ANDRÉ TORRES FERRAZ DE MELLO
MARCELO CASTRO BARBOSA

TRANSCRITOR DE REDES DE PETRI PARA LINGUAGEM DE PROGRAMAÇÃO DE MÁQUINA

Monografia final apresentado à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Engenheiro Mecatrônico

São Paulo
2010

ANDRÉ TORRES FERRAZ DE MELLO
MARCELO CASTRO BARBOSA

TRANSCRITOR DE REDES DE PETRI PARA LINGUAGEM DE PROGRAMAÇÃO DE MÁQUINA

Monografia final apresentado à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Engenheiro Mecatrônico

Área de Concentração:
Engenharia Mecatrônica

Orientador:
Prof. Dr. Fabrício Junqueira

São Paulo
2010

Às nossas famílias e amigos, que nos apoiaram durante a elaboração do trabalho e durante todo o curso na Escola Politécnica.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DA ENGENHARIA MECÂNICA/NAVAL DA ESCOLA POLITÉCNICA (EPMN) – USP.

Mello, André Torres Ferraz de
Transcritor de redes de Petri para linguagem de programação de máquina / A.T.F de Mello, M.C. Barbosa. -- São Paulo, 2010.
95 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Rede de Petri 2.Linguagem de máquina I.Barbosa, Marcelo Castro II.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III.t.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Rede de Petri é uma poderosa ferramenta gráfica para a modelagem de sistemas a eventos discretos. No entanto, a transcrição da Rede de Petri para as linguagens de programação de controladores lógicos programáveis (CLPs) é ainda feita manualmente dada a inexistência de transcritores que transformem automaticamente o formato em uma linguagem programável. O alvo deste trabalho é a solução deste problema por meio do projeto e implementação de um transcritor capaz de interpretar Rede de Petri e traduzí-la para linguagens de programação de CLP. Para alcançar este objetivo, foi realizada uma pesquisa preliminar, de modo a avaliar o cenário atual da utilização de Rede de Petri, seus diferentes tipos e capacidades de modelagem. As linguagens de programação da IEC 61131-3 também foram abordadas. Depois, os padrões de leitura e escrita do transcritor foram definidos: o programa lê redes de Petri do tipo SIPN (S*ignal Interpreted Petri Nets*) em formato PNML (*Petri Net Markup Language*), um padrão de intercâmbio recentemente estabelecido, e as compila em três linguagens de programação de CLP: *ladder diagram*, *sequential function chart* e *structured text*, todas através de algoritmos específicos. O formato de saída é o PLCOpen XML, uma linguagem que tem como objetivo a padronização das linguagens de programação da IEC 61131-3 e do seu intercâmbio. O transcritor ainda permite a simulação da rede carregada de maneira passo-a-passo, permitindo ao usuário final a verificação de possíveis erros de modelagem durante o desenvolvimento. Com isso, o transcritor cria um fluxo de trabalho completamente automatizado para o desenvolvimento e programação de CLPs utilizando-se redes de Petri como linguagem base.

ABSTRACT

The Petri net is a powerful graphic tool for discrete event systems modeling. However, translation from Petri nets to programmable logical controller (PLC) programming language, in order to provide dynamic project integration, is still done manually, given the unavailability of translators to compile the format into direct programming language. The goal of this paper is to address this drawback by developing and implementing a comprehensive translator, capable of interpreting a Petri net and translating it to PLC-ready language. To achieve this goal, a preliminary research was made, in order to evaluate the many extensions of the original Petri net language, their characteristics and modeling power. The IEC 61131-3 PLC programming languages were also studied. After that, the inputs and outputs of the translator were chosen: the software supports SIPN (Signal Interpreted Petri Nets) in PNML format (Petri Net Markup Language), a recently stablished interchange standard, and compiles it into three different PLC languages: ladder diagram, sequential function chart and structured text, through specific algorithms. The output language is the PLCOpen XML, a format that is bent on the standardization of the PLC programming languages and their interchange. The program also supports stepwise simulation of the chosen Petri net, allowing for error-checking during the translation process. In this way, the translator creates a seamless development workflow for PLCs, using Petri Nets as the base starting language.

1. Introdução

Os Controladores Lógicos Programáveis (CLPs) são poderosas ferramentas na área de automação sendo aplicados tanto em máquinas pequenas e simples quanto em grandes plantas de manufatura (linha de produção) (FREY, 2001). Os CLPs são muito utilizados e difundidos para a automação industrial. No entanto, existe uma deficiência na área de linguagem de programação para CLPs no que diz respeito à uma linguagem gráfica.

Na programação, existe uma separação comum entre os tipos de linguagem textual e gráfico, que se dá em função da ênfase dada à utilização de textos ou de gráficos e diagramas[1]. Devido à essa diferença, há também inúmeras discussões que enumeram vantagens e desvantagens de ambas num modelo comparativo. No caso das linguagens de programação de computadores e notações matemáticas, a linguagem textual predomina (BARROS, 2006). Apesar das linguagens gráficas já terem conquistado seu espaço (OMG, 2003), elas são vistas com frequência como inerentemente menos precisas que as linguagens textuais. Entretanto, Harel e Rumpe (2004) *apud* (BARROS, 2006) afirmam que essa visão não é verdade: uma linguagem textual pode não ser precisa enquanto que uma gráfica pode sê-lo.

Especificamente, as deficiências estão na dificuldade de aplicação e implementação da linguagem e na incapacidade de descrever graficamente algoritmos sequenciais e concorrentes e permitir uma análise visual sobre a evolução dos estados (estado atual, próximo estado, etc.) (FREY, 2001).

No entanto, existe uma linguagem gráfica que possui atributos fortes para modelagem e controle de sistemas a eventos discretos: a rede de Petri. Segundo Valk (2003) *apud* (Barros,2006), as vantagens comparativas da Rede de Petri sobre as demais linguagens consistem na representação gráfica simples (círculos, retângulos e setas), representação algébrica simples (na Rede de Petri de baixo nível), estruturação do algoritmo com dualidade de *lugar-transição* e, por último, a localidade do efeito das *transições*. Com isso, a Rede de Petri permite modelar e validar a sequência do algoritmo de controle

[1] Deve-se notar que as linguagens gráficas podem ser geradas a partir de uma descrição textual.

por meio da simulação. Entretanto, não há a conversão direta de Rede de Petri para as linguagens de programação de CLP.

1.1. Objetivo

Portanto, o trabalho se propõe à reduzir tal deficiência, encontrada na área de programação de CLPs, propondo um transcritor de Rede de Petri para linguagem de máquina de modo a permitir que todas as vantagens comparativas da Rede de Petri possam ser diretamente aplicadas à programação de CLP.

1.2. Estrutura do trabalho

De modo a apresentar a pesquisa e o trabalho realizado, este documento está dividido em três seções: primeiramente, apresenta-se o levantamento bibliográfico realizado sobre os temas abordados: redes de Petri e as linguagens da IEC 61131-3, bem como suas linguagens de intercâmbio entre *softwares*. Em seguida, apresenta-se o algoritmo utilizado para a transcrição. Por fim, é mostrada a estrutura do transcritor proposto e desenvolvido.

2. Rede de Petri

2.1. Definição

2.1.1. A definição original

Uma Rede de Petri é uma ferramenta para a descrição e análise de processos concomitantes que surgem em sistemas com muitos componentes (os sistemas distribuídos). O conceito de Rede de Petri foi proposto em 1962 por Carl Adam Petri, com o intuito de modelar reações químicas (PETRI, 2008). A Figura 2.1 retrata uma Rede de Petri aplicada numa reação química.

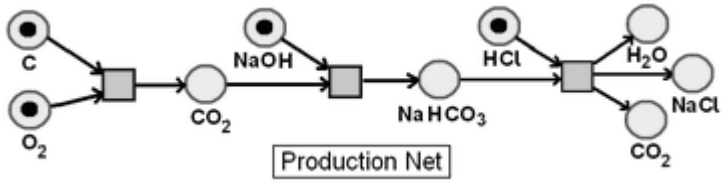


Figura 2.1 Rede de Petri descrevendo uma reação química

2.1.2. A definição do ponto de vista de sistemas a eventos discretos

Dado o intuito do projeto, é necessário abordar a Rede de Petri através da ótica do estudo de sistemas a eventos discretos. Assim, uma Rede de Petri é uma linguagem, ou uma forma de descrição, do procedimento de controle de sistemas a eventos discretos (MIYAGI, 1996).

2.2. Componentes

Existem dois principais tipos de componentes na Rede de Petri: os distribuidores e as atividades, representados, respectivamente, por círculos e retângulos.

O retângulo observado na Figura 2.2 denota uma atividade. Atividades são componentes ativos, que representam entidades reais capazes de produzir, transportar ou alterar ítens (MIYAGI, 1996).

Já os distribuidores são componentes passivos, que podem assumir estados, representando entidades capazes de armazenar ou mostrar itens. Os distribuidores podem estar “vazios” ou “cheios”, o que pode ser denotado pela presença ou não de uma *marca* preta no mesmo (MIYAGI, 1996).

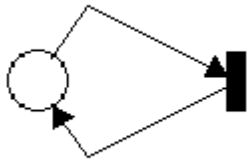


Figura 2.2 Rede de Petri simples com dois componentes

Ligando estes dois componentes principais, pode-se observar uma linha, que é denominada *arco*. Existem dois tipos de *arco*, já que estes são direcionados: o *arco* orientado de atividade à distribuidor e o *arco* orientado de distribuidor à atividade. Como requisito para a modelagem de Rede de Petri, um *arco* nunca pode ligar uma atividade a outra ou um distribuidor a outro. Caso isto ocorra durante a criação de uma Rede de Petri, diz-se que o modelo assumido está omitindo um nível de informação (MIYAGI, 1996).

2.3. Os tipos existentes de Rede de Petri

O tipo básico de Rede de Petri é a Rede de condição-evento. A partir dela, conforme a necessidade, foram desenvolvidas diversas extensões, capazes de auxiliar a modelagem de sistemas complexos.

Ao adicionar *peso* aos *arcos* e capacidade aos estados da Rede condição-evento, chega-se à Rede de Petri *lugar transição*. Ao individualizar o caráter das *marcas*, chega-se à Rede de Petri com *marcas* individualizadas, ou Rede de Petri colorida (MORORÓ, 2008). Há ainda sub-tipos destas Redes, que variam as inscrições nos *arcos*: *arcos* com inscrições fixas ou variáveis (MIYAGI, 1996).

No entanto, ao se pensar em Rede de Petri como ferramenta para modelar um controlador lógico programável, como é o intuito deste projeto, precisa-se utilizar uma extensão capaz de interagir com o ambiente externo, o ambiente controlado (Gomes *et al.*, 2007).

Existem algumas extensões de Rede de Petri que trabalham a conexão do modelo com o ambiente controlado. São elas: as SIPNs (*Signal Interpreted Petri Nets*) (FREY, 2001) e as IOPTs (*Input-Output Place-Transition Petri Nets*) (Gomes *et al.*, 2007).

A seguir, será feito um estudo de cada uma das extensões da Rede de Petri apresentadas até o momento.

Na Figura 2.3, é possível observar um esquemático de hierarquia para os diferentes tipos de Rede de Petri.

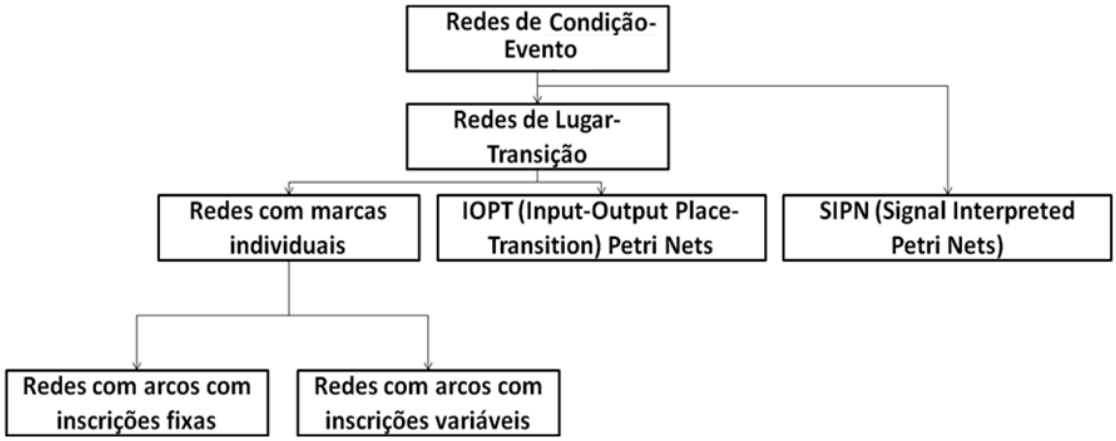


Figura 2.3 Algumas extensões da Rede de Petri

2.3.1. Rede de Petri Condição-Evento

A idéia por trás deste tipo de modelo é criar condições lógicas para a realização de eventos (atividades), por meio de *arcos* e distribuidores. Assim, eventos com múltiplas condições podem ser descritos graficamente (PETRI, 2008).

Para ocorrer uma determinada atividade em Rede de Petri, é necessário que todas os distribuidores cujos *arcos* tem como destino a atividade em questão estejam preenchidos. Estas são as chamadas pré-condições.

No exemplo da Figura 2.4, duas condições, P0 e P1 precisam estar preenchidas para que a atividade T0 ocorra (PETRI, 2008).

Do mesmo modo, é necessário que o(s) distribuidor(es) posterior(es) à atividade esteja(m) livre(s) para que a atividade ocorra. Estas são as chamadas pós-condições. Assim, pode-se observar na Figura 2.4, que, para que a atividade T0 ocorra, não bastam os estados anteriores estarem preenchidos: o estado posterior (P2) precisa estar desocupado (PETRI, 2008).

A notação gráfica que denota o preenchimento de uma condição é chamada de *marca* (MIYAGI, 1996).

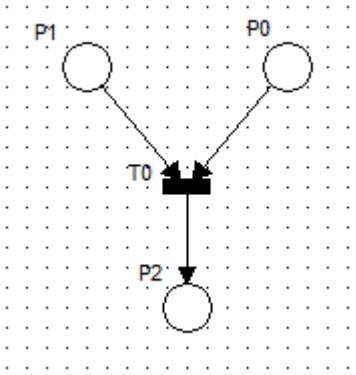


Figura 2.4 Exemplo de atividade com duas pré condições, P0 e P1

Existem alguns conceitos relacionados a Rede Condição-Evento, explicados em seguida (MIYAGI, 1996).

O primeiro, chamado de *conflito*, denota a situação na qual duas atividades dependem de uma condição em comum, criando assim um evento não determinístico. O outro conceito é o de *contato*. Um *contato* ocorre quando todas as pré-condições de uma determinada atividade estão completas, mas ao menos uma das pós-condições não está, impedindo a ocorrência da atividade. É possível evitar situações de *contato*, tornando as atividades dependentes unicamente das suas pré-condições, por meio do uso de *complementos*. Condições x e y são consideradas complementares quando:

- x é uma pré-condição da atividade z e y é uma pós-condição da mesma atividade;

- y é uma pré-condição da atividade z e x é uma pós-condição da mesma atividade;

- Para o instante inicial da Rede de Petri, x deve estar *marcada* se e somente se y não estiver.

2.3.2. Rede de Petri Lugar-Transição

A principal diferença desta extensão para a Rede de Petri Condição-Evento é que cada estado pode conter, ao invés de uma única *marca* (que significava que a condição estava satisfeita), várias *marcas*, representanto quantidades (MIYAGI, 1996).

Por isso, não se pode falar em condições, e sim em *lugares*. Cada *lugar* na rede de Petri possui uma capacidade de armazenamento, determinando

assim o número máximo de *marcas* que podem ser colocadas nestes (MIYAGI, 1996).

Ao trabalhar com Rede de Petri *Lugar-Transição*, existe um conceito importante: o de *arcos ponderados*. Os *arcos ponderados* simbolizam o fato de que é possível realizar uma *transição* que retire/coloque mais do que uma *marca* em um determinado *lugar* (MIYAGI, 1996).

Na Figura 2.5, os dois *arcos* possuem pesos diferentes: 2 e 3, respectivamente no *arco* da esquerda (que vai de P0 até T0) e da direita (que vai de P1 até T0). Isto significa que, assim que ocorrer o *disparo* da *transição*, duas *marcas* serão retiradas do *lugar* da esquerda e três *marcas* serão retiradas do *lugar* da direita (MIYAGI, 1996).

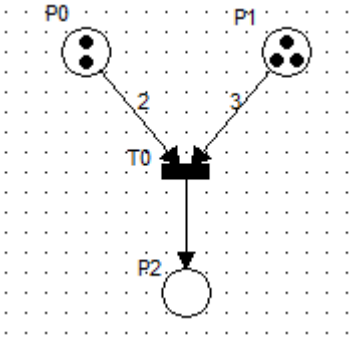


Figura 2.5 Arcos ponderados (Miyagi, 1996)

Como o *arco* inferior não possui peso, subentende-se que ele possui peso um. Ou seja, no momento do *disparo*, a *transição* irá colocar apenas uma *marca* no *lugar* inferior (MIYAGI, 1996). Após o *disparo* da *transição*, a Rede de Petri fica conforme a Figura 2.6.

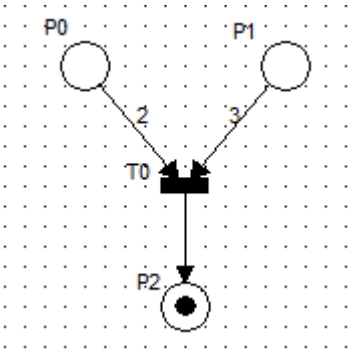


Figura 2.6 Rede de Petri após a transição T0

Em Rede de Petri *Lugar-Transição*, os conceitos de contato e *conflito* também estão presentes, de maneira ligeiramente diferente: duas *transições*

estarão em *conflito* se, em um determinado momento, ambas estiverem *habilitada*s e o *disparo* de uma implica na desabilitação da outra (MIYAGI, 1996).

Já o *contato* ocorre quando o *disparo* de uma *transição* é impedido pela capacidade insuficiente de armazenamento do *lugar* posterior à atividade. Ou seja, se há *p marcas* no *lugar* A, com capacidade de armazenamento *q* e o *arco* de *transição* para *lugar* possuir peso maior do que *q* - *p*, a *transição* não pode ocorrer. Assim como na Rede de Petri Condição-Evento, pode-se utilizar o conceito de complementos para evitar *contatos*. Dois *lugares*, x e y, são complementares quando (MIYAGI, 1996):

- Há exatamente o mesmo número de *arcos* de saída de x quanto de entrada em y, com os respectivos pesos idênticos;

- Há exatamente o mesmo número de *arcos* de entrada em x quanto de saída de y, com os respectivos pesos idênticos;

- A capacidade de x e y é igual;

- A *marcação* inicial de x é igual à sua capacidade menos a *marcação* inicial de y.

A Figura 2.7 contém um exemplo de dois *lugares* complementares. Na Rede em questão, ambos *lugares* tem capacidade máxima de 7 *marcas*.

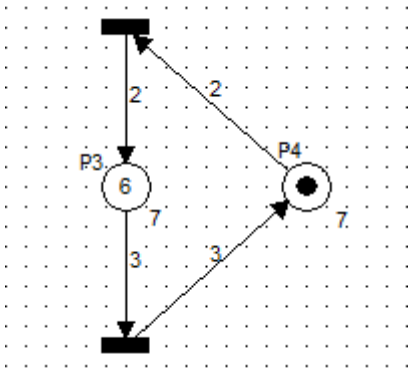


Figura 2.7 Exemplo de lugares complementares em Redes lugar-transição. # fora: capacidade; # dentro: marcação atual

2.3.3. Rede de Petri com marcas individuais ou Rede de Petri colorida

É possível representar diversos sistemas a eventos discretos reais com o auxílio de um dos tipos de Rede de Petri vistos anteriormente. No entanto, o nível de sofisticação de alguns sistemas exige a utilização de Rede de Petri mais flexíveis.

É o caso da Rede de *marcas* individuais. Nesse tipo de Rede, as *marcas* são diferentes umas das outras, podendo simbolizar ferramentas, peças, recursos, etc. de maneira exclusiva (MIYAGI, 1996). É uma maneira de simplificar Redes que, de outra maneira, exigiriam *lugares* e *transições* adicionais: uma *marca* do tipo “A” pode simbolizar um tipo de peça usinada, diferente de outra peça representada pela marca “B”.

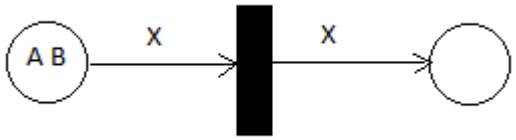
Dentro da classificação de Rede de Petri de *marcas* individuais, pode-se citar dois tipos diferentes: a Rede de Petri com *arcos* com inscrições fixas e a Rede de Petri com *arcos* com inscrições variáveis.

A Rede com *arcos* de inscrições fixas difere da Rede de *Lugar-Transição* pelo fato de que os *arcos* possuem pesos definidos por tipos de *marca*, e não apenas por quantidade. A regra é, portanto, que uma *marca* pode apenas fluir por um *arco* se ela corresponder à inscrição do *arco* (MIYAGI, 1996).

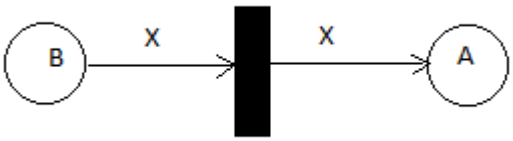
A inscrição no *arco* pode ser simplesmente qual a *marca* correspondente, mas pode também incluir quantidades e combinações de *marcas* (MIYAGI, 1996). Portanto, supondo que uma Rede contenha *marcas* do tipo A e B, é possível haver *arcos* com inscrições como “2A” (que representa um *arco* com peso 2, que só “transporta” *marcas* “A”) ou “2A + B” (neste caso, o *arco* vai “transportar”, de uma vez, duas *marcas* “A” e uma marca “B”)

Maiores descrições do funcionamento de Redes com *arcos* com inscrições fixas serão vistas nos exemplos estudados.

O outro tipo de Rede de Petri de *marcas* individuais é a Rede com *arcos* com inscrições variáveis. Neste tipo de Rede, as inscrições presentes nos *arcos* não possuem quantidades determinadas, como o *arco* “2A + B” visto anteriormente. Ao invés disso, as quantidades nas inscrições são descritas por meio de variáveis: “x” ou “y”. Assim, se um estado possuir duas *marcas*, “A” e “B”, da seguinte maneira, e uma *transição* x, pode-se afirmar que “x” irá assumir o valor de uma das duas *marcas* (MIYAGI, 1996). Portanto se x assumir, na Rede da Figura 2.8 (a), o valor de “A”, o resultado pode ser observado na Figura 2.8 (b). A variável “x” poderia, por outro lado, assumir o valor de “B”, transportando esta marca ao invés de “A”.



(a)



(b)

Figura 2.8 (a) Exemplo de Rede com arcos com inscrição variável ; (b) Exemplo de Rede com arcos de inscrição variável após transição

2.3.4. SIPN

A SIPN é concebida diretamente como uma de linguagem de programação de CLPs. (FREY, 2001). Ela não é um sub-tipo de Rede de *marcas* individuais. Na verdade, é uma extensão direta das Rede de Condição-Evento, no sentido de que a *marcação* é binária: um estado pode estar apenas *marcado* ou sem *marca*.

Como a SIPN foi criada como método de modelagem de controladores, é preciso analisá-la sob essa ótica: para se modelar um controlador, é essencial que haja uma interface entre o modelo e o ambiente controlado. Para isso, é preciso implementar na Rede de Petri as entradas (*inputs*) e saídas (*outputs*) que representam esta interface (FREY, 2001).

