total de fichas na rede se mantém.
5. Uma rede é *viva* quando todas as transições são vivas. Uma transição é viva se para toda marcação alcançável, existe uma seqüência de disparos, tal que a mesma torne-se habilitada.
6. Ocorre um *bloqueio* na rede quando uma transição ou conjunto de transições não dispara. Um caso especial de bloqueio é o *deadlock*, quando nenhuma transição está habilitada para disparo.

2.4 Representação Matricial

As Redes de Petri podem ser representadas sob a forma de duas matrizes e três vetores. Uma RdP com marcação inicial μ_0 é definida por:

$$RP = (P , T , I , O , \mu_0) ,$$

onde μ_0 é um vetor com componentes inteiras não negativas e sua dimensão é o número de lugares da rede:

$$\mu_0 = (\mu_0^1, \mu_0^2 , ... , \mu_0^n).$$

As componentes μ_0^i , representam o número inicial de fichas no lugar correspondente (Pi).

O vetor P representa os lugares da rede. Possui como dimensão o número de lugares da rede.

O vetor T representa as transições da rede. Possui como dimensão o número de transições da rede.

A matriz ***I*** é a que representa a função de entrada, enquanto a matriz ***O*** representa a função de saída. Em ambas as matrizes cada **linha i** está associada ao **lugar i**, enquanto a **coluna j** está associada à **transição j**. Cada matriz tem ***m*** linhas (uma para cada transição) por ***n*** colunas (uma para cada lugar). Na matriz ***I***, as linhas estão associadas às transições de entrada e as colunas aos lugares de destino. Na interação linha/coluna é colocado o peso do arco de entrada. Na matriz ***O***, as linhas estão associadas às transições de saída e as colunas aos lugares de destino. Na interação linha/coluna é colocado o peso do arco de saída

Define-se a matriz de incidência ***D*** associada a uma RdP, com base na sua estrutura, como $\boldsymbol{D = O - I}$, em que ***O*** e ***I*** são as matrizes de entrada e saída. Com base na informação contida nesta matriz ***D*** é possível representar as alterações na marcação da rede provocadas pelo disparo de uma transição.

As características da matriz de incidência são: o elemento $\boldsymbol{D_{ij}}$ representa a variação no número de fichas em $\boldsymbol{P_i}$ quando é disparada a transição (habilitada) $\boldsymbol{T_j}$. A coluna ***j*** representa a diferença entre os estados final e inicial da rede de Petri provocada pelo disparo da transição $\boldsymbol{T_j}$ (ou seja, as colunas são associadas às transições). As linhas estão associadas aos lugares e representam seus ganhos ou perdas em função do disparo de qualquer transição.

Como exemplo vamos construir os vetores P, T e μ_0, e as matrizes *I, O e D* da rede abaixo.

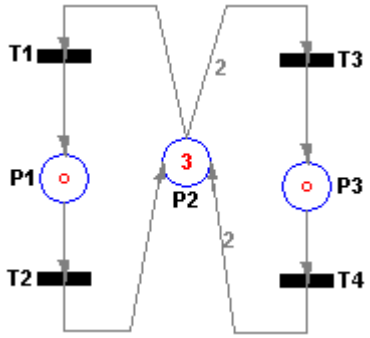


Figura 2.2 Representação Matricial: RdP de exemplo

P = (P1, P2, P3),

T = (T1, T2, T3, T4)

μ_0 = (0, 3 , 0)

$$I = \begin{array}{c|ccc} & P1 & P2 & P3 \\ T1 & 0 & 1 & 0 \\ T2 & 1 & 0 & 0 \\ T3 & 0 & 2 & 0 \\ T4 & 0 & 0 & 1 \end{array} \qquad O = \begin{array}{c|ccc} & P1 & P2 & P3 \\ T1 & 1 & 0 & 0 \\ T2 & 0 & 1 & 0 \\ T3 & 0 & 0 & 1 \\ T4 & 0 & 2 & 0 \end{array} \qquad D = (O\text{-}I) = \begin{array}{|ccc|} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & -2 & 1 \\ 0 & 2 & -1 \end{array}$$

2.5 Árvore de Alcançabilidade

Árvore de Alcançabilidade, também chamada de acessibilidade ou ocorrência, é a construção do espaço de estado associado á execução da rede. Neste grafo todos os estados e transições são representados individualmente permitindo, para o caso das RdP limitadas, em que o grafo é finito, verificar todas as propriedades de interesse por simples inspeção. Para o caso mais geral de redes não limitadas, em que o grafo não é finito, é utilizada uma técnica para permitir sua representação finita, utilizando-se o termo árvore de cobertura.

Dado uma marcação inicial da rede, este grafo poderá ser construído tendo por base todas as seqüências de disparos de transições possíveis. Cada marcação será representada por um nó, enquanto cada arco de interligação será associado à transição que lhe deu origem.

Como a árvore de marcações acessíveis associadas a redes de Petri não limitadas não é finita, aplica-se à sua representação um corte de ramos, possibilitando a utilização de um número finito de nós para representar a execução

da rede. Representa-se um número infinito de marcações que resultam a partir de outra, usando um símbolo especial ϖ e que significa um número de fichas que se tornar arbitrariamente grande, resultando da execução de múltiplos ciclos em que se verifica o acréscimo em determinado lugar da rede.

Para uma constante x qualquer, define-se operações para construção da árvore de alcançabilidade:

$$\varpi + x = \varpi$$

$$\varpi - x = \varpi$$

Antes de vermos o algoritmo de construção da árvore de alcançabilidade temos que definir quando uma marcação é maior que outra, pois essa informação será necessária para a construção da árvore.

Diz-se que a marcação μ_n é estritamente maior que a marcação μ_m , isto é: $\mu_n > \mu_m$, se e somente se,

com M = Número de lugares da rede,

$$\forall i : i \in [1,M] , \mu_n^i >= \mu_m^i$$

e

$$\exists j : j \in [1,M] , \mu_n^j > \mu_m^j.$$

<u>Algoritmo</u>

Seja X nó a ser processado:

1) Se existe um nó Y na árvore que não é fronteira e tem a mesma marcação que X, ou seja, μ(x) = μ(y), então x é um nó ***<u>duplicado</u>***.
2) Se não existem transições sensibilizadas associadas à marcação μ(x), ou seja {δ (μ(x),t_j) é indefinida $\forall$ $t_j \in$ T}; então x é um nó ***<u>terminal</u>***.
3) Para todas as transições $t_j \in$ T, que são habilitadas em μ[x], cria-se um novo nó *N*, na árvore de alcançabilidade, ligados ao primeiro através de um arco ao qual se associa a transição disparada. Para cada novo nó desenvolvido escreveremos o símbolo ϖ na componente i de μ_n se $\mu_n > \mu_m$ e $\mu_n^i > \mu_m^i$ para alguma marcação μ_m já existente no caminho de μ_n até a raiz da

árvore. Um arco, designado t_j, é dirigido do nó X para o nó N. O nó X é redefinido como nó *interior* e N torna-se um nó *fronteira*.

O primeiro nó a ser processado é o que representa o estado inicial. A partir dele vão sendo criados novos nós. Quando todos os nós tiverem sido classificados como terminal, duplicado ou interior, o algoritmo pára.

2.6 Elementos de Modelagem

As RdP são usadas principalmente para modelagem. Diversos tipos de sistemas, tais como hardware, software, sistemas físicos, sistemas de produção, especialmente aqueles com componentes independentes, podem ser modelados por uma RdP. As redes são usadas para modelar a ocorrência de vários eventos ou atividades de um sistema, fluxo de informações ou outros recursos e permitem a visualização de diversos conceitos e relações, designadamente : paralelismo e concorrência, partilha de recursos, sincronização, memorização, limitação de recursos e leitura. As RdP não oferecem apenas uma representação para estrutura e funcionamento de um sistema. Possibilitam também a visualização do comportamento do sistema através do movimento das fichas.

2.6.1 Conceitos Básicos

Inicialmente vamos definir dois conceitos básicos da modelagem:

- Eventos: São as ações que ocorrem no sistema e são controladas por seu estado.
- Condições: São os predicados ou descrições lógicas dos estados do sistema. Como tais, elas podem ser verdadeiras ou falsas.

Para a ocorrência de um determinado evento, certas condições precisam ser satisfeitas. São as pré-condições para o evento.

A ocorrência de um evento pode encerrar as pré-condições e causar a ocorrência de outras condições no sistema. São as chamadas pós-condições.

2.6.2 Elementos

Agora veremos alguns dos principais elementos que compõe a modelagem com Redes de Petri.

- **Paralelismo**

As RdP clássicas não possuem a noção de tempo, ou seja, não fica explicitado na rede quando as coisas vão acontecer, ou se elas vão acontecer. A rede apenas informa a possibilidade da ocorrência informando as condições e que eventos podem ocorrer.

O paralelismo ocorre justamente quando, ao mesmo tempo, dois ou mais eventos podem ocorrer. Isso é representado quando várias transições podem disparar ao mesmo tempo. A Figura 2.3 mostra duas transições que podem disparar ao mesmo tempo ou em qualquer ordem.

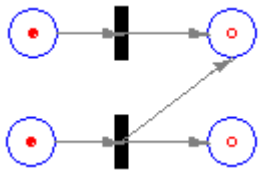


Figura 2.3 Paralelismo

- **Conflito**

O conflito pode ser modelado por transições que possuem pelo menos um lugar de entrada em comum. As transições da rede da Figura 2.4 estão em conflito pois o disparo de qualquer uma retira a ficha do lugar em comum, desabilitando a outra transição.

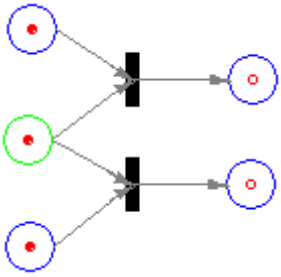


Figura 2.4 Conflito: duas transições em conflito com o lugar comum no meio

- **Sincronização (exclusão mútua)**

Quando a solução de um problema requer a cooperação de processos, faz-se necessário o compartilhamento de informações ou recursos entre os processos. O compartilhamento deve ser controlado para assegurar a correta operação do sistema, através de uma função de sincronismo.

A exclusão mútua é uma técnica para definir um código de entrada e saída de modo que, no máximo, um processo tenha acesso a uma informação compartilhada em um determinado tempo. O código que acessa o objeto e precisa de proteção contra interferência do outro processo é chamado de seção crítica.

Ao entrar na seção crítica o processo bloqueia o acesso e executa sua função, abandona a seção crítica e desbloqueia sua entrada.

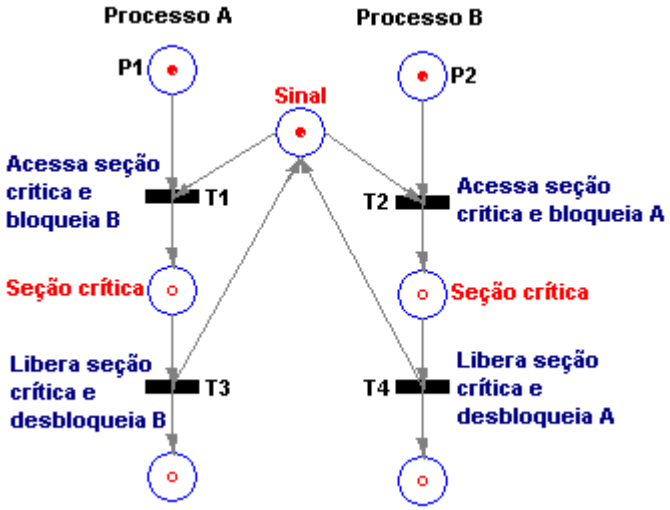


Figura 2.5 Exclusão mútua

Veja a Figura 2.5. O lugar "Sinal" representa a permissão para entrar na seção crítica. Para isso um processo precisa obter a ficha em P1 ou P2, sinalizando que vai entrar na seção critica e obter a ficha de "Sinal". Somente uma transição pode disparar. O disparo de T1 desabilita T2 e estabelece uma espera para o processo B, até o processo A sair da região crítica, disparando a transição T3, e repondo a ficha em "Sinal".

2.7 Exemplos de Modelagem

Nas três subseções a seguir vamos modelar três sistemas, um atelier simples e mais dois exemplos clássicos de Sistemas Operacionais: Leitores e Escritores, e Produtores e Consumidores.

2.7.1 Atelier Simples

Vamos modelar agora um sistema simples composto de um torno e um robô que o alimenta, carregando e descarregando peças de um armazém [01]. Primeiro veremos qual o significado dos lugares na modelagem:

- P0 – Representa o armazém. A quantidade de fichas é o número de peças.

- P1 – Fichas em P1 significa que o robô está carregando.
- P2 – Fichas em P2 significa que a peça está sendo trabalhada na usina.
- P3 – Fichas em P3 significa que o robô está livre para operar.
- P4 – Fichas em P4 significa que o torno está livre.
- P5 – Fichas em P5 significa que o robô está descarregando.

  Em segundo veremos qual o significado das transições:
- T0 – Robô pega peça nova.
- T1 – Robô carrega o torno.
- T2 – Robô descarrega o torno.
- T3 – Robô solta a peça pronta.

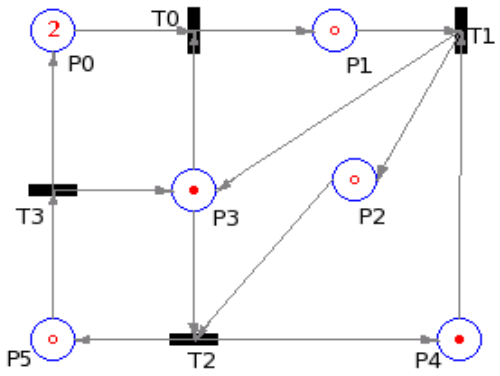


Figura 2.6 Armazém no estado inicial

No estado inicial (Figura 2.6) vamos usar a marcação (2,0,0,1,1,0). Cada posição no vetor é um lugar, variando de P0 a P5. Pela marcação inicial temos que o armazém possui 2 peças, o robô está pronto para operar e o torno está livre.

Ao dispararmos a seguinte seqüência de transições: T0, T1, T2, T3; estamos fazendo com que o sistema se comporte fazendo com que o robô pegue uma peça nova, a coloque no torno, depois descarregue o torno e finalmente solte a peça pronta no armazém, voltando assim para o estado inicial.

Se dispararmos a seguinte seqüência de transições: T0, T1, T0; chegaremos a um estado de *deadlock* (Figura 2.7), pois fizemos com que o robô tomasse as ações de pegar uma peça nova, colocá-la no torno e pegasse outra peça nova enquanto o torno estava ocupado.

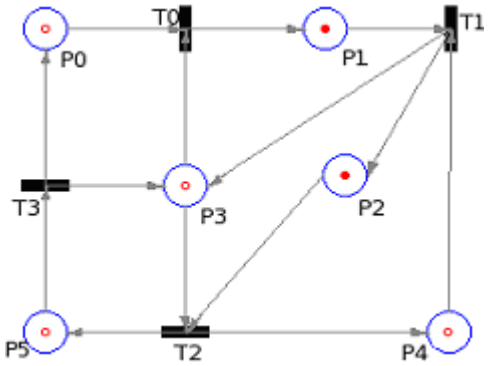


Figura 2.7 Sistema em estado de *deadlock*

A árvore de alcançabilidade dessa rede é mostrada na Figura 2.8.

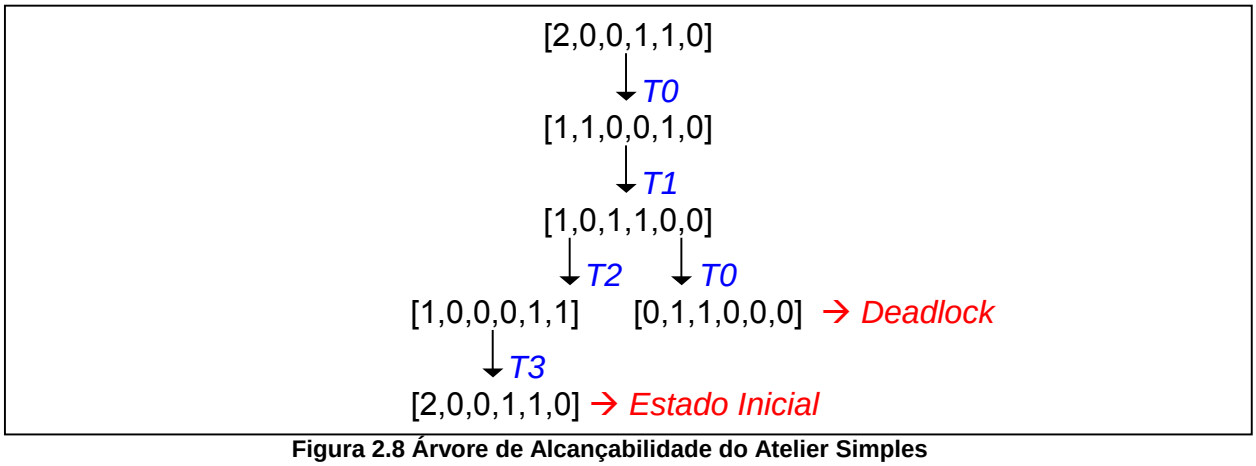


Figura 2.8 Árvore de Alcançabilidade do Atelier Simples

Podemos verificar as propriedades da Rede de Petri nesse exemplo. Ela é limitada, pois a quantidade de fichas máxima encontrada em algum lugar da rede é 2; não é segura, pois o limite é 2; não é reinicializável, pois para a marcação de *deadlock* obviamente não existe seqüência de disparos que a faça voltar para o estado inicial; não é conservativa pois a quantidade de fichas na rede varia; não é viva, pois nem todas as transições são vivas (*deadlock*); e finalmente verificou-se que ocorre bloqueio na rede.

2.7.2 Leitores e Escritores

Neste problema múltiplos leitores e escritores compartilham uma mesma base de dados (região crítica). Deseja-se manter o estado consistente dos registros. Portanto, as mudanças de estado devem ser isoladas. Para isso, os processos escritores devem examinar e atualizar a base de dados com acesso exclusivo e qualquer número de processos leitores pode usar a base quando não houver escritor a alterando.

Em resumo, um processo escritor deve ter acesso exclusivo à região crítica, ou seja, quando ele estiver escrevendo nenhum outro processo pode estar escrevendo ou lendo a base de dados. Por outro lado, quando um processo leitor está lendo a base de dados, outros leitores também podem acessá-la.

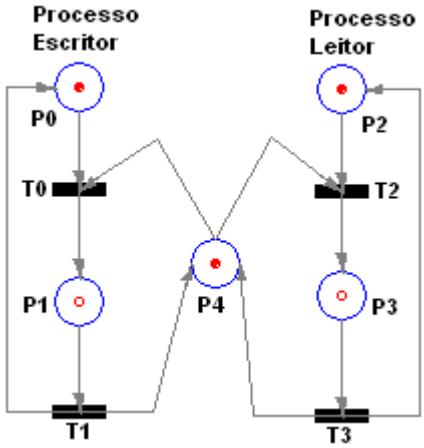


Figura 2.9 Rede Leitor e Escritor no estado inicial

Em nosso modelo simplificado temos apenas um processo escritor e um processo leitor. Os significados dos lugares, quando possuem fichas, e das transições, quando habilitadas, são:

- P0 – Processo Escritor pronto para escrever.
- P1 – Processo Escritor escrevendo.
- P2 – Processo Leitor pronto para ler.
- P3 – Processo Leitor lendo.
- P4 – Base de dados livre.
- T0 – Iniciar escrita.
- T1 – Terminar escrita.
- T2 – Iniciar leitura.
- T3 – Terminar leitura.

A Figura 2.10 mostra a rede no estado inicial onde o escritor está pronto para escrever, o leitor pronto para ler e a área de memória livre.

Construindo a Árvore de Alcançabilidade da rede temos:

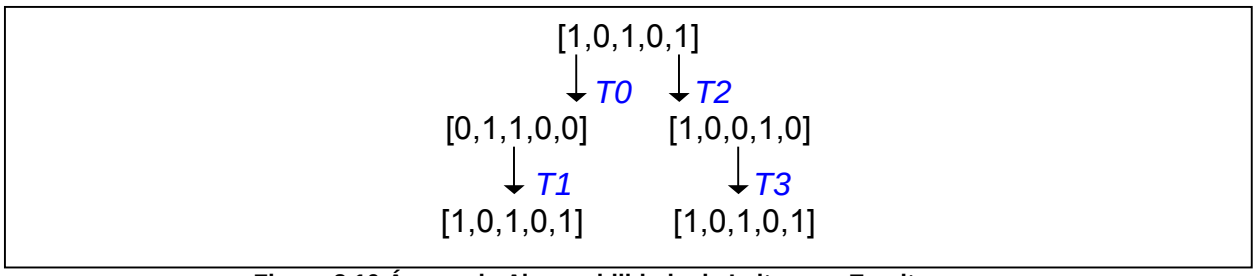


Figura 2.10 Árvore de Alcançabilidade de Leitores e Escritores

Os nós terminais da árvore são duplicatas do estado inicial, concluímos então que a rede é viva e reinicializável. Como a maior quantidade de fichas encontrada em algum lugar da rede é 1 então dizemos que ela é limitada e seu limite é 1. Limite 1 é caso especial da limitação. Significa que a rede é segura. Ela não é conservativa pois o total de fichas varia entre 2 e 3. Essa rede não possui nenhum estado de bloqueio, pois sempre é possível disparar alguma transição.

2.7.3 Produtores e Consumidores

Este é outro problema clássico de Sistemas Operacionais. O problema consiste de dois tipos de processos: os produtores e os consumidores. Ambos os processos acessam o mesmo *buffer*, o produtor coloca objetos no *buffer* enquanto o consumidor retira objetos do *buffer*.

Como regra, o consumidor não pode remover objetos do *buffer* caso este esteja vazio, o que causaria *underflow*. Caso consideremos que o *buffer* possui limite, o produtor não pode ser capaz de acrescentar itens no buffer quando o buffer estiver completamente cheio, o que causaria *overflow*.

A diferença deste problema para o de “Leitores e Escritores” está mais na definição, aqui o processo consumidor só pode “consumir” após o produtor ter escrito algo na memória (ou base de dados), ou seja, é um problema de sincronização. Enquanto no problema de “Leitores e Escritores” o objetivo é fazer com que processos distintos tenham acesso exclusivo a área de memória, ou seja, é um problema de exclusão mútua.

Em nosso modelo temos um único processo produtor e um único processo consumidor. Descrevemos abaixo o significado dos lugares e transições, quando aqueles possuem fichas e estes últimos estão aptos para disparo.

- P0 – Produtor pronto para produzir.