Na SIPN, as entradas são associadas às *transições* (FREY, 2001). Assim, uma *transição*, para ocorrer, deve possuir todas as suas pré-condições completas (diz-se *habilitada*) e, caso seja associada a uma entrada, somente se esta condição estiver satisfeita.

Já as saídas são associadas aos *lugares* (FREY, 2001). Em SIPN, uma variável de saída pode ser definida por mais de um *lugar*. Assim, há uma matriz que relaciona estados e variáveis de saída.

Para trabalhar como linguagem de controle, a SIPN precisa ser determinística. Assim, existe um conceito conhecido como sincronização dinâmica: se dois ou mais eventos estão *habilitados* simultaneamente

(ambos/todos estão *habilitados* e as suas condições de entrada estão satisfeitas) ocorre um *disparo* simultâneo de todos eles (FREY, 2001).

Isso acarreta uma outra mudança na SIPN em relação à Rede de Petri Condição-Evento: a dinâmica do fluxo de *marcas*. Uma SIPN continua sua movimentação de *marcas* até atingir uma combinação estável para um dado conjunto de entradas. Ou seja, ao se alterar uma das entradas da Rede, esta irá percorrer estados ditos instáveis até atingir a estabilidade novamente (FREY, 2001).

O último conceito chave das SIPN é a hierarquia. Como a modelagem de eventos com muitas variáveis e estados pode tornar uma Rede de Petri muito complexa, as SIPN permitem criar “sub-Redes”, que facilitam a modularização e diminuição das representações gráficas (FREY, 2001). Isso pode ser observado na Figura 2.9.

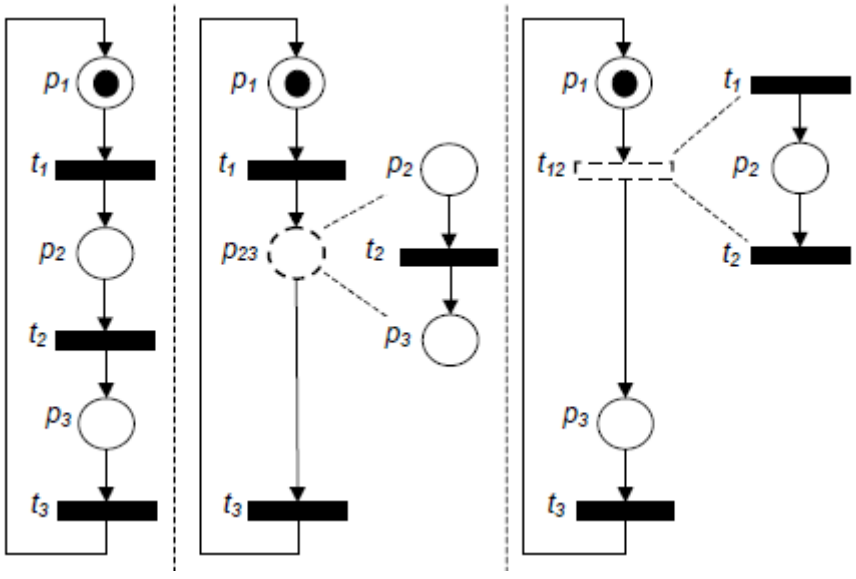


Figura 2.9 Refinamentos de lugar (no centro) e transição (à direita) em SIPNs (FREY, 2001)

As sub-Redes aparecem na SIPN como refinamentos de *lugares* ou *transições*. Quando são refinamentos de *lugares*, como na Figura 2.9 (b), a *marca* só pode deixar a sua posição quando o *lugar* final da sub-Rede (representado na Figura 2.9 por p_3) for preenchido. Analogamente, quando uma *transição* possui refinamento, ela só é *disparada* quando a última *transição* da sub-Rede (é o caso, na Figura 2.9 (c) à direita, da *transição* t_2) é *disparada*.

Embora os refinamentos de *lugar* e *transição* apareçam como ferramentas de auxílio no momento de modelagem, é sempre possível representar Redes com refinamentos como Redes que não as possuem, “abrindo-se” as sub-Redes.

É preciso notar que a descrição feita neste trabalho se restringe ao caráter da rede de Petri, não possuindo rigor matemático. Para mais informações sobre s SIPN, inclusive seu formalismo matemático, pode-se consultar (FREY, 2001).

2.3.5. IOPT

A IOPT (*Input-Output Place-Transition Petri Net*) é outra extensão da rede de Petri que, assim como a SIPN, se propõe a transformar as redes de Petri em elementos não autônomos, ou seja, que possuam comunicação com o ambiente (GOMES *et al.*, 2007).

Toda a descrição formal da IOPT aqui apresentada baseia-se em (GOMES *et al.*, 2007), obra na qual o conceito da Rede é apresentado.

A IOPT associa sinais e eventos de entrada e saída a elementos da Rede. Ao contrário da SIPN, que associa entradas com *transições* e saídas com *lugares*, a IOPT possui maior flexibilidade: toda *transição* pode ser associada tanto a uma entrada quanto a uma saída (ou múltiplas), e os *lugares* podem ser associados a saídas. Alternativamente, saídas podem ser definidas por combinações de diversos estados.

Como pode ser visto na Figura 2.3, as IOPTs são extensões da Rede de Petri *Lugar-Transição* e, como tal, permite diversas *marcas* por *lugar*. Com isso, é possível criar funções booleanas de saída baseadas na quantidade de *marcas* de um *lugar*.

Na Figura 2.10, é possível ver um exemplo de função Booleana de saída no *lugar* “GateOutOpen”: $GateOut = 1\ if\ marking > 0$.

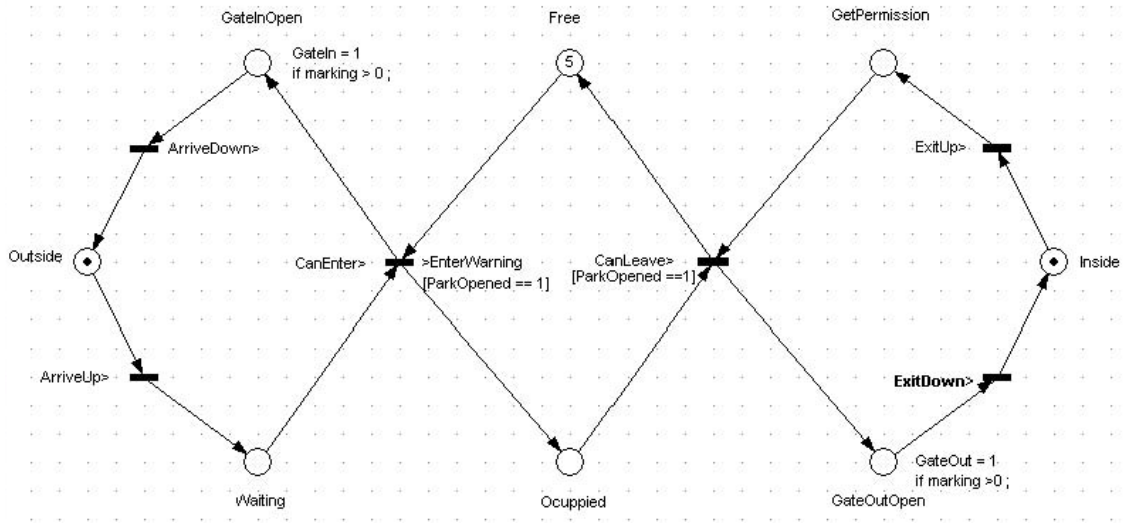


Figura 2.10 Exemplo de modelo de estacionamento em IOPT (Gomes et al.*, 2007).*

Em termos dinâmicos, a IOPT resolve o problema não-determinístico dos *conflitos* na Rede de Petri por meio de *prioridades*. Cada *transição* possui, além de um possível evento de entrada, uma prioridade. Assim, para ser *disparada*, uma *transição* precisa estar *habilitada* (isto é, todas as suas pré-condições precisam estar *marcadas*), o evento de entrada precisa ser ativado e sua *prioridade* deve ser a maior possível. Assim, a dinâmica da Rede segue pela *transição* com maior prioridade até a menor, até atingir um estado estável.

2.4. Método de representação – A *Petri Net Markup Language* (PNML)

2.4.1. Introdução à PNML

Em novembro de 2009, a ISO (International Organization for Standardization) publicou a norma ISO/IEC 15909-2, que descreve um formato de transferência para a Rede de Petri baseado em XML (eXtended Markup Language – para maiores informações sobre XML consultar o Anexo A). Este formato, a *Petri Net Markup Language* (PNML), possibilita o intercâmbio de Redes entre diversas ferramentas de trabalho e diferentes grupos de trabalho. A norma define, ainda, certos conceitos sobre a aparência gráfica da Rede de Petri, bem como sua sintaxe na linguagem (ISO, 2010).

A PNML foi proposta durante a *International Conference on Application and theory of Petri Nets 2000*, e posteriormente desenvolvida e publicada por Michael Weber e Ekkart Kindler (KINDLER e WEBER, 2003).

2.4.2. Conceitos

Os conceitos relativos a PNML descritos a seguir foram propostos em (WEBER, KINDLER, *et al.*, 2003), e posteriormente publicados pela (ISO, 2010)

- *Princípios de projeto*: a PNML e o seu uso são guiados pelos objetivos da flexibilidade, da compatibilidade e da desambiguidade.
- *Tipos de Rede de Petri*: como visto anteriormente, há diversos tipos e extensões da Rede de Petri. A PNML leva isso em conta ao representar cada rede de Petri e definir qual é o seu tipo constituinte;
- *Estrutura da PNML:* a linguagem possui alguns módulos. São eles:
    - *Meta model:* arquivo que contém a Rede de Petri propriamente dita;
    - *Type definition interface*: uma interface para a definição de tipos de Rede de Petri;
    - *Feature definition interface*: módulo no qual são definidas as características novas, exclusivas de certas ferramentas e tipos de rede de Petri;
    - *Conventions document*: dado que diversos tipos de Rede de Petri são amplamente utilizados, eles podem ser padronizados. Este é o objetivo do documento de convenções, um sistema de padronização que é atualizado progressivamente por usuários e desenvolvedores;
- *Rede de Petri e objetos:* cada arquivo PNML pode conter apenas uma ou diversas Rede de Petri. Tanto a Rede quanto os elementos que a compõem são denominados *objetos*. Na versão básica da linguagem há quatro tipos de objetos:
    - *Arco;*
    - *Lugar;*
    - *Transição;*
    - *Rede;*
- *Labels:* cada objeto, incluindo a Rede de Petri em si, pode ter anotações. Essas anotações podem representar nomes, observações, variáveis, condições, etc., dependendo do tipo de Rede;

- *Informações gráficas*: para a exibição gráfica da Rede de Petri, a PNML reserva informações específicas, como coordenadas de exibição, posições relativas a outros *objetos*, etc., mais uma vez dependendo do tipo da Rede representada;
- *Informações de ferramenta*: alguns editores e simuladores da Rede de Petri possuem funcionalidades adicionais únicas. Assim, é possível registrar informações adicionais. Quando o arquivo é aberto por outros editores que não o original, a informação é ignorada e a Rede é aberta normalmente.

2.4.3. A sintaxe da PNML

Como descrito anteriormente, a PNML é uma linguagem baseada em XML. Os seus documentos de validação estão disponíveis publicamente em http://www.pnml.org/version-2009/version-2009.php. Os arquivos são implementados na extensão .rng, ou RELAX-NG, um formato de *XML schema*.

Os *schemas* da linguagem incluem definições para objetos (*lugares*, *transições*, *arcos*), anotações (nomes, *labels*), tipos de variáveis (*integer, boolean*, etc.) e outros aspectos da linguagem.

A critério de ilustração da linguagem da PNML, segue um exemplo da sintaxe, descrevendo a Rede de Petri observada na Figura 2.11.

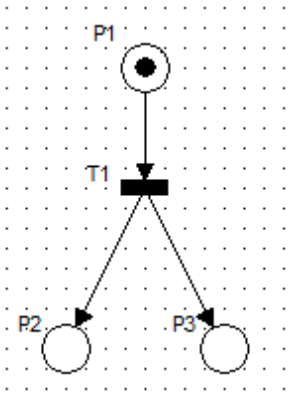


Figura 2.11 Rede ilustrativa para análise da sintaxe da PNML

No código, presente na Figura 2.12, é possível distinguir as diferentes seções do código: o cabeçalho do arquivo, definindo os schemas; a área de declaração de *lugares*, que contém os *lugares P1*, *P2* e *P3* e suas respectivas posições, *marcações* iniciais e capacidades; a área de declaração das

transições, na qual tem-se *T1* e sua posição; e por último, a área de relações, na qual os *arcos* são declarados, assim como seus objetos de entrada e saída.

A PNML ainda possui espaço reservado para informações específicas da ferramenta de edição de Rede de Petri utilizada. Neste exemplo, esta parte da sintaxe foi omitida.

```
<?xml version="1.0" encoding="ISO-
8859-1"?><pnml xmlns =
"http://www.irt.rwth-
aachen.de/download/netlab/pntd/pns
mNet">
  <net id = "n1"
       type =
"http://www.irt.rwth-
aachen.de/download/netlab/pntd/pns
mNet">
    <name>
<text>exemplo_relatorio(fig10)</te
xt>
    </name>
    <place id = "p1">
      <graphics>
        <position x = "240"
                  y = "200"/>
        <dimension x = "40"
                   y = "40"/>
      </graphics>
      <initialMarking>
        <text>1</text>
      </initialMarking>
      <capacity>
        <text>1</text>
      </capacity>
    </place>
    <place id = "p2">
      <graphics>
        <position x = "420"
                  y = "160"/>
        <dimension x = "40"
                   y = "40"/>
      </graphics>
      <initialMarking>
        <text>0</text>
      </initialMarking>
      <capacity>
        <text>1</text>
      </capacity>
    </place>
    <place id = "p3">
      <graphics>
        <position x = "420"
                  y = "240"/>
        <dimension x = "40"
                   y = "40"/>
      </graphics>
      <initialMarking>
        <text>0</text>
      </initialMarking>
      <capacity>
        <text>1</text>
      </capacity>
    </place>
    <transition id = "t1">
      <graphics>
        <position x = "340"
                  y = "200"/>
        <dimension x = "40"
                   y = "40"/>
      </graphics>
    </transition>
    <arc id = "a1"
         source = "p1"
         target = "t1"/>
    <arc id = "a2"
         source = "t1"
         target = "p2"/>
    <arc id = "a3"
         source = "t1"
         target = "p3"/>
```

Figura 2.12 Exemplo de código PNML

3. Linguagem de Máquina

3.1. O CLP

O controlador lógico programável (CLP) é um dispositivo eletrônico para aplicações industriais que, para execução de funções como operações lógicas, sequencialização, temporização e computação numérica, possui uma memória

onde ficam gravadas sob a forma de uma lista de comandos que definem o procedimento de controle. Baseado no conteúdo desta memória, a operação de máquinas e/ou processos são controlados por meio de sinais de saída digitais e/ou analógicos (MIYAGI, 1996).

3.2. Norma IEC61131-3

3.2.1. Introdução

Nesse trabalho serão apresentadas cinco linguagens que são padronizadas pela norma IEC61131-3: *Sequential Function Chart* (SFC), *Ladder Diagram* (LD), *Function Block Diagram* (FBD), *Instruction List* (IL) e *Structured text* (ST). Essas cinco linguagens são demonstradas na Tabela 1 (MIYAGI, 1996).

Tabela 1 - Classificação das possíveis linguagens de CLPs

Tipo	**Linguagem**	**Caráter**		
		Lógica	Ordenação	Funções Complexas
Textual	Álgebra de Boole	X		
	IL	X		
	ST	X		
Gráfica	LD	X		X
	FBD	X		X
	Fluxograma		X	X
	Elementos SFC		X	
Tabular	Tabela de decisão		X	

Tal diversidade de linguagens de programação pode ser atribuída à competição entre os fabricantes que procuravam achar maneiras mais simples e eficientes de programar seus produtos e adquirir vantagem de mercado e um diferencial de produto (WARNER *et al.*, 2006) *apud* (SARMENTO, 2008). Entretanto, é nessa diversidade que se encaixa a proposta desse trabalho, a qual visa idealmente à flexibilidade do transcritor para todos os tipos de linguagem. Um fabricante que tivesse o seu transcritor adaptado aos diversos tipos de linguagem teria isso como uma vantagem competitiva contra os

demais. A definição das linguagens suportadas pelo transcritor (LD, SFC e ST) e a motivação da escolha pode ser vistas na seção 5.2.2.

3.2.2. Representação – a PLCOpen XML

A norma IEC 61131-3, apesar de definir as linguagens descritas anteriormente, não apresenta um padrão de formato para o intercâmbio destas linguagens entre aplicativos e desenvolvedores.

A PLCOpen é uma organização cuja proposta é ser independente de fabricantes de CLPs ou consumidores dos mesmos. Sua missão é resolver problemas relacionados ao trabalho com CLPs, de modo a chegar em um nível internacional de padronização (PLCOPEN, 2008).

Com esse intuito, um comitê técnico denominado TC6 elaborou uma proposta de linguagem, baseada em XML (vide Anexo A), que dá suporte ao intercâmbio de informações escritas nas linguagens da IEC 61131-3.

A sintaxe desta linguagem pode ser encontrada em *http://www.plcopen.org/pages/tc6_xml/index.htm*, na forma de *schemas* de XML no formato “.xsd”.

3.3. O ladder diagram (LD)

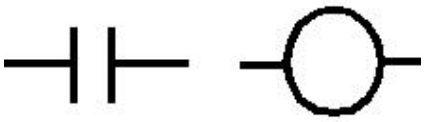
O ladder diagram corresponde a uma representação lógica baseada no diagrama de circuito de relés, cuja utilização era ampla antes do surgimento dos CLPs (MIYAGI, 1996).

3.3.1. Conceitos introdutórios

Antes de iniciar uma descrição detalhada do LD, convém explicar seus elementos básicos.

- *Contato:* corresponde a uma chave elétrica que pode estar fechada, transmitindo o sinal, ou aberta, interrompendo o sinal;
- *Bobina:* corresponde a uma variável lógica que armazena um tipo de informação, como por exemplo um estado.

Na Figura 3.1 há o símbolo utilizado para o contato e para a bobina.



(a) (b)

Figura 3.1 Símbolos do contato (a) e da bobina (b)

3.3.2. Regras quanto ao posicionamento

Por ser uma linguagem gráfica, o LD segue regras de posicionamento dos elementos. Para que as regras de posicionamento sejam descritas de forma textual, deve se considerar que há uma matriz de *n* linhas e *m* colunas sobre a qual serão posicionados os elementos da linguagem de *ladder*. Os limites para *n* e *m* são impostos pelo compilador da linguagem e pelo CLP a ser utilizado.

- Todo elemento deve ser representado graficamente sobre uma posição definida pela matriz, ou seja, o elemento terá uma coordenada referente a sua linha (de 1 a *n*) e outra coordenada referente a sua coluna (de 1 a *m*);
- O elemento bobina deve ser representado somente na última coluna, que é a coluna mais a direita definida pela coordenada m.
- Nas extremidades esquerda e direita da matriz há uma linha chamada de *linha mestre*. No caso da linha à esquerda, ela também recebe o nome de *linha mãe*;
- Cada posição da matriz com coordenadas genéricas *i* e *j* pode ou não estar conectada ao seu elemento imediatamente adjacente na horizontal (na mesma linha, com coordenadas *i* e *j±1*). Não há conexões entre elementos adjacentes na vertical (na mesma coluna). No entanto, na vertical é permitida a ligação entre conexões adjacentes. As conexões receberão o nome de linhas ao longo do documento. Um diagrama exemplo pode ser observado na Figura 3.2:

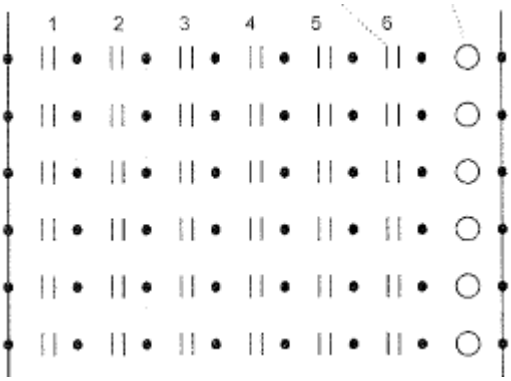


Figura 3.2 Exemplo de LD(MIYAGI, 1996)

3.3.3. Regras quanto ao fluxo

A energia, também denominada de força (MIYAGI, 1996), está submetida a regras de fluxo. Trata-se de regras muito importantes para o projeto pois definem a ordem pela qual serão atribuídos valores nas variáveis lógicas, que podem resultar em saídas diferentes para uma mesma entrada de sinal caso a ordem seja alterada. Em outras palavras, a propriedade da comutação não é sempre válida. Isso será abordado de maneira mais detalhada mais a frente no trabalho.

- A energia flui da esquerda para a direita, não podendo percorrer nunca no sentido oposto;
- A energia pode fluir em qualquer sentido na direção vertical;
- O processamento do diagrama é feito de cima para baixo;
- Somente serão atribuídos os valores às saídas ao final do processamento (*scan time*) de modo que se uma variável ligada a uma saída é alterada uma vez e recebe seu valor inicial após uma segunda alteração, seu valor não será alterado na saída.

3.3.4. Elementos do LD

Alguns elementos utilizados no LD já foram brevemente descritos nesse trabalho. Essa seção se destina a uma descrição mais detalhada dos principais elementos básicos do LD explicando suas características e tipos a fim de criar o ferramentário necessário para desenvolver sistemas de controle discreto.

- CONTATO

Como já descrito antes, o contato corresponde a uma chave elétrica que tem dois estados: aberto e fechado. Os contatos estão associados uma

entrada, uma saída ou uma variável lógica. Há quatro tipos de contato que se subdividem em duas classes:

1) Contatos associados ao sinal
    a. Tipo *A* (*make*: normalmente aberto);
    b. Tipo *B* (*break*: normalmente fechado).
2) Contatos detectores de variação
    a. Tipo *p* (positivo: detector de borda de subida);
    b. Tipo *n* (negativo: detector de borda de descida).

A teoria aplicada nos contatos do LD é simples, o que facilita o projeto dos processos de controle (MIYAGI, 1996). A teoria consiste na associação direta de um contato a uma entrada, saída ou variável lógica (que pode ser uma bobina, por exemplo) de modo que qualquer alteração no estado da entrada, saída ou variável lógica resultará numa alteração no estado do contato.

A diferença entre as duas classes está no tipo de detecção de cada uma. A primeira classe detecta o estado de energização da variável associada ao contato. Se o estado de energização é 0, o contato está em seu estado normal. Se o estado de energização é alterado para 1, o contato sai do seu estado normal. Se o estado de energização da variável associada retornar a 0, o contato retorna ao seu estado normal e assim sucessivamente.

Por outro lado, a segunda classe está associada à variação do estado de energização a sua esquerda (e não à variável lógica propriamente dita, como a outra classe de contatos). Há dois tipos de variação que podem ocorrer no lado esquerdo do contato que disparam sua atuação: a variação de 0 para 1 (que é identificada pelo contato tipo *p*) e a variação de 1 para 0 (que é identificada pelo contato do tipo *n*). Em outras palavras, se o lado esquerdo de um contato do tipo *p* é alterado para ON, ou seja, sofre uma alteração de 0 para 1, o contato é fechado transmitindo o sinal ON para o lado direito a ele. Esse sinal transmitido dura apenas um período de ciclo de controle do CP. Portanto, a cada borda de subida de energia, o contato permanece um período de ciclo[2] de CP em estado FECHADO, transmitindo o sinal ON da esquerda

[2] Período entre duas atualizações seguidas de um controlador programável

para a direita, e depois volta a abrir. O contato do tipo *n* responde da mesma maneira para bordas de descida de sinal.