- P1 – Produtor pronto para armazenar no *buffer*.
- P2 – Consumidor pronto para remover do *buffer*.
- P3 – Consumidor pronto para consumir.
- P4 – *Buffer*.
- T0 – Produzir.
- T1 – Armazenar no *buffer*.
- T2 – Remover do *buffer*.
- T3 – Consumir.

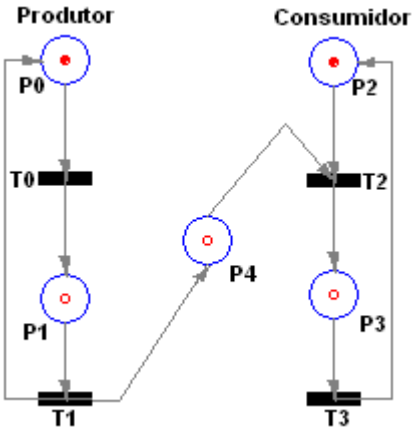


Figura 2.11 Rede Produtor e Consumidor no estado inicial

A Figura 2.11 mostra a rede no estado inicial, onde o produtor está pronto para produzir, o consumidor pronto para consumir e o buffer está vazio.

A Árvore de Alcançabilidade da rede é mostrada na Figura 2.12. Através dela podemos efetuar a análise.

Vemos, nesse exemplo, que o *buffer* é o lugar em que foi verificado um crescimento grande de fichas. Podemos conferir isso, se fizermos a simulação da rede executando repetidamente os disparos T0 e T1 (produzir e armazenar no *buffer*, respectivamente). Se considerássemos que o *buffer* possui algum limite, esse fato verificado nos diria que deveria ocorrer *overflow.* Percebemos também que a situação de *underflow* nunca ocorre, pois a transição de remoção de objetos do *buffer* nunca está habilitada quando ele está vazio.

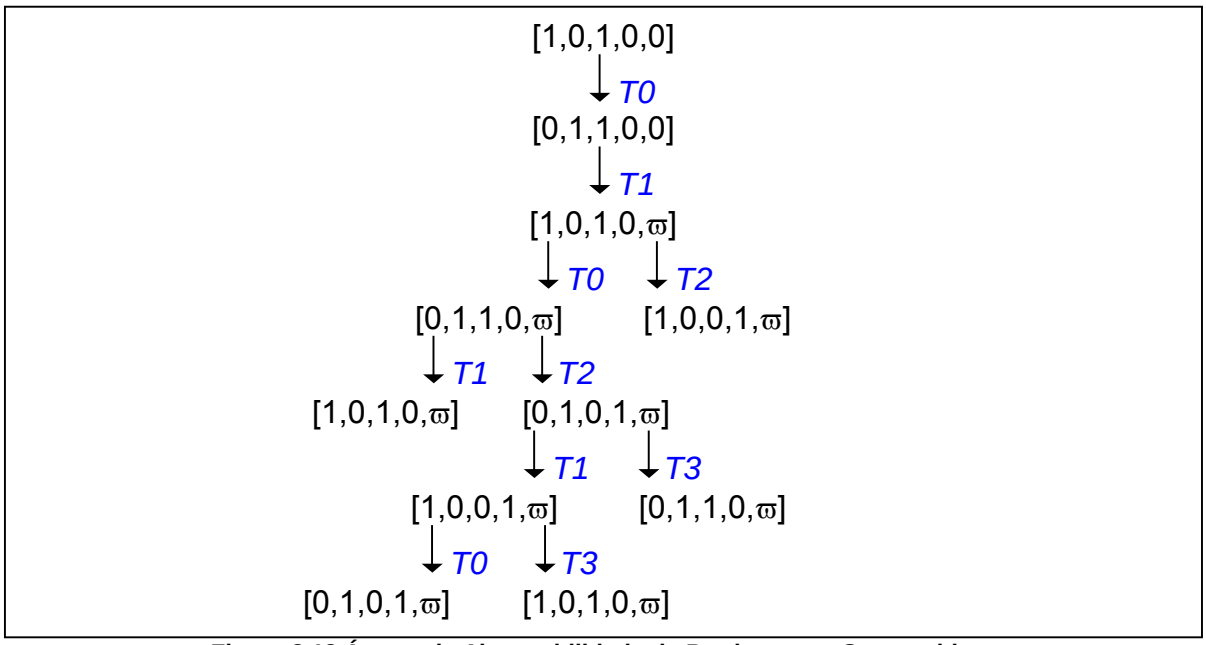


Figura 2.12 Árvore de Alcançabilidade de Produtores e Consumidores

Outras propriedades que encontramos são que a rede é viva não limitada e não conservativa. Podemos observar que a construção da árvore de alcançabilidade reduzida pode fazer desaparecer alguma informação e assim, não poderemos, em geral, determinar, através da árvore, se uma rede ilimitada é viva e própria.

Capítulo 3

Trabalhos Relacionados

Em [02] é mantido um repositório sobre Redes de Petri. Entre as informações mantidas, uma lista de ferramentas acadêmicas e comerciais é disponibilizada, juntamente com uma análise das características relevantes de cada uma delas.

Neste capítulo vamos tratar de alguns trabalhos relacionados ao nosso e em destaque a ferramenta ARP que usamos como base para o desenvolvimento do nosso *software*.

3.1 JARP

O programa JARP ainda está em desenvolvimento em um projeto disponível no *Source Forge* administrado por Ricardo Padilha [05]. Tem como objetivo inicial ser um auxiliar para a ferramenta de análise de redes de Petri ARP que foi desenvolvida no LCMI do Departamento de Engenharia Elétrica da Universidade Federal de Santa Catarina.

Segundo [05], suas principais características são:

- Composição visual de redes de Petri;
- Simulação interativa de redes;
- Exportação de redes para o formato de arquivo do ARP;
- Exportação de para o formato padronizado PNML (Petri Net Markup Language - baseada em XML);
- Exportação para os formatos gráficos GIF, JPEG, PNG e PPM.

As expansões previstas são:

- A implementação dos algoritmos básicos de análise de redes de Petri (componentes conservativos e repetitivos, marcações acessíveis, árvore de cobertura, propriedades da rede...);
- A possibilidade de importar redes a partir de arquivos no formato ARP (o formato ARP não define informações de posição dos lugares e transições).

Por ser ainda uma ferramenta em desenvolvimento, algumas de suas funcionalidades não estão completamente implementadas, tais como: a exportação da rede para diversos formatos e a simulação, que ainda não permite que se possa voltar a estados anteriores.

A Figura 3.1 mostra a rede Produtores e Consumidores desenhada na ferramenta JARP.

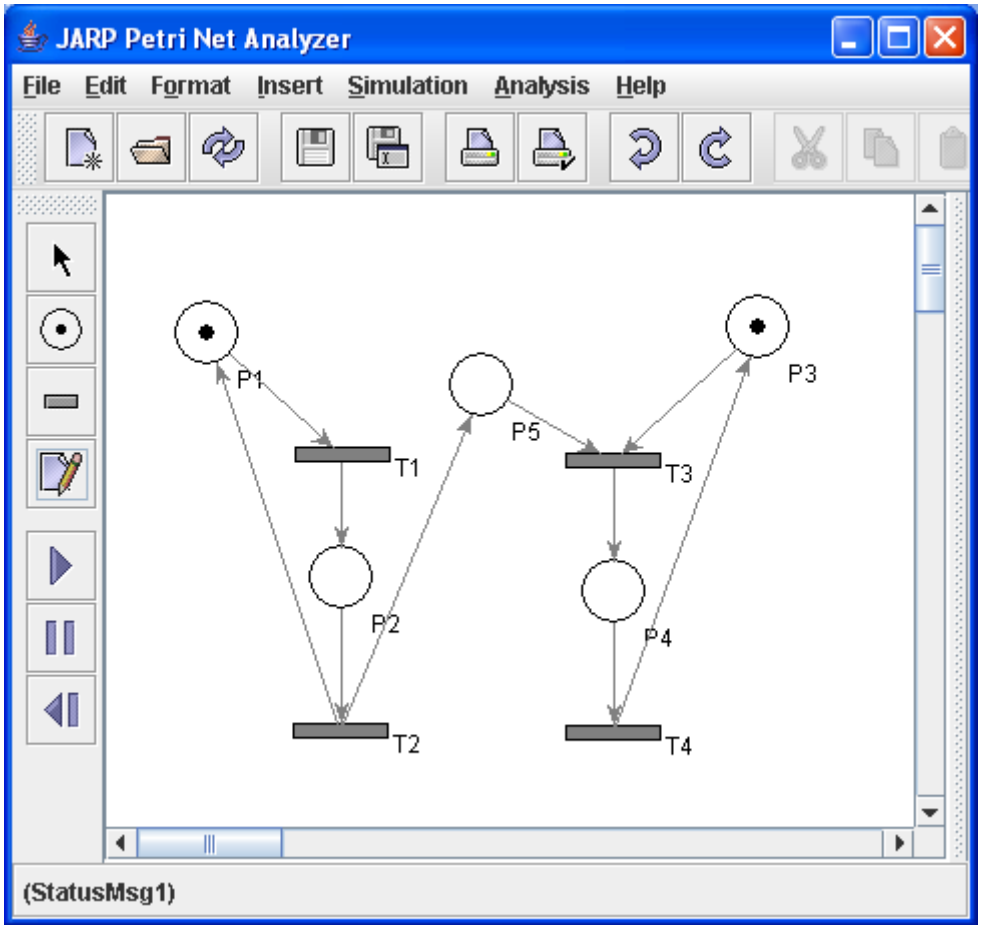


Figura 3.1 Ferramenta JARP

3.2 PetriTool

Esta ferramenta está disponível em [06] e é tratada em [07]. Ela possui um editor gráfico para o desenho de RdP. É de fácil manuseio por parte do usuário, possui ainda a possibilidade de simular a execução da rede e verificar propriedades usando a técnica da árvore de alcançabilidade. A ferramenta constrói a árvore de alcançabilidade, analisa então a árvore verificando propriedades como: segurança, limitação, vivacidade, e conservação.

Uma observação a ser feita é que a ferramenta não produz resultados gráficos na geração da árvore de alcançabilidade. Como exemplo, a Figura 3.2 mostra a tela do programa com a RdP Leitores e Escritores com sua árvore de alcançabilidade e análise correspondente.

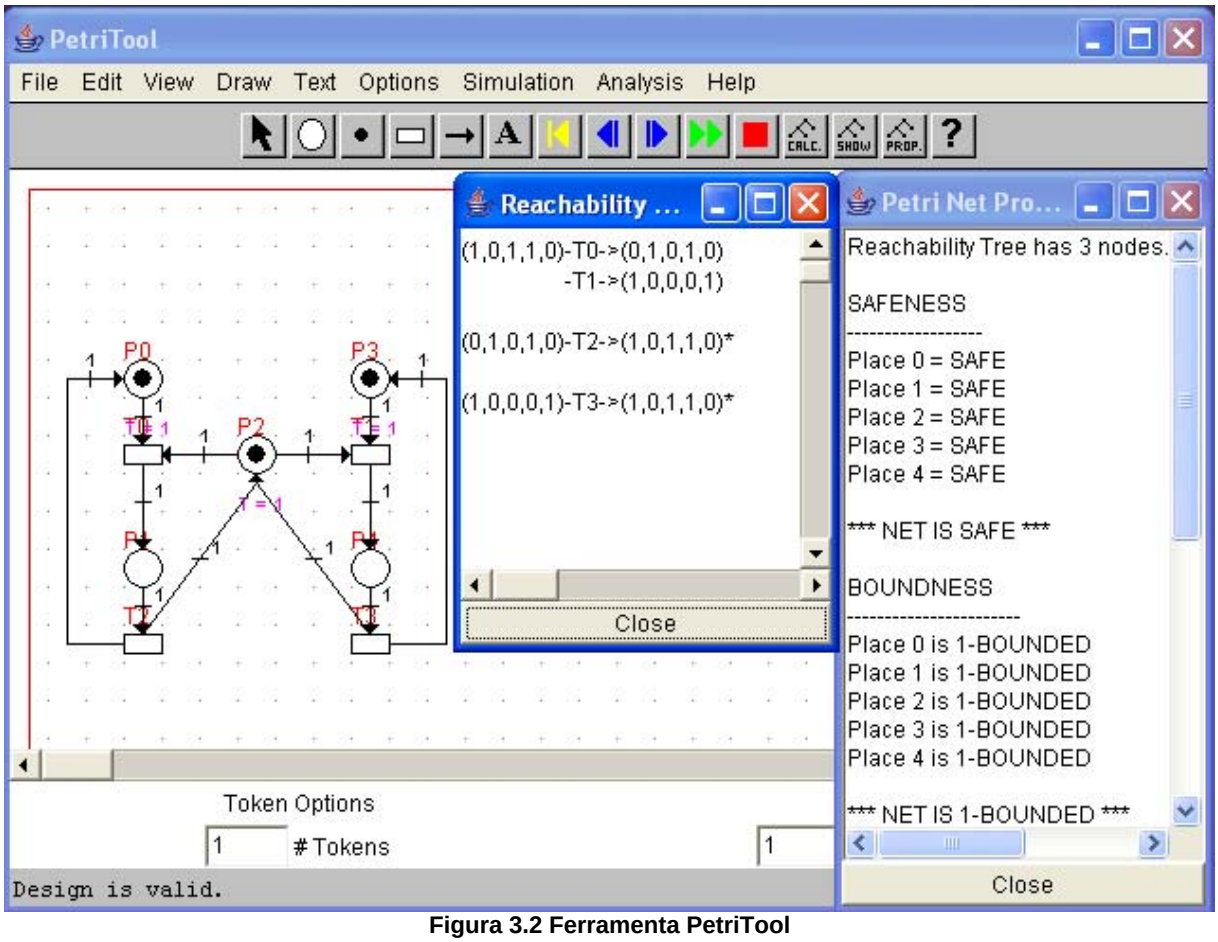


Figura 3.2 Ferramenta PetriTool

3.3 jPNS

Este programa é um *applet* disponível em [08]. Foi desenvolvido por estudantes da universidade de Stuttgart na Alemanha. O foco do programa é a edição e a simulação da rede de petri.

A ferramenta permite que a simulação seja automática, onde o programa dispara as transições disponíveis de maneira aleatória e vai exibindo as marcações da rede; ou simulação interativa, onde o usuário seleciona as transições que deseja disparar.

O ponto fraco desta ferramenta é que ela não faz nenhuma análise sobre a rede.

A Figura 3.3 mostra a rede que modela o problema dos filósofos desenhada no programa jPNS.

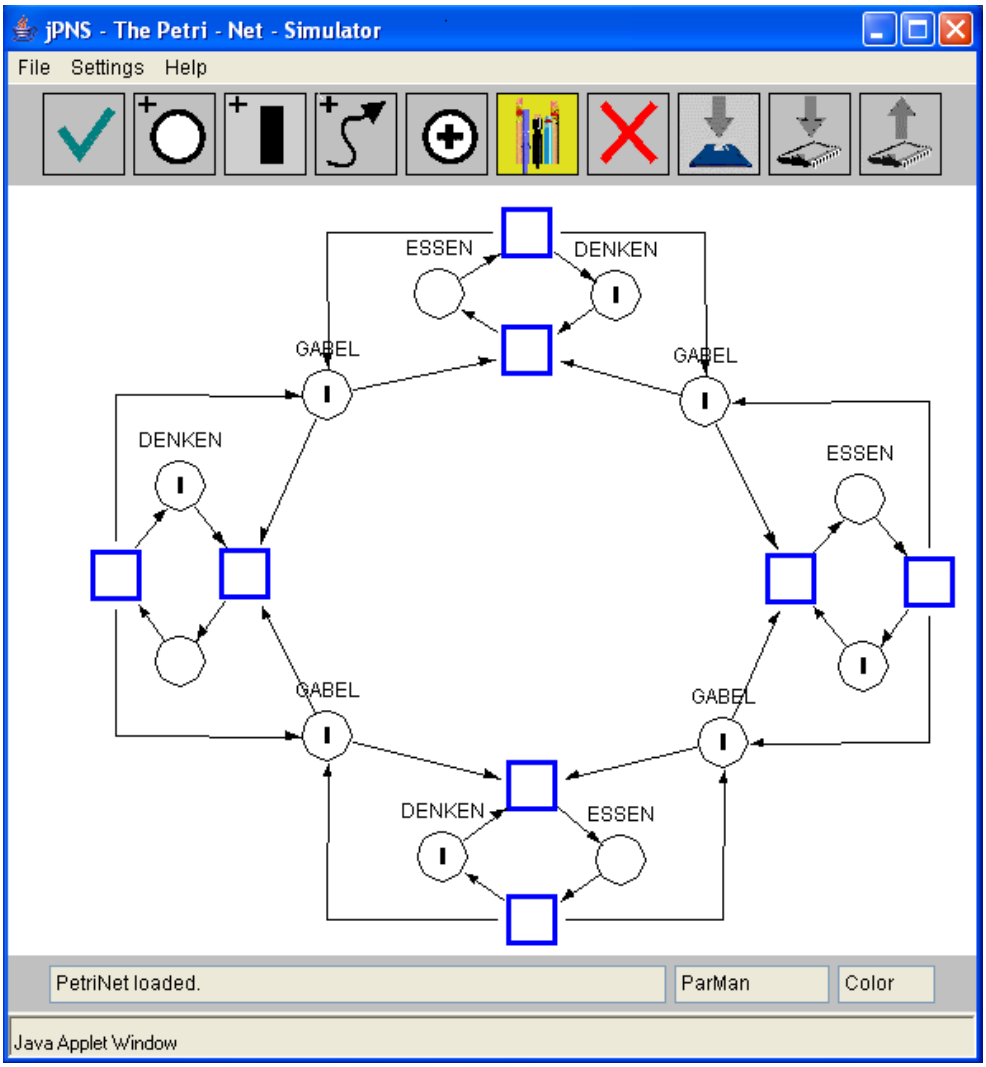


Figura 3.3 Ferramenta jPNS

3.4 ARP

ARP é o Analisador/Simulador de Redes de Petri desenvolvido no Laboratório de Controle e Microinformática (LCMI) do Departamento de Engenharia Elétrica da Universidade Federal de Santa Catarina, entre 1985 e 1990. As informações contidas no texto a seguir foram extraídas de [01] e [09], onde podem ser encontrados mais detalhes.

Foi construído em Pascal de forma modular, permitindo facilmente projetar e conectar novas ferramentas de análise ou tratamento de Redes de Petri. Roda sobre o sistema operacional DOS 3.0 ou superior. É um programa para o auxílio ao projeto com RdP, contando com várias ferramentas para RdP ordinárias, com temporização

e com temporização estendida.

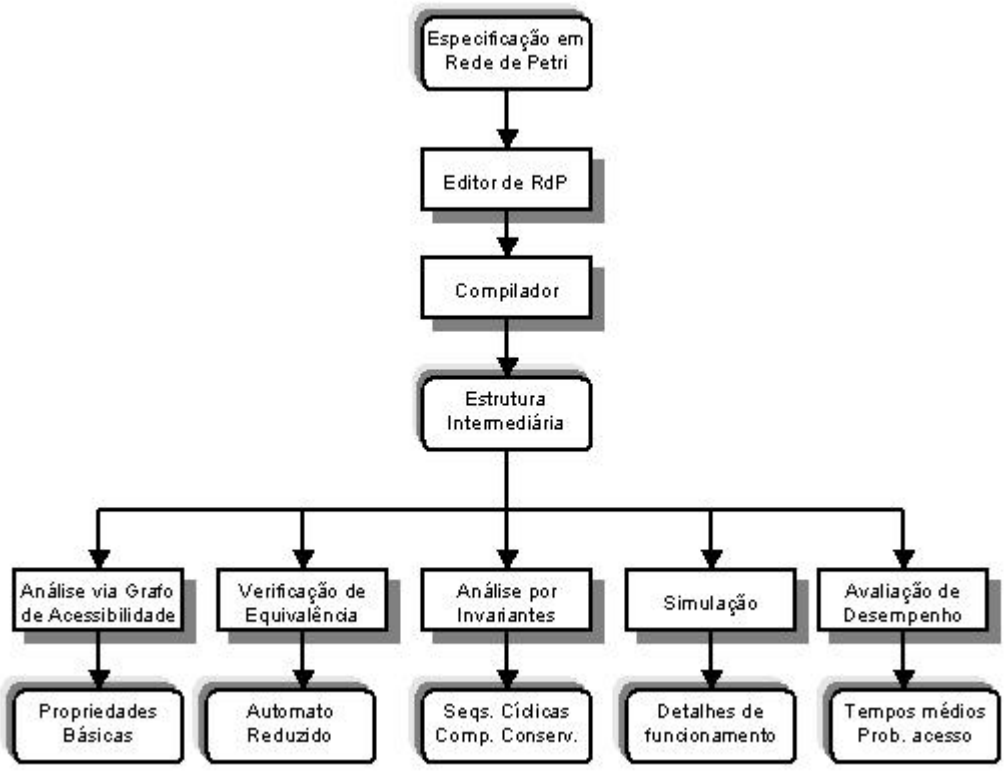


Figura 3.4 Arquitetura do Sistema ARP

A Figura 3.4 mostra a arquitetura do sistema ARP. Sua construção modular permite o desenvolvimento e a rápida inserção de novas ferramentas, pois todo acesso a dispositivos (vídeo, teclado, arquivos) está a cargo de interfaces padrão.

A entrada de uma RdP no sistema ARP efetua-se através de uma linguagem de descrição, que emprega uma sintaxe semelhante à descrição de dados em Pascal.

A descrição da rede na linguagem de entrada é posteriormente compilada na forma de uma estrutura intermediária que contém a descrição da rede e que servirá como entrada das demais ferramentas.

A partir dessa estrutura intermediária e de entradas específicas, cada ferramenta realiza sua operação e produz como saída um texto contendo os resultados obtidos.

3.4.1 Ferramentas do Programa

O ARP é composto de várias ferramentas. As próximas subseções falam um pouco sobre cada uma delas.