- BOBINA

As bobinas representam variáveis lógicas no LD e são responsáveis pela característica de memória do sistema de controle. É nela, por exemplo, que é atribuído um estado ou uma saída. A mudança de estado da bobina se dá com mudança da sua energização: a bobina é acionada quando energizada e desacionada quando desenergizada. Como já dito anteriormente, as bobinas são processadas de cima para baixo e apenas após o processamento é feita a atualização da saída.

Numa bobina, assim como numa entrada, pode haver contatos associados. Os contatos associados a uma bobina são acionados imediatamente após o acionamento de sua respectiva bobina.

Há os seguintes tipos de bobinas (MIYAGI, 1996):

- Bobina comum: quando energizada seu valor lógico fica 1;
- Bobina inversa: quando energizada seu valor lógico fica 0;
- Bobina de *set* / *reset*: o valor lógico da bobina de *set* fica 1 quando energizada (o valor lógico da bobina *reset* fica 0 quando ela é energizada);
- Bobina com memória: mantém memorizado seu valor mesmo quando é desenergizada; pode ser associada a bobinas de *set* ou *reset*;
- Bobina detectora de variação positiva (ou negativa): é a bobina correspondente aos contatos detectores de variação, isto é, detecta a variação de 0 para 1 e fica acionada durante o período de um ciclo de CP. A bobina de variação negativa responde de maneira inversa à positiva. A bobina de detecção de borda de subida (positiva) recebe a notação *P* enquanto a outra (negativa), *N*.

- TEMPORIZADOR

Em sistemas de controle de sistemas discretos nota-se a necessidade de elementos vinculados à temporização. Há diversos tipos de temporizadores, que se baseiam nos pulsos de *clock* gerados pelo cristal oscilador do CP, que

são capazes de introduzir atrasos em sinais. Em linhas gerais, um temporizador possui os seguintes componentes: entrada, saída, valor de ajuste (*set-point*) e valor atual. A saída é função da entrada e de como o temporizador foi configurado. O valor de ajuste corresponde à configuração do temporizador. O valor atual, por outro lado, é uma variável interna que representa o tempo decorrido e é compartilhada com o meio externo para demais usos. A seguir, seguem algumas descrições de tipos de temporizadores.

- *On delay timer* (TON): após a ativação do temporizador, sinal ON na entrada, há a inserção de um atraso antes de transmitir o sinal ON para a saída; quando a entrada recebe sinal OFF a saída imediatamente recebe OFF;
- *Off delay timer* (TOF): após a ativação do temporizador, sinal ON na entrada, o sinal ON é transmitido imediatamente para a saída; quando o sinal de entrada é levado a OFF é inserido um atraso antes de levar a OFF a saída;
- Pulso (TP): após a ativação do temporizador, sinal ON na entrada, o sinal ON é transmitido imediatamente para a saída mas tem um período de pulso determinado pelo valor de ajuste do temporizador ou pela desativação do sinal de entrada (essa última, somente se ocorrer antes do período do temporizador) que retorna a saída a OFF.

3.3.5. Funções básicas de controle

Com as regras quanto ao posicionamento e regras quanto ao fluxo, além dos componentes básicos descritos anteriormente, há a possibilidade de implementação de funções básicas de controle por meio do LD.

A primeira função básica é a capacidade de fazer operações lógicas. As operações lógicas que podem ser realizadas em LD são AND lógico, OR lógico e NOT lógico, além de suas demais combinações. Por ser um diagrama que respeita regras de posicionamento matricial, as operações podem ser visualizadas também na forma matricial.

A função AND lógico é representado pela associação em série de componentes, isto é, uma associação horizontal de componentes. As regras

quanto ao fluxo garantem que essa associação siga uma ordem de ativação sequencial e dependente entre si (energia vai somente da linha mestre à esquerda para a linha mestre à direita).

A função OR lógico é representado pela associação em paralelo de componentes, isto é, uma associação vertical de componentes. As regras quanto ao fluxo garantem que essa associação permita ativação não sequencial e independente entre si.

Conforme visto, é uma característica do LD a representação da função AND lógico e a função OR lógico de maneira oposta e complementar. Essa distinção visual (horizontal e vertical, série e paralelo) é uma vantagem do diagrama pois facilita a leitura e síntese de projetos.

Outra função que pode ser implementada é a capacidade de memorização do sistema. Por se tratar muitas vezes de o controle de um sistema dinâmico, essa função é muito importante pois permite que as saídas sejam determinadas pelas entradas atuais e anteriores. Com a utilização de bobinas, que são circuitos de auto-retenção, pode-se implementar a estrutura de memória.

Por último, a função de temporização pode ser implementada no LD por meio dos complementos já descritos anteriormente. O processo de controle precisa da execução vinculada ao tempo, o que faz dos temporizadores, componentes vitais para a aplicabilidade do LD.

3.4. O sequential flow chart (SFC)

O *Sequential Flow Chart* é a segunda linguagem gráfica padronizada pela IEC61131-3 que também é abordada neste trabalho. O SFC é uma linguagem com técnicas presentes em linguagens derivadas de Rede de Petri, como GRAFCET (*Graphic de Commande Etape-Transition*) da França e o MFG (*Mark Flow Graph*) do Japão, que já são largamente utilizadas em seus respectivos países. O desenvolvimento da linguagem SFC é motivado pelo interesse em ferramentas gráficas que representem de forma explicita as funções para descrever processos sequenciais para aplicações industriais (MIYAGI, 1996).

3.4.1. Conceitos introdutórios

Mantendo a mesma estrutura desenvolvida para a descrição do LD, inicia-se esta seção com algumas definições de conceitos que se aplicarão ao SFC.

O SFC é uma linguagem gráfica cujo objetivo é descrever um processo sequencial de maneira clara e explícita aproximando-o do raciocínio lógico do desenvolvedor e facilitando a sua implementação. Para isso a linguagem adota uma estrutura de passos e eventos conectados entre si por objetos de relacionamento. Como já dito anteriormente, há uma semelhança com a Rede de Petri.

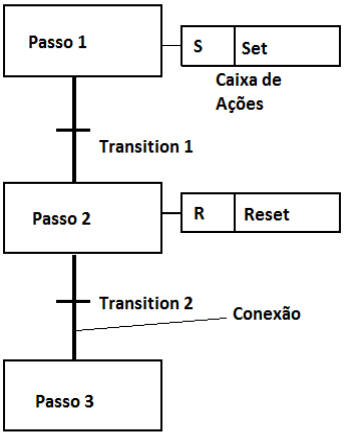


Figura 3.3 Exemplo de componentes SFC

O diagrama é composto por elementos como:

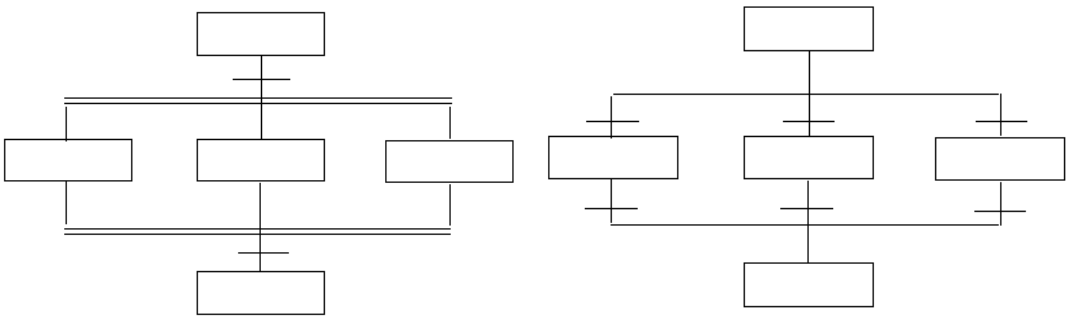
- *Step* (*passo*): passo no diagrama sequencial SFC
- *Transition* (*transição*): evento que serve de condição para a mudança de estado ou de passo;
- *Link* (conexão): elemento gráfico que conecta *transições* com passos e atribui ao diagrama uma sequência lógica;
- *Action (ação):* ações presentes em um *step* (opcional).

3.4.2. Regras do SFC

Há regras que definem como deve ser a evolução do SFC. Algumas delas, colocadas por (MIYAGI, 1996), serão apresentadas abaixo.

- Define-se como estado inicial da sequência a situação em que apenas o passo inicial está ON e o restante está OFF;

- O estado do SFC é determinado pela combinação de todos os passos;
- Entre duas *transições* existe pelo menos um passo;
- Entre dois passos existe pelo menos uma *transição;*
- Estando satisfeitas todas as condições da *transição* e estando todos os passos antecedentes em ON, os passos seguintes à *transição* em questão ficam ON e todos os antecedentes ficam OFF;
- A regra de evolução anterior se aplica para criação de sequências em paralelo de maneira que todos os passos subsequentes fiquem ON simultaneamente – a notação utilizada é uma linha dupla horizontal (Figura 3.4 a).
- Essa regra se aplica tanto para a abertura de sequências em paralelo como para o seu fechamento;
- Assim como há a regra das sequencias em paralelo, há a regra de evolução por seleção de sequência; nessa situação um passo está conectado a mais de uma *transição* que competem entre si; em caso de *conflito*, as *transições* conflitadas respeitam a uma ordem de preferência que pode ser definida em projeto ou determinada como padrão (a *transição* mais à esquerda tem preferência sobre a outra) – a notação utilizada é uma reta horizontal simples (Figura 3.4 b).
- Essa regra se aplica tanto para a abertura de sequências por seleção como para o seu fechamento.



(a) (b)

Figura 3.4 Sequência em paralelo (a); sequência de seleção (b)

3.4.3. Elementos do SFC

- STEP (PASSO)

O passo representa uma etapa de uma sequência lógica de um processo. Por definição, o passo pode permanecer em dois estados lógicos, ON e OFF. Entretanto, o passo pode trazer mais informações (além do seu estado lógico) quando está associado a um bloco de ação (*action block*).

- AÇÕES

As ações são atributos opcionais de um passo contidos em seu bloco de ação. A ação pode ser conectada diretamente ao passo ou por meio de declarações STEP e END_STEP. Sempre que o estado vinculado ao bloco de ações estiver em estado ON

- TRANSIÇÕES

A *transição* indica a condição necessária para mudar de passo caso todos os passos antes dessa *transição* estejam em estado ON, indicando que essa *transição* pode ser habilitada.

Há algumas notações que serão descritas aqui para as *transições* especiais:

  - Abertura de paralelo: um passo, seguido de uma *transição*, seguida de uma linha horizontal dupla, seguida dos respectivos passos do paralelo (Figura 3.4 a);
  - Fechamento de paralelo – Mesma notação que a anterior (Figura 3.4 a);
  - Abertura de seleção: um passo, seguido uma linha horizontal simples, seguida de mais de uma *transição* (Figura 3.4 b)*;*
  - Fechamento da seleção– Mesma notação que a anterior (Figura 3.4 b).

- LINKS

É o elemento gráfico responsável pela conexão e direcionamento SFC, que é de cima para baixo a menos que o sentido seja indicado.

3.4.4. Funções básicas de controle

Assim como descrito para a linguagem LD, as mesmas funções podem ser implementadas com essa linguagem. A flexibilidade desta linguagem, por

outro lado, se dá pela inserção de parâmetros por meio de ações. As ações podem acionar as saídas, iniciar temporizadores, entre outros. Então, com isso, é possível implementar as tais funções operacionais lógicas, funções de temporização e funções de memória.

3.5. O structured text (ST)

3.5.1. Conceitos introdutórios

O ST é uma das duas linguagens textuais propostas pela IEC 61131-3. Ela tem sintaxe semelhante a Pascal (THAYER, 2009). É uma linguagem de alto nível, por não usar operadores orientados a máquina e sim declarações abstratas que descrevem funcionalidades complexas (KARL-HEINZ e TIEGELKAMP, 2001).

A linguagem possui uma série de vantagens, obtidas em (KARL-HEINZ e TIEGELKAMP, 2001):

- Formulação compacta da tarefa de programação;
- Construção clara do programa em blocos de declarações;
- Construções lógicas potentes para controlar o fluxo de comando.

No entanto, a linguagem acarreta algumas desvantagens:

- A tradução para código de máquina não pode ser influenciada pelo usuário, dado que é realizada por meio de um compilador;
- O alto nível de abstração pode levar a uma perda de eficiência, dado que programas compilados são geralmente mais longos e menos rápidos do que programas desenvolvidos diretamente em código de máquina.

3.5.2. A sintaxe da linguagem

A seguir cabe uma breve descrição da sintaxe da linguagem, obtida em (KARL-HEINZ e TIEGELKAMP, 2001).

O ST é composto por uma série de declarações. Cada declaração pode ser realizada em uma linha e deve ser prosseguida por um ponto-e-vírgula. As declarações podem utilizar valores de variáveis únicas, combinações lógicas[3]

[3] Combinações podem ser tanto de lógica Booleana (AND, OR, etc.) quanto de lógica aritmética (+, -, etc.)

de variáveis e constantes. Um exemplo de declaração na qual há a atribuição de um valor a uma variável pode ser visto a seguir:

$$A := B\ AND\ C;$$

Além das declarações básicas há estruturas de programação auxiliares. São elas:

- RETURN
- IF
- CASE
- FOR
- WHILE
- REPEAT
- EXIT

Como mencionado, as estruturas lógicas são semelhantes a estruturas de outras linguagens de programação de alto nível.

4. Softwares de apoio utilizados

4.1. CoDeSys

4.1.1. Descrição

O CoDeSys é um *software* para automação criado pela S3 – *Smart Software Solutions*. Com o CoDeSys, é possível programar qualquer CLP, dentre uma série de modelos compatíveis, com as linguagens propostas na IEC 61131-3. O CoDeSys possui uma versão gratuita, que pode ser utilizada por desenvolvedores e projetistas (SMART SOFTWARE SOLUTIONS, 2010).

O pacote é composto pelo *software* CoDeSys propriamente dito, que contém editores de linguagens de programação, e do CoDeSys SP, um *runtime system* que faz a interface com os controladores compatíveis. Uma lista destes dispositivos pode ser encontrada em *http://www.automation-alliance.com/index.shtml?aa_products*.

Atualmente, a *Automation Alliance*, que compõe os fabricantes de produtos compatíveis com o *software* CoDeSys, possui cerca de 100 membros. Este foi uma das principais características para a escolha do CoDeSys como ferramenta-alvo de trabalho do transcritor, cujo intuito é possuir a maior flexibilidade possível.

4.1.2. Funcionalidades

O CoDeSys é compatível com as linguagens de programação da IEC 61131-3, como já exposto. Além de compilar as linguagens, o *software* possui uma ferramenta de edição gráfica que permite ao usuário diretamente criar os programas no CoDeSys. Na Figura 4.1 é possível observar o ambiente mencionado com uma visualização de LD na parte inferior da tela.

O CoDeSys utiliza como unidade básica de programação o POU (*Program Organization Unit*), definido pela IEC61131-3. Cada POU pode ser composto em uma das linguagens de trabalho definidas pela norma.

A unidade macro de programação, o *Project*, também padronizado pela IEC61131-3, pode conter diversas POUs, com variáveis locais e globais. Além disso, o *Project* contém os *devices*, que representam as unidades de *hardware*-

alvo. Para a definição de um *device* é necessário um arquivo de extensão (do tipo *.devdesc), baseado em XML, ou outros tipos de arquivos suportados. Estes arquivos podem ser encontrados nas páginas dos fabricantes suportados pelo CoDeSys ou na própria página da *Smart Software Solutions*.

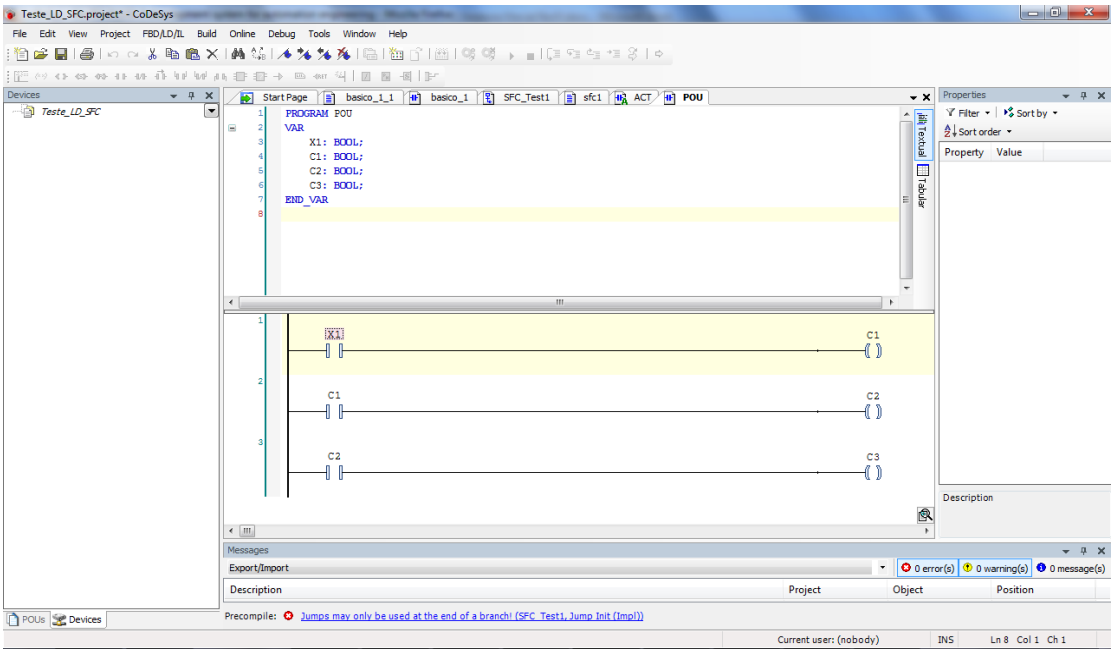


Figura 4.1 O ambiente de desenvolvimento do CoDeSys, com um exemplo de Ladder Diagram

O CoDeSys apresenta ainda uma funcionalidade chamada *simulation*, que permite a simulação de compilação e execução do código sem a presença física do controlador-alvo. Assim, é possível realizar tarefas importantes do desenvolvimento, como verificação de erros, validação com as propostas de projeto e *debugging*.

4.1.3. Linguagens de importação e exportação do CoDeSys

Apesar do CoDeSys ser compatível com as linguagens de programação do IEC 61131-3 e possuir editores gráficos para as mesmas, ele não possui suporte completo à linguagem PLCOpen XML, descrita anteriormente neste trabalho.

O suporte à PLCOpen XML é restrito às linguagens ST e IL, que são textuais, e não se estende às linguagens SFC e LD.

Dessa maneira, o CoDeSys será utilizado como ferramenta para análise de integridade de código do ST gerado pelo transcritor.

A partir deste código o desenvolvedor pode realizar simulações, estudos de eficiência do modelo desenvolvido e eventual *debugging*. Depois destas etapas o desenvolvedor pode programar o CLP escolhido de maneira direta.

4.2. NetLab – editor de Rede de Petri

4.2.1. Descrição

O NetLab é uma ferramenta para a edição e simulação de Rede de Petri, proposto por Dirk Abel em 1990. Em sua última versão, passou a aceitar o formato de arquivo da PNML (ABEL, 2008).

4.2.2. Funcionalidades

O NetLab (Figura 4.2) é capaz de editar e mostrar graficamente Rede de Petri do tipo *lugar-transição*. Além disso, é possível simular passo-a-passo da Rede de Petri da maneira *stepwise*, isto é, uma *transição* por vez. Para criar a simulação da Rede, o usuário clica em uma *transição* que está *habilitada*, *disparando*-a.

O NetLab também é compatível com *labels*, como propostos na PNML, que serão utilizados para determinar o nome da variável associada ao *lugar* ou *transição*. Assim, o aplicativo se torna adequado ao intuito deste trabalho e também de fácil acesso para os desenvolvedores.

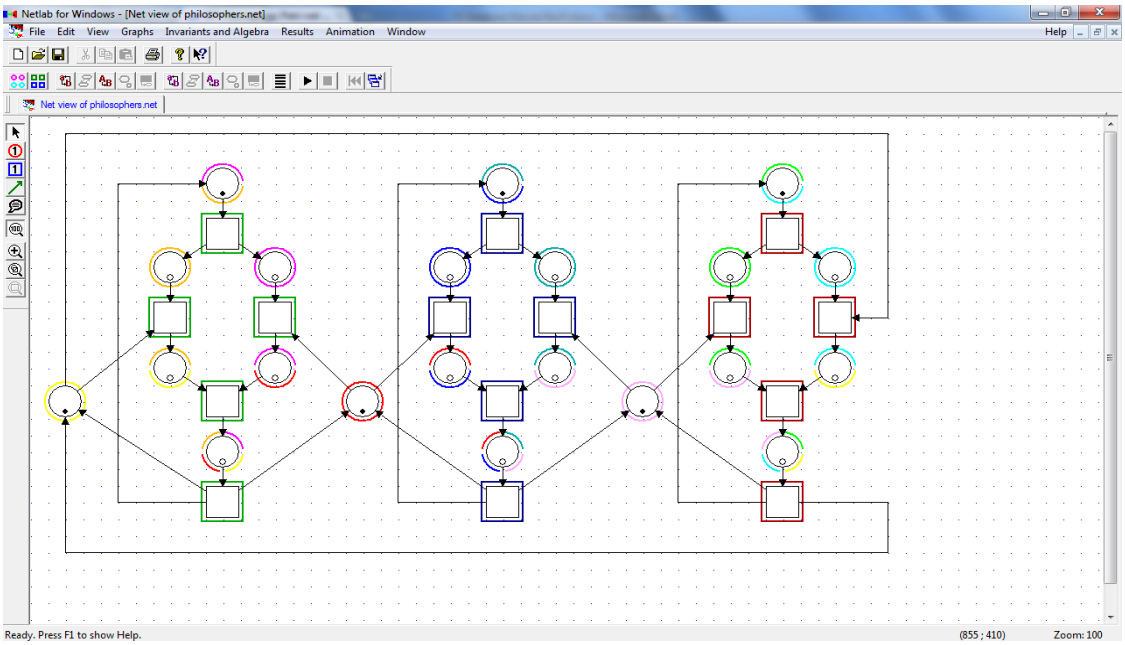


Figura 4.2 Tela de edição do NetLab

4.3. Beremiz – IDE de linguagens da IEC 61131-3

4.3.1. Descrição

O Beremiz é um *software* de código livre desenvolvido por pesquisadores portugueses e franceses para possibilitar ao usuário acadêmico o contato com linguagens da IEC 61131-3 de maneira gratuita, sem necessidade de licenças (Tisserant, Bessard e de Sousa, 2007).

4.3.2. Funcionalidades

O programa possui uma interface gráfica ligada diretamente à PCLOpen XML, possibilitando o intercâmbio entre diversas ferramentas de desenvolvimento.

Além da interface gráfica o programa realiza um processo de compilação das linguagens utilizadas para a linguagem C++, de modo a simular o processo em um microcomputador, sem a necessidade direta de um CLP(TISSERANT, BESSARD e DE SOUSA, 2007).

O estrutura de dados do Beremiz foi criada sobre os esquemas oficiais divulgados pela PLCOpen. Portanto, como a estrutura de dados foi criada em cima das relações entre objetos definidas pelo esquema (.xsd) o Beremiz pode ser utilizado como um validador do eschema PLCOpen TC6 XML. Com isso, para efeitos deste trabalho, o Beremiz é utilizado como um validador da consistência da saída do transcritor. Ao ler e validar o arquivo XML gerado, o Beremiz cria uma visualização gráfica, possibilitando a conferência dos diagramas gerados.