3.4.1.1 Edição de Redes

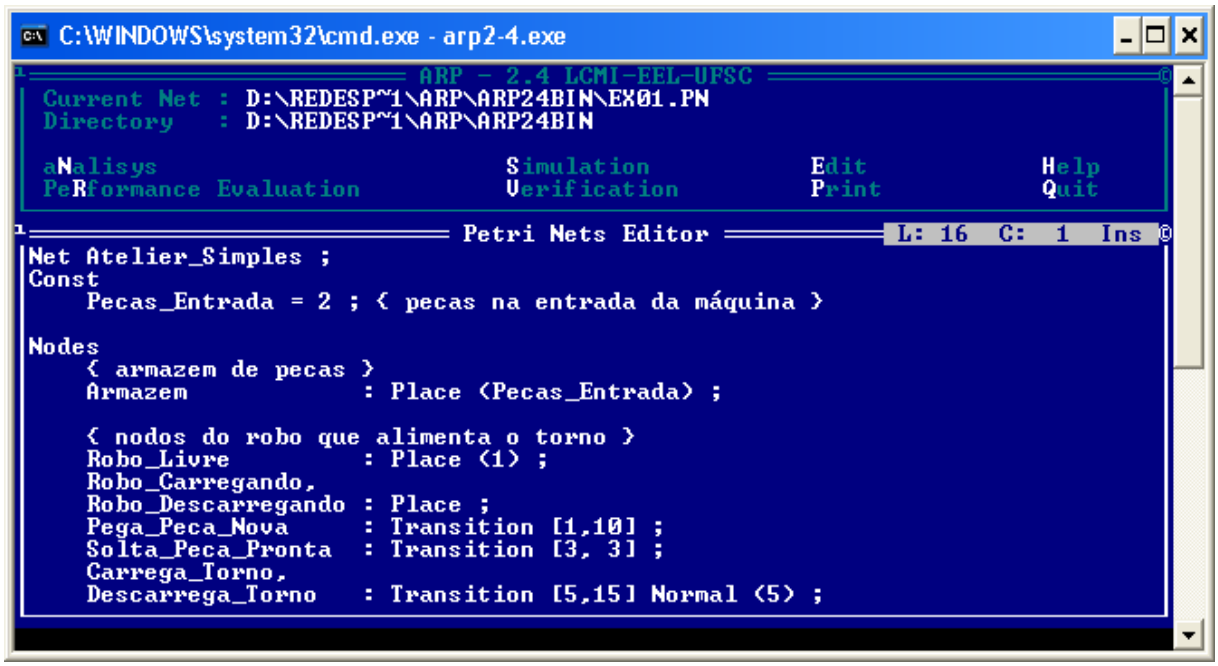


Figura 3.5 Tela de edição mostrando a descrição da rede Atelier Simples

A janela de edição concentra as funções encarregadas da manipulação dos arquivos que contém as redes, em disco. Um arquivo de rede é um arquivo ASCII normal, com extensão *default* "RDP", que contém a descrição de uma Rede de Petri.

3.4.1.2 Enumeração de Estados

A enumeração de estados é uma análise baseada na procura de todos os estados alcançáveis pela rede, disparando suas transições. O algoritmo usado é basicamente o de *Karp-Miller* [10] [11], com uma pequena modificação na determinação do crescimento de fichas nos lugares, que permite uma análise mais detalhada de redes com crescimento de fichas. A capacidade do programa depende da memória disponível.

Para redes com temporização, os estados são agrupados em classes de estados com características comuns. Cada classe de estado é composta por uma marcação e um domínio de disparo (conjunto das transições sensibilizadas, cada qual com seu respectivo intervalo dinâmico de disparo).

O texto das propriedades observadas informa as propriedades da rede pesquisadas sobre o grafo de estados gerado:

- **Limitação**: é calculado o número máximo de fichas (limite) em cada lugar da rede, para os estados alcançáveis. Os lugares podem ser nulos, quando

nunca receberam fichas; binários, sempre possuindo uma ou nenhuma ficha; limitados, quando o número de fichas é sempre igual ou inferior a um limite finito maior que 1 ou não-limitados, quando o número de fichas tende ao infinito (simbolizado por *w*). Caso sejam detectados lugares não-limitados, são indicadas as seqüências de disparos de transições que levam ao crescimento de fichas nesses lugares.

- **Conservação**: é verificado se a soma total de fichas na rede é constante para qualquer marcação alcançável, indicando se a rede é estritamente conservativa ou não.
- **Vivacidade**: este teste é relativo ao disparo das transições. Uma transição é viva se, a partir de qualquer estado do grafo gerado, existe uma seqüência de disparos que a contenha, ou seja, que leve a seu disparo. Uma transição é quase-viva se foi disparada ao menos uma vez durante a construção do grafo.
- **Multi-sensibilização**: a enumeração de classes de estados em redes com temporização pode levar a resultados incorretos caso alguma transição esteja multi-sensibilizada no grafo. Uma transição está multi-sensibilizada se, para alguma marcação, o número de fichas na entrada da transição é maior ou igual a duas vezes o peso da entrada, para todos os lugares de entrada.
- **Reiniciação**: a rede é reiniciável se todos os seus estados forem reiniciáveis. Um estado é reiniciável se, partindo dele, existe alguma seqüência de disparos de transições que leve a rede de volta ao estado inicial.
- **Livelocks**: um *livelock* é um ciclo de disparos de transições para o qual a rede não possui saída, repetindo sempre os mesmos estados sem possibilidade de mudança de rumo. Caso sejam detectados *live-locks* na rede, são indicados os estados que iniciam cada um dos mesmos.
- **Deadlocks**: um estado em está em bloqueio ou *deadlock* quando não possui nenhuma transição sensibilizada e portanto nenhum estado sucessor. Caso sejam detectados *deadlocks* na rede, são indicadas as seqüências de disparos de transições que levam aos mesmos.

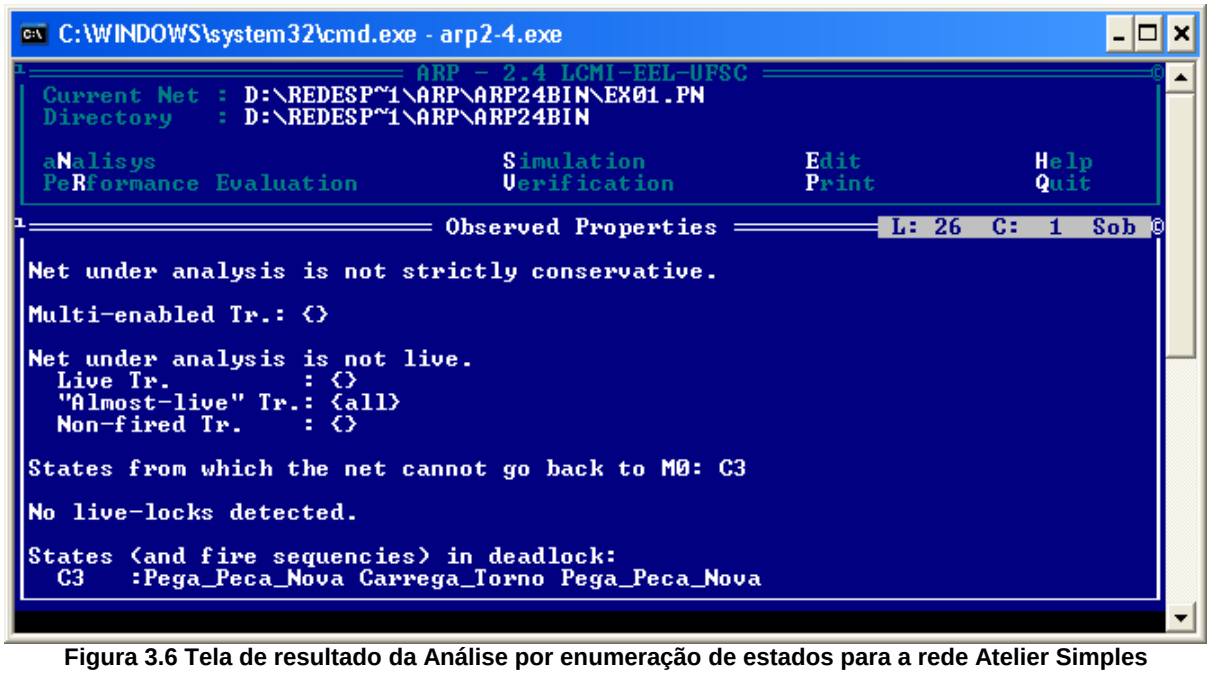


Figura 3.6 Tela de resultado da Análise por enumeração de estados para a rede Atelier Simples

3.4.1.3 Cálculo de Invariantes

A análise de invariantes procura, através de cálculos sobre a matriz de incidência da rede, conjuntos de lugares ou transições com características especiais, como a conservação de fichas ou o funcionamento cíclico.

Os invariantes lineares de lugar indicam as componentes conservativas da rede, ou seja, conjuntos de lugares da rede nos quais a soma ponderada do número de fichas seja constante para qualquer marcação alcançável. Todo vetor inteiro x_n tal que $D_{mn}\ x = 0$ é um invariante linear de lugar, onde x_i é a ponderação do i-ésimo lugar da rede na formação do invariante (D_{mn} é a matriz de incidência da rede de Petri, com m transições e n lugares).

Os invariantes lineares de transição indicam as componentes repetitivas estacionárias da rede, ou seja, conjuntos de transições que, quando disparadas na ordem e freqüência adequada, formam um ciclo, retomando ao estado de partida. Todo vetor inteiro y, tal que $y_m\ D_{mn} = 0$ é um invariante linear de transição, onde y_i indica o numero de disparos da i-ésima transição para formar o ciclo (a ordem de disparo das transições não é diretamente calculável). Deve-se observar que os invariantes dependem somente das características estruturais da rede, e não de seu estado inicial. Portanto, alguns invariantes de transição indicados podem não constar do grafo de estados acessíveis da rede, para o estado inicial definido.

Como resultado, o programa fornece a base linearmente independente dos invariantes encontrados e todos os invariantes positivos mínimos (que não sejam superposições de outros invariantes positivos), combinações lineares das linhas da base de invariantes. Também é efetuado um teste de cobertura sobre os invariantes encontrados e um teste de sub-redes, indicando quais invariantes são máquinas de estados ou grafos de eventos, conforme sua topologia:

- Um invariante linear de lugar pode ser uma **máquina de estados** se cada transição ligada a um lugar do invariante possui somente um arco de entrada e um arco de saída, ambos de peso unitário.
- Um invariante linear de transição pode ser um **grafo de eventos** se cada lugar ligado a uma transição do invariante possui somente um arco de entrada e um arco de saída, ambos de peso unitário.

Deve-se observar que este módulo ignora a informação temporal da rede, considerando somente sua topologia. Portanto, alguns invariantes de transição detectados podem não ser realizáveis, caso as restrições temporais o impeçam.

As propriedades verificadas com a análise de invariantes são assim chamadas por considerarem apenas a topologia da RdP, ou seja, não variam conforme a distribuição de fichas na rede. Sua utilidade reside justamente neste fato, quando não interessa a marcação, ou seja, não importa o estado inicial do sistema modelado, mas apenas sua topologia.

3.4.1.4 Verificação de Equivalência

A verificação por equivalência de linguagem permite observar em detalhe o comportamento de uma RdP, enfocando a atenção em algumas partes da rede e abstraindo as demais. Ela consiste em considerar um conjunto de transições da rede como sendo eventos "visíveis", sendo as demais consideradas invisíveis, e obter um autômato finito determinístico mínimo que indica o seqüenciamento entre os eventos considerados visíveis.

3.4.1.5 Simulação

O módulo de simulação permite acompanhar a evolução do funcionamento de uma rede, escolhendo as transições a disparar e observando o estado da rede e sua evolução, a qualquer instante.

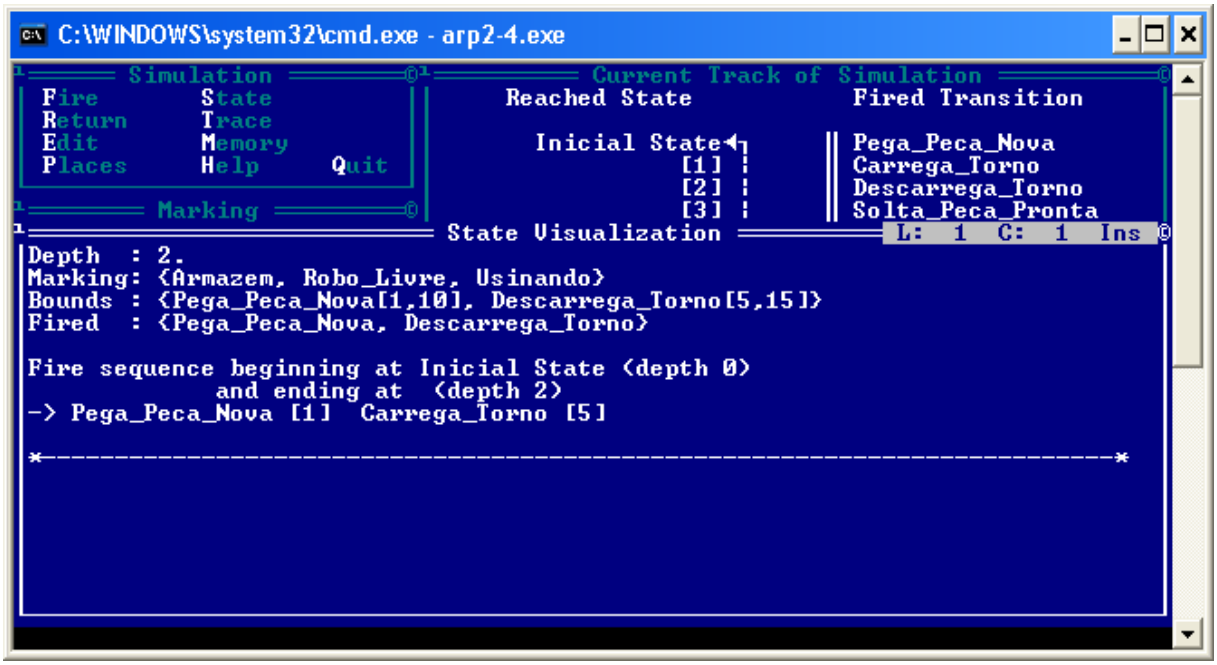


Figura 3.7 Tela de Simulação do ARP para rede Atelier Simples

3.4.1.6 Avaliação de Desempenho

Este módulo trabalha com RdP com temporização estendidas, que possuem uma função de densidade de probabilidade associada ao intervalo de disparo de cada transição. Para a avaliação de desempenho são possíveis duas abordagens: a abordagem orientada a estados e a abordagem orientada a eventos.

No modo **orientação a estados** é definido um estado (marcação) inicial e até 10 estados destino, sendo então realizados ciclos de simulação. Cada ciclo de simulação consiste no disparo das transições sensibilizadas em instantes aleatórios (sorteados de acordo com o intervalo de disparo e a curva de densidade de probabilidade de cada transição) até alcançar um destino, um bloqueio ou superar o número máximo de disparos fixado pelo usuário. Ao fim de cada ciclo são atualizados os contadores estatísticos e restaurada a marcação inicial. Os ciclos são repetidos até ser alcançada a precisão desejada ou ocorrer uma interrupção pelo usuário, pressionando uma tecla qualquer.

No modo **orientação a eventos** é definido um evento (transição) inicial e até 10 eventos destino. As transições são disparadas como na orientação a estados, sendo as medidas estatísticas efetuadas entre o disparo do evento inicial e o disparo de algum evento destino. Com isso pode-se medir o tempo médio entre eventos e a probabilidade de ocorrência de eventos.

Resulta da avaliação um texto contendo as seguintes medidas estatísticas efetuadas sobre a rede:

- Tempo médio (e desvio padrão) de acesso a cada destino.
- Tempos mínimo e máximo de acessos detectados a cada destino.
- Probabilidades de acesso relativas entre destinos.
- Número de acessos efetuados a cada destino.
- Número médio de disparos de cada transição por ciclo de simulação.
- Marcação média registrada nos lugares, por unidade de tempo.
- Tempo médio de espera de cada transição para disparar, a partir de seu instante de sensibilização.
- Número de ciclos improdutivos, onde não foi alcançado nenhum destino.

3.5 Conclusão

Utilizamos o ARP como base para o desenvolvimento da nossa ferramenta, por ser uma ferramenta abrangente no que diz respeito à análise das RdP, e pelo código modular que nos permite a extração do algoritmo usado a partir dele.

Capítulo 4

Implementação

4.1 Descrição Geral

Nossa implementação é baseada na tecnologia de orientação a objetos fazendo uso de técnicas como herança, encapsulamento e polimorfismo, com o objetivo de prover reusabilidade e extensibilidade ao código do programa [12]. Utilizamos a linguagem Java. A parte gráfica utiliza o pacote *Swing* e a API *Forms* fornecida pela *JGoodies* [13]. Outra API, também *open source*, utilizada foi a *XStream* [14], uma biblioteca simples para serializar objetos (no caso, os objetos representando a RdP sendo editada) em XML, bem como para recompor os objetos a partir de sua representação "*flat*" em XML. Para nossa implementação criamos um pequeno *framework* para suporte multilíngüe, basicamente fazendo uso de arquivos *properties*, classes de interface e uma classe que liga esses elementos. Atualmente o programa possui como idiomas opcionais o português brasileiro e o inglês.

4.2 Arquitetura e Estrutura do Código

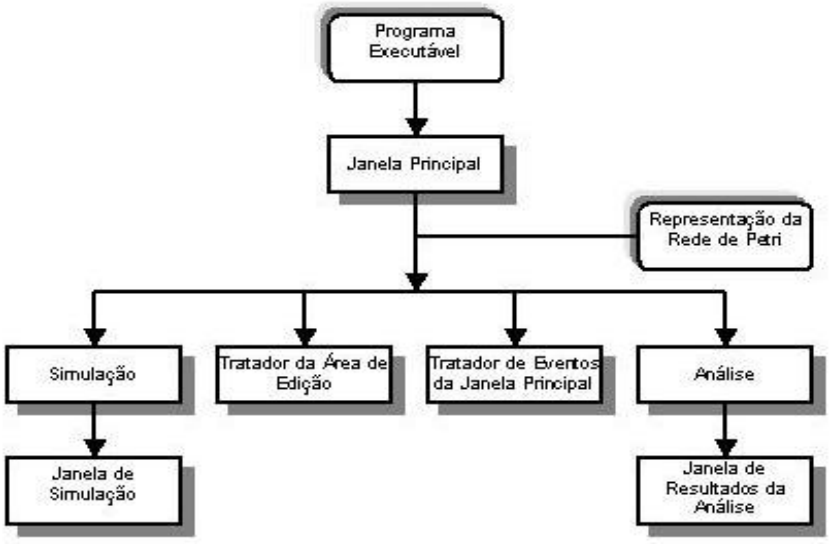


Figura 4.1 Arquitetura da ferramenta

A Figura 4.1 apresenta a arquitetura geral da ferramenta. A base da ferramenta é o modelo de objetos, que separa a parte gráfica e a análise. Através dessa separação é possível, por exemplo, reutilizar a *engine* de simulação e análise e adaptá-la a outra interface gráfica, ou utilizar a representação de uma RdP com outras ferramentas de análise. O modelo de uma RdP consiste de classes-base para

representar os elementos lugar, transição, arco e Rede de Petri. Através de herança são criadas classes especializadas com atributos e métodos necessários para a interface gráfica. Este modelo é utilizado pelos outros módulos da ferramenta.

Na camada seguinte temos, então, os módulos que operam sobre o modelo de objetos da rede através da captura de eventos e emissão de notificações para os módulos de exibição. Assim, temos o Tratador da Área de Edição (*PetriNetEditorCanvas*), Tratador da Janela Principal (*MenuActionListener*, *ButtonActionListener*, *PlaceActionListener*, *ArcActionListener* e *TransActionListener*), Simulação (*SimulationAction*) e Análise (*AnalyzerAction*).

Na última camada temos os módulos de exibição: Janela de Simulação e a Janela de Resultados da Análise, que apresentam os resultado dos eventos de simulação e análise.

Observa-se que as classes referentes à análise e simulação independem da interface gráfica e precisam apenas das informações contidas nos objetos-base do modelo de uma rede. A estrutura do código se divide em quatro tipos: (i) classes-base, usadas para trafegar informações entre a interface gráfica e a *engine*; (ii) classes da interface com usuário; (iii) classes da *engine*; e (iv) classes mediadoras.

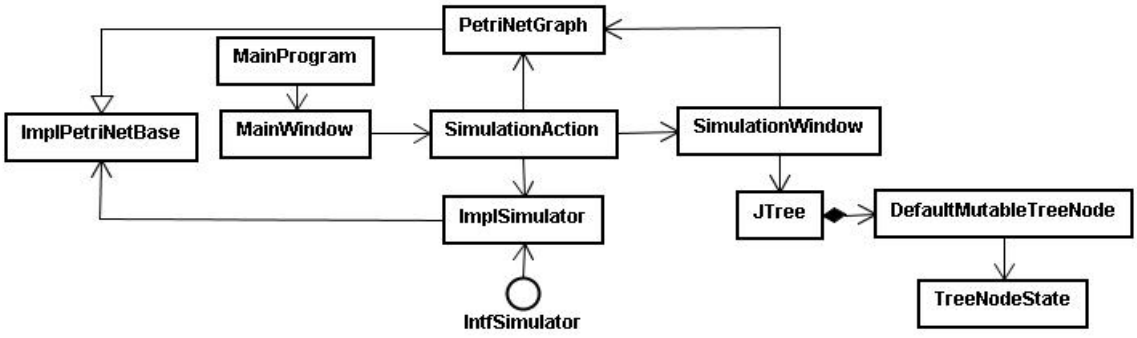


Figura 4.2 Separação entre a Simulação e o Modelo da Rede

Na Figura 4.2, por exemplo, temos o diagrama de classes do caso da *engine* de simulação. As ações que dirigem a simulação são capturadas através da classe *SimulationAction*, que também notifica as ações para a interface gráfica, classe *SimulationWindow*, e para a classe que representa a rede estendida, *PetriNetGraph* (que contém informações de posicionamento, cor, etc. dos elementos básicos da rede). A simulação, propriamente dita, é realizada pela classe *ImplSimulator*, que opera sobre a classe *ImplPetriNetBase*. Diagrama semelhante se aplica ao caso da

análise e é mostrado na Figura 4.3.

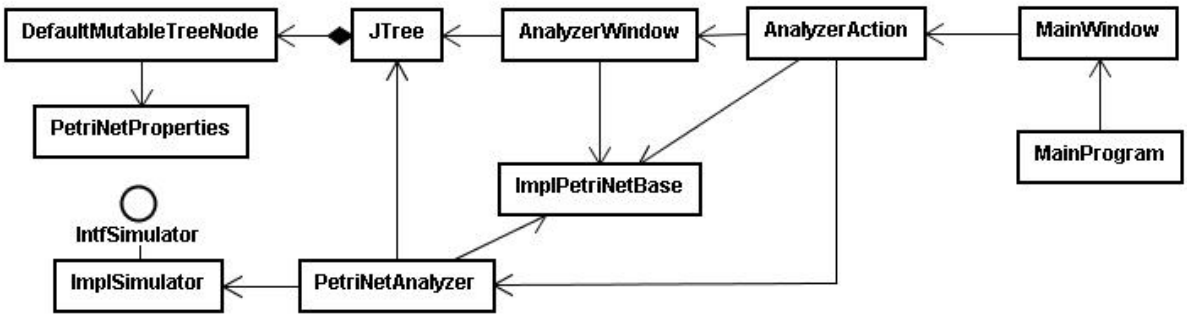


Figura 4.3 Separação entre Análise e Modelo da Rede

Nas próximas seções vamos descrever resumidamente sobre os principais pacotes que compõem o programa.

4.2.1 Pacote *br.uerj.petrinetanalyzer.common*

Neste pacote estão as classes fundamentais usadas tanto para interação com usuário quanto para operações sobre a rede. Nela estão as classes que definem os objetos elementares de uma Rede de Petri e a própria rede:

- *ImplPetriNetBase*. Define uma RdP com seus atributos elementares (Arrays de Lugar, Arco e Transição; nome; e representações, Matriz de Entrada, Matriz de Saída e Matriz de Incidência). Apesar de nossa implementação atual não fazer uso dessas representações, elas permitem um reuso e extensão no código para análises que exijam uma representação matricial.
- *PlaceBase*. Define atributos básicos de um lugar: número de fichas, nome, identificador e sua posição no *array* de lugar na classe *ImplPetriNetBase*.
- *TransitionBase*. Define atributos básicos de uma transição: nome, identificador e posição no *array* de transições na classe *ImplPetriNetBase*. Possui, além desses atributos elementares, outros relacionados à RdP temporizadas, visando uma possível extensão: *Curva de Densidade*, podendo ser normal, uniforme ou exponencial; *Static Earliest Firing Time* e *Static Latest Firing Time* que são usados para temporização segundo o modelo de Merlin.
- *ArcBase*. Define atributos básicos de um arco: peso; um booleano informando se o arco é de entrada (*true*) ou de saída (*false*); lugar ligado ao arco; transição ligada ao arco; e posição no *array* de arcos da classe *ImplPetriNetBase*.

Os atributos de posição são necessários para auxiliar na identificação única de cada objeto, uma vez que o nome é opcional.

Todas as classes encapsulam seus atributos e possuem métodos que permitem o acesso aos mesmos.

4.2.2 Pacote *br.uerj.petrinetanalyzer.engine*

Classes referentes a *engine* estão neste pacote, abaixo um pequeno resumo das principais:

- *ImplSimulator.* Implementa os algoritmos relacionados à execução de uma RdP. Recebendo no construtor como parâmetro a Rede de Petri a ser simulada (executada), possui ainda mais dois atributos chaves: um *array* de inteiros com a marcação dos lugares, e um *array* booleano que informa se uma transição está ou não habilitada para disparo. Possui um método chave *fireTransition* que recebe a transição a ser disparada, e ao ser invocado altera a marcação dos estados da rede, segundo algoritmo da Listagem 4.1.

```
01 Algoritmo:
02   Para todo Lugar de Entrada da Transição Faça
03       Lugar.fichas = Lugar.fichas - peso;
04   Fim Para
05   Para todo Lugar de Saída da Transição Faça
06       Lugar.fichas = Lugar.fichas + peso;
07   Fim Para
```

Listagem 4.1 Método *fireTransition*

Outro método importante é o que retorna as transições habilitadas, *getAvailableTransition*, segundo o algoritmo da Listagem 4.2.

```
01 Algoritmo:
02    Para toda Transicao Faça
03       Transição.habilitada = true;
04    Fim Para
05    Para todo Arco de entrada Faça
06       Se NOT(Arco.Lugar.fichas >= arco.peso
07       E Arco.transicao.habilitada = true) Então
08          Arco.transicao.habilitada = false;
09       Fim Se
10   Fim Para
```

Listagem 4.2 Método *getAvaliableTrasitions*

Possui ainda, outros métodos como o *setState* que recebe um *array* de marcação e um *array* booleano, atribuindo o estado da RdP e métodos para acessar os atributos.

- *PetriNetAnalyzer*. É a classe que cuida de verificar as propriedades da RdP e

construir a árvore de alcançabilidade. Para isso, faz uso da classe *ImplSimulator*. A árvore é encapsulada em um atributo *tree* instância da classe *JTree* [15]. Apesar de *JTree* pertencer ao pacote Swing do Java SDK e parecer ser relacionado somente a parte gráfica, aqui ele é usado como uma estrutura de dados comum muito adequada aos propósitos do programa. Destacamos nessa classe o conjunto de métodos que constrói a árvore de alcançabilidade, que é a base para as análises da rede, são eles: *generateReachabilityTree, generateTree, processNode, createNewState* e *compareMarking*. Na listagem a seguir temos uma versão do algoritmo de construção da árvore de alcançabilidade, baseado no apresentado no Capítulo 2, adaptado para linguagens de programação.

```
01 Algoritmo:
02 generateReachabilityTree()
03 {
04   root = Initial State;
05   processNode(root);
06   generateTree(root);
07 }
08
09 generateTree(node N)
10 {
11    Para i de 1 até Número de filhos de N Faça
12       X = N.getFilho(i);
13       processNode(X);
14       generateTree(X);
15    Fim Para
16 }
17
18 processNode(node X){
19    (Condição 1)
20    Para cada nó Y da arvore Faça
21       Se (compareMarking(Y,X)=0) E (Y!=X) E (Y.tipo<>FRONTEIRA) Então
22          X.tipo = DUPLICADO;
23          retorna;
24       Fim Se
25    Fim Para
26
27    (Condição 2)
28    Se (X.temTransicaoDisponivel = false) Então
29       X.tipo = TERMINAL;
30       retorna;
31    Fim Se
32
33    (Condição 3)
34    Para cada transição habilitada T de X Faça
35       Z = createNewState(X, T);
36       Z.tipo = FRONTEIRA;
37       AddNode(Z);
38    Fim Para
39    X.tipo = INTERIOR;
40 }
41 createNewState(node X, transition T)
42 {
```

```
43    Z = Disparo de T a partir de X;
44    Y = X;
45    Enquanto ( Y <> null ) Faça
46       Se compareMarking(Z , X) > 0 Então
47          Para i de 1 até total de lugares Faça
48             Se (Z.lugar[i].fichas > Y.lugar[i].fichas) Então
49                Z.lugar[i].fichas = w;
50          Fim Para
51       Fim Se
52       Y = Y.getPai();
53    Fim Enquanto
54    retorna Z;
55 }
56
57 compareMarking(node Z, node X)
58 {
59    /* Existe um lugar de Z que possui fichas > Lugar de X */
60    boolean existe = false;
61    /* Todo lugar de Z possui >= fichas Todo lugar de X */
62    boolean maior  = true;
63
64    Para i de 1 até numero de lugares Faça:
65       Se (Z.lugar[i].fichas >= X.lugar[i].fichas) E (maior = true)Então
66          maior = true;
67          Se (Z.lugar[i].fichas > X.lugar[i].fichas)
68             existe = true;
69          Fim Se
70       Senão
71          maior = false;
72       Fim Senão
73    Fim Para
74
75    Se ( maior = true ) E ( existe = true) Então
76       retorna 1;  /* marcação do estado Z > Marcação do estado X */
77    Se ( maior = true ) E ( existe = false ) Então
78       retorna 0;  /* marcação do estado Z = Marcação do estado X */
79    Se (maior = false) Então
80       retorna -1; /* marcação do estado Z < Marcação do estado X */
81 }
```

Listagem 4.3 Algoritmo gerador da árvore de alcançabilidade

- *PetriNetState*. Encapsula os dados referentes ao estado corrente de uma RdP.
- *PetriNetProperties*. Classe que encapsula as propriedades inerentes a uma RdP, possuindo diversos atributos que retornam informações sobre a rede sendo editada. Por exemplo, atributos booleanos que informam se a rede é viva, limitada e conservativa. Outro atributo importante a destacar é a lista informando todas as seqüências de disparos que levam a rede ao estado de *deadlock*.

4.2.3 Pacote *br.uerj.petrinetanalyzer.gui.objects*

As classes referentes à interface gráfica com o usuário estão distribuídas no pacote *br.uerj.petrinetanalyzer.gui* e seus subpacotes.

No pacote *br.uerj.petrinetanalyzer.gui.objects* estão as classes que modelam os objetos da Rede de Petri, herdando de suas classes bases correspondentes, porém acrescidas de atributos e métodos necessários para a interface com o usuário.

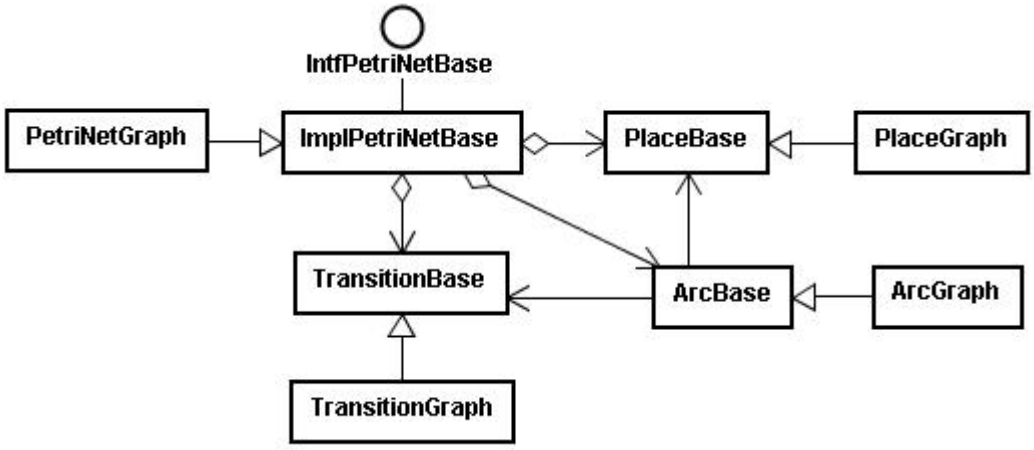


Figura 4.4 Diagrama de Classes com as classes-base e as estendidas para modelar Redes de Petri

- *PetriNetGraph*. Herda de *PetriNetBase*. A especialização está nos atributos informando qual objeto está correntemente selecionado pelo usuário no campo de desenho. O seu método principal é o *getObjectPosition* que recebe como parâmetro as coordenadas *X* e *Y* de um ponto na tela e retorna o objeto da rede que está sob o ponto, caso exista.

- *PlaceGraph* e *TransitionGraph*. Herdam de *PlaceBase* e *TransitionBase* respectivamente. Ambos são apenas acrescidos de atributos de coordenada, e métodos que verificam se o objeto está ou não sob um ponto recebido como parâmetro (métodos *inPlace e inTransition*), além de métodos de acesso.

- *ArcGraph*. Herda de *ArcBase*. Possui atributos que auxiliam no desenho do Arco, como uma lista de pontos que compõe o arco, qual sua posição em relação ao lugar ao qual está ligado (norte, sul, leste, oeste). Possui como métodos principais: *setStartObject*, que informa qual o objeto de origem, *setEndObject*, informa qual o objeto de destino. Outro método importante é o *refreshEndPoints*, invocado quando algum dos objetos inicial ou final tem sua posição alterada.

4.2.4 Pacote *br.uerj.petrinetanalyzer.gui.listener*

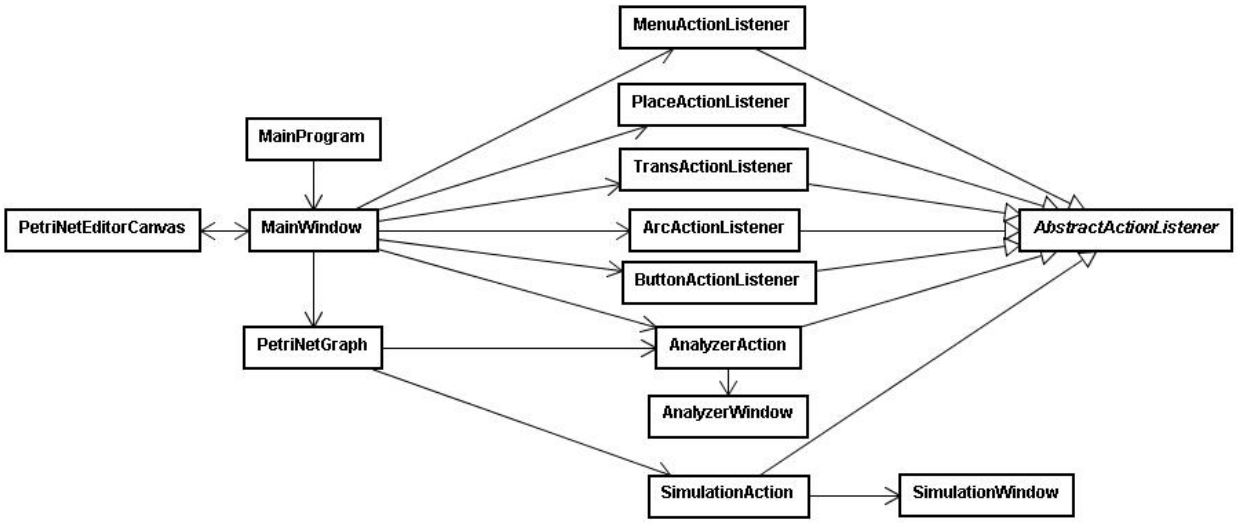


Figura 4.5 Diagrama de classe com as classes mediadoras

Neste pacote é que estão as classes mediadoras, a maioria das classes tem apenas objetivo de tratar os eventos da interface gráfica: *MenuActionListener, ButtonActionListener, PlaceActionListener, ArcActionListener, TransitionActionListener*, por isso não entraremos em detalhes sobre elas. As que realmente fazem a mediação entre as operações executadas sobre a rede e as telas do usuário são as seguintes:

- *SimulationAction.* Classe que faz a ligação entre o programa principal, a janela de simulação e a classe *ImplSimulator*. A RdP a ser simulada e o estado corrente compõe os seus atributos principais.

  Seu funcionamento é o seguinte. O programa, na janela principal com o desenho da rede, recebe os eventos solicitando o disparo de uma transição, e para isso invoca o método *tryFireTransition*. Este método recebe como parâmetro a transição que se quer disparar e tenta fazer o disparo invocando o método *fireTransition* da classe *ImplSimulator.* A *renderização* das figuras na tela, colocando em vermelho as transições habilitadas para disparo, é feita quando o programa principal pergunta a classe *SimulationAction* se a transição está habilitada, através do método *canFireTransition.*

  Durante a simulação é possível que o usuário faça a rede retornar para qualquer dos estados pelo qual a mesma tenha passado anteriormente. Para tal, é necessário que seja selecionado algum estado na Árvore de Disparos na janela

de simulação. O programa principal invocará, então, o método *goToState* da classe *SimulationAction*, este método simplesmente pegará o estado selecionado na janela de simulação, alterará o estado corrente para o estado selecionado, e passará o novo *array* de marcação e *array* de transições para a instância da classe *ImplSimulator*. A simulação interativa prosseguirá então a partir desse estado.

- *AnalyzerAction.* Faz a ligação entre o programa principal, a janela de resultados da análise e a classe *PetriNetAnalyzer,* que faz a verificação das propriedades da RdP. Possui apenas o atributo da rede a ser analisada. Seu método principal é o construtor, que instancia a classe *PetriNetAnalyzer* e passa os resultados para uma instância da janela de resultados (Listagem 4.4).

```
01 public AnalyzerAction(PetriNetBase pn)
02 {
03   ...
04   this.pn = pn;
05   analyzerEngine = new PetriNetAnalyzer(this.pn);
06   analyzerEngine.geraArvoreAlcancabilidade();
07   anWindow = new AnalyzerWindow(this.pn,
08                                 analyzerEngine.getTree(),
09                                 analyzerEngine.getProperties());
10   ...
11 }
```

Listagem 4.4 Construtor *AnalyzerAction*

4.3 Interface Gráfica

A Figura 4.6 apresenta alguns exemplos de uso da ferramenta, destacando-se o editor gráfico. (a) e (b) apresentam detalhes da edição de arcos e transições. (c) e (d) apresentam a janela principal, com os menus de edição/ação, redes editadas e uma janela de simulação. (e) e (f) apresentam resultados de análises.

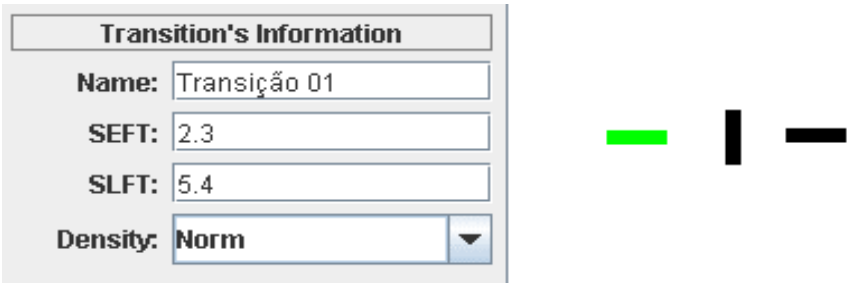


Figura 4.6 (a)

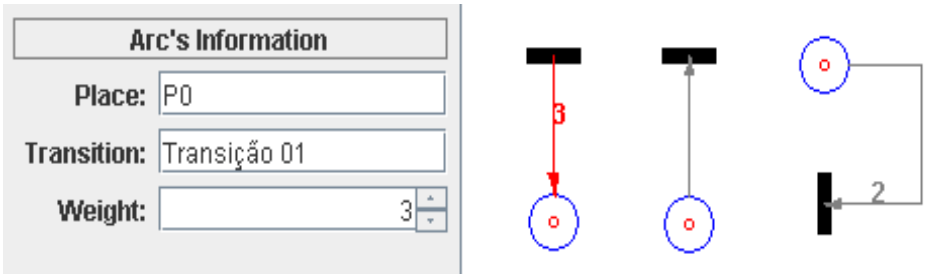


Figura 4.6 (b)

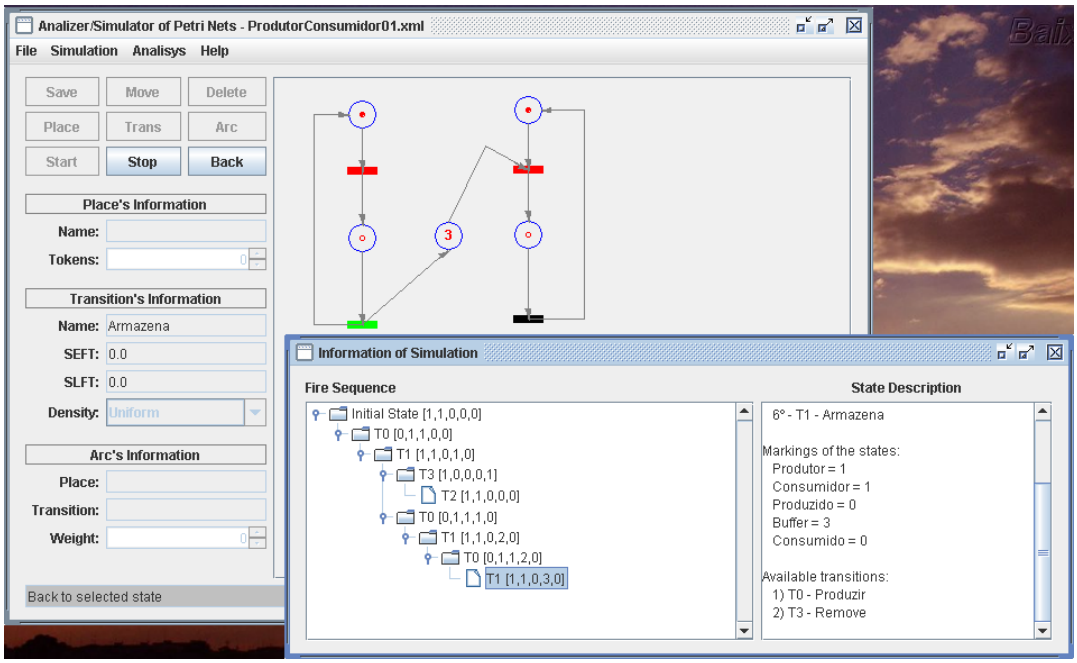


Figura 4.6 (c)

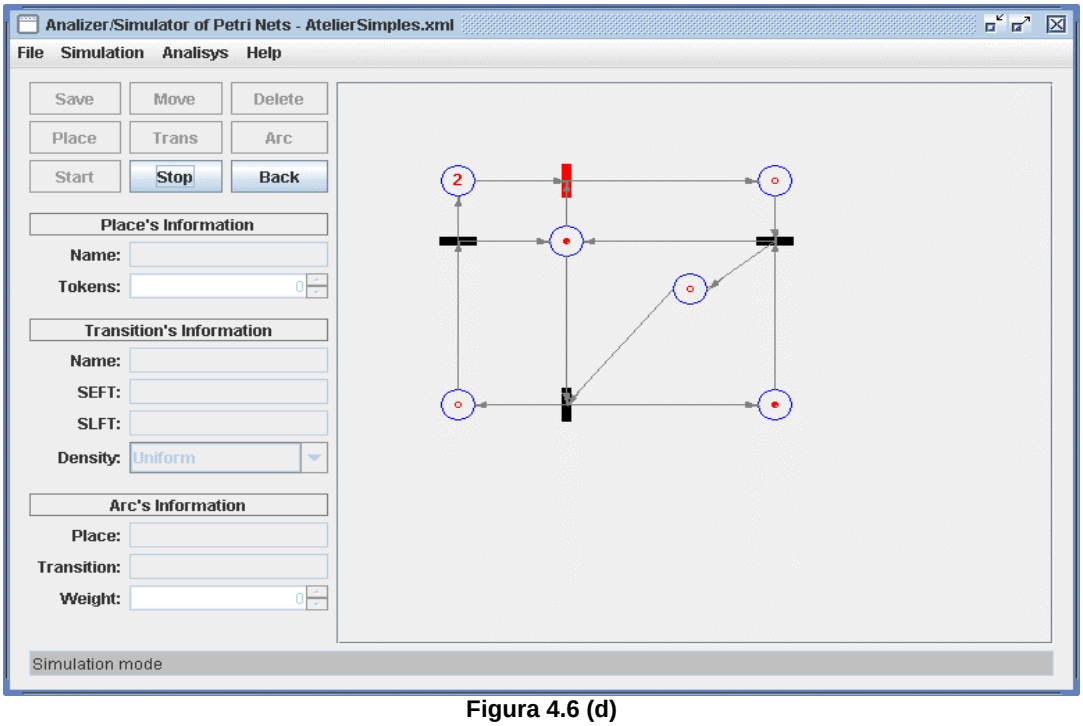


Figura 4.6 (d)