Um exemplo de visualização gráfica de um diagrama LD pode ser observado na Figura 4.3.

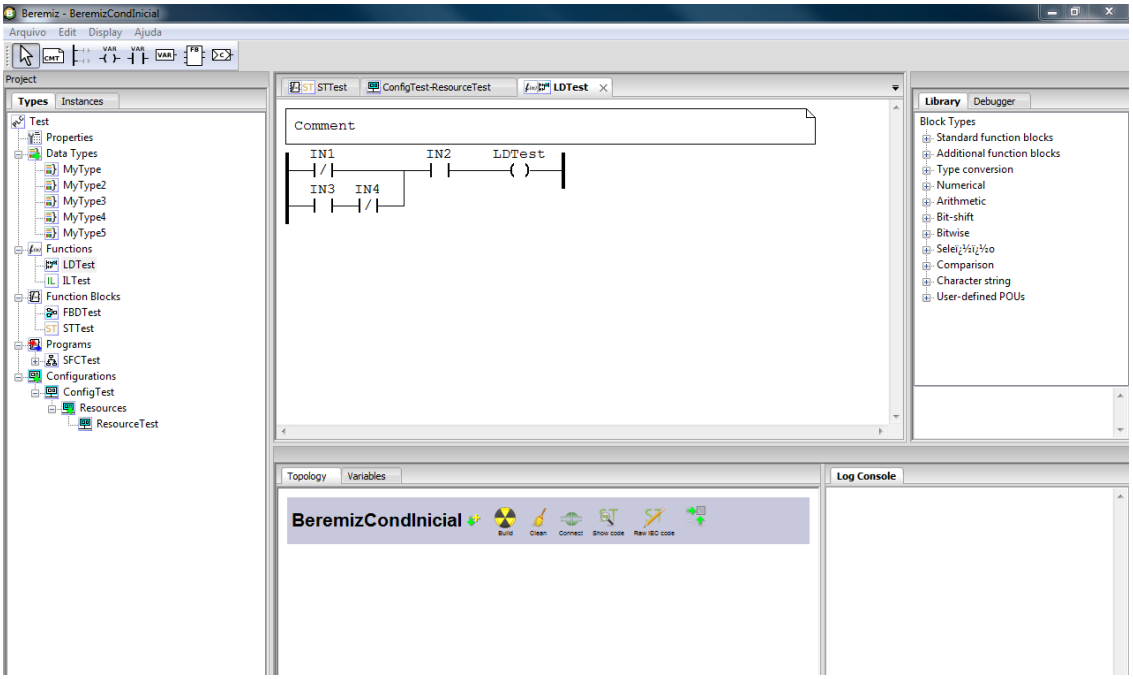


Figura 4.3 O ambiente de trabalho do Beremiz

5. A proposta do transcritor

Este capítulo se preza a esclarecer os parâmetros do trabalho realizado. Para isso, são definidas as linguagens de entrada e de saída do transcritor, o formato de arquivo suportado pelo mesmo e, por fim, as funcionalidades adicionais do programa.

5.1. Linguagem de entrada – representação da Rede de Petri

Com a recente padronização, por parte da IEC, da linguagem de intercâmbio de Rede de Petri, a PNML, a comunicação entre diferentes desenvolvedores fica mais viável, facilitando trabalhos conjuntos e cooperação a nível internacional (KINDLER e WEBER, 2003). Justamente por causa desta normatização, a linguagem de entrada escolhida para trabalho no transcritor será a PNML, com sua sintaxe básica, como definida em (PNML ORGANIZATION, 2010).

5.2. Linguagem de saída – representação das linguagens da IEC 61131-3

5.2.1. O alvo da saída do transcritor

Para alcançar o objetivo de gerar código em linguagem de máquina, após realizar o levantamento bibliográfico sobre o assunto, dispunha-se de duas alternativas:

A primeira seria adotar um CLP como padrão, e gerar a saída do transcritor na linguagem proprietária do seu fabricante. No caso da escolha desta opção, o transcritor seria uma ferramenta restrita e atrelada a linguagem de apenas um fabricante. A vantagem desta linha de trabalho seria a possibilidade de geração de linguagem a nível de máquina, pronta para ser carregada no controlador.

A segunda opção seria a utilização de uma ferramenta de desenvolvimento de linguagem de controle com suporte a uma linguagem padronizada, pertencente a IEC 61131-3, e que pudesse gerar código de máquina para mais de um controlador, de mais de um fabricante. Existem algumas destas ferramentas existentes no mercado atualmente, como o CoDeSys, da S3, e o ControlBuild, da Geensoft.

A maioria das instituições, tanto acadêmicas quanto privadas, trabalha com mais de um CLP, de fabricantes diferentes. Assim, programadores precisam ser treinados em diferentes linguagens para ser proficientes na operação dos mesmos. As linguagens da IEC 61131-3 reduziram esta necessidade, possibilitando a um programador familiarizado com elas a operação de diversos CLP's (KARL-HEINZ e TIEGELKAMP, 2001).

Portanto, a geração de código apenas para um CLP seria restritiva e incondizente com as tendências de desenvolvimento no mercado atual. A opção escolhida foi então o trabalho com uma plataforma de desenvolvimento com suporte às linguagens da IEC 61131-3 e geração de código para diversos CLPs.

O CoDeSys V3 é frequentemente tido como referência no ramo da padronização das linguagens de controle (WITSCH e VOGEL-HEUSER, 2009), tanto por causa do seu suporte às linguagens da IEC 61131-3 quanto pela quantidade de fabricantes que participam da Automation Alliance, uma Rede de fabricantes parceiros da 3S, criadora do CoDeSys (AUTOMATION ALLIANCE, 2010). Além disso, o CoDeSys em sua versão para desenvolvedores é gratuita (SMART SOFTWARE SOLUTIONS, 2010). Assim, esta foi a ferramenta de trabalho escolhida.

O CoDeSys possui capacidade de importação de linguagens textuais da IEC61131-3 no formato proposto pela PLCOpen, o PLCOpen XML. Como visto, a PLCOpen XML é mais uma tentativa de padronização das linguagens de programação de controladores de máquina (PLCOPEN, 2008) e, como tal, pode ser utilizada por desenvolvedores para aplicação em inúmeras ferramentas, para simulação, validação, teste, e outros fins.

Portanto o transcritor realizará a exportação das linguagens escolhidas no formato padrão da PLCOpen XML.

5.2.2. As linguagens da IEC 61131-3 escolhidas

Dada a utilização do PLCOpen XML como linguagem de intercâmbio, o transcritor utilizará três linguagens da IEC 61131-3: o LD, o SFC e o ST, de modo a:

- Possibilitar a abertura do programa gerado em ST no CoDeSys, permitindo simulação e programação direta de CLPs;
- Utilização dos modelos em LD e SFC em diversas outras aplicações que dão suporte à linguagem PLCOpen XML.

A escolha do LD se deu por ser a lógica de projeto de controle de sistemas mais popular na indústria americana (LUCAS e TILBURY, 2005). A sua aplicação é variada e flexível (LUCAS e TILBURY, 2005), e a concepção deriva diretamente do conceito de relês (FREY, 2001), tornando-a uma linguagem gráfica de fácil utilização, quando as variáveis do sistema controlado são *booleanas* (KARL-HEINZ e TIEGELKAMP, 2001). Como a Rede de Petri proposta para trabalho utiliza variáveis deste tipo, o LD pode ser considerado uma boa linguagem para representação.

Já o SFC, por ser adequado para a modelagem de processos que podem ser divididos em passos (KARL-HEINZ e TIEGELKAMP, 2001), se torna uma lógica de modelagem compatível com as Rede de Petri e, portanto, com o transcritor.

Por último, o ST foi escolhido como saída para o CoDeSys por ser a linguagem textual da IEC61131-3 com o maior crescimento em termos de utilização e aceitação (THAYER, 2009). Além disso, o seu entendimento é simples para usuários familiarizados com linguagens de programação de alto nível (THAYER, 2009).

5.3. Tipo de Rede de Petri suportado

5.3.1. Introdução

Após um estudo dos tipos de Rede de Petri existentes atualmente e do conceito original da Rede básica de condição-evento, é possível chegar à conclusão que este modelo não se comunica com o ambiente, sendo uma ferramenta de processamento autônoma (FREY, 2001).

Para realizar a interface da Rede de Petri com o ambiente, será utilizada uma de suas extensões, a SIPN proposta por (FREY, 2001), utilizando uma sintaxe simplificada, de modo a garantir a sua utilização por um número maior de desenvolvedores.

5.3.2. Os elementos da SIPN

A SIPN é uma extensão direta das Redes de condição-evento. Ela adiciona, como visto anteriormente, mapeamento de variáveis de entrada às *transições* e variáveis de saída aos *lugar*es.

O transcritor faz o mesmo com as Redes de condição-evento, utilizando uma sintaxe de atribuição de variáveis que será vista adiante.

Assim, toda rede de Petri de condição-evento pode também ser interpretada pelo transcritor. Este mecanismo foi assim idealizado de maneira a flexibilizar o uso do mesmo por não haver necessidade de aprendizado, por parte do desenvolvedor, de mais uma linguagem de programação e representação.

5.3.3. Os inputs e outputs e a dinâmica da Rede

O transcritor utiliza o seguinte esquema de variáveis e dinâmica de processamento da Rede:

- *Lugares e variáveis de saída*: na SIPN, as variáveis de saída são representadas por *lugar*es. O transcritor, para realizar o mapeamento nas redes de Petri condição-evento, utiliza os *labels*: interpreta, portanto, o nome de um *lugar* como nome de uma variável de saída associado a ele. Vale ressaltar que as variáveis de saída controladas são do tipo *BOOL*, representadas como *FALSE* (0) no caso de ausência de *marca* e *TRUE* (1) no caso de presença de *marca*.
- *Transições e variáveis de entrada*: cada *transição*, para ocorrer, precisa estar *habilitada* (na acepção clássica da palavra nas redes de Petri de condição-evento) e a variável de entrada associada a ela, que é do tipo *BOOL*, precisa ser *TRUE*. A variável associada à *transição*, assim como nos *lugar*es, é descrita pelo nome da *transição* expresso no *label*.
- *Variáveis internas da Rede – lugar*es e *transições*, no entanto, podem precisar ser utilizados na rede de Petri como variáveis internas ou auxiliares. Assim, faz-se necessária a criação de uma sintaxe dedicada para *lugar*es e transições que não estão

associados a nenhuma variável de entrada ou saída. Isso é feito por meio da expressão *default*, inserida no início do nome do *lugar* ou *transição*. Assim:

- *defaultLugar1;*
- *defaultTransição2;*
- *defaultVariável3.*

São exemplos de nomes de *lugar*es e *transições* que não possuem variáveis de entrada ou saída correspondentes.

- *Variáveis negadas*

  Ao desenvolver Rede de Petri, o desenvolvedor pode precisar utilizar uma variável de saída ou de entrada negada. Isto é, uma variável de saída que deve ser *TRUE* quando não há *marca* no seu *lugar* correspondente e *FALSE* caso contrário. De outra maneira, pode-se precisar de uma *transição* cuja ocorrência só será possível quando a variável de entrada associada a ela seja *FALSE*, e vice-versa. Para isso, o desenvolvedor precisa inserir o caractere “!” no início do nome do *lugar* ou *transição*. Assim:

  - !x1;
  - !lugar5;
  - !*transição*4.

  São exemplos de nomes de *lugar*es e *transições* cujas variáveis de entrada e saída associadas são do tipo negado. É importante notar que o desenvolvedor deve garantir o caráter único dos nomes de *lugares* e *transições*. Ou seja, para efeitos de transcrição, um *lugar* e uma *transição* não podem ter o mesmo nome, o que acarretaria em uma variável concomitantemente de saída e de entrada.

5.3.4. A ordem de disparo – transições default

A presença de *transições* do tipo *default* cria uma necessidade de definição de ordem de execução das mesmas, dado que podem ocorrer situações não-determinísticas.

Nas SIPNs o processamento da Rede é feito da seguinte maneira: para um dado conjunto de variáveis de entrada, a Rede irá percorrer e ativar todas

as *transições* possíveis, até atingir a estabilidade, e em seguida gerar os *outputs* associados aos *lugar*es com *marca*s. Desta maneira, o disparo de duas *transições default*, quando *habilitadas* conjuntamente, será observado pelo ambiente externo como simultâneo.

5.3.5. Considerações sobre o projeto das Redes

Como pode ser observado, a ordem de disparo dos eventos das SIPNs é regida pelas variáveis de entrada associadas. No entanto, isto não previne a ocorrência de conflitos ou *deadlocks*.

Portanto, é essencial a um desenvolvedor conhecer os conceitos por trás da Rede de Petri e dominar a modelagem e controle de SED como disciplina ao projetar e desenvolver um modelo da mesma, de modo a evitar conflitos e *deadlocks* advindos de uma modelagem incorreta. O transcritor, assim, não irá se prezar a detectar possíveis “defeitos” na dinâmica de *disparo* da rede de Petri.

5.4. Funcionalidades do transcritor

5.4.1. Exibição das Redes

O transcritor proposto é capaz de abrir arquivos PNML, como descrito adiante, e construir graficamente a rede de Petri para verificação visual. No entanto, o programa não é um editor de rede de Petri.

5.4.2. Sequência de disparos

De modo a possibilitar ao desenvolvedor validar o modelo em Rede de Petri, o programa possui uma ferramenta de simulação de *disparo* das *transições* na rede de Petri. Como as *transições* estão associadas a variáveis de entrada, o usuário poderá simular, por meio de botões, a mudança de cada uma destas variáveis, observando então as saídas decorrentes do evento.

5.4.3. Geração de arquivos

Como descrito anteriormente neste trabalho, após o processamento, o transcritor gera três tipos de arquivo, todos na sintaxe da PLCOpen XML.

É dada a opção ao desenvolvedor de escolher a linguagem IEC 61131-3 de saída: LD, SFC ou ST.

5.5. Linguagem de desenvolvimento

O transcritor foi implementado na linguagem Visual Basic .NET. Há alguns motivos para a escolha desta linguagem.

O primeiro é a licença que a Escola Politécnica possui do ambiente de desenvolvimento Visual Studio 2008. O Visual Studio, além de possuir suporte direto ao Visual Basic .NET, possui bibliotecas de trabalho em XML com suporte a *schemas*. Desta maneira, a manipulação, importação e exportação dos arquivos em XML é facilitada (MICROSOFT CORPORATION, 2003). Além disso, o Visual Studio possui um ambiente de desenvolvimento de GUI (*graphical user interface*), elemento essencial para o desenvolvimento dos módulos de simulação e exibição de redes de Petri do transcritor.

Em segundo lugar, uma linguagem orientada a objetos é útil para a modelagem do transcritor, dado que os objetos dos tipos *Rede de Petri*, *arco*, *lugar*, *transição* observados na PNML poderão ser mapeados como instâncias de classes.

5.6. Fluxograma de trabalho com o transcritor

Com as definições postuladas no item 5, é possível esboçar o fluxograma contínuo de trabalho ao se utilizar o transcritor. O usuário é capaz de realizar a modelagem em rede de Petri e programá-la diretamente no CLP escolhido.Com isso, o fluxograma do trabalho fica como na Figura 5.1.

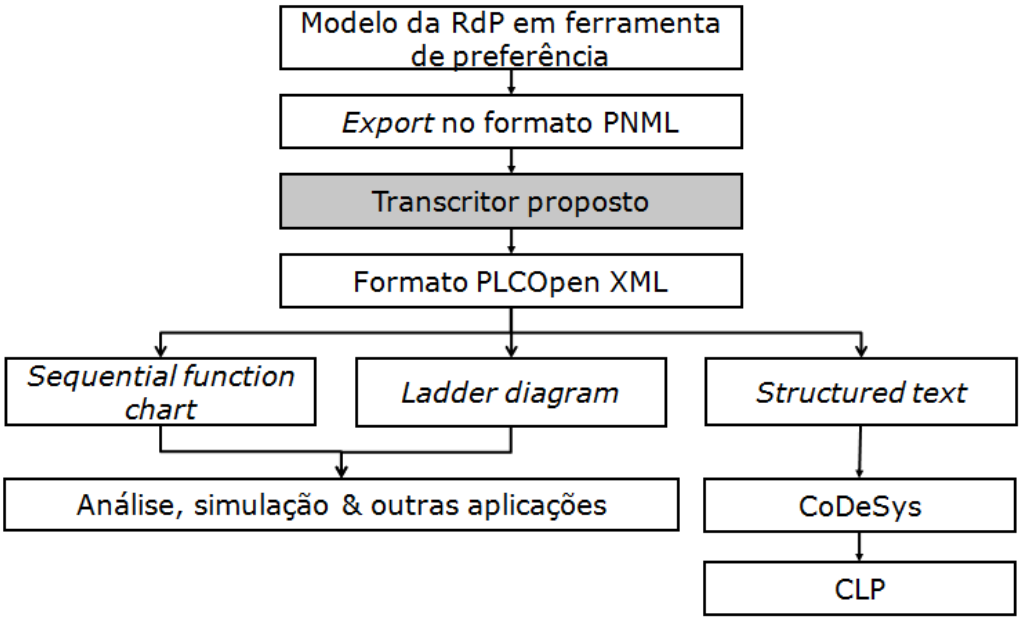


Figura 5.1 Fluxograma do trabalho

6. O algoritmo de transcrição

6.1. Rede de Petri para LD

O algoritmo de transcrição da rede de Petri para o Ladder Diagram escolhido para uso no projeto pode ser encontrado em (SANTOS FILHO e MIYAGI, 1997). Nesse artigo, o Prof. Dr. Diolino José dos Santos Filho propõe um mapeamento da rede de Petri em um Ladder Diagram separado em três blocos.

Como descrito previamente, o diagrama é processado por completo uma vez em cada *scan time*. Portanto, para cada mudança de estado na rede de Petri (ou seja, para cada movimentação das *marcas*) o *Ladder Diagram* correspondente é percorrido uma vez.

Dessa maneira a proposta de (SANTOS FILHO e MIYAGI, 1997) é realizar a adaptação em três passos: inicialmente, o programa verifica quais *transições* estão *habilitadas*. Em seguida, executa as *transições*, trocando as *marcas* de *lugares*. Por último, baseado na posição atual das *marcas*, o programa *habilita* as variáveis de saída correspondentes aos *lugares* com *marcas*. Adicionalmente é preciso inserir no LD um bloco de inicialização, para definir as condições iniciais (de acordo com as *marcações* iniciais na rede a ser transcrita). Graficamente, o processo pode ser observado na Figura 6.1.

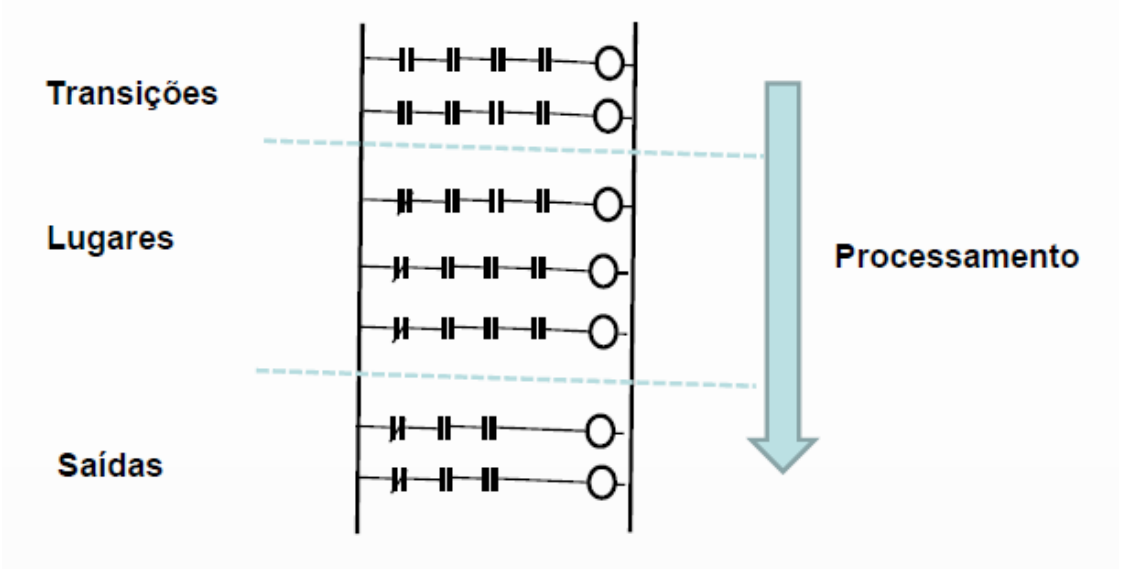


Figura 6.1 O algoritmo de transcrição utilizado (SANTOS FILHO e MIYAGI, 1997)

Para a descrição detalhada do algoritmo de transcrição será utilizada a rede da Figura 6.2, obtida em (SANTOS FILHO e MIYAGI, 1997). Nela há três

transições, ligadas às variáveis de entrada B1, B2 e B3. Como pode ser visto, a terceira *transição* possui uma combinação lógica das entradas B2 e B3. Há três *lugares*, com dois deles ligados a variáveis de saída (BM1 e BM2), que são acionadas quando há *marcas* nos respectivos *lugares*.

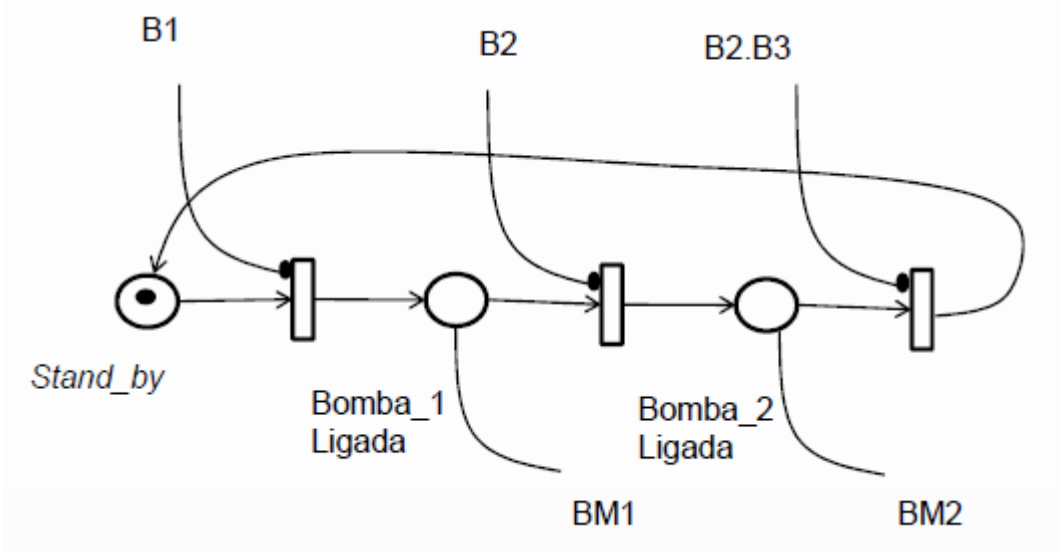


Figura 6.2 Exemplo de rede de Petri para descrição do algoritmo(SANTOS FILHO e MIYAGI, 1997)

6.1.1. Variáveis internas do LD

Para realizar a transcrição é necessário criar variáveis auxiliares internas, que representam os estados dos elementos *lugar* e *transição* da rede de Petri transcrita.

Portanto, cada *lugar* da rede de Petri terá duas variáveis associadas: primeiramente, a variável de saída do programa, representada pelo nome do *lugar*; em seguida, uma variável interna que representa a presença de uma *marca* no mesmo. Esta variável, para efeitos deste trabalho, é definida como o nome do *lugar* seguido da terminação *Local*. Assim, um *lugar* denominado *L1* possui as variáveis associadas *L1* e *L1Local*.