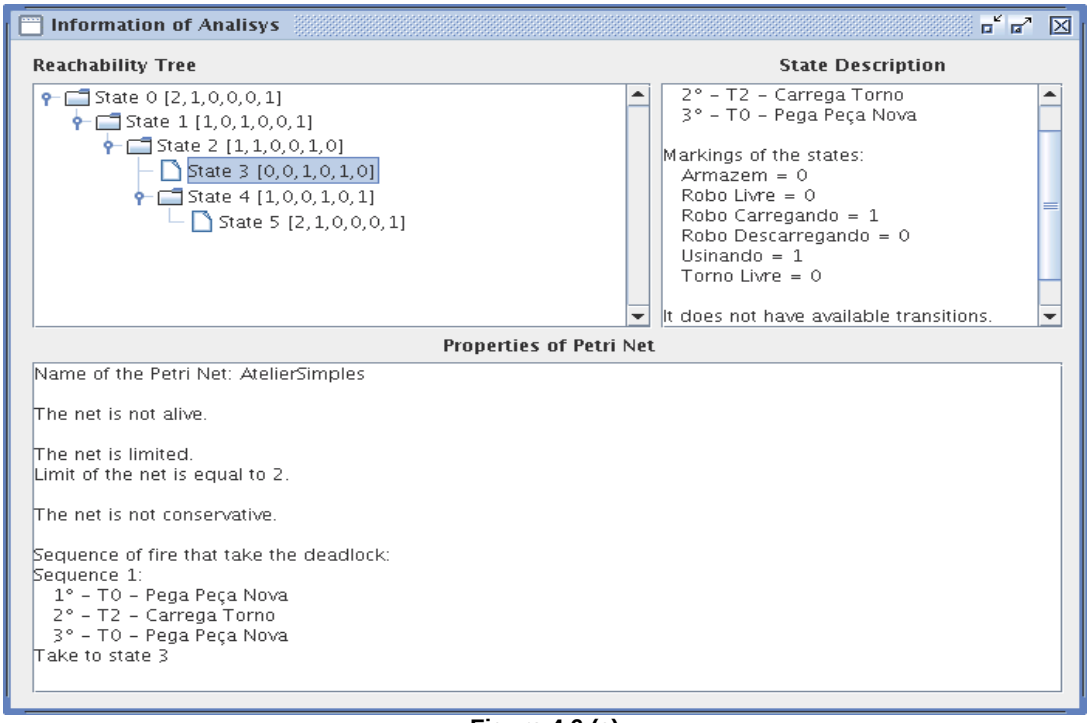


Figura 4.6 (e)

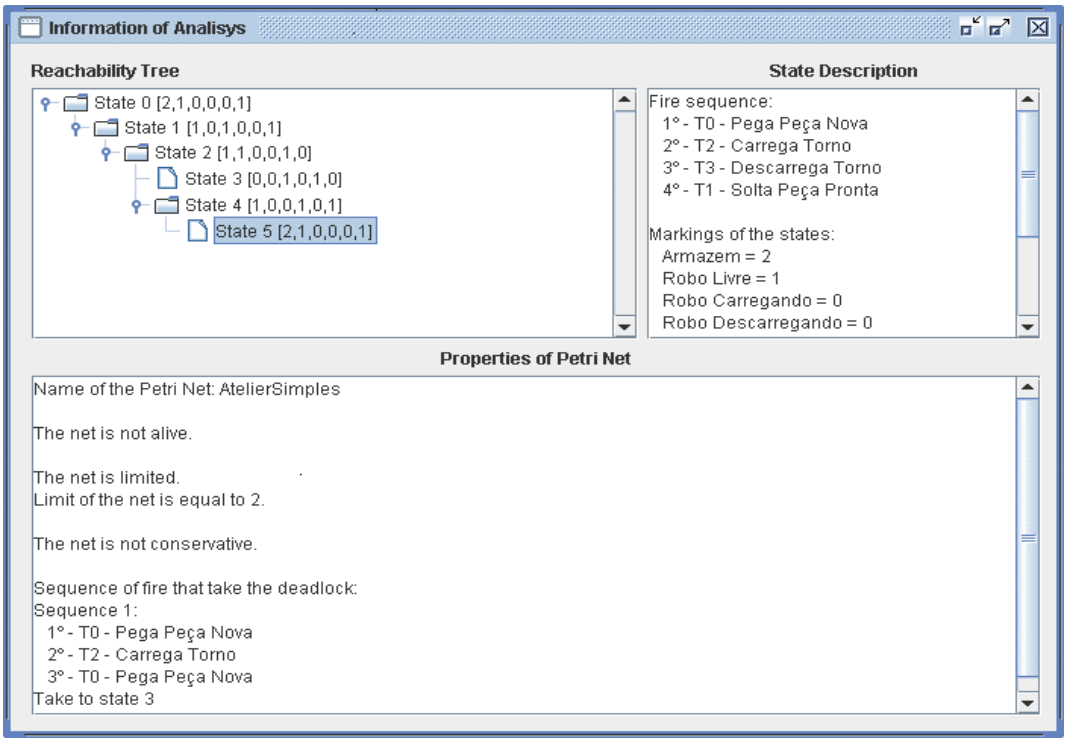


Figura 4.6 (f)

Figura 4.6 Interface gráfica, exemplos de uso

4.4 Extensões

As classes que modelam o núcleo de uma Rede de Petri possuem atributos e métodos atualmente não utilizados pelo aplicativo já visando uma extensão. Por exemplo, o código já está preparado para Redes de Petri Temporizadas segundo o

modelo de Merlin [16]. A classe *TransitionBase* possui os atributos: *curvaDensidade*; *seft* (*Static Earliest Firing Time*) e *slft* (*Static Latest Firing Time*), facilitando a configuração destas informações, na Figura 4.7 é mostrado o diagrama UML desta classe.

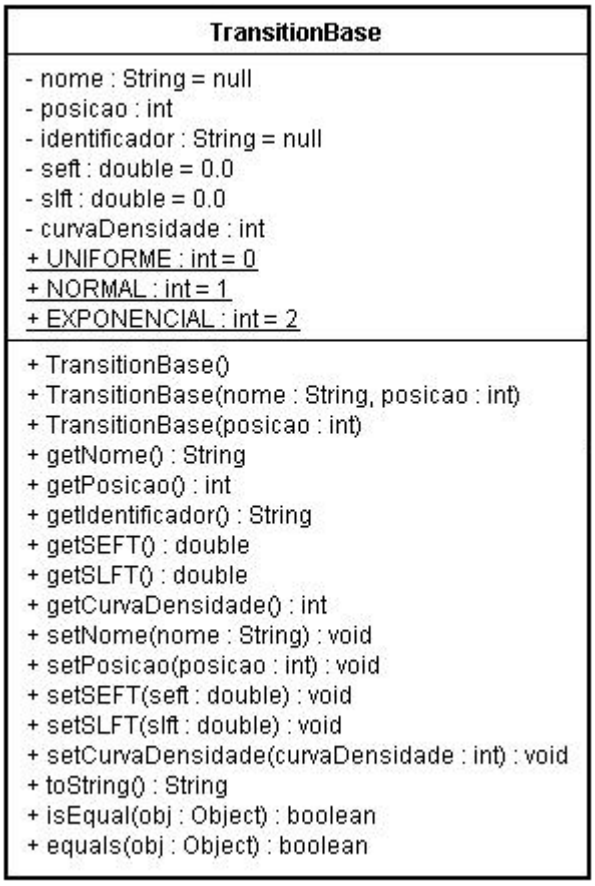


Figura 4.7 Classe *TransitionBase*

Novas análises podem ser adicionadas consultando-se a representação matricial da Rede de Petri, construída pelos métodos *buildInputAndOutpuMatrix* (listagem 4.5) e *buildIncedenceMatrix* (listagem 4.6), e é fornecida pelos métodos *getInputMatrix, getOutputMatrix* e *getIncidenceMatrix*. Para mais verificações de propriedades, devem ser alteradas ou estendidas as classes *PetriNetAnalyzer* e *PetriNetProperties*. Todas as verificações feitas atualmente na rede, são executadas durante a construção da árvore de alcançabilidade, outras podem ser feitas após a construção da árvore o que exige que ela seja percorrida algumas vezes. Como as classes de simulação e análise da RdP foram construídas apenas usando-se a modelagem dos elementos básicos da rede, elas são completamente reutilizáveis.

O formato do arquivo XML gerado para persistir uma rede editada assemelha-se à estrutura da classe *PetriNetGraph*, uma vez que o arquivo persistido é espelho

da classe. Contudo, pelas classes envolvidas na formação de uma RdP possuírem apenas os atributos essenciais, o arquivo gerado torna-se parecido com outras representações de RdP em XML, como *PNML* por exemplo [17]. Assim sendo, é simples capturar os atributos principais de um arquivo em um formato qualquer e colocá-los na representação de XML de nossa implementação. O inverso também é possível, mas um pouco mais trabalhoso, pois sempre dependerá da complexidade do formato do arquivo alvo.

```
01 public void buildInputAndOutputMatrix()
02 {
03    if(listArco != null)
04    {
05         matrizEntrada = new int[getNumTransicao()][getNumLugar()];
06         matrizSaida   = new int[getNumTransicao()][getNumLugar()];
07
08         for(int i = 0; i < listArco.size(); i++)
09         {
10            ArcBase        arco     = listArco.get(i);
11            TransitionBase transicao = arco.getTransicao();
12            PlaceBase      lugar     = arco.getLugar();
13
14            int l = transicao.getPosicao();
15            int c = lugar.getPosicao();
16
17            if(arco.verifyEntrada())
18                 matrizEntrada[l][c] = arco.getPeso();
19            else
20                 matrizSaida[l][c]   = arco.getPeso();
21
22         }
23    }
24 }
```

Listagem 4.5 Método *buildInputAndOutputMatrix*

```
01 public void buildIncidenceMatrix()
02 {
03   matrizIncidencia = new int[getNumTransicao()][getNumLugar()];
04
05   for(int l = 0; l < getNumTransicao(); l++ )
06     for(int c = 0; c < getNumLugar(); c++)
07       matrizIncidencia[l][c] = matrizSaida[l][c] - matrizEntrada[l][c];
08 }
```

Listagem 4.6 Método *buildIncidenceMatrix*

Nas Listagens 4.7 e 4.8 temos a comparação do formato do XML gerado pelo nosso aplicativo e o formato *PNML*.

Percebe-se claramente que os atributos essenciais que existem no formato *PNML* também fazem parte do nosso formato.

```
01 <br.uerj.petrinetanalyzer.gui.objects.PetriNetGraph>
02   <nome>RedeSimples.xml</nome>
03   <listLugar>
04       <br.uerj.petrinetanalyzer.gui.objects.PlaceGraph>
05       <x>27</x>
06       <y>69</y>
07       <nome>P0</nome>
08       <posicao>0</posicao>
09       <fichas>2</fichas>
10       <identificador>P0</identificador>
11       </br.uerj.petrinetanalyzer.gui.objects.PlaceGraph>
12   </listLugar>
13   <listTransicao>
14     <br.uerj.petrinetanalyzer.gui.objects.TransitionGraph>
15       <orientation>1</orientation>
16       <x>83</x>
17       <y>69</y>
18       <nome>T0</nome>
19       <posicao>0</posicao>
20       <identificador>T0</identificador>
21       <seft>0.0</seft>
22       <slft>0.0</slft>
23       <curvaDensidade>0</curvaDensidade>
24     </br.uerj.petrinetanalyzer.gui.objects.TransitionGraph>
25   </listTransicao>
26 ...
```

Listagem 4.7 Trecho de XML persistido pelo JSARP

```
01 <?xml version="1.0" encoding="ISO-8859-1"?><pnml xmlns = "">
02   <net id = "n1"type = "http://www.irt.rwth-
03 aachen.de/download/netlab/pntd/pnsmNet">
04     <name><text>Petri net1</text></name>
05     <place id = "p1">
06       <graphics>
07         <position x = "300"
08                   y = "100"/>
09         <dimension x = "40"
10                    y = "40"/>
11       </graphics>
12       <initialMarking>
13         <text>1</text>
14       </initialMarking>
15       <capacity>
16         <text>1</text>
17       </capacity>
18     </place>
19     <transition id = "t1">
20       <graphics>
21         <position x = "300"
22                   y = "200"/>
23         <dimension x = "40"
24                    y = "40"/>
25       </graphics>
26     </transition>
27 ...
```

Listagem 4.8 Trecho de arquivo PNML

4.5 Exemplos

Nesta seção apresentaremos quatro exemplos de modelagem com Redes de Petri e o resultado apresentado pelo nosso programa. O primeiro é o “Atelier Simples” (o mesmo do Capítulo 2). O segundo é uma variante do problema “Produtores e Consumidores” também apresentado no Capítulo 2. O terceiro é um problema que envolve a integração de sistemas. Como os três exemplos anteriores mostram redes limitadas, o quarto exemplo apenas mostra como o programa se comporta com redes ilimitadas.

4.5.1 Atelier Simples

Veremos agora os resultados apresentados pelo nosso programa no problema do Atelier Simples. Não detalharemos aqui a modelagem do problema, uma vez que ela se encontra na Seção 2.7.1.

Apenas lembrando que o sistema modelado é composto de um de um torno e de um robô que o alimenta, carregando e descarregando peças de um armazém.

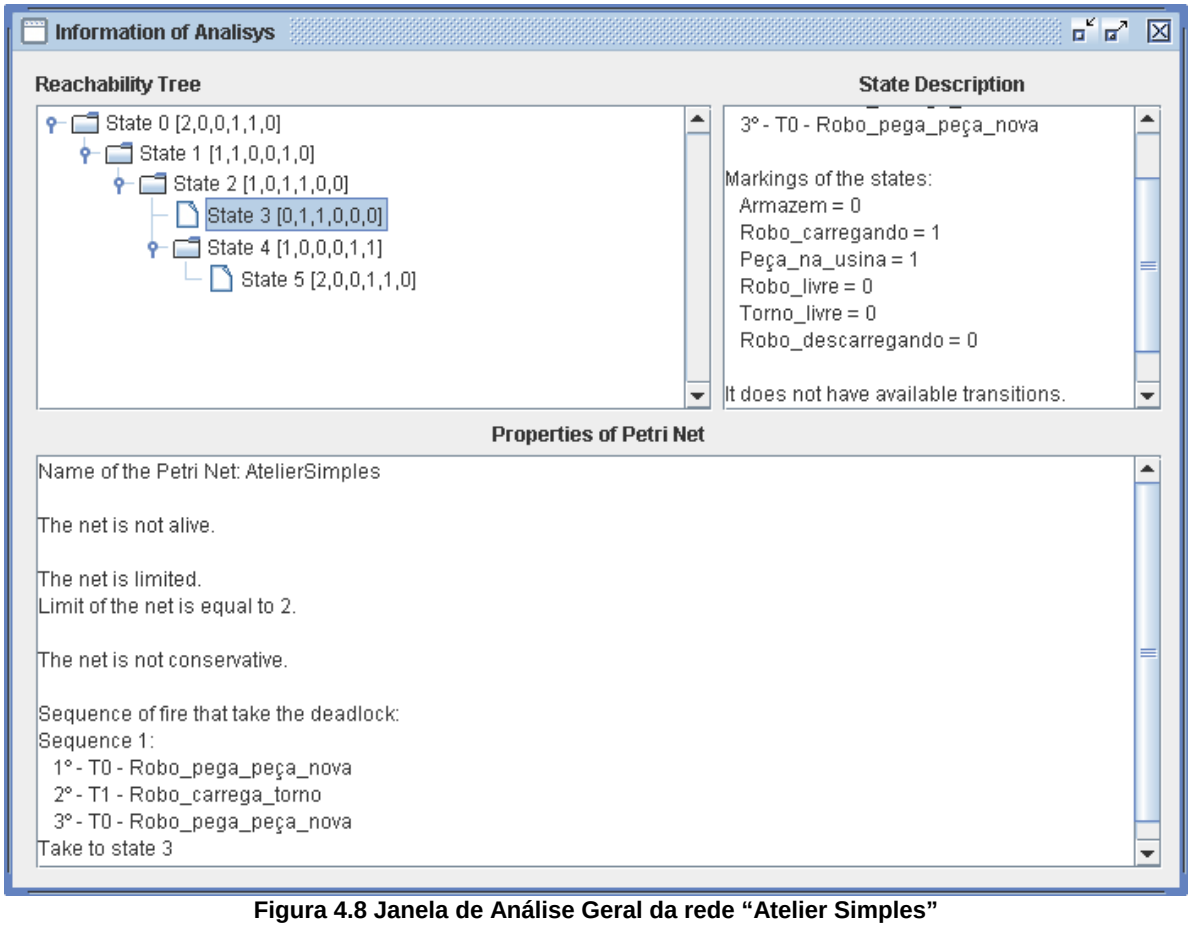
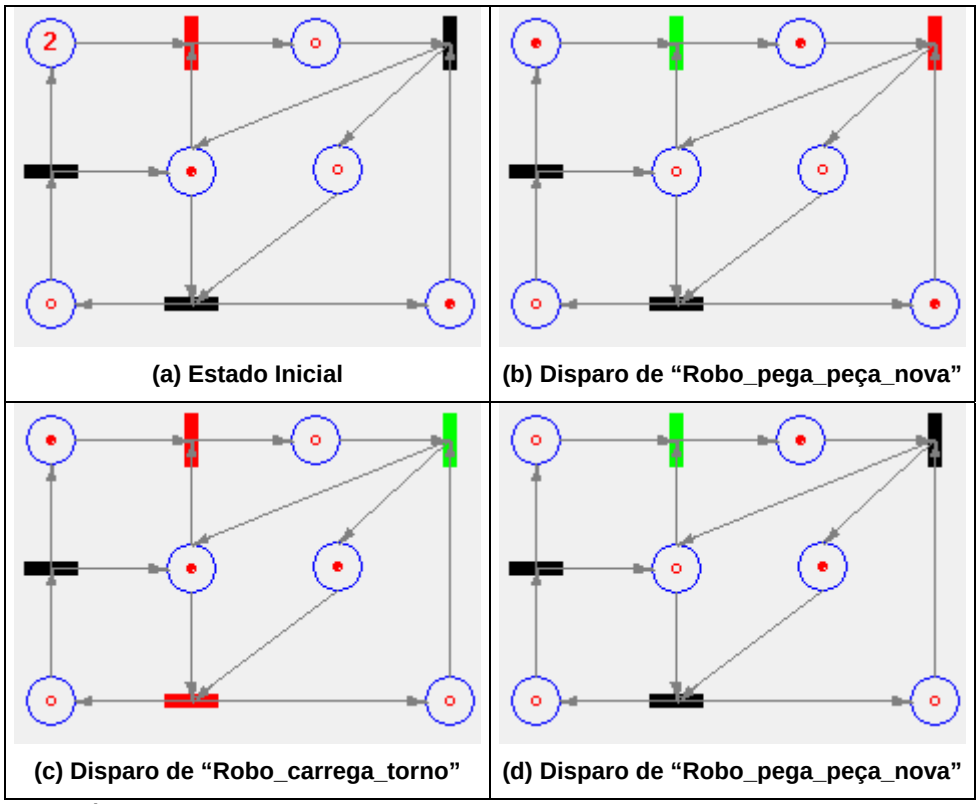


Figura 4.8 Janela de Análise Geral da rede “Atelier Simples”

Mostramos na Figura 4.8 a tela de análise geral da rede com o estado 3 (*deadlock)* selecionado na árvore de alcançabilidade.

O resultado mostrado pelo nosso programa é idêntico aos observados no Capítulo 2. A rede não é viva, com limite igual a 2, não é conservativa e possui uma seqüência de disparos que leva a um estado de *deadlock*.

Podemos ver graficamente como fica a evolução da rede até chegar ao estado de *deadlock* através do modo de simulação na Figura 4.9.



(a) Estado Inicial | **(b) Disparo de “Robo_pega_peça_nova”**

(c) Disparo de “Robo_carrega_torno” | **(d) Disparo de “Robo_pega_peça_nova”**

Figura 4.9 Seqüência de disparos que levam a *deadlock*. Transições habilitadas em vermelho e última transição disparada em verde

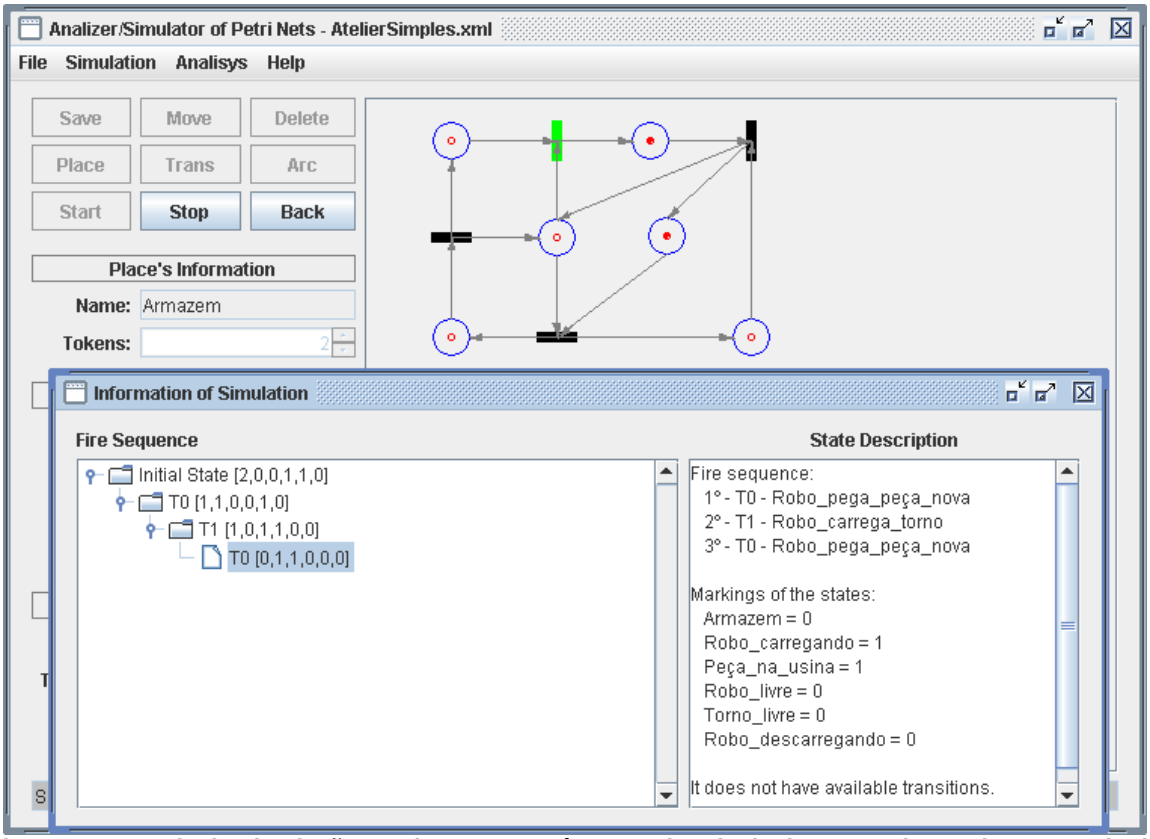


Figura 4.10 Janela de Simulação em destaque. Atrás a Janela principal mostrando a rede em estado de *deadlock*

4.5.2 Produtores e Consumidores com Buffer Limitado

Essa é uma variante do exemplo mostrado na Seção 2.7.3, onde o sistema era composto por um produtor, um consumidor e um buffer sem restrições de limite explícito.

Agora nosso modelo será acrescido de um lugar, que representa um contador limitador do buffer, e dois arcos que vão incrementar e decrementar o valor desse "contador".

O significado das transições não muda. Os lugares e seus respectivos significados na ordem são:

- P0 – Produtor pronto para armazenar.
- P1 – Consumidor pronto para retirar do buffer.
- P2 – Consumidor pronto para consumir.
- P3 – Limitador do buffer ("contador").
- P4 – Buffer.
- P5 – Produtor pronto para produzir.

Vamos impor ao buffer o limite igual a 3, e no estado inicial os processos produtor e consumidor estão prontos para iniciarem suas execuções. Temos portanto no estado inicial a marcação [0,1,0,3,0,1].

A Figura 4.11 mostra a rede modificada para ter o buffer limitado.

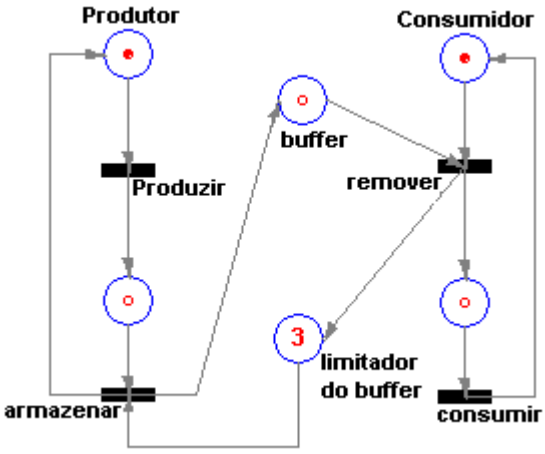


Figura 4.11 Rede de Petri: Produtores e Consumidores com Buffer Limitado

Apresentamos na Figura 4.12 o resultado da análise geral da rede. Comparando os resultados dessa rede modificada com a rede original do Capítulo 2, temos que enquanto a original era ilimitada e não conservativa, esta possui limite igual a 3 e é conservativa com máximo de fichas na rede sendo 5. As duas são vivas.

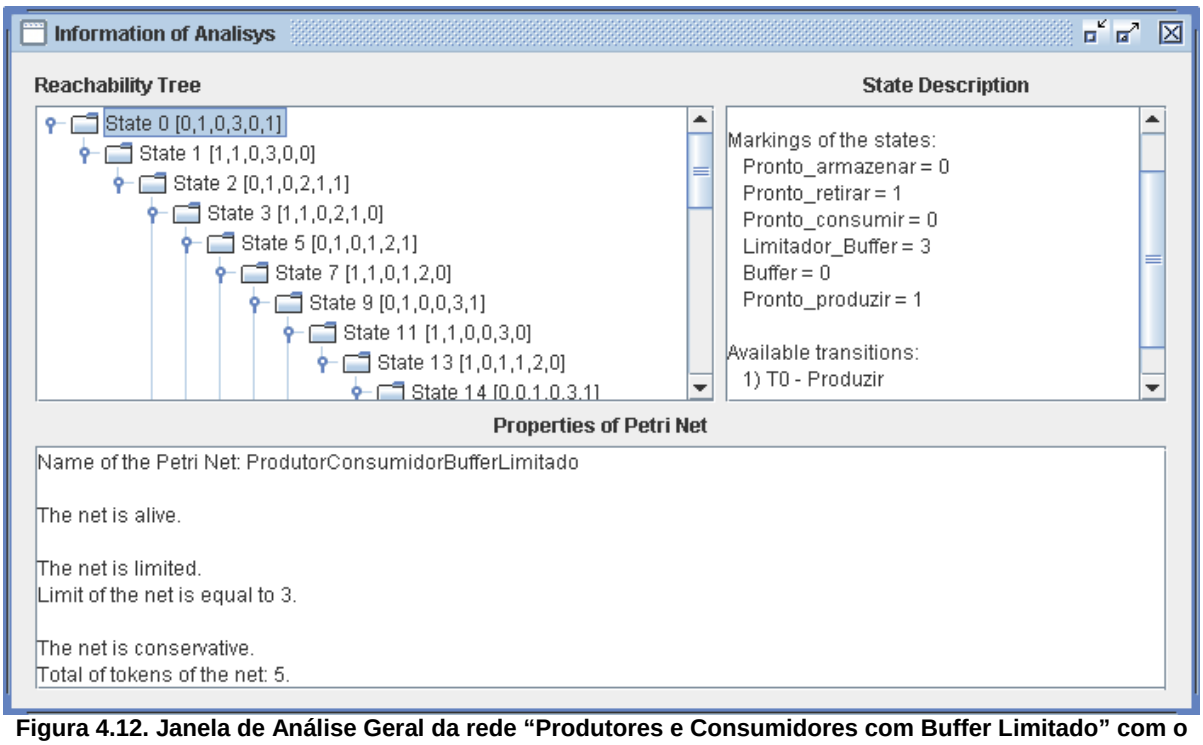


Figura 4.12. Janela de Análise Geral da rede "Produtores e Consumidores com Buffer Limitado" com o estado inicial selecionado na árvore de alcançabilidade

4.5.3 Integração de Sistemas Simples

Este modelo será tratado mais detalhadamente que os anteriores por ser completamente novo.

Vamos construir um modelo da integração de sistemas. O modelo é composto de um sistema origem que faz as requisições e de um sistema destino que processa as requisições. Contudo, existem 3 comportamentos distintos para tipos diferentes de requisição.

- Tipo 0: a requisição é enviada para o sistema destino que responde emitindo um recibo. Paralelamente, processa a requisição e envia a resposta do processamento. Por fim, a resposta é repassada para o sistema origem. O sistema origem então envia uma confirmação de recebimento.
- Tipo 1: a requisição é enviada para o sistema destino como no Tipo 0, mas o recibo também deve ser repassado para o sistema origem como um status da requisição. O sistema origem, ao receber esse status, envia uma confirmação de recebimento. O fluxo final é o mesmo do tipo 0, ou seja, após o processamento o sistema de destino envia a resposta que é repassada ao sistema origem que encerra enviando uma confirmação de recebimento.
- Tipo 2: a requisição é enviada para o sistema destino que processa a requisição e responde o processamento sem emitir recibo. Essa resposta não é repassada ao sistema origem e o fluxo é encerrado.

O significado dos lugares é o seguinte:

- P0 – Sistema Origem pronto para enviar requisição.
- P1 – Sistema Destino processando recibo de requisição tipo 0.
- P2 – Sistema Destino processando requisição.
- P3 – Resposta pronta para ser enviada para o Sistema Origem.
- P4 – Sistema Origem recebe resposta.
- P5 – Sistema Destino processa requisição tipo 2.
- P6 – Sistema Destino processa requisição tipo 1.
- P7 – Sistema Origem recebe status da requisição.
- P8 – Encerramento da Requisição.

O significado das transições é:

- T0 – Sistema Origem envia requisição tipo 0.

- T1 – Sistema Origem envia requisição tipo 1.
- T2 – Sistema Origem envia requisição tipo 2.
- T3 – Sistema Destino envia recibo (tipo 0).
- T4 – Sistema Destino envia resposta do processamento.
- T5 – Enviar resposta de processamento para o Sistema Origem.
- T6 – Sistema Origem envia recibo da resposta de processamento.
- T7 – Sistema Destino envia recibo (tipo 1, status).
- T8 – Sistema Origem envia recibo do status de processamento (tipo 1).
- T9 – Sistema Destino envia resposta de processamento (tipo 2).
- T10 – Reinicializa o ciclo (não está relacionado à modelagem).

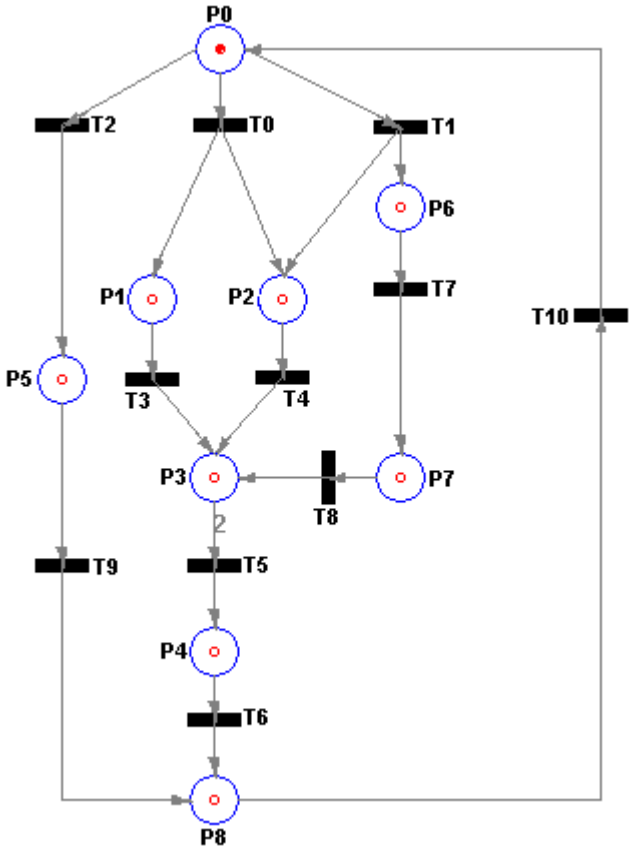


Figura 4.13 Rede de Petri: Integração de Sistemas Simples

Um detalhe do modelo, mostrado na Figura 4.13, deve ser observado. O peso 2 no arco que tem origem em P3 e destino em T5. O valor 2 no peso representa a condição de enviar a resposta de processamento para o sistema origem apenas quando, tanto o recibo quanto a resposta, tiverem chegado do sistema destino para as requisições tipo 0. Para as requisições tipo 1, representa a condição de enviar a resposta de processamento para o sistema origem, somente após o envio do status

da requisição para o sistema origem e sua confirmação de recebimento, e a chegada da resposta de processamento.

Os resultados da análise realizada pelo nosso programa nos mostra que a rede é viva. Observe que isso ocorre graças à transição que reinicializa o sistema, pois caso contrário haveria bloqueio quando o total de fichas na rede fosse 1 e ela estivesse em P8.

A rede não é conservativa e é limitada com limite igual a 2. Uma interpretação desse resultado é que existem, no máximo, dois fluxos em paralelo: o processamento da requisição e o processamento do recibo da requisição.

O fato de não existir bloqueio na rede é algo positivo para a integração de sistemas modelado.

A Figura 4.14 mostra o estado 8 selecionado que representa a seguinte seqüência de disparos: T0, T5, T6, T7 e T8. Esta seqüência é o tratamento dado a requisições tipo 0.

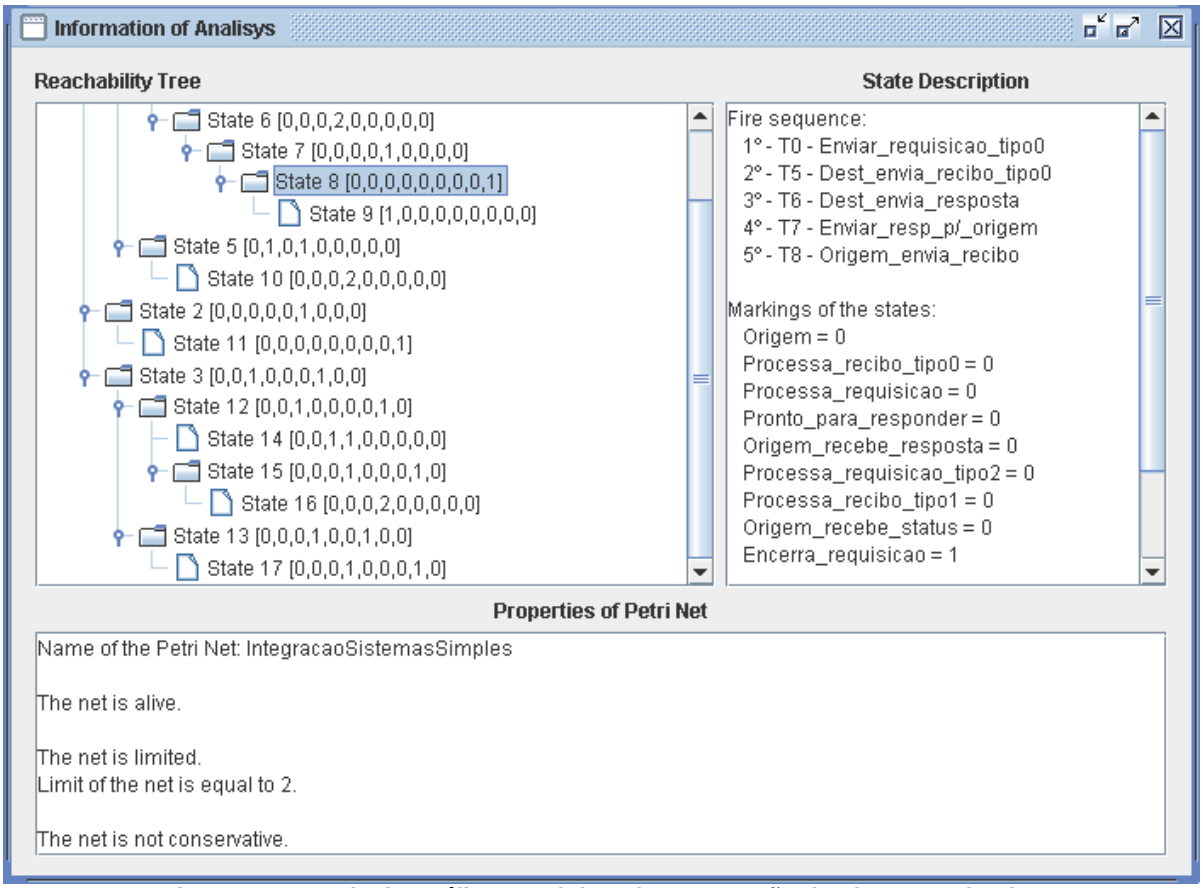


Figura 4.14 Janela de Análise Geral da rede "Integração de Sistemas Simples

4.5.4 Rede Ilimitada

Como foi dito no início da Seção 4.5, esse exemplo visa apenas mostrar o comportamento do programa para redes ilimitadas [18].

A rede a seguir apresenta um crescimento infinito de fichas em todos os seus lugares, após disparos consecutivos de suas transições.

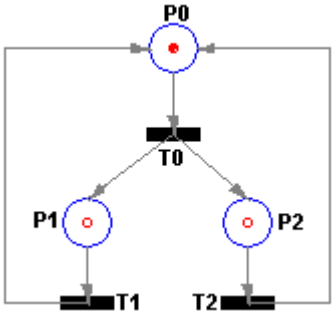


Figura 4.15 Rede de Petri ilimitada

A Figura 4.16 mostra a árvore de alcançabilidade completa e o estado inicial selecionado.

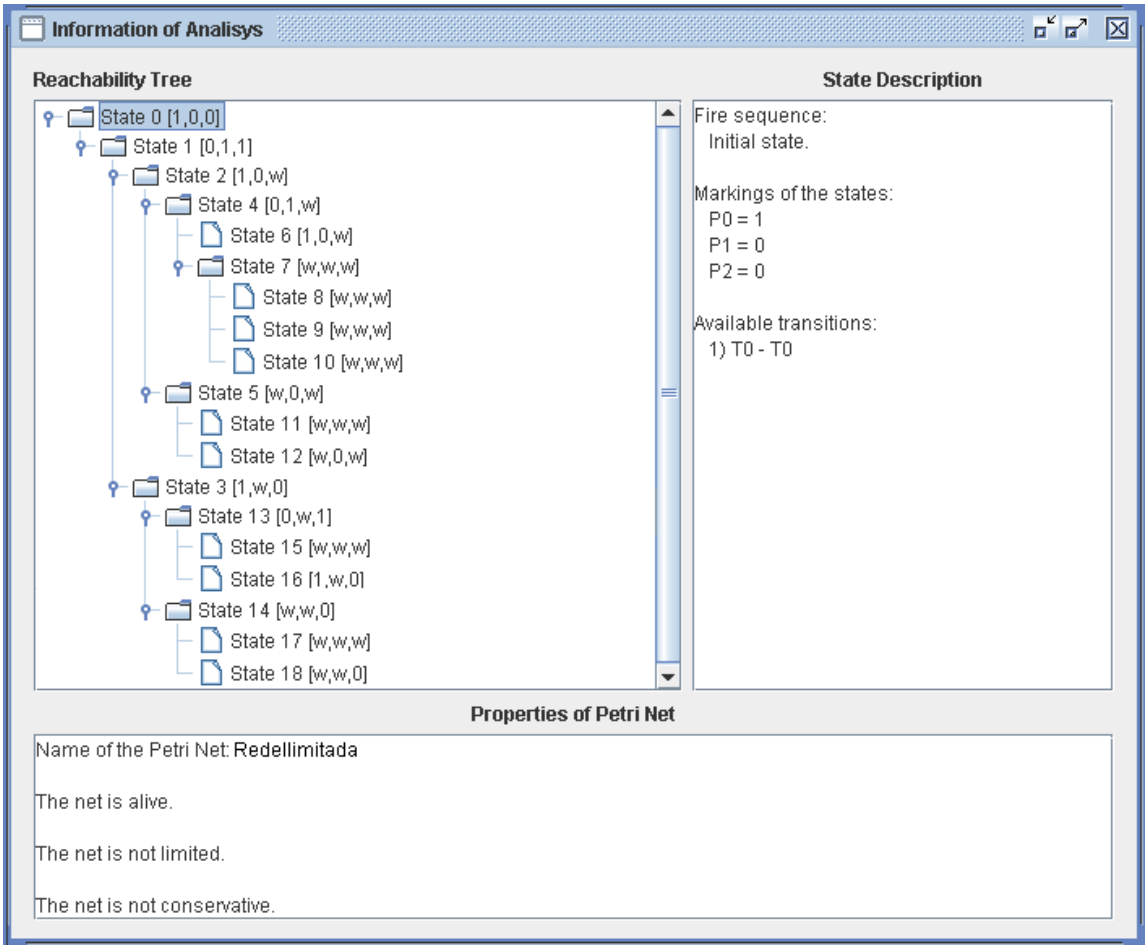


Figura 4.16 Janela de Análise Geral para a rede ilimitada

Capítulo 5

Considerações Finais

A ferramenta foi concebida para estar em constante desenvolvimento. Na versão atual ela trabalha apenas com RdP clássicas, porém o núcleo do código foi estruturado para facilitar a inclusão de novas características, tanto em relação ao modelo da rede, quanto às análises e à própria interface gráfica. Como exemplo, estamos trabalhando na inclusão de características temporais nas RdP editadas. Como discutido, outras extensões como RdP estocásticas ou redes coloridas podem ser adicionadas.

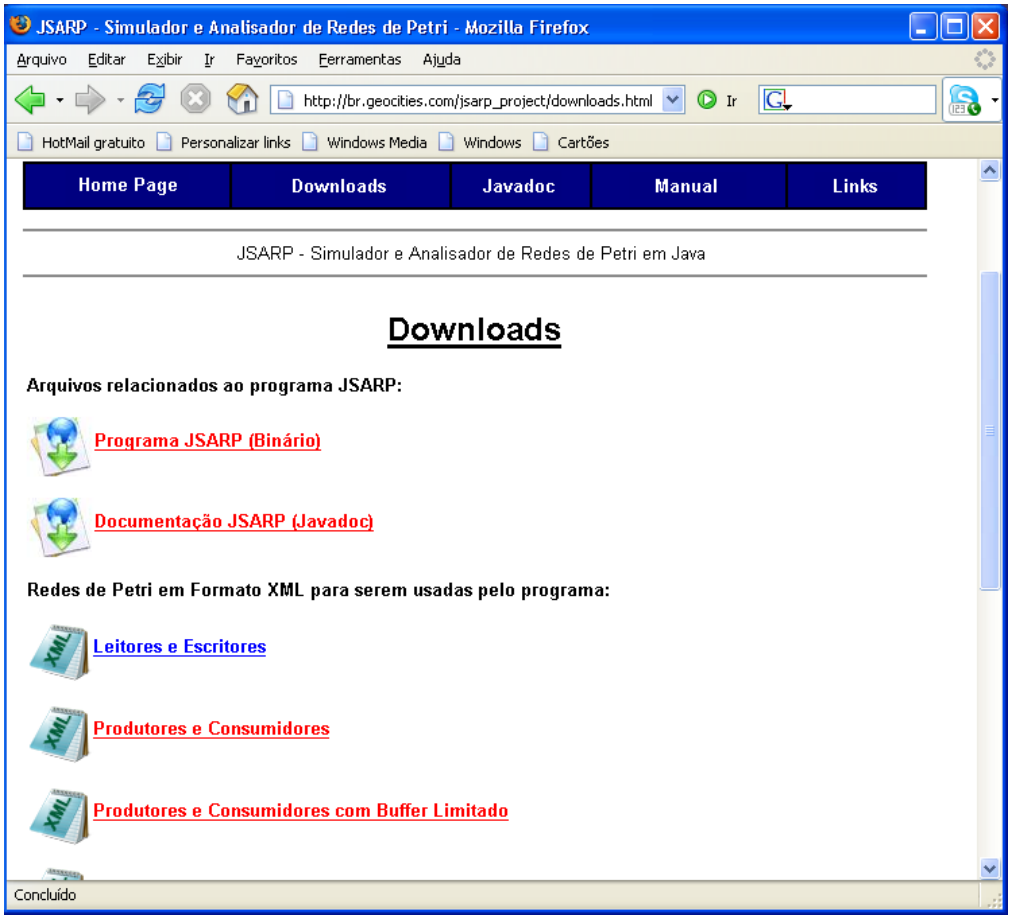


Figura 5.1 Página do projeto

O código da ferramenta será aberto para a comunidade interessada em trabalhar em extensões e, obviamente, a ferramenta será disponibilizada livremente. Uma versão inicial do aplicativo pode ser "baixada" na página da internet http://br.geocities.com/jsarp_project. As redes apresentadas nesta monografia também estão disponíveis para *download*, bem como a documentação do código

gerada pelo *javadoc* e o manual do programa.