O mesmo é válido para as *transições*: há uma variável de mesmo nome denotando o estado da variável de entrada e uma variável de terminação *Local* para denotar a *habilitação* da *transição*. Portanto, uma *transição* de nome *T1* possui as variáveis *T1* e *T1Local*.

6.1.2. Habilitação das transições

Na primeira etapa do *Ladder Diagram* transcrito há a definição de quais *transições* estão *habilitadas*, como visto na Figura 6.3.

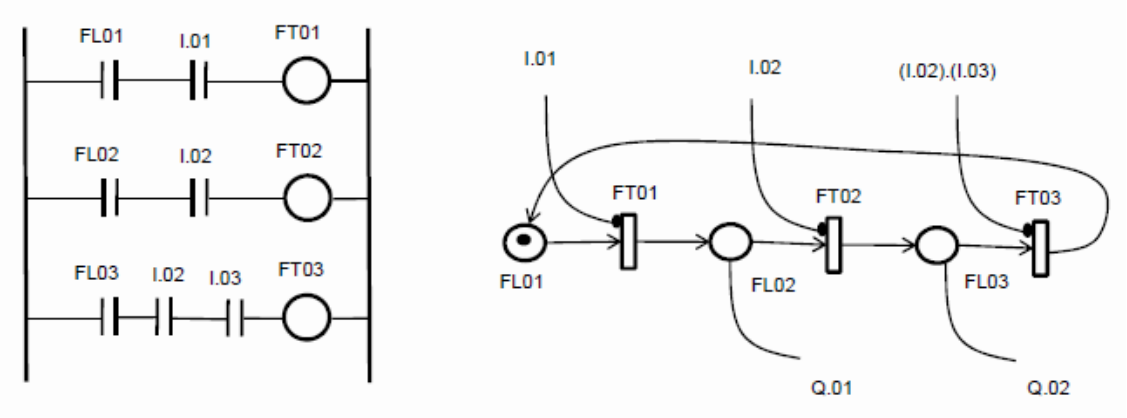


Figura 6.3 A habilitação das transições expressa no LD (SANTOS FILHO e MIYAGI, 1997)

Isso é feito definindo-se cada *transição* como uma bobina, utilizando a variável auxiliar local (na Figura 6.3, as variáveis locais são FT01, FT02 e FT03). Em seguida, define-se os *lugares* que a *habilitam* como contatos. Por último, no mesmo *rung*[4], deve-se definir a variável de entrada associada à *transição* como um contato (ainda na Figura 6.3, as variáveis de entrada associadas a *transições* são I01 e I02).

6.1.3. Fluxo de marcas

Uma vez processada a primeira seção que *habilita* as *transições*, é preciso definir a movimentação das *marcas*, como apresentado na Figura 6.4.

Para tanto, é preciso criar dois tipos de *rung* para cada *transição*: um com bobinas de *set* e outro com bobinas de *reset*. Os *rungs* com bobinas de *reset* irão retirar as *marcas* dos *lugares* cujos *arcos* apontam para a *transição* considerada Isso é realizado por meio das variáveis locais associadas aos *lugares* (na Figura 6.4 as variáveis locais associadas a *lugares* possuem sufixo “FL”). Os *rungs* com bobinas de *set* irão colocar *marcas* nos *lugares* para os quais os *arcos* que saem da *transição* considerada apontam, igualmente com o uso de variáveis locais.

Assim uma *transição* terá tantos *rungs* correspondentes quantos forem os *arcos* ligados a ela.

[4] Uma linha do diagrama, como descrito na seção 3.3

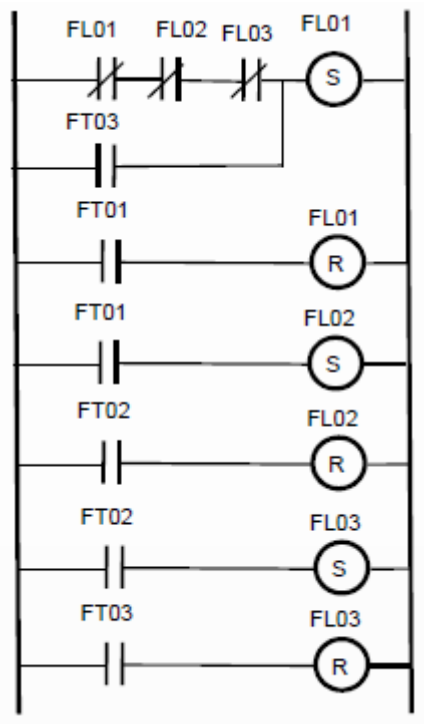


Figura 6.4 A movimentação das marcas no LD (SANTOS FILHO e MIYAGI, 1997)

Adicionalmente há uma seção para a inicialização das variáveis do programa. Isto é representado como um *rung* que possui contatos negados para cada um dos *lugares* da rede. Desta maneira, as bobinas neste *rung* serão acessadas somente durante a primeira varredura a ser realizada pelo CLP, quando todos os contatos estiverem em *false*. As bobinas deste *rung* devem ser do tipo *set*, sendo cada uma correspondente aos *lugares* que possuem *marcação* inicial.

Quanto a este passo da transcrição é preciso ressaltar que há duas maneiras de inicialização de acordo com a *marcação* inicial. A primeira, proposta em (SANTOS FILHO e MIYAGI, 1997), prevê a inicialização por meio de *rungs* auxiliares associados aos *rungs* de *set* de cada *lugar*. Tal método pode ser observado na Figura 6.4, na primeira linha: pode-se observar um *rung* adicional que contém os contatos negados ligado à bobina de *set*.

Outra maneira de inicialização é por meio de um *rung* adicional exclusivo para a *marcação* inicial, localizado em qualquer posição dentro do LD. Este *rung* só será ativado uma única vez, durante a primeira varredura. Nele há a presença dos contatos negados associados a cada um dos *lugares* e diversas bobinas, correspondentes a cada um dos *lugares* com *marcação* inicial.

Portanto, para uma rede com três *lugares*, sendo que dois deles possuem *marcação* inicial, o *rung* de inicialização seria similar ao observado na Figura 6.5. Este é o método adotado para a transcrição neste trabalho.

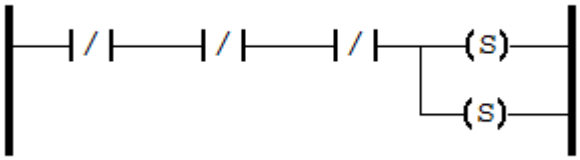


Figura 6.5 Exemplo de inicialização das variáveis em um LD

6.1.4. Habilitação das saídas

Ao fim do *scan time*, quando a movimentação das *marcas* já ocorreu, é preciso *habilitar* as variáveis de saída associadas a cada *lugar*. Isso é feito como ilustrado na Figura 6.6.

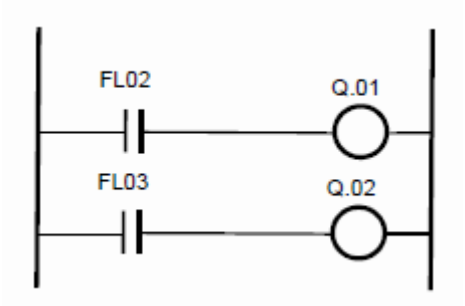


Figura 6.6 A definição das variáveis de saída no LD (SANTOS FILHO e MIYAGI, 1997)

Para este passo cada *lugar* possui um *rung* correspondente, no qual há um contato associado à variável local e uma bobina associada à variável de saída. Esta é uma bobina comum, e não de *set* ou *reset*. Desta maneira, ao fim do *scan time*, apenas os *lugares* com *marca* (aqueles cujas variáveis locais tem estado *true*) tem as suas respectivas variáveis de saída habilitadas.

6.2. Rede de Petri para SFC

O algoritmo de transcrição de Rede de Petri para SFC também é fundamentado no trabalho de (SANTOS FILHO e MIYAGI, 1997). Desta vez, diferentemente da transcrição para LD, há uma semelhança explícita entre a linguagem de entrada e saída do transcritor. Na verdade, essa semelhança não é apenas uma coincidência pois o SFC é uma linguagem derivada da Rede de Petri (MIYAGI, 1996).

Essa semelhança entre as linguagens simplifica bastante a transcrição pois, em linhas gerais, a transcrição consiste em estabelecer relação entre os diferentes elementos estruturais das duas linguagens.

6.2.1. Os elementos estruturais do SFC

Foram descritos na seção 3.4.1 quatro elementos estruturais do SFC: o passo; a transição; a conexão; e a ação. Para seguir a notação proposta por (SANTOS FILHO e MIYAGI, 1997) será introduzido um novo elemento estrutural chamado *Receptividade*.

A *Receptividade* consiste na condição associada à uma *transição* tal que a *transição* só pode ocorrer caso seja satisfeita essa condição.

Com isso, é possível determinar o comportamento de uma *transição* ao se definir o estado da sua receptividade. Caso a condição modelada seja satisfeita, a receptividade fica ativa permitindo que a *transição* seja realizada. No caso oposto, em que a condição modelada não é satisfeita, a receptividade fica inativa e a *transição* não se realiza.

6.2.2. O mapeamento entre Rede de Petri e SFC

A seguir, serão mapeados os elementos estruturais da Rede de Petri por elementos estruturais do modelo SFC (nessa ordem). O elemento *lugar* da Rede de Petri pode ser mapeado como um *passo* do modelo SFC, a *transição* pode ser mapeada como a *transição* e o *arco* como a *conexão*. Tendo esses elementos mapeados, é necessário ainda mapear os elementos da rede SIPN, escolhida para desenvolver o transcritor. Esse tipo de Rede de Petri possui as *variáveis de entrada*, ligadas às *transições*, e as *variáveis de saída*, ligadas aos *lugares*. As *variáveis de entrada* podem ser mapeadas como a *receptividade* de cada *transição* do modelo SFC. Por último, as *variáveis de saída* podem ser mapeadas como as ações do modelo SFC.

Seguindo as diretrizes de mapeamento propostas acima, a transcrição é concluída deixando por especificar apenas as situações particulares da transcrição.

6.2.3. Particularidades da transcrição

O mapeamento é simples para o caso base que pode ser observado na Figura 6.7. No caso base não há concorrência de eventos ou processos, havendo apenas um ramo na rede.

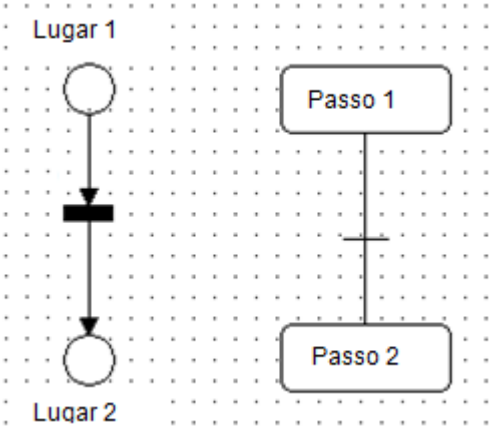


Figura 6.7 Transcrição simples

O exemplo da Figura 6.7 mostra a transcrição de uma Rede de Petri simples por meio do mapeamento direto entre os elementos estruturais de ambas as linguagens. Na figura, apenas é escrito o mapeamento do elemento *lugar*, apesar de o *arco* e a *transição* também terem sido mapeados.

A particularidade está nas situações de conflito e de paralelismo, previstas pelo modelo SFC. Os casos de conflito, conforme explicado na seção 5.3.5, não serão tratados no transcritor (há um aviso de possível ocorrência de conflito, que deve ser tratado pelo desenvolvedor na etapa de projeto do sistema). Dessa maneira, deve ser tratado apenas o caso de paralelismo.

O paralelismo na Rede de Petri acontece quando dois ou mais *arcos* saem de uma mesma *transição*. Quando essa *transição* é *habilitada* e ativada, todos os *lugares* que sucedem a *transição* receberão *marcas*, caracterizando uma situação de paralelismo. A Figura 6.8 mostra três ramos em paralelo.

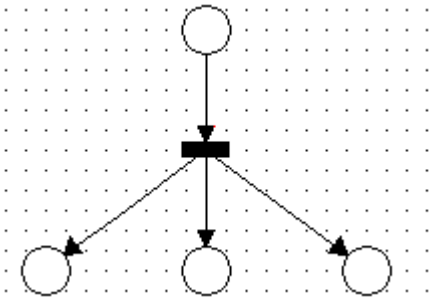


Figura 6.8 Criação de paralelismo numa Rede de Petri

O procedimento para transcrição consiste na inclusão de uma linha dupla de paralelismo da qual sairão os ramos em paralelo, ou seja, ao invés de conexões para novos ramos saindo diretamente da *transição*, as conexões saem da linha dupla de paralelismo. A Figura 6.9 permite uma melhor compreensão.

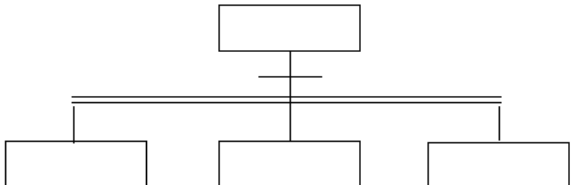


Figura 6.9 Criação de paralelismo no SFC

Da mesma forma que na Rede de Petri é possível criar um paralelismo, também é possível encerrar o paralelismo. Essa situação acontece quando dois ou mais *arcos*, vindos de *lugares* distintos, apontam para a mesma *transição*. Repara-se que segundo as regras da Rede de Petri definidas na seção 2.1, a *transição* para a qual estão sendo apontados os dois ou mais *arcos* só estará habilitada caso todos os *lugares* precedentes estejam *marcados*. Essa situação caracteriza o encerramento do paralelismo.

6.2.4. Variáveis locais do SFC

Para permitir a transcrição do código e o posterior processamento do mesmo é preciso criar variáveis auxiliares internas no diagrama do SFC.

As variáveis externas, que representam as entradas e saídas do sistema de controle, são definidas de acordo com os nomes das *transições* e *lugares* da rede de Petri a ser transcrita. Essas variáveis vão entrar nos blocos de ação associados a cada *step* do SFC.

Portanto, o nome de cada *passo* (*step*) deve ser uma variável local de processamento. Como no LD, a variável é definida como o nome do *lugar* seguido do sufixo “Local”.

6.2.5. Cuidados adicionais

É necessário ter alguns cuidados quanto à construção da Rede de Petri para obter um melhor resultado na transcrição ao SFC.

- *Passo inicial*

O primeiro cuidado a ser tomado é a identificação do passo inicial. Como o diagrama SFC não possui marcas como a rede de Petri, é necessário que a Rede de Petri a ser transcrita possua apenas um lugar que contenha marcação inicial. Caso contrário, o transcritor não será capaz de fazer a transcrição.

- *Para sistemas discretos cíclicos utilizar malha fechada*

Outro cuidado consiste na construção de uma rede de Petri com malha fechada quando houver o interesse na modelagem de sistemas com repetição. O sucesso da transcrição depende da existência de um único *arco* que aponta para o *lugar inicial*. Com essa prática, o transcritor é capaz de identificar a conexão corresponde ao arco em questão e a substituir pelo elemento *Jump*, que indica o passo inicial após acionamento da última transição.

- *Fluxo da Rede de Petri na sua construção*

O último cuidado importante consiste na maneira como a Rede de Petri deve ser construída a fim de permitir um diagrama SFC melhor distribuído e balanceado. Como o posicionamento de cada elemento do SFC é feito com base no posicionamento do elemento equivalente na rede de Petri após feita a transposição (a abscissa vira ordenada e viceversa), deve-se construir uma Rede de Petri com um fluxo principal da esquerda para a direita de tal forma que o SFC resultante tenha o fluxo de cima para baixo. No caso desse cuidado não ser tomado, a transcrição ocorre sem nenhum problema estrutural, apenas com uma diagramação ruim.

6.3. Rede de Petri para ST

6.3.1. Variáveis internas do ST

Para a transcrição da rede de Petri para a linguagem textual do ST foi criado um algoritmo baseado no algoritmo apresentado em 6.1, também dividido em três blocos de execução.

Assim como o algoritmo anterior, a transcrição para ST utiliza variáveis locais que representam a *habilitação* das *transições* e a presença de *marcas* nos *lugares*.

6.3.2. Habilitação das transições

Como no algoritmo de transcrição de rede de Petri para LD, o primeiro passo a ser realizado em cada *scan time* é definir quais *transições* estão *habilitadas*. Para isso é atribuído um valor à variável interna associada a cada transição. Este valor é a combinação lógica “AND” de três condições:

- Existência de *marcas* nos *lugares* que precedem a *transição* (ou seja, TRUE nas variáveis locais dos *lugares*);
- Ausência de *marcas* nos *lugares* que procedem à *transição* (ou seja, FALSE nas variáveis locais dos *lugares*);
- Ativação da variável externa de entrada associada à transição (TRUE na variável local da *transição*).

Portanto, para a rede apresentada na Figura 6.2, a seção de *habilitação* das *transições* pode ser observada na Figura 6.10.

```
FT01 := FL01 AND NOT FL02 AND I.01;
FT02 := FL02 AND NOT FL02 AND I.02;
FT03 := FL03 AND NOT FL01 AND (I.02 AND I.03);
```

Figura 6.10 A habilitação das transições expressa no ST

6.3.3. Fluxo de marcas

Para realizar o fluxo das *marcas* na rede transcrita é preciso criar estruturas de *set* e *reset* nas variáveis locais associadas aos *lugares*. Em ST, a estrutura utilizada para representar esta declaração é o IF. Portanto, cria-se uma estrutura do tipo IF para cada *transição*, tendo como condição a variável local associada à *transição*. Dentro da estrutura IF atribui-se os valores TRUE a todas as variáveis locais associadas aos *lugares* que se encontram depois da

transição e os valores FALSE a todas as variáveis locais associadas aos *lugares* que se encontram antes da *transição*.

Desta maneira o fluxo de *marcas* em ST que representa a rede da Figura 6.2 pode ser observado na Figura 6.11.

```
IF FT01 = TRUE THEN
    FL01 := FALSE;
    FL02 := TRUE;
END_IF

IF FT02 = TRUE THEN
    FL02 := FALSE;
    FL03 := TRUE;
END_IF

IF FT03 = TRUE THEN
    FL03 := FALSE;
    FL01 := TRUE;
END_IF
```

Figura 6.11 O fluxo de marcas expresso no ST

É importante observar que a estrutura IF se comporta como uma bobina de *set* ou *reset* dado que não há cláusula de ELSE, ou seja, o mesmo procedimento não é realizado uma vez que a transição deixe de estar habilitada.

6.3.4. Habilitação das saídas

Ao final de cada *scan time* é preciso habilitar as variáveis de saída de cada *lugar*. Isso é realizado ao se atribuir o valor das variáveis locais associadas a cada *lugar* (que representam a presença de *marca* no *lugar*) à cada variável de saída.

Desta maneira pode-se observar a habilitação das saídas para a rede exemplo na Figura 6.12.

```
Q.01 := FL02;
Q.02 := FL03;
```

Figura 6.12 A habilitação das saídas expressa no ST

6.3.5. Marcação inicial

Para se definir a marcação inicial na rede é preciso criar uma declaração que será percorrida apenas uma vez pelo CLP, no momento inicial da execução, quando todos os *lugares* estiverem sem *marcas*. Para isso é utilizada uma estrutura IF que tem como condição o valor FALSE em todas as

variáveis locais associadas aos *lugares*. O resultado da transcrição da rede exemplo pode ser visto na Figura 6.13.

```
IF NOT FL01 AND NOT FL02 AND NOT FL03 THEN
    FL01 = TRUE;
END_IF
```

Figura 6.13 A inicialização da marcação expressa no ST

7. A estrutura do transcritor

7.1. Projeto segundo a UML

Nesta seção procura-se fazer uma rápida revisão sobre UML (*Unified Modeling Language*). Segundo (OMG, 2005) a modelagem de um *software* é o planejamento e preparação realizado antes de realizar a implementação do mesmo. A UML é uma representação padrão para descrever esta etapa.

Existem diversas metodologias de projeto para *software*, envolvendo os diagramas propostos pela UML (OMG, 2005). Dos treze diagramas propostos, três foram utilizados para a elaboração e estruturação do *software* do transcritor:

- *Diagrama de casos de uso:* este diagrama descreve quais devem ser as funcionalidades de um programa, e como elas se relacionam com um *ator*, a representação do usuário final. Este diagrama possui alto nível de abstração e é, segundo (OMG, 2005), um Diagrama de Comportamento, frequentemente utilizado no começo do planejamento para se definir os requerimentos detalhados do programa.
- *Diagrama de componentes:* neste diagrama são representados os componentes do programa, que são representações lógicas de unidades de programa com funções individuais e interfaces de comunicação entre si. É um Diagrama de Estrutura (OMG, 2005).
- *Diagrama de classes*: neste diagrama são representadas as estruturas básicas da programação orientada a objetos, as classes. Cada classe é representada por um retângulo e pode possuir atributos e métodos, além de relações com outras classes. Neste diagrama o nível de abstração é muito menor do que no diagrama de casos de uso, definindo claramente ao desenvolvedor quais seções de código deverão ser implementadas (OMG, 2005).

7.2. Os casos de uso do programa

Este diagrama define quais são as funções que um usuário deve encontrar ao utilizar o transcritor. Ele pode ser encontrado na Figura 7.1.

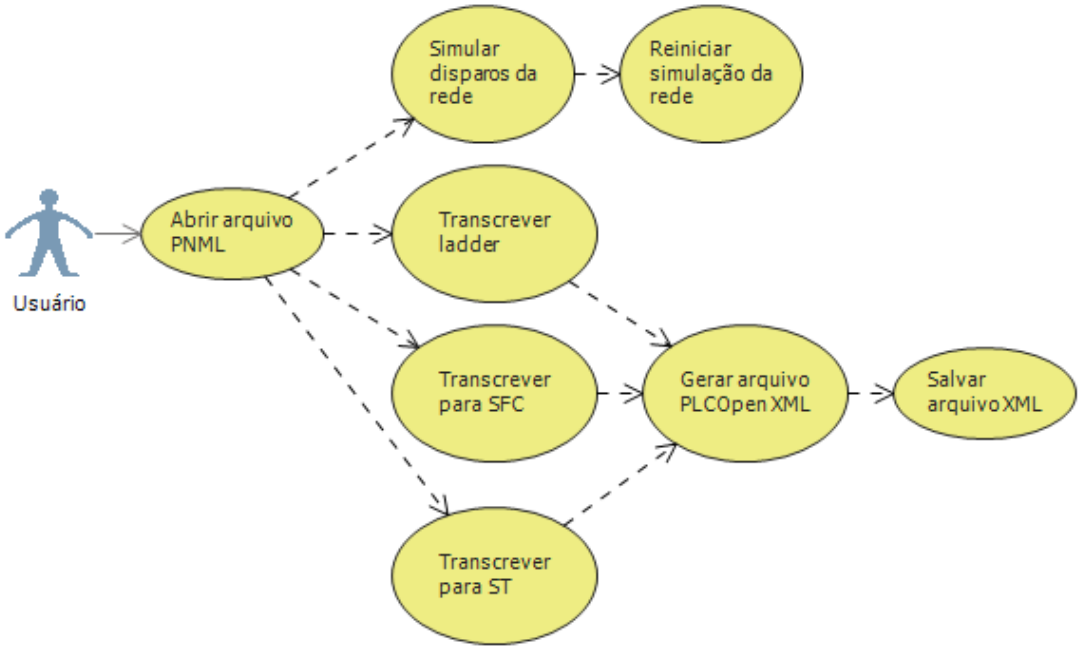


Figura 7.1 Diagrama de casos de uso

O primeiro passo para se utilizar o transcritor é abrir um arquivo PNML. Esta é uma condição para a habilitação de todas as outras funções do transcritor. Por isso, no diagrama de casos de uso, há relações de *dependência* que saem da ação de abertura do arquivo PNML.

Com uma rede PNML aberta, quatro opções de uso se tornam disponíveis para o usuário: uma de simulação de rede de Petri e três de transcrição propriamente dita.

7.2.1. Simulação

A simulação da rede de Petri se dá por meio de botões que representam cada uma das entradas do programa, respectivamente associadas às *transições*. A partir desses botões, o usuário é capaz de realizar o *disparo* das transições da rede de Petri.

Uma vez iniciada a simulação, o usuário habilita mais uma função do transcritor: reiniciar a simulação. Esta função retorna a rede de Petri carregada ao seu estado inicial.

7.2.2. Transcrição

Com a rede de Petri, as três funções de transcrição disponíveis ao usuário são: transcrição para LD, transcrição para SFC e transcrição para ST. Estas são as funções principais do transcritor, e são habilitadas por botões específicos do programa.

A transcrição é uma operação executada no *background*; enquanto é executada, o usuário não pode realizar outras ações no programa.

Uma vez realizada a transcrição o usuário pode realizar a ação de escrita. Como descrito na seção 5.4.3 a escrita é realizada em PLCOPen XML.

7.3. Os componentes do programa

Este diagrama, que pode ser encontrado na Figura 7.2, foi elaborado de maneira a criar elementos independentes para cada função do transcritor. Dessa maneira, há quatro elementos principais mais a interface, descritos a seguir:

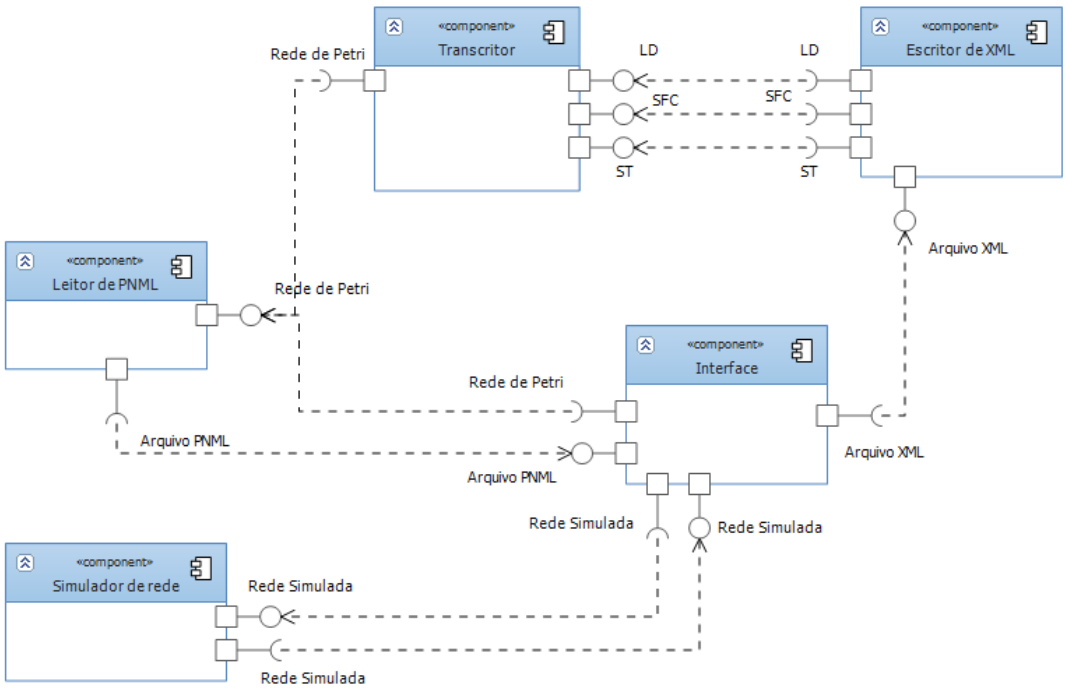


Figura 7.2 O diagrama de componentes do transcritor

7.3.1. Interface

Como todo programa, o transcritor precisa de uma interface de comunicação com o usuário. A interface é gráfica, isto é, os comandos não

precisarão ser inseridos via linha de comando e sim por botões e caixas de diálogo.

A *Interface* é uma classe *Windows Form*, classe padrão da linguagem Visual Basic para criação de elementos gráficos. Esse componente é responsável por ativar as outras funções do transcritor, de acordo com as entradas dadas pelo usuário.

7.3.2. Leitor de PNML

Como visto nos casos de uso, a primeira função do transcritor, que habilita as outras, é a abertura de um arquivo PNML que descreve uma rede de Petri.

Isso será feito por meio do *Leitor de PNML*, que recebe da *Interface* um arquivo com essa extensão e gera um objeto do tipo *rede de Petri*.

Este objeto é o modelo utilizado dentro do transcritor para todas as funções de simulação e transcrição. É a representação básica de uma rede.

7.3.3. Transcritor

Este componente é responsável pela ação de transcrição propriamente dita. Ele recebe um objeto do tipo *rede de Petri* do *Leitor de PNML*, e realiza a transcrição para LD, SFC ou ST, de acordo com o comando proveniente da *Interface.*

O transcritor cria três tipos de objeto: um *LDiagram,* um *SFCDiagram* ou um *STDiagram*. Estes objetos são as representações, para o transcritor, das linguagens de saída (LD, SFC e ST).

7.3.4. Escritor de XML

O *Escritor de XML* é o elemento responsável pela operação de criação do arquivo de extensão “.xml”, escrito PCLOpen XML.

Ao receber o comando da *Interface*, o *Escritor de XML* acessa o *Transcritor*, que possui os elementos do tipo *LDDiagram*, *SFCDiagram* ou *STDiagram* e em seguida realiza a escrita do arquivo correspondente. Este arquivo, de extensão “.xml”, fica disponível para a *Interface*, na qual a operação de “Salvar como” pode ser realizada.

7.3.5. Simulador de rede

Este é o componente responsável pela simulação da rede de Petri. Ele só é habilitado depois de um arquivo PNML ter sido carregado pelo *Leitor de PNML* e um objeto do tipo *rede de Petri* ter sido criado.

Uma vez iniciada a simulação, o *Simulador de rede* recebe a versão inicial do objeto *rede de Petri*. Em seguida, dadas as variáveis externas, realiza a movimentação das *marcas*. Ao finalizar a movimentação, retorna então o objeto modificado para a *Interface*, que a renderiza na tela do computador. Este procedimento é repetido para cada passo da simulação.

A dinâmica dos passos da simulação segue o comando do usuário. Este pode definir o estado de cada uma das variáveis de entrada da rede e então simular o passo.

Caso o usuário ative o comando de reinicialização da rede, a *Interface* acessa o objeto *rede de Petri* que está armazenado no *leitor de PNML*, e o processo de simulação é retomado.

7.4. As classes do transcritor

Depois de definido o diagrama de componentes, é criado o diagrama de classes. Como o Visual Basic é uma linguagem orientada a objetos, cada componente do transcritor é uma instância de uma classe. Além disso, os elementos que são trocados entre cada elemento do programa também são objetos, que por sua vez também são instâncias de classes.

7.4.1. Metodologia de criação das classes

Há uma ferramenta no Microsoft Visual Studio 2008 que se chama *XSD.exe*. Essa ferramenta, que pode ser rodada no *prompt* de comando, tem a funcionalidade de gerar esquemas XSD a partir de arquivos XML e a funcionalidade de criar classes em VB (ou em demais linguagens suportadas pelo aplicativo) seguindo a estrutura de dados de um esquema XSD. A Figura 7.3 mostra o programa.

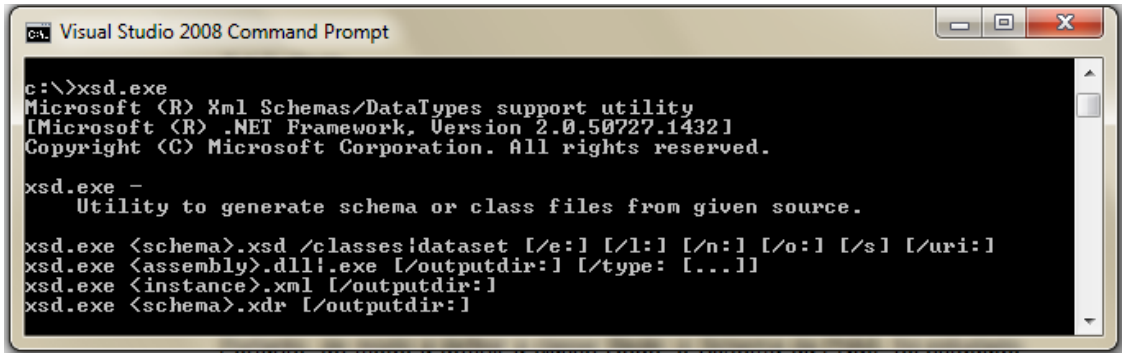


Figura 7.3 Aplicativo XSD.exe no prompt de comando

Como o sucesso do presente trabalho esta diretamente relacionado ao sucesso da leitura do arquivo de entrada, que é no formato PNML (.pnml), e ao sucesso na escrita dos arquivos de saída, que são no formato XML dentro do esquema PLCOpen TC6 XML, optou-se por utilizar a ferramenta citada para a criação automática das respectivas classes.

A vantagem dessa metodologia existe pois as classes geradas pela ferramenta *XSD.xls* possuem instruções de serialização de objeto. O conceito de serialização permite que objetos, instâncias de uma classe, possam ser armazenados no formato XML. Por outro lado, a desserialização, que nada mais é que o processo inverso, permite que informações armazenadas em um arquivo XML possam ser atribuidas ao objeto. Para que tais processos sejam possíveis, há uma série de instruções ligadas a cada classe, a cada atributo, a cada método e a cada propriedade que indicam ao computador como deverá ser feito o processo.

Exemplificando, no caso de existirem atributos que não devem ser armazenados a instrução é de ignorar:

```
<System.Xml.Serialization.XmlIgnoreAttribute()>
```

No caso de as instruções indicarem que o atributo deve criar um novo elemento XML obrigatório chamado “data”, o comando é:

```
<System.Xml.Serialization.XmlArrayItemAttribute("data",
IsNullable:=false)>
```

Com essa metodologia o trascritor seguiria, de maneira simplificada, os seguintes passos para a transcrição: 1) Desserialização do arquivo PNML; 2) Algoritmo de transcrição; 3) Serialização do arquivo no formato XML.

Para criação das classes para PNML, a metodologia utilizada partiu da construção de um exemplo de arquivo PNML (.pnml) com base no esquema

em RELAX-NG (.rng) que fosse bastante completo e detalhado, contendo exemplos de todas as possíveis composições do esquema. Com esse arquivo, foi gerado o esquema XSD equivalente. Por último, com esse esquema foram geradas automaticamente as classes que serviram de base para o trabalho.

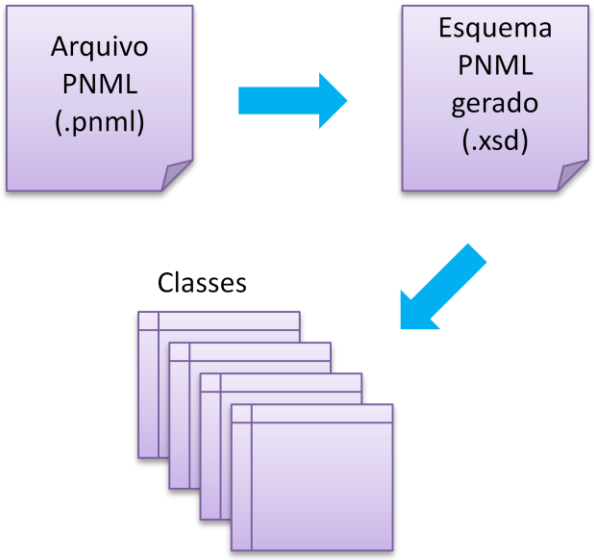


Figura 7.4 Criação das classes para PNML

Para criação das classes para SFC, LD e ST, a metodologia utilizada consistiu na geração automática de classes com o esquema oficial da PLCOpen. As classes obtidas serviram como base para o trabalho

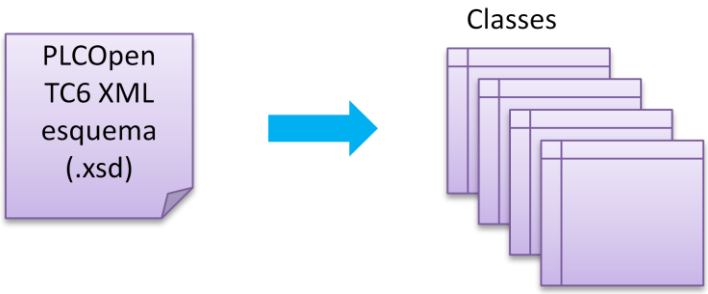


Figura 7.5 Geração das classes para SFC, LD e ST

Classes geradas automaticamente são bastante precisas. Entretanto, qualquer funcionalidade que precise ser incluída numa dessas classes requer que sejam tomadas os devidos cuidados para não perder o corpo inicial das mesmas. Um dos cuidados é manter o corpo inicial da classe inalterado, fazendo inclusões auxiliares apenas. Outro cuidado é manter coerente as instruções de serialização. Os cuidados são importantíssimos para garantir que as ferramentas de serialização de desserialização continuem funcionando.

7.4.2. Rede

A classe básica que define o objeto que representa a rede de Petri dentro do transcritor é a classe *Rede*. A composição de suas subclasses pode ser observada na Figura 7.6.

A classe *Rede* foi construída tendo como base a composição da linguagem PNML. Por ser uma linguagem de XML, a PNML possui elementos definidos por um *XML Schema*, como descrito anteriormente.

Portanto, de modo a definir a classe *Rede*, o *Schema* da PNML foi estudado. Cada subclasse representa um tipo de elemento presente na PNML.

Desta maneira, o *Leitor de PNML* é capaz de traduzir o conteúdo de um arquivo PNML para um objeto do tipo *Rede*.

As subclasses da *Rede* são:

- *Transição*
- *Lugar*
- *Arco*
- *Graphics*
- *graphicsPosition*
- *graphicsDimension*

Como pode ser observado na Figura 7.6, cada rede pode ter diversas *transições*, *lugares* e *arcos*, que por sua vez só podem pertencer a uma rede.

Cada *Transição, Lugar* e *Arco* é ligado a um objeto da classe *Graphics*. Esse objeto, por sua vez, é ligado a dois objetos: *graphicsPosition* e *graphicsDimension*. Estes objetos são responsáveis por armazenar a posição e dimensão do elemento, informações que podem ser utilizadas para o programa desenhar a rede.

7.4.1. LDiagram

A classe LDiagram é a representação do Ladder Diagram dentro do transcritor. Assim como na PNML, a sua criação foi baseada em uma representação existente, a PLCOpen XML. Por meio do Schema desta representação foi possível criar a classe e suas subclasses, que podem ser observadas na Figura 7.7.

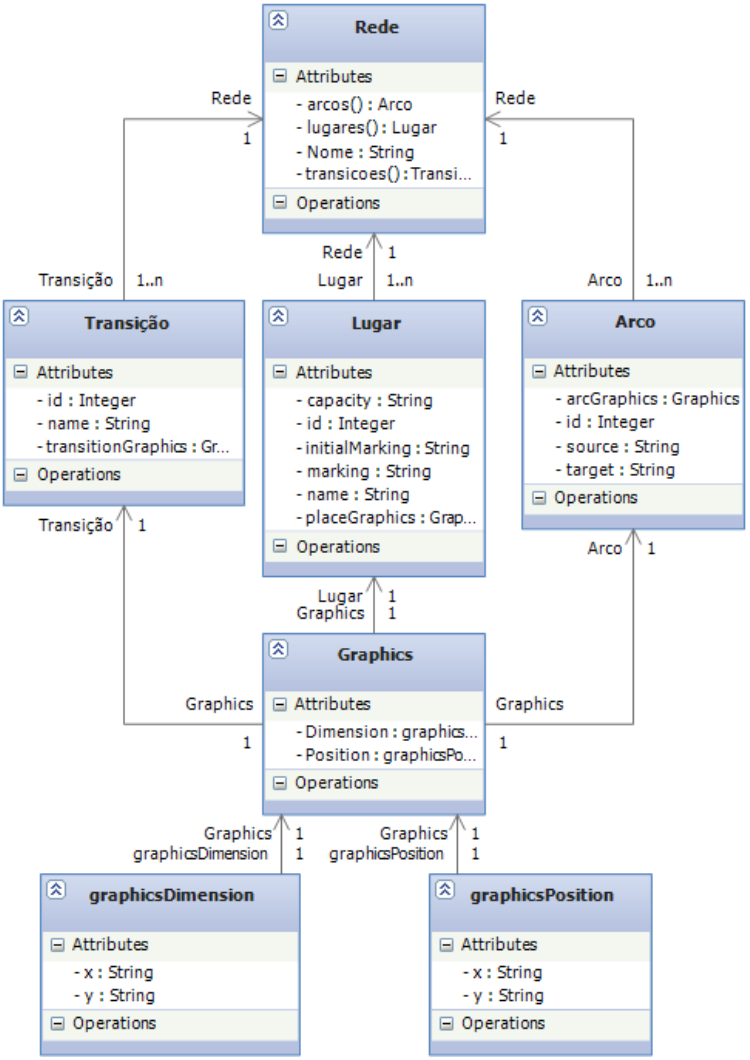


Figura 7.6 A classe Rede e suas subclasses

As denominações foram mantidas em sua língua original, o inglês, para facilitar o mapeamento no arquivo XML. São elas:

- LeftPowerRail
- Contact
- Coil
- RightPowerRail
- connectionPointOut
- connectionPointIn
- connection
- Position

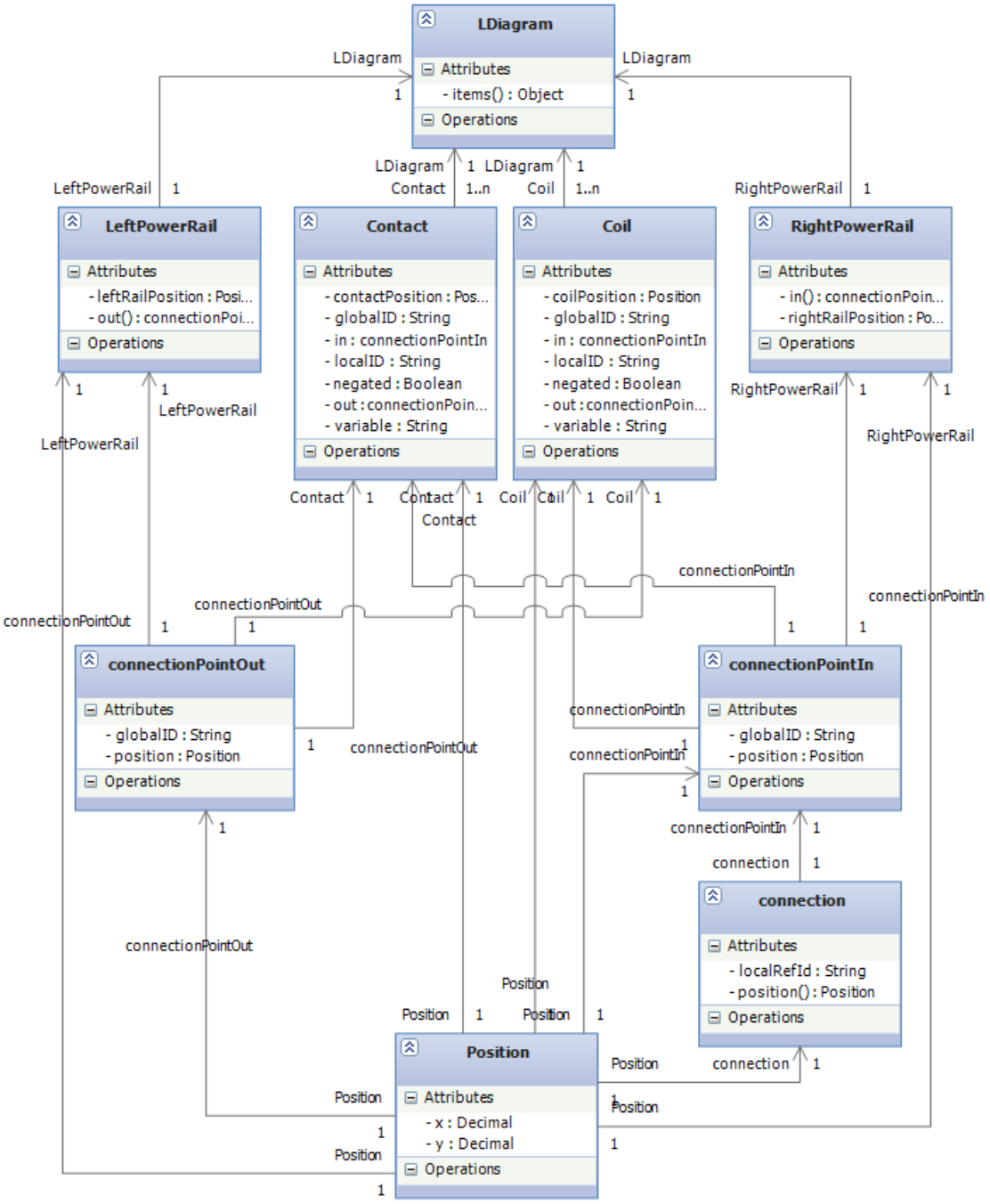


Figura 7.7 A classe LDiagram e suas subclasses

Cada *LDiagram* tem apenas um *LeftPowerRail* e um *RightPowerRail*. Estes objetos representam as linhas mestres do diagrama, respectivamente à esquerda e à direita. O *LeftPowerRail* possui um *connectionPointOut* e o *RightPowerRail* possui um *connectionPointIn*. A função desses objetos é descrita adiante.

Além disso, o *LDiagram* pode possuir diversos *Contacts* e *Coils*. Cada um desses elementos possui, por sua vez, um *connectionPointOut* e um

connectionPointIn. Com destes dois tipos de objeto é possível montar as linhas de conexão que ligam cada *rung* do diagrama, que são representadas por objetos da classe *connection*. Esses dois elementos possuem identificadores globais no programa. Assim, sabendo-se o *connectionPointOut* de um elemento, pode-se descobrir a qual elemento ele se liga, ao obter o *connectionPointIn* com o mesmo ID global.

Todas as subclasses do *LDiagram* possuem objetos do tipo *Position*, que definem as coordenadas do objeto no plano x-y, possibilitando assim a um editor desenhar o diagrama.

7.4.2. SFCDiagram

O *SFCDiagram* descreve um diagrama SFC no transcritor. Assim como o *LDiagram,* a sua criação foi baseada na linguagem PLCOpen XML. A sua composição e subclasses pode ser observada na Figura 7.8.

Como nas outras classes, sua denominação em inglês foi preservada de modo a facilitar o mapeamento da escrita do objeto para um arquivo PLCOpen XML.

As suas subclasses são:

- selectionConvergence
- selectionDivergence
- SFC_Step
- actionBlock
- action
- Transition
- transitionCondition

Um *SFCDiagram* pode possuir diversos objetos das classes *SFC_Step*, *Transition*, *selectionConvergence* e *selectionDivergence*. Eles representam os elementos básicos do SFC. Uma *Transition* é a representação de uma *transição*. O *step* representa o elemento homônimo. A *selectionConvergence* e *selectionDivergence* são elementos auxiliares que possuem a unificação de diferentes ramos do SFC ou a ramificação de apenas uma linha em duas ou mais.