Um ponto importante a ser ressaltado em nosso trabalho é que, além da ferramenta poder ser utilizada para fins de pesquisa, pode ser também usada para fins didáticos. Sua interface gráfica amigável, a interação com o usuário e a forma gráfica de apresentar os resultados de análise, facilitam o aprendizado e despertam o interesse de novos alunos.

Por fim, nossa implementação conseguiu reunir as características básicas desejadas no início de sua concepção: ser independente de plataforma, objetivo atingido por usarmos a linguagem Java; ser extensível e reutilizável, objetivo conseguido através da boa prática da programação orientada a objetos. Cumprimos também com a meta de reunir em um só programa um editor gráfico, um analisador e um simulador interativo.

Referências

[01] Maziero, C. (1990). “ARP - Analisador de Redes de Petri”. Disponível [Online] http://www.ppgia.pucpr.br/~maziero/diversos/petri/, dezembro , 2006.

[02] Petri Nets World. Disponível [Online] http://www.informatik.uni-hamburg.de/TGI/PetriNets/ TGI group at the University of Hamburg, Germany, dezembro, 2006.

[03] Petri, C. A. Kommunikation mit Automaten. Dissertation, Rheinisch Westfalisches Institut fur Instrumentelle Mathematik an der Universitat Bonn, Bonn, 1962.

[04] Marranghello, N. (2005). “Redes de Petri: Conceitos e Aplicações”. Disponível [Online] http://www.dcce.ibilce.unesp.br/~norian/cursos/mds/ApostilaRdP-CA.pdf, janeiro, 2007.

[05] JARP. Disponível [Online] http://jarp.sourceforge.net/br/index.html, dezembro, 2006.

[06] PetriTool. Disponível [Online] http://www.csh.rit.edu/~rick/thesis/, abril, 2007.

[07] Brink, R. S. (1996) Tese de Mestrado: “A Petri Net Design, simulation and verification tool”, Department of Computer Engineering College of Engineering Rochester Institute of Technology Rochester, New York, setembro, 1996.

[08] jPNS. Disponível [Online] http://robotics.ee.uwa.edu.au/pns/, dezembro, 2006.

[09] Maziero, C. (1990). Tese de Mestrado “Um ambiente para a análise e simulação de sistemas modelados por redes de Petri”, Universidade Federal de Santa Catarina - UFSC.

[10] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. SIAM Journal of Appl. Math. , 14(6):13901411, November 1966.

[11] Richard M. Karp, Raymond E. Miller, and Shmuel Winogard. The organization of computations for uniform recurrence equations. Journal of the Association for Computing Machinery, 14(3):563590, July 1967.

[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (2003) “Design Patterns Elements of Reusable Object-Oriented Software”, Edited by Addison Wesley.

[13] JGoodies (2006). Disponível [Online] http://www.jgoodies.com, dezembro, 2006.

[14] XStream. Disponível [Online] http://xstream.codehaus.org, dezembro, 2006.

[15] Sun Mircrosystems, “The Java Tutorials: How to Use Trees”. Disponível [Online] http://java.sun.com/docs/books/tutorial/uiswing/components/tree.html, janeiro, 2007.

[16] Merlin, P.M. e Farber, D.J (1976), “Recoverability of Communication Protocols – Implication of a Theoretical Study”. IEEE Transactions on Communications, vol. COM-24, pp. 1036-1042, Setembro. 1976.

[17] PNML, (Software and Systems Engineering – High-level Petri Nets, Part 2: Transfer Format. International Standard ISO/IEC 15909-2. Working Draft Version 0.9.0, June 2005. (Submitted for a combined ISO/IEC SC7 WD/CD registration and CD ballot.).

[18] de Souza Leão, J. L. (2004). “Programação e Validação de Sistemas Multitarefa – Capítulo 5 Redes de Petri”, VI , 76 p, 29,7cm (Rio de Janeiro) COPPE/UFRJ. Disponível [Online] http://www.gta.ufrj.br/%7Eleao/coe717-2004-1/, janeiro, 2007.

[19] Sun Microsystems, “The Java Tutorials: How to Use File Choosers”. Disponível [Online] http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html, janeiro 2007.

[20] Cardoso, J. e Valette, R. (1997) “Redes de Petri”, Edited by UFSC.

[21] Peterson, J. L. (1981). “Petri Net Theory and the Modeling of Systems”, Prentice-Hall, .J., ISBN: 0-13-661983-5.

Apêndice A

Manual do Usuário – JSARP

A.1. Introdução

JSARP é um simulador e analisador de Redes de Petri. É um aplicativo que permite desenhar uma rede, fazer análises, verificar propriedades e simular sua execução de forma interativa.

O programa foi desenvovido usando a linguagem Java, tornando-o independente de plataforma, exigindo apenas que a máquina possua o *Java Runtime Enviroment (JRE)* instalado. A versão do Java utilizada foi o J2SE 1.5 fornecido pela *Sun Microsystems*.

Este manual não tratará da instalação do JRE ou da configuração do sistema operacional para utilização do Java. Também não está no escopo deste manual abordar sobre a teoria de Redes de Petri e suas características. Falaremos apenas do aplicativo, partindo do princípio de que a versão correta do JRE está instalada e que as variáveis de ambiente do sistema operacional estão devidamente configuradas.

O programa foi testado nas plataformas do Windows XP Home Edition e Mandriva Linux Free 2006. Contudo, as informações contidas no manual referentes aos sistemas operacionais sob o qual o programa rodará, servem tanto para as versões do Windows compatíveis com o XP (Win95, Win98, WinME, Win2000 e WinXP) quanto para outras distribuições Linux ou baseadas no Unix. A partir de agora todas as versões do Windows serão tratadas apenas como Windows e as plataformas baseadas no Unix como Unix.

A distribuição do programa é formada pelas seguintes pastas e arquivos:

- **lib**: possui 3 arquivos "*.jar*".
  - **forms-1.0.7.jar** : API Forms do JGoodies usada para criação da interface gráfica.
  - **xpp3_min-1.1.3.4.O.jar** : API para tratar arquivos XML.
  - **xstream-1.2.1.jar** : API para persistir arquivos XML em disco.
- **resource**: possui 2 arquivos "*.properties*".

- **jsarp_pt_br.properties** : Arquivo de idioma português brasileiro.
- **jsarp_eng_us.properties** : Arquivo de idioma inglês.

Mais 4 arquivos estão na pasta raíz:

- **jsarp.jar** : Arquivo jar executável caso o sistema operacional esteja configurado para executar arquivos de extensão jar com o JRE adequado.
- **start.sh** : Script que inicia o programa para sistemas operacionais compatíveis com o Unix, como o Linux por exemplo.
- **start.bat** : Script que inicia o programa no Windows.
- **Version.txt** : Arquivo de texto com as informações sobre o autor, contato, e data e hora de criação da distribuição.

Para iniciar o programa através do Windows, pode-se dar um duplo clique no arquivo jsarp.jar, caso o sistema operacional esteja configurado para executar arquivos de extensão jar usando o JRE. Outra opção no Windows é executar o arquivo start.bat. Para iniciar o programa no Unix é só executar o script start.sh.

Ainda é possível executar o programa através de um *shell*, chamando o java:

```
java –jar jsarp.jar
```

Após a Visão Geral, o manual é dividido da seguinte forma, na Seção A.2 falamos sobre as opções do Menu Arquivo; como editar/desenhar uma Rede é tratado na Seção A.3; o modo de simulação é discutido na Seção A.4; na Seção A.5 a análise é abordada.

A.1.1. Visão Geral

A Figura A.1 apresenta a Janela Principal do programa assim que ele é carregado. O idioma por *default* ao se carregar o programa é o inglês.

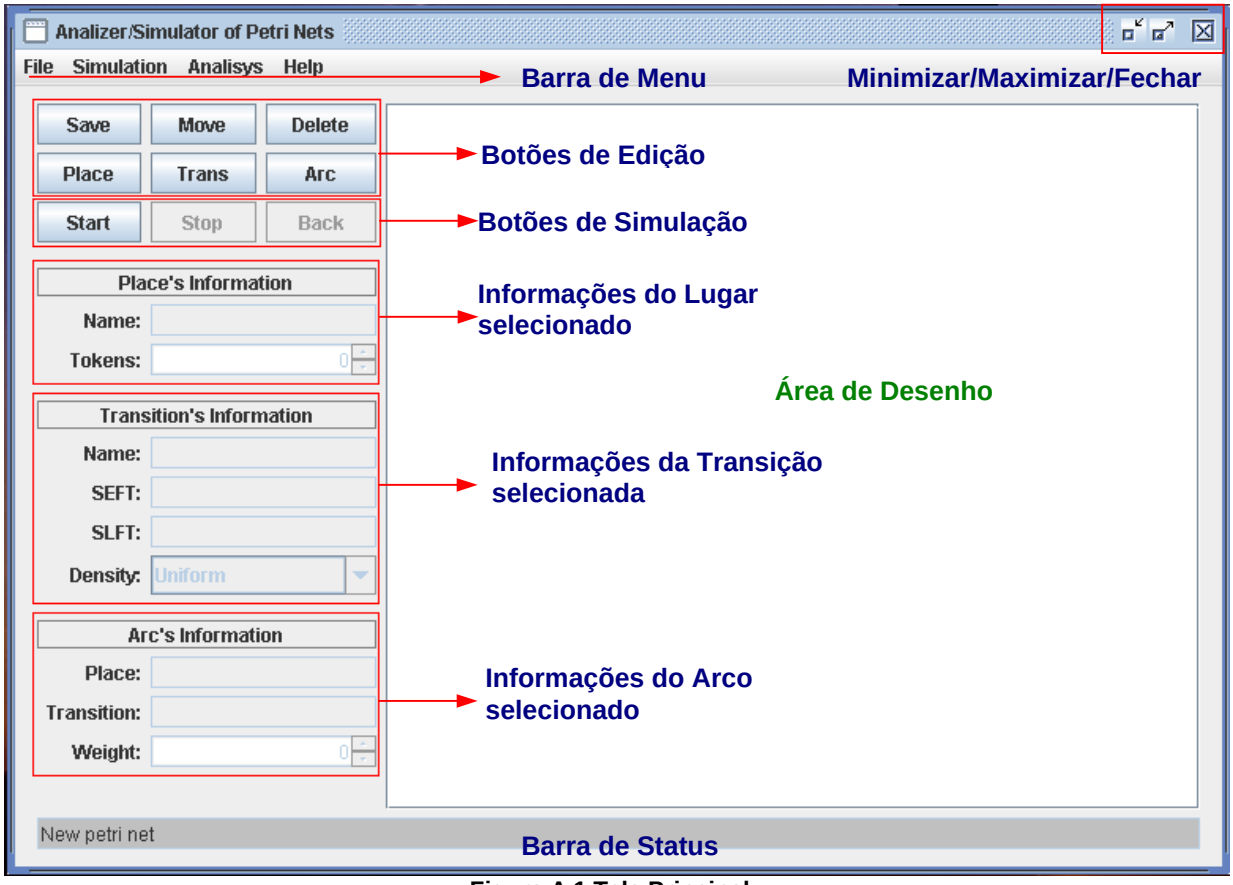


Figura A.1 Tela Principal

Na parte superior temos a Barra de Menu; no canto superior direito os botões Minimizar, Maximizar e Fechar.

Ocupando a maior parte da janela temos a Área de Desenho. Nela é que se fará a edição e o desenho da Rede.

A parte esquerda da tela apresenta: os botões relacionados à edição da Rede de Petri; botões referentes à simulação; painéis onde serão apresentadas e editadas as informações dos componentes de uma Rede de Petri correntemente selecionados; painel de Lugar; painel de Transição e painel de Arco.

Na parte inferior temos a Barra de Status. Nela serão apresentadas informações importantes ao usuário conforme se vai utilizando o programa.

A.2. Menu Arquivo

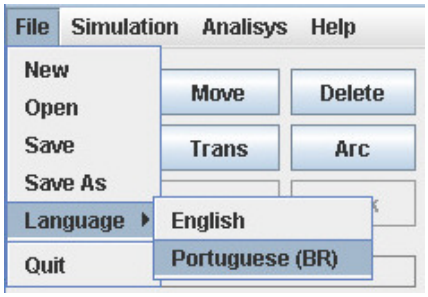


Figura A.2 Menu Arquivo

O Menu Arquivo possui as opções de: criar uma nova rede, abrir uma rede existente, salvar o arquivo, alterar o idioma e sair do programa.

A.2.1. Novo Arquivo

Ao se clicar na primeira opção do Menu, *File → New*, a área de desenho será completamente apagada, e será criado um novo contexto para o desenho de uma nova rede.

A.2.2. Abrir Arquivo

Quando escolhemos essa opção, *File → Open*, é aberta uma caixa de diálogo que permitirá que se escolha algum arquivo de extensão XML para ser aberto.

Para abrir efetivamente o arquivo selecionado é só clicar em *Open*. Caso haja desistência da operação, deve se clicar em *Cancel*.

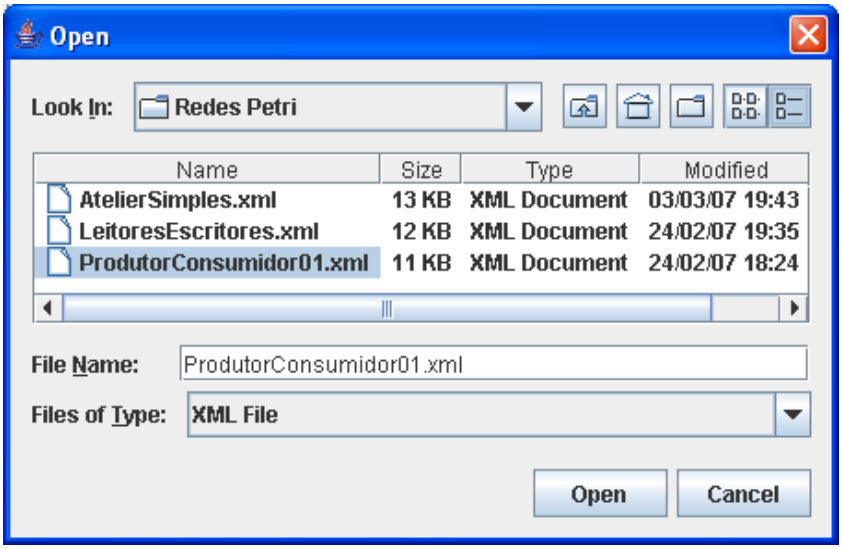


Figura A.3 Abrir Arquivo

A.2.3. Salvar Arquivo

Existem 2 opções para salvar o arquivo. A primeira na realidade sobrescreve o último arquivo aberto ou salvo anteriormente. A segunda abre uma caixa de diálogo perguntando onde o arquivo deve ser salvo e qual o nome que será dado.

O nome do arquivo será o nome da Rede de Petri. Automaticamente o programa acrescentará a extensão XML caso não tenha sido colocada essa extensão no nome do arquivo.

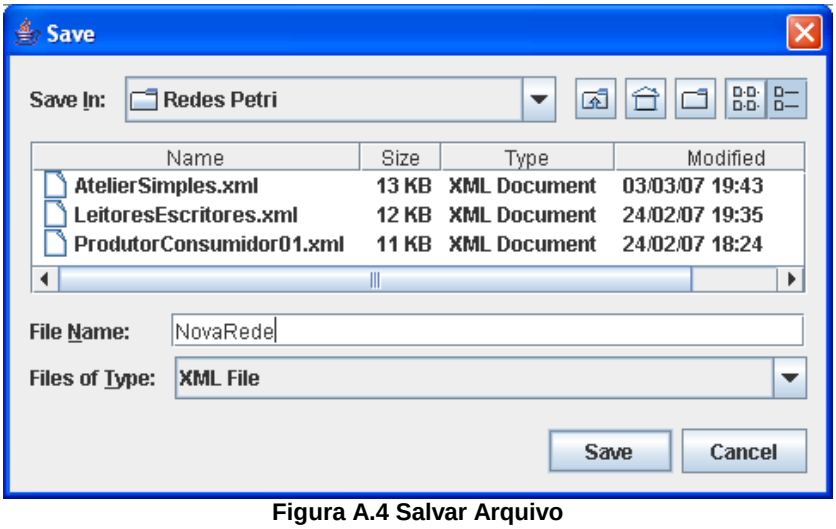


Figura A.4 Salvar Arquivo

Para salvar efetivamente o arquivo deve se clicar em *Save*. Para cancelar a operação deve se clicar em *Cancel*.

A.2.4. Alterar Idioma

Os únicos idiomas disponíveis nessa versão do programa são o inglês e o português brasileiro.

Para alterar o idioma basta ir em *File* → *Language* e no sub-menu aberto clicar na linguagem desejada.

A.3. Modo de Edição

O modo de edição é o padrão do programa. Este modo permite que se desenhe a rede na área de desenho e que seus objetos sejam configurados.

Dividimos esse trecho do manual nos três objetos que compõe as Redes de Petri: lugar (*Place*), transição (*Trans*) e arco (*Arc*).

Figura A.5 Botões: Lugar, Transição e Arco

A.3.1. Editar Lugar

Primeiramente para editar um lugar é necessário que ele seja adicionado à Rede de Petri. Para tal, é preciso clicar com o botão esquerdo do mouse no botão correspondente ao lugar e depois em uma região livre da área de desenho.

Uma vez que o lugar exista na rede, pode-se editá-lo alterando seu nome e quantidade de fichas. Para edição também é necessário que o botão correspondente ao lugar seja clicado. Após feito isso seleciona-se um lugar, clicando sobre ele com o botão esquerdo do mouse na área de desenho.

O lugar correntemente selecionado fica em verde enquanto os demais permanecem em azul. A parte da tela referente às informações de lugar é habilitada permitindo a edição enquanto as partes referentes à transição e arco são desabilitadas.

A Figura A.6 mostra 3 lugares, da esquerda para direita. O primeiro possui 3 fichas; o segundo uma única ficha (representada por um ponto vermelho), e o terceiro não possui nenhuma ficha (representado pela circunferência vermelha). Na mesma figura é mostrado que o lugar do meio está selecionado, suas informações são exibidas no painel à esquerda e é permitida sua edição.

Figura A.6 Edição de Lugar

Para mover o lugar de posição precisa-se clicar no botão *Move* e em seguida pressionar o botão esquerdo do mouse sobre o lugar desejado na área de desenho, arrastá-lo até a posição que se queira, e soltar o botão do mouse.

Para remover um lugar da rede clicamos no botão *Delete* e em seguida clicamos com o botão esquerdo do mouse sobre o lugar.

Figura A.7 Botões: Salvar, Mover e Apagar

A.3.2. Editar Transição

Transições são trabalhadas no programa de forma semelhante ao lugar. Para adicionar ou editar uma transição deve se clicar primeiro sobre o botão que representa a transição (*Trans*). Depois sobre uma região livre da área de desenho. Quando se quer editar uma transição existente, ao invés de se clicar sobre uma região livre, clica-se sobre a transição existente.

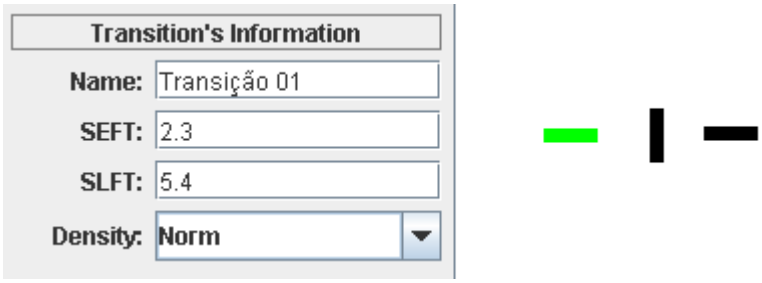


Figura A.8 Edição de transição

Quando uma transição é selecionada ela fica com a cor verde. A parte da tela com as informações de transição torna-se habilitada, enquanto as demais são desabilitadas.

A posição da transição pode ser horizontal (*default*) ou vertical. Para alternar as posições clica-se com o botão direito do mouse sobre a transição.

Para mover a transição, de forma semelhante ao lugar, clica-se sobre o botão *Move* depois pressiona-se o botão esquerdo do mouse sobre a transição, arrasta-se até a posição desejada e solta-se o botão.

Remover uma transição também é simples. Clica-se no botão *Delete* depois com o botão esquerdo sobre a transição.

A.3.3. Editar Arco

Arcos são tratados de forma diferenciada de lugares e transições. Eles obrigatoriamente devem ter como extremos o par transição-lugar.

Para adicionar um arco deve se clicar primeiramente no botão relacionado ao Arco (*Arc*). Depois, na área de desenho clica-se sobre o objeto de origem do arco. Caso o objeto origem seja um lugar o objeto destino deve ser uma transição, caso o objeto origem seja uma transição o objeto destino deve ser um lugar. Após clicar sobre o objeto origem pode-se clicar livremente sobre qualquer região livre da área

de desenho, fazendo com que o arco seja composto por vários pontos, e, por fim, clicar sobre o objeto destino.

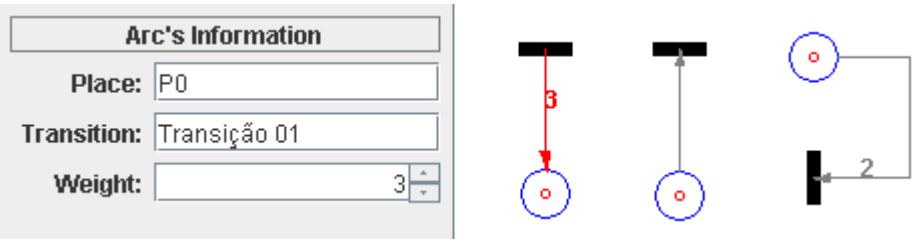


Figura A.9 Edição de Arco

A Figura A.9 apresenta 3 arcos; o primeiro selecionado em vermelho com suas informações aparecendo na parte de informações de Arco, com peso 3, origem em uma transição e fim em um lugar; a segunda tem origem em um lugar e fim em uma transição, possui peso unitário (a ausência de rótulo significa peso igual a 1); a última transição é composta por 2 pontos intermediários entre o objeto de início e fim, possui peso 2, tem origem em um lugar e fim em uma transição.

Quando um arco é selecionado a parte com informações de Arco é habilitada enquanto as demais são desabilitadas.