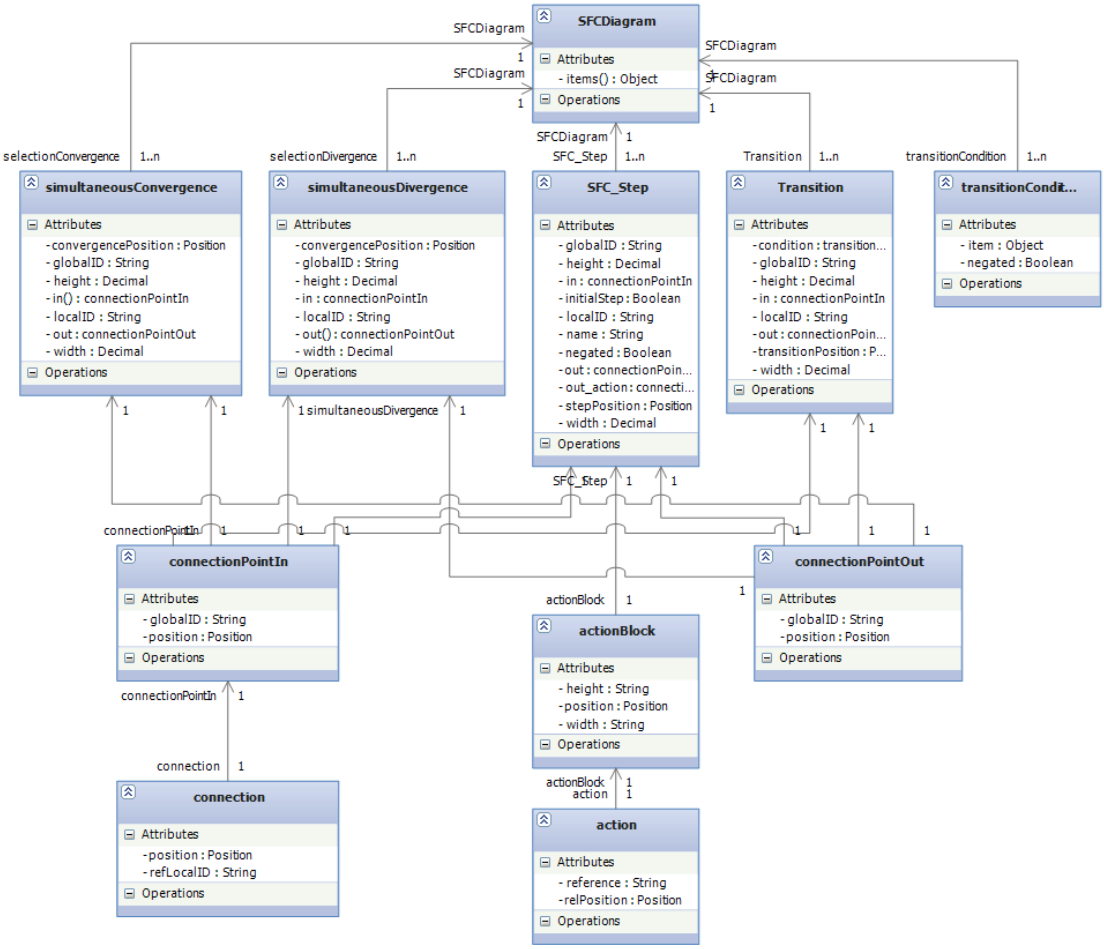


Figura 7.8 A classe SFCDiagram e suas subclasses

Cada *Transition* pode possuir uma *transitionCondition*, que é um conjunto definido de parâmetros que devem ser obedecidos para que a *transição* ocorra.

Cada *SFC_Step* possui um *actionBlock*, que por sua vez possui um *action*. Dentro deste objeto é possível encontrar o atributo *reference*, que indica qual variável de saída está associada àquele *step*.

Assim como na classe *LDiagram*, todos os elementos possuem *connectionPointIn* e *connectionPointOut*. Dessa maneira é possível ao programa estabelecer as ligações entre os diferentes objetos do *SFCDiagram*. Além disso, a conexão entre dois elementos diferentes no gráfico é realizada por meio do objeto *connection*.

7.4.3. O Structured Text

Por ser uma linguagem textual, o ST é representado dentro do programa como uma *String*. Assim sendo, não há uma classe dedicada exclusivamente para esta linguagem.

7.4.4. Interface

A interface do transcritor pode ser dividida em 4 grandes partes:

- Características da rede
- Diagrama de simulação da rede
- Console de simulação da rede
- Controles para a transcrição

Além desses elementos existe uma barra de menu na seção superior, que dá acesso às funções de abertura, escrita e fechamento de arquivos.

Com uma rede de Petri aberta o usuário pode verificar o seu nome e número de lugares e transições, para validação, na seção de características da rede.

O diagrama de simulação de rede de Petri é a visualização gráfica da rede carregada. Esta imagem é criada com base nas informações gráficas presentes no arquivo PNML carregado.

No console de simulação da rede de Petri é possível definir o estado de cada variável de entrada da rede em *true* ou *false*. Isso é feito por meio de caixas de escolha (*comboBoxes*) e de botões dedicados (*radioButtons*).

Para se definir o estado de uma variável, escolhe-se a mesma em uma das caixas de escolha e ativa-se o botão que contém o estado correspondente. Isso pode ser feito em duas caixas diferentes, de modo a agilizar o processo de simulação.

Uma vez definidos os estados desejados das variáveis o usuário pode clicar em “Dar passo”. Este botão realiza o procedimento de movimentação das *marcas* na rede de Petri.

Por último, a seção de controles de transcrição possui os comandos para realizar a transcrição da rede de Petri para as três linguagens de saída: o LD, SFC e ST.

Uma vez realizada a transcrição, o usuário recebe a opção de salvar o arquivo gerado, por meio do botão “Salvar em XML”.

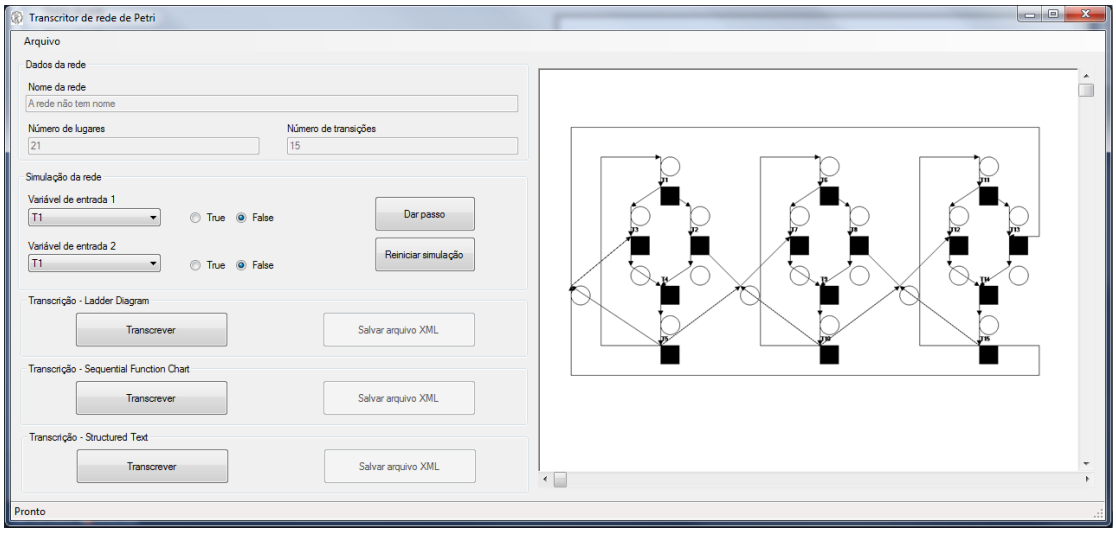


Figura 7.9 Imagem da interface com rede ilustrativa exibida

7.4.5. leitorDePNML

Esta classe tem como intuito implementar a leitura do arquivo de extensão “.PNML” e mapeá-lo como um objeto da classe *Rede.* Sua representação no diagrama de classes pode ser visto na Figura 7.10.

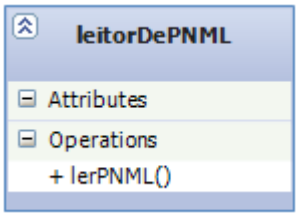


Figura 7.10 A classe leitorDePNML

Dado que a PNML é uma linguagem XML, é possível usar o ferramental dedicado dessa linguagem, não sendo necessária a implementação de um *parser*[5] exclusivo.

A Microsoft disponibiliza uma classe de manipulação de XML denominada *XML Serializer*. Esta classe possui métodos de leitura (*deserializing*) e escrita (*serializing*) de arquivos XML para instâncias de classes criadas pelo usuário. Portanto, ao criar classes com base nas

[5] *Parsing* é o procedimento de análise de um texto para determinar a sua estrutura gramatical em respeito a uma gramática formal de programação. No caso, a análise de um arquivo XML para determinar a sua estrutura em classes de Visual Basic.

linguagens XML equivalentes é possível realizar a leitura e escrita de maneira imediata.

Com isso, o método *lerPNML* abre uma caixa de diálogo, na qual o usuário pode acessar a pasta que contém o arquivo desejado, e repassa o mesmo para o *XML Deserializer*. Este realiza o processo de *parsing*, criando um objeto do tipo *Rede*. Este objeto é então armazenado na interface, de onde pode ser utilizado pelo *Transcritor* ou *Simulador*.

7.4.6. Transcritor

A classe transcritor representa o funcionamento do algoritmo de transcrição no programa. A instância do *Transcritor* não possui atributos nem propriedades. Sua representação da classe no diagrama de classes pode ser vista na Figura 7.11.

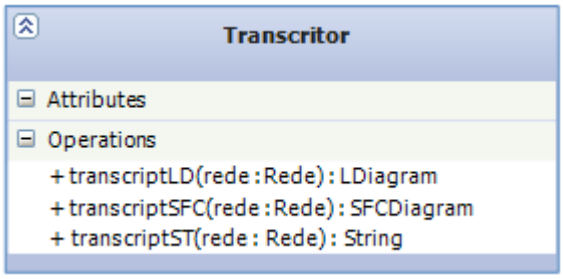


Figura 7.11 A classe Transcritor

Os três métodos da classe *Transcritor* representam os três processos de transcrição da rede de Petri, um para LD, outro para SFC e o terceiro para ST. Todos recebem um objeto do tipo *Rede* e retornam um objeto do tipo *LDiagram, SFCDiagram* ou *String*. Nenhum outro parâmetro de entrada é necessário.

7.4.7. escritorDeXml

A classe *escritorDeXML* é similar à classe *Transcritor*, não possuindo atributos ou propriedades. Ela possui três métodos distintos, utilizados para gerar o arquivo XML de saída do programa.

Os métodos tomam como entrada os objetos que representam cada uma das linguagens: LD, SFC e ST, para então abrir uma caixa de diálogo que permite ao usuário decidir o local para o arquivo “.xml” ser salvo. Estes métodos não retornam nenhum valor.

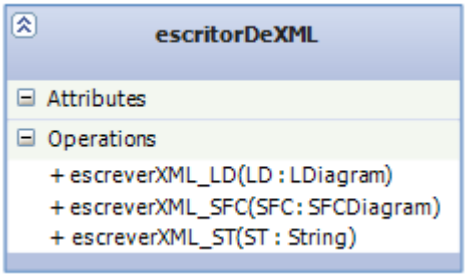


Figura 7.12 A classe escritorDeXML

8. Exemplo de transcrição

8.1. O sistema representado

Para exemplificar a utilização do transcritor como ferramenta de programação é necessária a apresentação de um sistema a eventos discretos a ser controlado.

Para o exemplo foi escolhido um misturador, baseado no exemplo proposto em (MIYAGI, 1996).

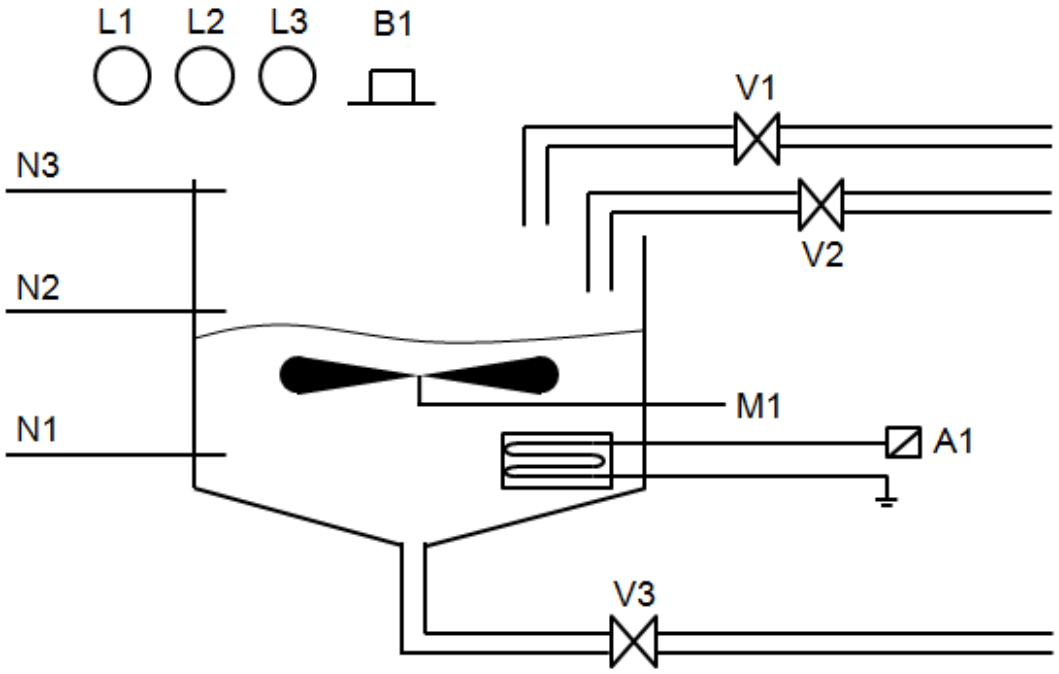


Figura 8.1 O exemplo do sistema a ser controlado

8.1.1. A estrutura física

O exemplo pode ser visto na Figura 8.1, e representa um misturador, que pode estar presente em diversos processos da indústria química.

O misturador proposto possui duas entradas de líquido controladas por meio das válvulas V1 e V2, pelas quais entram respectivamente o solvente e o soluto. Além disso possui a válvula V3, pela qual a mistura final é escoada.

O processo de mistura ainda precisa de um aquecedor que é ativado pelo relê A1 e um misturador, que por sua vez é ativado pelo relê M1.

Para as detecções de nível do líquido dentro do reservatório há três sensores: N1, N2 e N3.

Para o monitoramento do processo há três lâmpadas, L1, L2 e L3. A lâmpada L1 denota que o sistema está pronto, aguardando o comando de

início. A lâmpada L2 indica que o reservatório está se enchendo e a lâmpada L3 indica que o mesmo está esvaziando.

Há por último o botão B1, utilizado para inicializar o processo.

8.1.2. O processo de mistura

Uma vez inicializado, o processo deve ocorrer da seguinte maneira: a válvula V1 deve se abrir para permitir a entrada do solvente. Ao mesmo tempo deve ser ativado o aquecedor A1 para garantir a temperatura constante regulada durante o processo. A lâmpada L2, indicadora de “reservatório enchendo”, também deve ser ligada.

O nível de solvente deve subir até atingir o nível N2, quando então a válvula V1 se fecha e a válvula V2 se abre, permitindo a entrada do soluto. Concomitantemente à abertura da válvula é acionado o misturador M1, para garantir a homogeneidade da mistura.

Quando o nível de mistura chegar até o sensor N3, a válvula V2 deve ser fechada e o misturador M1 e o aquecedor A1 devem ser desligado. Além disso a lâmpada L2 deve ser desligada. A partir deste momento inicia-se o processo de esvaziamento, por meio da abertura da válvula V3. Para o monitoramento do processo é preciso ligar a lâmpada L3, indicadora de “reservatório esvaziando”.

Por fim, ao se atingir o nível mínimo N1, a válvula V3 deve se fechar, a lâmpada L3 ser apagada e o processo retornado ao seu estado inicial de prontidão. Este estado é representado pela lâmpada L1, que deve ser ligada.

8.2. A rede de Petri a ser transcrita

Para controlar o sistema proposto foi criada a rede vista na Figura 8.2. Nela, como previsto em uma SIPN, as variáveis de entrada foram mapeadas nas *transições* e as variáveis de saída foram mapeadas nos *lugares*.

Portanto, cada *lugar* possui um nome que corresponde à variável de saída associada, e o mesmo é válido para as *transições*, cujos nomes são variáveis de entrada.

A rede foi criada no NetLab e posteriormente carregada no transcritor, produzindo a imagem observada na Figura 8.2.

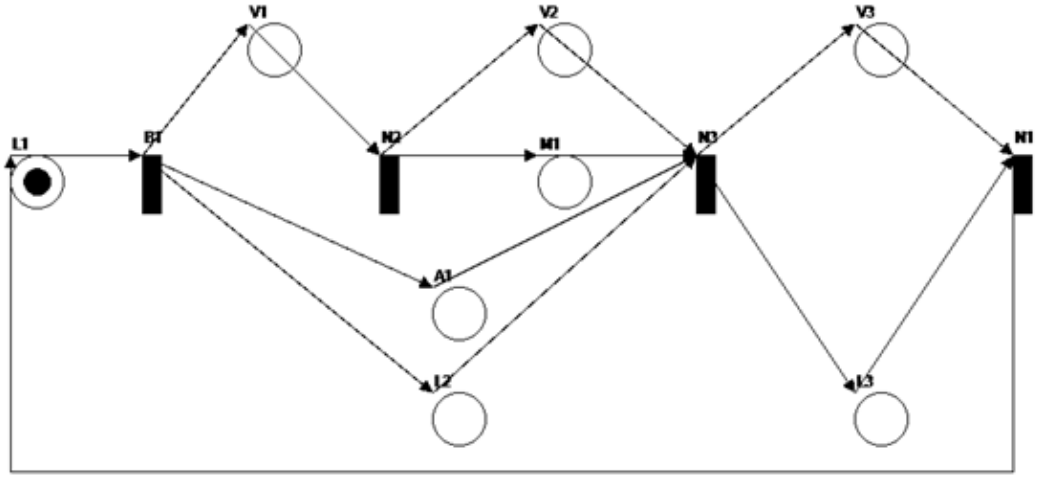


Figura 8.2 A rede de Petri utilizada no exemplo, como vista no transcritor

8.3. O LD resultante

Primeiramente a rede de Petri foi aberta no transcritor. Em seguida, foi transcrita para LD. Para a visualização, o arquivo XML gerado foi aberto no Beremiz.

Como previsto no algoritmo, o programa é dividido em três blocos, mais uma seção para a definição das condições iniciais.

8.3.1. A habilitação das transições

O primeiro bloco transcrito equivale à *habilitação* de cada *transição* de acordo com as *marcas* nos *lugares* que a antecedem e procedem, além da(s) variável(is) externa(s) associada(s) a ela. Essa *habilitação* se dá por meio de um *rung* por *transição*. Portanto para as quatro *transições* da rede exemplo é gerado um bloco de LD com quatro *rungs*, que pode ser observado na Figura 8.3.

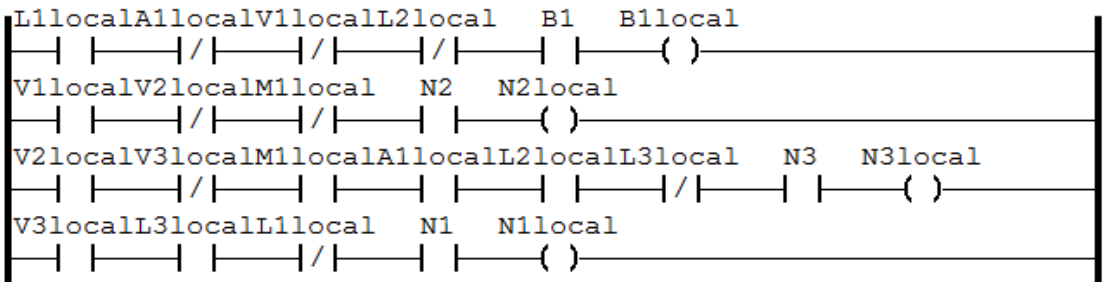


Figura 8.3 Bloco de habilitação das transições no LD

É possível observar que os contatos que representam os *lugares* posteriores à *transição* são negados, denotando que a *transição* somente está habilitada caso não haja *marcas* nesses *lugares*.

8.3.2. O fluxo de marcas

O próximo bloco do LD é responsável por representar a movimentação das *marcas* na rede de Petri. Portanto, para cada *transição* existem tantos *rungs* quantos forem os *arcos* conectados a ela. Para *arcos* de saída existem *rungs* com bobinas de *set* e para os *arcos* de entrada existem *rungs* com bobinas de *reset*. O resultado deste algoritmo pode ser visto na Figura 8.4.

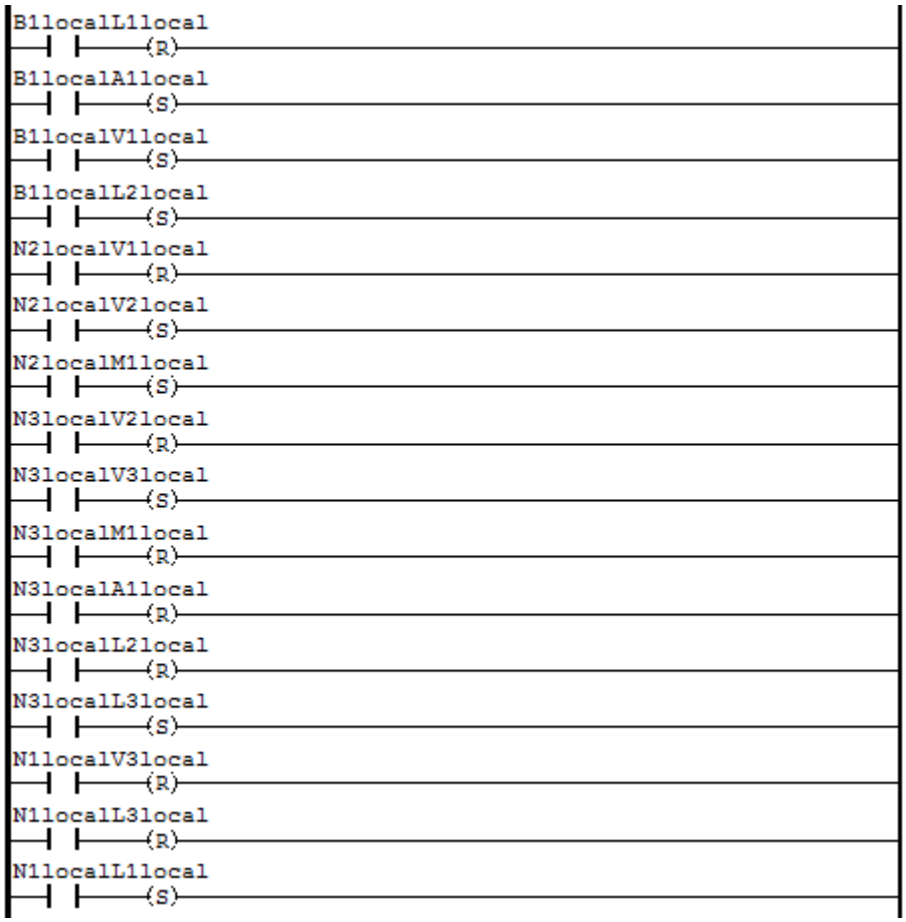


Figura 8.4 Bloco de fluxo das marcas no LD

8.3.3. A habilitação das saídas

O último bloco de execução criado no LD é utilizado para se definir o estado de cada variável de saída de acordo com a existência de uma *marca* no *lugar* correspondente. Isso é realizado criando-se um *rung* por *lugar*, com uma bobina representando a variável de saída. O resultado pode ser observado na Figura 8.5.

Figura 8.5 Bloco de habilitação das variáveis de saída no LD

8.3.4. A criação da condição inicial

Por último é preciso criar no LD a condição inicial da execução do programa. Isso é realizado por meio de um *rung* que possui contatos negados associados a cada um dos *lugares* da rede, e bobinas de *set* associadas aos *lugares* que possuem *marcas* na condição inicial.

Dessa maneira, na primeira varredura que o CLP realizará sobre o LD as bobinas do *rung* criarão *marcas* nos *lugares* adequados.

Como na rede exemplo apenas um dos *lugares* possui *marcação* inicial, há apenas uma bobina de *set*. O resultado pode ser observado na Figura 8.6.

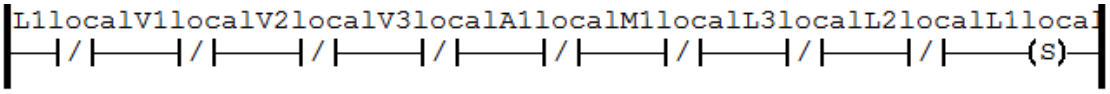


Figura 8.6 Bloco de inicialização das marcações iniciais

8.4. O SFC resultante

Assim como para o exemplo do LD, a rede de Petri foi aberta no transcritor, e a opção de transcrição para SFC foi selecionada. Em seguida, o arquivo PLCOpen XML gerado foi aberto no Beremiz para visualização.

Conforme previsto no método de transcrição, há semelhança entre o SFC de saída e a rede de Petri de entrada. Observando a Figura 8.7, percebe-se que cada elemento da rede de Petri foi convertido no elemento estrutural

equivalente do diagrama SFC, que o posicionamento de cada um desses elementos convertidos foi feito a partir da transposição da posição do elemento equivalente na rede de Petri e que o fluxo se dá de cima para baixo, como era de se esperar uma vez que a rede foi construída seguindo o cuidado propostos pelo relatório com relação ao fluxo da esquerda para a direita.

Outro ponto a ser observado no exemplo é como ficaram as variáveis após a transcrição. O nome de cada *passo* ficou como o nome do *lugar* correspondente acrescido da palavra 'Local'. Como o nome do *lugar* na rede de Petri era na verdade a variável de saída, a mesma se tornou uma *ação* do *bloco de ação* com o qualificador N, que significa que enquanto o *lugar* estiver ativo, também estará a variável. Em paralelo a isso, pode-se observar que o nome da *transição* da Rede de Petri, que era na verdade uma variável de entrada, resultou na *receptividade* da *transição*, disposta sobre a forma de texto estruturado, o ST.

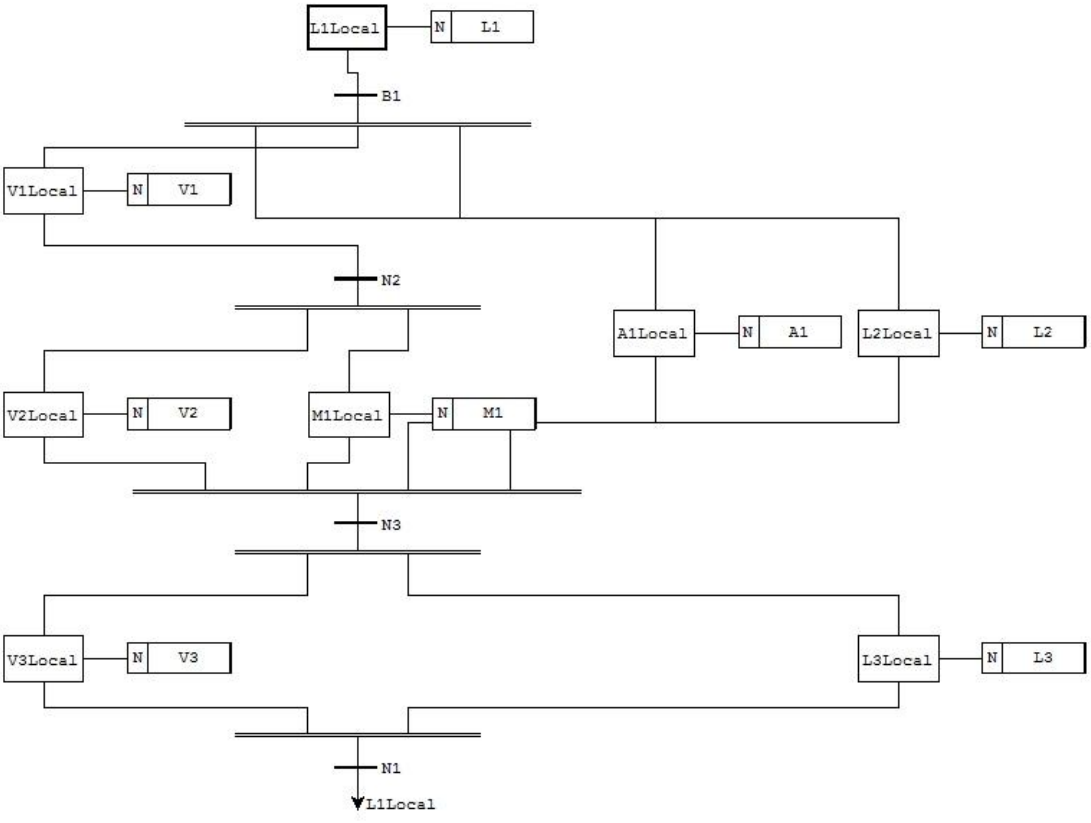


Figura 8.7 O diagrama SFC gerado pelo transcritor

No arquivo XML gerado pelo transcritor pode-se encontrar o trecho de código mostrado na Figura 8.8. A condição, representada pelo elemento

condition do XML corresponde ao que chamamos de *receptividade* anteriormente. A vantagem de essa receptividade poder ser escrita sob a forma de ST, ou ainda qualquer outra linguagem padronizada pela PLCOpen, é a flexibilidade de encadeamento de programas e a simplificação da codificação. Na Figura 8.8, apenas uma variável N1 foi incluída no ST. Tal variável foi previamente inicializada na interface como *Booleana*.

```
<condition>
  <inline name="">
    <ST>
      <![CDATA[N1]]>
    </ST>
  </inline>
</condition>
```

Figura 8.8 Trecho do arquivo XML gerado na transcrição para SFC

No final do exemplo, foi necessário a inclusão do objeto *Jump* que permite que o ciclo recomece após seu término. A Rede de Petri que foi usada no exemplo, há um arco que liga a última transição (N1) ao primeiro lugar (L1), que é definido assim por ser o único que contém *marca* inicial. Após feita a transcrição, ao invés de existir uma conexão ligando os objetos equivalentes do SFC, a transição N1 e o *passo inicial*, respectivamente, é utilizado o *Jump* que indica o próximo passo.

8.5. O ST resultante

Para o exemplo de transcrição para ST, a rede de Petri foi aberta no transcritor e a opção de transcrição para ST foi selecionada. Em seguida, o arquivo PLCOpen XML foi importado para o CoDeSys, ambiente de desenvolvimento no qual sua sintaxe foi verificada. Na Figura 8.9, Figura 8.10, Figura 8.11 e Figura 8.12 o código é observado como mostrado na interface do CoDeSys.

8.5.1. A habilitação das transições

Como previsto no algoritmo apresentado, a primeira seção do ST transcrito se assemelha em função ao primeiro bloco da transcrição para o formato do LD, sendo responsável pela habilitação das *transições*. O código pode ser observado na Figura 8.9.

```
B1Local := L1Local AND  NOT A1Local AND  NOT V1Local AND  NOT L2Local AND B1;
N2Local := V1Local AND  NOT V2Local AND  NOT M1Local AND N2;
N3Local := V2Local AND  NOT V3Local AND M1Local AND A1Local AND L2Local AND  NOT L3Local AND N3;
N1Local := V3Local AND L3Local AND  NOT L1Local AND N1;
```

Figura 8.9 A habilitação das transições no ST gerado

8.5.2. O fluxo de marcas

A segunda seção do código ST gerado realiza a movimentação das *marcas* na rede e pode ser observada na Figura 8.10.

```
IF B1Local = TRUE THEN
L1Local := FALSE;
END_IF
IF B1Local = TRUE THEN
A1Local := TRUE;
END_IF
IF B1Local = TRUE THEN
V1Local := TRUE;
END_IF
IF B1Local = TRUE THEN
L2Local := TRUE;
END_IF
IF N2Local = TRUE THEN
V1Local := FALSE;
END_IF
IF N2Local = TRUE THEN
V2Local := TRUE;
END_IF
IF N2Local = TRUE THEN
M1Local := TRUE;
END_IF
IF N3Local = TRUE THEN
V2Local := FALSE;
END_IF
IF N3Local = TRUE THEN
V3Local := TRUE;
END_IF
IF N3Local = TRUE THEN
M1Local := FALSE;
END_IF
IF N3Local = TRUE THEN
A1Local := FALSE;
END_IF
IF N3Local = TRUE THEN
L2Local := FALSE;
END_IF
IF N3Local = TRUE THEN
L3Local := TRUE;
END_IF
IF N1Local = TRUE THEN
V3Local := FALSE;
END_IF
IF N1Local = TRUE THEN
L3Local := FALSE;
END_IF
IF N1Local = TRUE THEN
L1Local := TRUE;
END_IF
```

Figura 8.10 O fluxo de marcas no ST gerado

8.5.3. A habilitação das saídas

A terceira seção do código gerado realiza a habilitação das saídas de acordo com o estado das variáveis locais do ST. O resultado pode ser visto na Figura 8.11.

```
L1 := L1Local;
V1 := V1Local;
V2 := V2Local;
V3 := V3Local;
A1 := A1Local;
M1 := M1Local;
L3 := L3Local;
L2 := L2Local;
```

Figura 8.11 A habilitação das saídas no ST

8.5.4. A criação da condição inicial

Para realizar a criação da condição inicial foi criada uma declaração de IF que coloca uma *marca* no *lugar* inicial. Tal procedimento é visto na Figura 8.12.

```
IF NOT L1Local  AND NOT V1Local  AND NOT V2Local  AND NOT V3Local  AND NOT A1Local
AND NOT M1Local  AND NOT L3Local  AND NOT L2Local THEN
L1Local := TRUE;
END_IF
```

Figura 8.12 A criação da condição inicial no ST

9. Conclusão

A motivação original deste trabalho era a criação de um fluxo de trabalho integrado e automatizado para a programação de CLPs, através da transcrição de modelos desenhados em rede de Petri para linguagens da IEC 61131-3.

Com o transcritor proposto criado com sucesso, o usuário pode utilizar a modelagem por meio de rede de Petri para a programação direta de um CLP, garantindo maior liberdade e flexibilidade ao estudo e implementação da programação do controle de sistemas a eventos discretos.

A rede de Petri pode ser utilizada, então, como uma linguagem direta de programação de CLPs, eliminando-se a necessidade de passos adicionais manuais de transcrição, que anteriormente tomavam tempo adicional do desenvolvedor.

Beneficiam-se com isso tanto os estudantes e professores de engenharia, que podem utilizar a ferramenta para testar modelos teóricos desenvolvidos e aplicá-los em laboratórios experimentais, quanto profissionais da indústria, que podem criar a programação de sistemas complexos por meio da rede de Petri e, posteriormente, aplicar o modelo desenvolvido a plantas reais.

As linguagens utilizadas no transcritor, a PNML e a PLCOpen XML, são demonstrações de uma tendência de padronização do controle de manufatura, que começou com a norma IEC 61131-3. Assim, a possibilidade de utilização do programa criado tende a crescer com o tempo. Adicionalmente, eventuais mudanças nas linguagens utilizadas podem ser aplicadas ao programa para continuar e melhorar a sua utilização.

Por fim, o trabalho obteve êxito em todas as suas propostas, contribuindo assim para a simplificação do acesso à programação de CLPs.

ANEXO A: XML (eXtended Markup Language)

A 1. Introdução

Antes de se iniciar qualquer aprofundamento na linguagem XML que faz parte do escopo deste trabalho, cabe uma breve apresentação de algumas outras normas e linguagens, todas elas visando a padronização.

A 1 i. Unicode

O *Unicode Consortium* é uma organização, sem fins lucrativos, voltada para o desenvolvimento, manutenção e promoção da internacionalização dos padrões de programas e dados[6] particularmente o Padrão Unicode[7] que especifica a representação de textos (THE UNICODE CONSORTIUM, 2010).

O Padrão Unicode é um padrão universal que permite aos computadores representar e manipular uma grande gama de caracteres de diferentes sistemas de escrita. Publicado no livro *The Unicode Standard* (THE UNICODE CONSORTIUM, 2006) o código em si consiste em mais de 107 mil caracteres, além de propriedades, diagramas e até regras de ordenação e renderização. Essa padronização universal permite o livre intercâmbio de programas o que é muito importante para o avanço tecnológico e avanço de conhecimentos.

A 1 ii. SGML (Standard Generalized Markup Language)

A *SGML* é uma metalinguagem capaz de definir outras linguagens de marcação. Ela foi desenvolvida para permitir a interpretação de informações por computadores e máquinas. A SGML é definida em uma norma ISO: "*ISO 8879:1986 Information processing--Text and office systems--Standard Generalized Markup Language (SGML)*"(ISO, 2010).

A 2. Extended Markup Language

A linguagem nomeada em inglês por *eXtended Markup Language*, cuja abreviação é XML, tem forte importância na realização desse trabalho por ser uma linguagem altamente adaptável e já disseminada e aceita pela

[6] Em inglês: *Softwares and datas*
[7] Em inglês: *Unicode Standard*

comunidade técnica. Em outras palavras, essa característica favorece a aplicabilidade da proposta desse projeto.

A 2 i. Recomendação World Wide Web Consortium

O documento da W3C (*World Wide Web Consortium*) que recomenda a linguagem XML (*eXtended Markup Language*) para determinados fins especifica que a linguagem XML é uma sintaxe apropriada criada dentro da linguagem SGML (como um subtipo da mesma). É a própria W3C que é responsável por avaliar as necessidades e determinar as diretrizes das mudanças nos padrões, realizando a manutenção da especificação e norma.

A 2 ii. Objetivos da linguagem

Em sua idealização e criação, em 1996, por um grupo dedicado para tal fim e sob a direção da W3C, a linguagem conhecida com XML tinha os principais objetivos:

- Deve ser diretamente usada para Internet;
- Deve suportar uma grande variedade de aplicações;
- Deve ser compatível com SGML;
- Facilidade para escrever o código de um programa que processe documentos em XML;
- O número de características opcionais na XML tem que ser o mínimo, idealmente zero;
- Deve ter uma sintaxe simples e clara que garanta a legibilidade do código por seres humanos;
- Facilidade e velocidade no desenvolvimento do código em XML;
- Concisão e formalidade no desenvolvimento do código em XML;
- Facilidade na criação de documentos XML;
- Concisão nos marcadores característicos da linguagem não é importante.

A especificação garante que a associação com outros padrões, principalmente o Padrão Unicode, aumenta ainda mais a universalidade da linguagem XML (W3C, 2008).

A 2 iii. Estrutura

Segundo a W3C (2008), um documento XML tem tanto a estrutura lógica quanto a estrutura física, que é o código. A estrutura física é composta por unidades chamadas de entidades. As relações entre entidades, as declarações, os elementos, as instruções e os procedimentos tem que estar dispostos e organizados segundo a norma compondo um documento corretamente formado[8] na estrutura lógica e física. São restrições e sintaxes de linguagem que garantem essa formação.

A principal característica da estrutura da linguagem XML é que ela é instanciada por marcadores, que definem o início e o fim dos elementos. Essa foi a herança da linguagem SGML que a antecedeu.

Um documento XML corretamente formado, de maneira simplificada, tem que compreender cada um de seus elementos dentro de seus respectivos marcadores obedecendo a estrutura lógica da norma. Há um elemento primário chamado de elemento raiz no qual todos os outros elementos e entidades estão contidos.

Por definição, todo o texto contido no arquivo que não é delimitado por um marcador constitui os caracteres de dados do documento. Os caracteres principais da linguagem XML são [<] e [>], que são utilizados para referenciar marcadores, comentários ou instruções e [”] e [&], que são utilizados para outros fins. Quaisquer caracteres destes que precisem ser utilizados para outra finalidade (senão as propostas pela linguagem) devem ser escapados[9] pelos caracteres de escape, que variam de situação para a situação segundo as especificações da norma.(W3C, 2008).

A 2 iv. Aplicabilidade a banco de dados

Uma das principais vantagens da linguagem XML é a sua aplicabilidade a estruturas de dados muito úteis à computação. Tal característica é possível devido à existência de ferramentas relevantes de estruturação de dados, como registros, listas e árvores, que possibilitam a transcrição e armazenagem de arquivos de dados no formato XML.

---

[8] Em inglês: *well-formed*

[9] Termo utilizado para indicar que o caractere reservado pela linguagem não está sendo usado para esse fim

Para estruturas lógicas com elevado nível de complexidade essas ferramentas permitem maior eficiência e mais simplicidade na síntese da informação. No caso da Rede de Petri, que podem alcançar níveis elevados de complexidade, faz-se necessária uma linguagem eficiente para a transcrição. Isso, inclusive, é feito em etapas com a identificação de padrões na Rede de Petri e realizando processos de modularização (BARROS, 2006).

Bibliografia

ABEL, D. Petri Net tool Netlab (Windows). **INSTITUT FÜR REGELUNGSTECHNIK WEBSITE**, 2008. Disponivel em: <http://www.irt.rwth-aachen.de/en/downloads/petri-net-tool-netlab.html>. Acesso em: 14 Junho 2010.

AUTOMATION ALLIANCE. Automation Alliance homepage. **Automation Alliance website**, 2010. Disponivel em: <http://www.automation-alliance.com/>. Acesso em: 02 Junho 2010.

BARROS, J. P. M. P. R. E. **Modularidade em Redes de Petri**. Universidade Nova de Lisboa. Lisboa. 2006.

FREY, G. **SIPN, Hierarchical SIPN and Extensions**. Kaiserslautern: Institue of Automatic Control - University of Kaiserslautern, 2001.

GOMES, L. et al. **The Input-Output Place-Transition Petri Net Class**. IEEE International Conference on Industrial Informatics. Vienna: [s.n.]. 2007. p. 509-514.

ISO. International Organization For Standarzation. **Information processing -- Text and office systems -- Standard Generalized Markup Language (SGML)**, 2010. Disponivel em: <http://www.iso.org/iso/catalogue_detail.htm?csnumber=16387>. Acesso em: 12 Abril 2010.

ISO. International Organization For Standarzation. **Information processing -- Text and office systems -- Standard Generalized Markup Language (SGML)**. Disponivel em: <http://www.iso.org/iso/catalogue_detail.htm?csnumber=16387>. Acesso em: 12 Abril 2010.

KARL-HEINZ, J.; TIEGELKAMP, M. **Programming industrial automation systems:** concepts and programming languages, requirements for programming systems, aids to decision-making tools. 1st Edition. ed. Heidelberg, Germany: Springer, 2001.

KINDLER, E.; WEBER, M. The Petri Net Markup Language. **Lecture Notes in Computer Science**, Berlin, 2003. 124-144.

LUCAS, M. R.; TILBURY, D. M. Methods of measuring the size and complexity of PLC programs in different logic control design methodologies.

The International Journal of Advanced Manufacturing Technology, London, p. 436-447, Setembro 2005. ISSN 1433-3015.

MICROSOFT CORPORATION. XML Serialization in the.NET Framework. **MSDN Library**, 2003. Disponivel em: <http://msdn.microsoft.com/en-us/library/ms950721.aspx>. Acesso em: 10 Junho 2010.

MIYAGI, P. E. **Controle Programável:** fundamentos do controle de sistemas a eventos discretos. 1. ed. São Paulo: Editora Blucher, 1996.

MORORÓ, B. O. **Modelagem Sistêmica do Processo de Melhoria Contínua de Processos Industriais Utilizando o Método Seis Sigma e Redes de Petri**. São Paulo: Escola Politécnica da USP, 2008.

OMG. Unified Modeling Language Specification, version 1.5. **Object Modeling Group**, 2003. Disponivel em: <http://www.omg.org/cgi-bin/doc?formal/03-03-01>. Acesso em: 13 Abril 2010.

OMG. Introduction to OMG's. **OMG**, 2005. Disponivel em: <http://www.omg.org/gettingstarted/what_is_uml.htm>. Acesso em: 10 set. 2010.

PETRI, C. A. Scholarpedia. **Petri Net**, 2008. Disponivel em: <http://www.scholarpedia.org/article/Petri_net>. Acesso em: 25 Março 2010.

PLCOPEN. Status T6C. **PLCOpen**, 2008. Disponivel em: <http://www.plcopen.org/pages/whats_new/tc6/status.htm>. Acesso em: 13 Abril 2010.

PNML ORGANIZATION. PNML Grammar, version 2009. **PNML.org**, 2010. Disponivel em: <http://www.pnml.org/version-2009/version-2009.php>. Acesso em: 14 Junho 2010.

RANDELL, B. A Begginer's Guide to the XML DOM. **MSDN Windows Developer Center**, 1999. Disponivel em: <http://msdn.microsoft.com/en-us/library/aa468547.aspx>. Acesso em: 2 out. 2010.

SANTOS FILHO, D. J.; MIYAGI, P. E. **Proposta de uma ferramenta automática de programação de CPS a partir de modelos MFG**. COBEM - Congresso Brasileiro de Engenharia Mecânica. Bauru: [s.n.]. 1997.

SARMENTO, C. A. **Modelagem De Programas E Sua Verificação Para Controladores Programáveis**. São Paulo: Ed. Rev., 2008.

SMART SOFTWARE SOLUTIONS. CoDeSys Homepage. **Smart Software Solutions Website**, 2010. Disponivel em: <http://www.3s-software.com/index.shtml?homepage>. Acesso em: 14 Junho 2010.

THAYER, T. Speaking in Tongues: Understanding the IEC 61131-3 Programming Languages. **Control Engineering**, Chicago, 30 January 2009.

THE UNICODE CONSORTIUM. The Unicode Standard: Version 5.0. In: CONSORTIUM, T. U. **The Unicode Standard:** Version 5.0. 5. ed. [S.l.]: Addison-Wesley Professional, 2006. ISBN 0321480910.

THE UNICODE CONSORTIUM. Sobre a Organização: The Unicode Consortium. **The Unicode Consortium**, 2010. Disponivel em: <http://www.unicode.org/>. Acesso em: 12 Abril 2010.

TISSERANT, E.; BESSARD, L.; DE SOUSA, M. **An Open Source IEC61131-3 Integrated Development Environment**. INDIN - International Conference on Indutrial Informatics. Vienna: IEEE. 2007.

W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). **World Wide Web Consortium (W3C)**, 2008. Disponivel em: <http://www.w3.org/TR/REC-xml/>. Acesso em: 12 Abril 2010.

WANER, F. et al. **Modeling Software with Finite State Machines**. Nova Iorque: Auerbach Publications, 2006.

WEBER, M. et al. The Petri Net Markup Language: concepts, technology and tools. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, Berlin, Germany, 2003. 483-505.

WITSCH, D.; VOGEL-HEUSER, B. **Close integration between UML and IEC 61131-3:** New possibilities through object-oriented extensions. IEEE Conference on Emerging Technologies and Factory Automation. Mallorca: IEEE. 2009. p. Article 5347155.