Outra particularidade quanto ao desenho do arco é que pode-se configurar a que orientação do lugar, o extremo estará ligado. Se ao norte, leste, sul ou oeste do lugar. Para trocar a orientação no sentido horário clica-se primeiro no botão referente ao Arco (*Arc*) e depois com o botão direito do mouse sobre o Arco desejado.

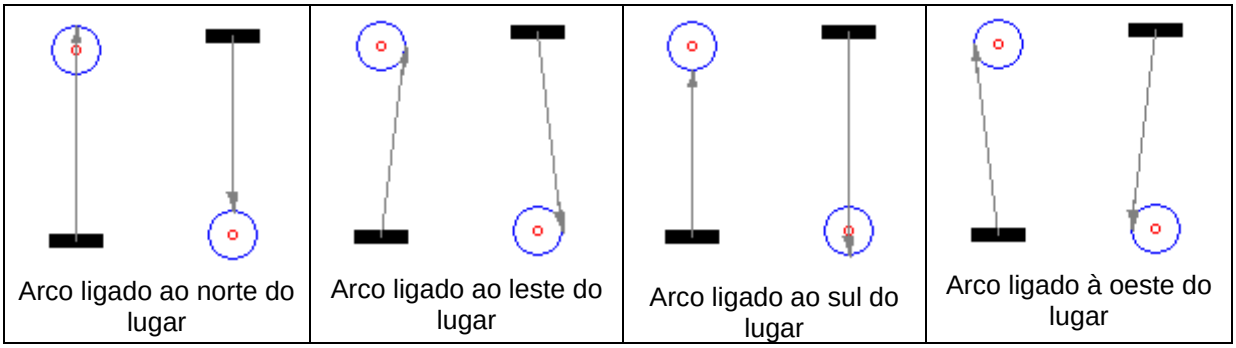


Figura A.10 Posicionamento do Arco em relação ao Lugar

É possível ainda mover os pontos intermediários de um arco, clicando no botão *Move*, depois pressionando o botão esquerdo do mouse em alguma parte do segmento de reta que forma o arco próximo do ponto que se quer mover, arrastar o mouse até a posição desejada e soltar o botão.

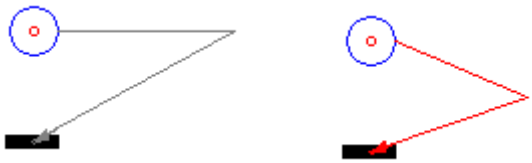


Figura A.11 Ponto intermediário do arco movido para outra posição

Para remover um arco simplesmente clica-se sobre o botão *Delete* depois com o botão esquerdo do mouse sobre o arco.

A.4. Modo de Simulação

O modo de simulação, como o próprio nome diz, é que fará com que a Rede possa ser "executada", ou seja, simulada através do disparo de suas transições.

Para entrar no modo de simulação clica-se no botão *Start*. Ao clicar nesse botão o fundo da área de desenho torna-se acinzentado e as transições habilitadas para disparo ficam em vermelho. Os botões relacionados à edição da Rede de Petri são desabilitados. Apenas os referentes à simulação são habilitados: *Stop* e *Back*.

Quando se deseja disparar uma transição, clica-se com botão esquerdo do mouse sobre ela. Se quiser apenas selecioná-la para exibir suas informações no painel de informações de transição, clica-se com o botão direito do mouse sobre ela. A transição correntemente selecionada ficará em verde (caso ela não esteja habilitada para disparo).

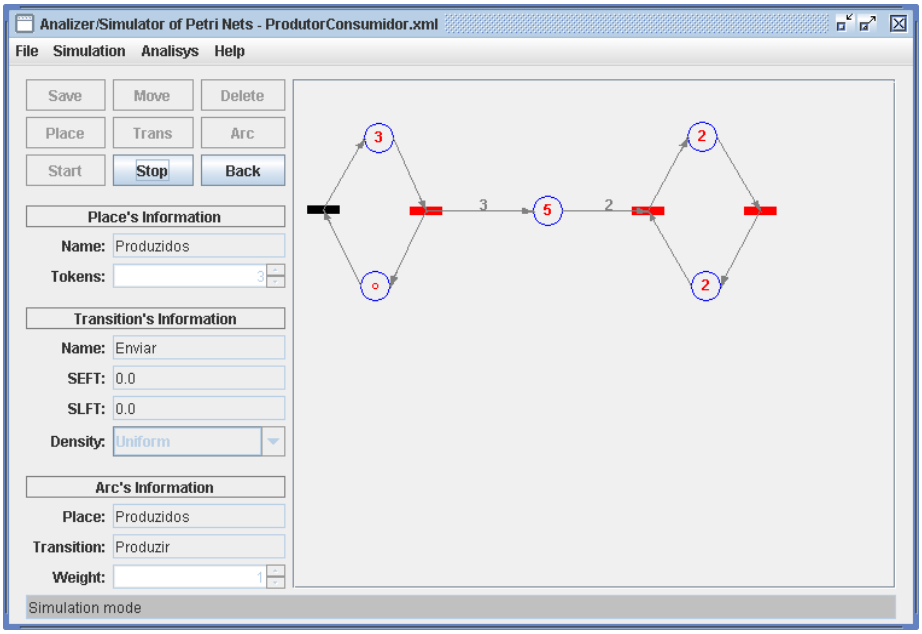


Figura A.12 Janela Principal no modo de simulação com as transições habilitadas em vermelho

Pode-se selecionar um arco ou lugar clicando-se sobre ele com qualquer botão do mouse. O lugar correntemente selecionado ficará em verde e um arco selecionado em vermelho. As informações serão exibidas no painel correspondente. Repare que apesar das informações serem exibidas elas não são editáveis.

A.4.1. Janela de Simulação

A Figura A.13 mostra a Janela de Simulação que é aberta assim que o botão *Start* é clicado. A tela é dividida em duas partes: no lado esquerdo fica a Árvore com a Seqüência de Disparos que, no começo, possui apenas o Estado Inicial. A árvore vai crescendo conforme as transições são disparadas. No lado direito é apresentada uma descrição do estado correntemente selecionado.

A Árvore com a Seqüência de Disparos apresenta o identificador da transição disparada (T*n*) além de, entre colchetes, a marcação da rede.

As informações apresentadas sobre o estado selecionado são:

- Seqüência de disparos: ordem nas quais as transições foram disparadas para chegar ao estado.
- Marcação do estado: quantidade de fichas em cada lugar da Rede de Petri.
- Transições disponíveis: informa quais transições estão habilitadas para disparo.

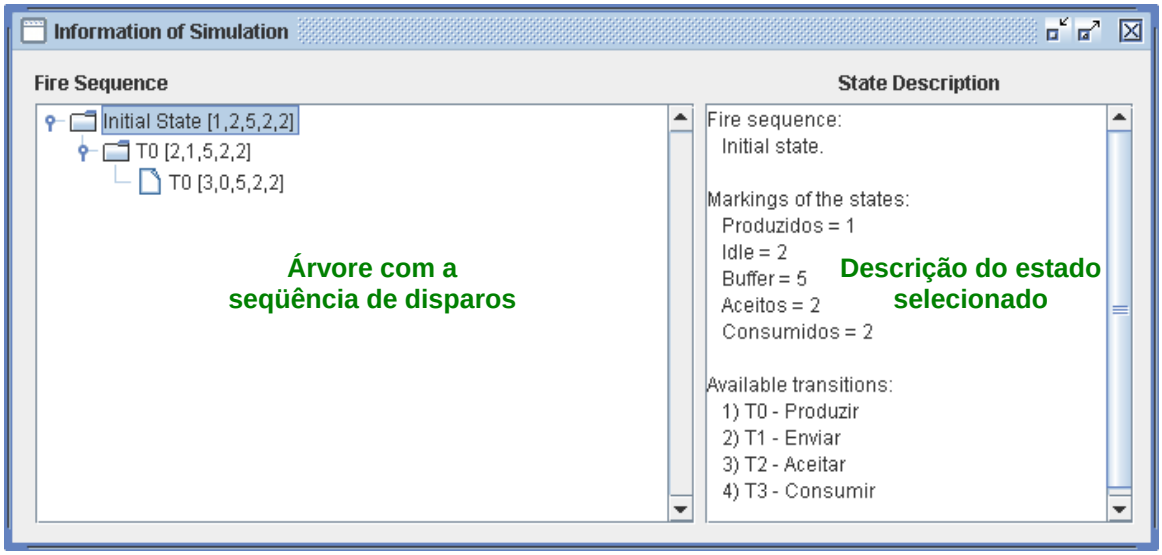


Figura A.13 Janela de Simulação

A.4.2. Voltando a um Estado da Rede

O programa permite ao usuário ir para qualquer estado presente na Árvore de Seqüência de Disparos. Para isso seleciona-se o Estado desejado na Janela de Simulação e clica-se no botão *Back* da Janela Principal.

A partir daí serão feitas ramificações na Árvore caso as próximas transições disparadas sejam diferentes das transições disparadas anteriormente.

A Figura A.14 mostra a Janela de Simulação à frente com o estado de marcação [2,1,3,3,1] selecionado. Ao fundo está a Janela Principal com a rede após voltar para o estado selecionado.

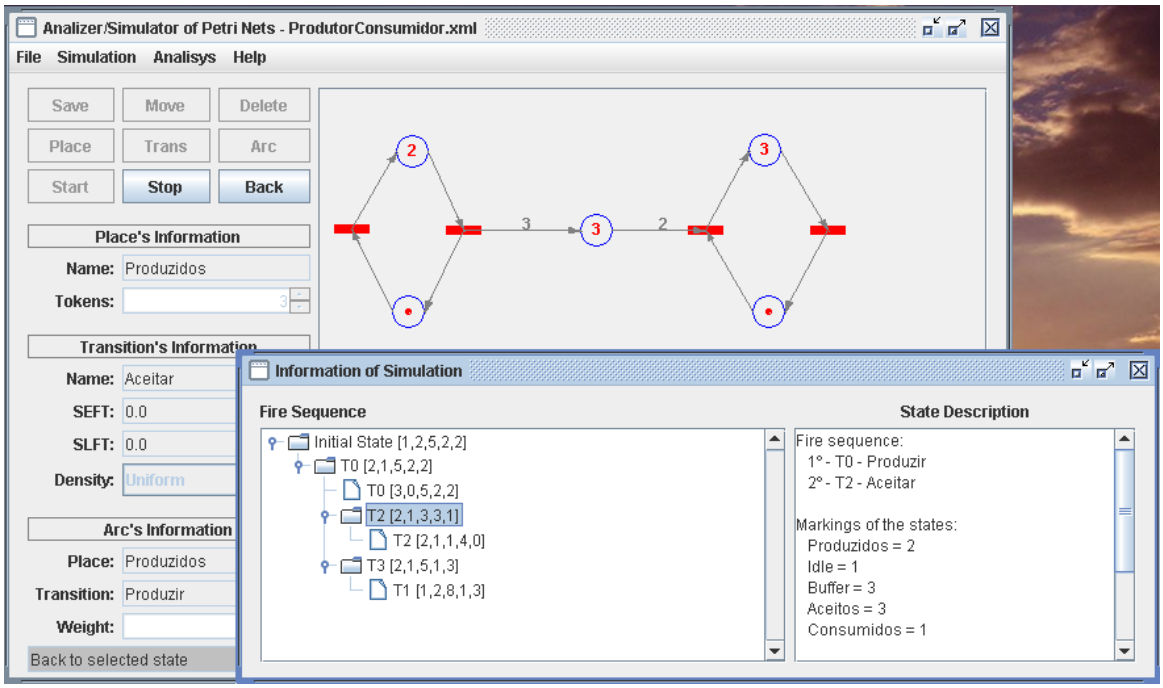


Figura A.14 Janela de Simulação na frente da Janela Principal

Para encerrar a simulação, clica-se no botão *Stop*. Automaticamente, a Janela de Simulação será fechada e o programa volta para o modo de edição.

A.5. Análise Geral

Várias análises e verificações podem ser feitas sobre uma Rede de Petri. Nosso aplicativo faz algumas análises gerais sobre a rede, testando vivacidade, limitação, bloqueio e conservação; além de gerar a Árvore de Alcançabilidade.

Para que o programa apresente ao usuário as análises sobre a rede é só ir no menu em: *Analisys* → *General Analisys*.

A.5.1. Janela de Resultados

A Janela de Resultados apresenta a Árvore de Alcançabilidade, propriedades verificadas na Rede e a descrição do estado correntemente selecionado na Árvore de Alcançabilidade.

A Árvore de Alcançabilidade tem como rótulo de seus nós o nome “State *n*”, onde *n* é a ordem na qual o estado foi criado durante a geração da árvore. Também faz parte do rótulo a marcação da rede no estado (entre colchetes).

A descrição do estado possui as informações:

- Seqüência de disparos: ordem nas quais as transições foram disparadas para chegar ao estado.
- Marcação do estado: quantidade de fichas em cada lugar da Rede de Petri. O símbolo “w” na marcação de um lugar representa que foi identificado um crescimento grande de fichas naquele lugar.
- Transições disponíveis: informa quais transições estão habilitadas para disparo.
- Estado duplicado: caso o estado selecionado seja duplicata de algum estado criado anteriormente na geração da rede, teremos a informação dizendo que ele é um estado duplicado e o identificador do estado original.

Por fim nesta janela apresentamos ainda as propriedades da Rede:

- Vivacidade: informa se rede é ou não viva
- Limitação: informa se a rede possui ou não limitação, caso seja limitada informará também qual o limite da rede.
- Conservativa: informa se a rede é ou não conservativa, caso seja conservativa informará também qual o total de fichas da rede.
- Bloqueio: caso a execução da rede leve a algum estado de *deadlock*, informará quais são os estados de *deadlock* e as seqüências de disparos que levam a eles.

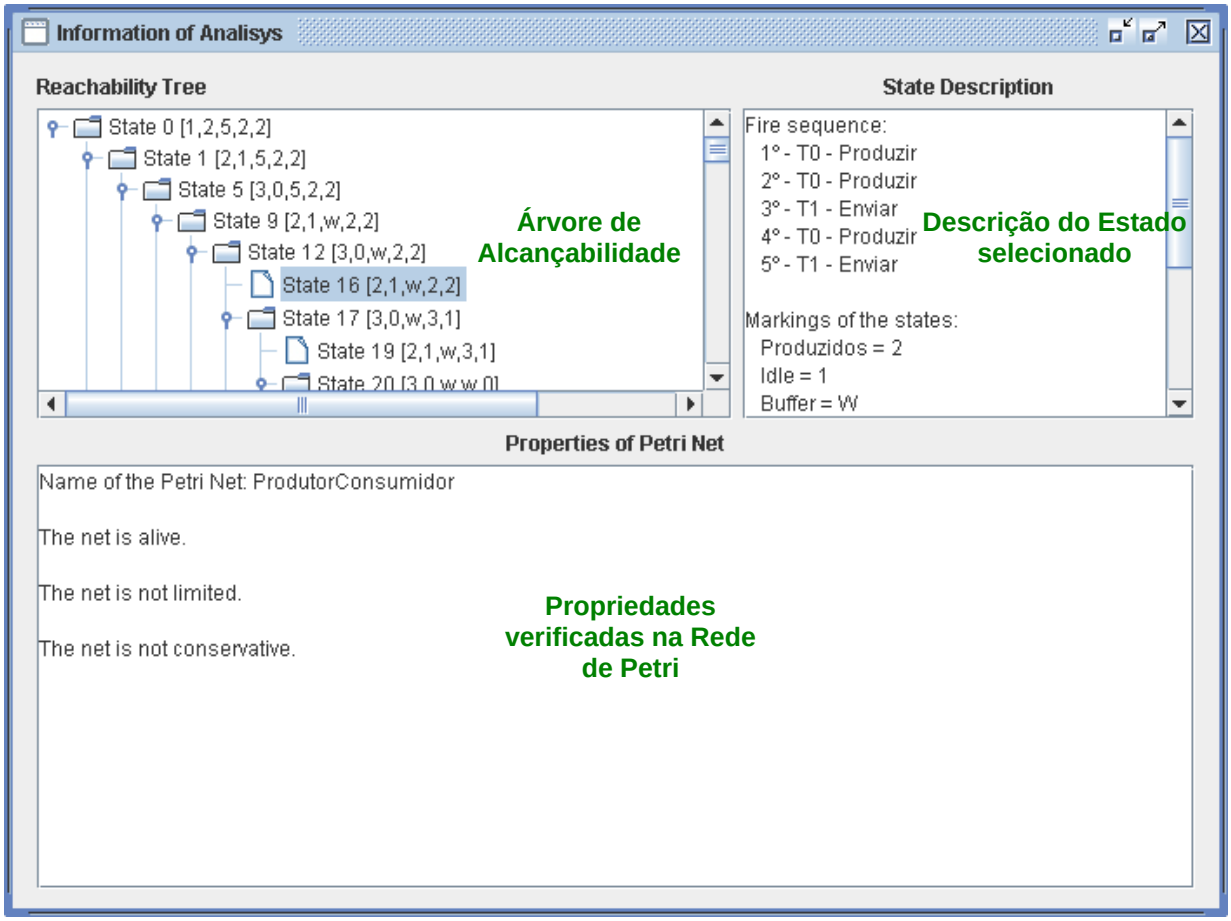


Figura A.15. Janela de Análise Geral

A qualquer momento podemos fechar a Janela de Análise clicando no botão fechar localizado no canto superior direito.

Apêndice B

Classes Auxiliares

Neste apêndice tratamos sobre as classes auxiliares criadas para dar suporte à implementação do simulador e analisador de Redes de Petri sem estarem diretamente associados ao estudo das Redes de Petri. Podem ser livremente usadas em contextos completamente diferentes do abordado pelo nosso projeto.

B.1. Classe *br.uerj.language.LanguageTool*

Esta classe foi criada com objetivo de tornar o programa multilingüe. Ela trabalha manipulando arquivos *properties* e uma tabela com uma *String* chave e uma *String* valor que será exibida na tela.

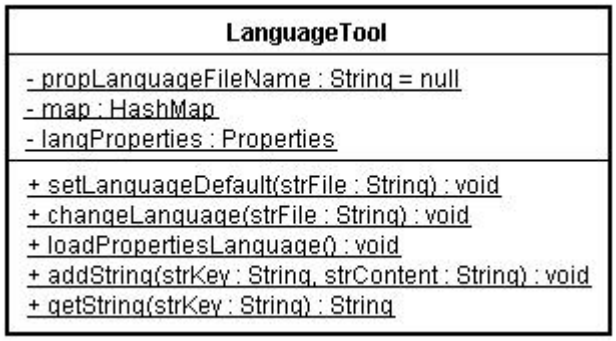


Figura B.1 Classe LanguageTool

Seus principais atributos são um objeto *Properties* e um objeto *HashMap*. Ambos tem o mesmo objetivo de armazenar o par chave e valor. O *Properties* é que será efetivamente usado para permitir o uso de vários idiomas no programa. A chave será usada pelo programa para pegar as informações contidas no arquivo de *properties*. A *HashMap* será usada como forma de contingência caso não se encontre a chave no arquivo *properties* ou ocorra falha ao tentar acessar o arquivo.

O resumo do funcionamento é o seguinte: primeiramente o programa invoca o método *setLanguageDefault* passando como parâmetro o nome do arquivo *properties*. Em seguida o arquivo é carregado através do método *loadProperties*. No próximo passo o aplicativo cadastra vários pares de nome e valor *default* a serem usados em caso de falha invocando o método *addString*. Após a execução desses passos o programa vai tentar pegar as *Strings* existentes no arquivo *properties* invocando o método *getString* que recebe como parâmetro a chave. Caso não se

encontre a chave no arquivo ou tenha ocorrido falha na abertura do arquivo será usado o valor *default* cadastrado na *HashMap*.

```
01 public static String getString(String key)
02 {
03  if(map != null)
04  {
05    if(langProperties == null)
06        return (String) map.get(key);
07    else
08        return lang.Properties.getProperty(key,(String)map.get(key));
09  }
10  else
11  {
12    if(langProperties!= null)
13        return langProperties.getProperty(key);
14    else
15        return null;
16  }
17 }
```

Listagem B.1 Método *getString*

Para alterar o arquivo *properties* usado como referência invoca-se o método *changeLanguage* passando como parâmetro o nome do arquivo.

Essa estrutura permite que o programa funcione corretamente mesmo que haja falhas na leitura de arquivo.

B.2. Classe *br.uerj.swing.JTextFieldExtended*

Essa classe herda de *JTextField*. Foi criada para ajudar no suporte multilingüe recebendo um novo atributo *String* que é a chave atualmente usada pela caixa de texto para exibir seu conteúdo.

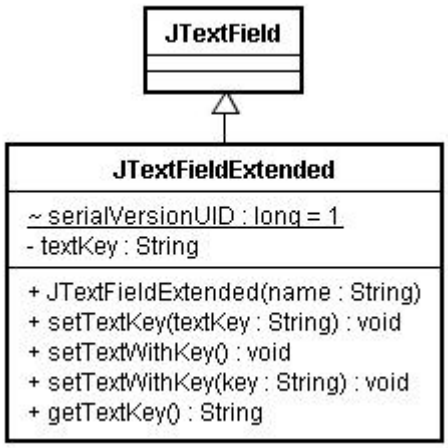


Figura B.2 Classe JTextFieldExtended

Uma vez que o objeto guarda a informação de qual é a chave atual, quando o idioma é trocado podemos recuperar qual deve ser o novo conteúdo da caixa de

texto usando a chave armazenada. Essas operações são feitas através dos métodos *setTextWithKey* tanto o que recebe a chave como parâmetro, que seta o novo valor da chave, quanto o que não recebe parâmetros e usa a chave armazenada.

```
01 public void setTextWithKey()
02 {
03   String text = LanguageTool.getString(getTextKey());
04   super.setText(text);
05 }
```

Listagem B.2 Método *setTextWithKey*

B.3. Classe *br.uerj.FileFilter.FileFilterXML*

Classe que herda de *FileFilter* sobrescrevendo os métodos: *accept* e *getDescription*.

O primeiro recebe como parâmetro um objeto *File* e retorna verdadeiro caso esse arquivo seja aceito pelo filtro, retorna falso caso contrário. O segundo método retorna uma *String* com uma descrição do filtro.

Em nossa implementação esse filtro é usado pelas caixas de diálogo Abrir e Salvar arquivo em disco.

B.4. Classe *br.uerj.FileFilter.FileFilterUtil*

Classe para auxiliar a construção de filtros de arquivo. Possui apenas um método estático *getExtensionOfFile* que recebe uma *String* que representa o nome do arquivo e retorna uma *String* que é a extensão do arquivo, ou seja, a substring após o último caractere “.” do nome do arquivo.