Gustavo Ribeiro Alves

Sistema Computacional de Representação e Manipulação de Redes de Petri

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Mecânica.

São Paulo
2006

Gustavo Ribeiro Alves

Sistema Computacional de Representação e Manipulação de Redes de Petri

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Mecânica.

Área de concentração:
Automação e Controle

Orientador:
Prof. Dr. Diolino José dos Santos Filho

São Paulo
2006

Ao leitor, que consumira uma fração do tempo que gastei neste no mesmo.

Agradecimentos

A minha família, que foi obrigada a conviver com um membro cada vez mais omisso e distante;

A minha namorada Fernanda Zeidan, que suportou todas as minhas mudanças de humor, desesperança, descrença e cansaço;

A todos aqueles que ajudaram na revisão deste texto, em especial[1]: Eng. Rodrigo Rizzi Starr, Jornalista Renata Miranda, Profa. Dra. Cristina Toshie Motohashi Matsusaki, Prof. Dr. Júlio Arakaki, Prof. André Cesar Martins Cavalheiro;

A quem, antes de orientador considero um amigo: Prof. Dr. Diolino José dos Santos Filho;

Aos colegas de grupo de Pesquisa e trabalho que me deram suporte de mais maneiras de que é possível descrever. Especialmente àqueles que tiveram que alterar sua rotina de trabalho nas semanas que antecederam a concretização deste;

Aos professores da USP que nunca se recusarram a discutir temas pertinentes(ou não) ao trabalho;

A todos aqueles que tiveram que fazer sacrifícios em diversos níveis para que este texto pudesse ser concretizado: são tantos que corro o risco de omitir alguém se tentar elencá-los;

Aos meus alunos que, sem culpa nenhuma, aguentaram um professor omisso durante a realização deste[2];

A todos estes e aqueles que de alguma maneira me ajudaram a chegar até aqui: os meus sinceros agradecimentos e o meu muito obrigado.

[1] Os nomes não estão organizados em nenhuma ordem específica.

[2] A estes devo ainda minhas sinceras desculpas.

Resumo

A diferença entre a forma com que o desenvolvedor manipula uma rede de Petri em um computador e a forma com que ele as manipula fora do mesmo é grande o suficiente para permitir uma perda considerável de produtividade quando do desenvolvimento de softwares baseados em redes de Petri. Desta maneira propôs-se a criação de um sistema que reduzisse a diferença entre estas. Este sistema faz isso através de dois desacoplamentos: separação entre a interface de manipulação da rede(*(*interface de acesso) e da descrição da rede(*meta-rede*). O segundo desacoplamento é feito entre a interface de acesso e o método de armazenamento para permitir que uma mesma implementação do armazenamento de redes possa ser utilizada para todas as *meta-redes* possíveis. A linguagem de meta-rede proposta é suficiente para a representação de ampla gama de redes. Ela também é capaz de ser utilizada para a representação de grafos que necessitam de um conjunto de restrições nas conexões entre os nós.

Abstract

The difference between the way a developer manipulates a Petri net in a computer and the way he manipulates them outside it is great enought to allow a significant productivity loss when he wants to develop Petri net based softwares. The creation of a system that can reduce this difference is then proposed. This system accomplishes this with two decouplings. The separation between the net manipulation interface and the net description language is the first one. The second is done between the net manipulation interface and the net storage classes in such a way that it allows the same implementation for the net storage classes to be used in all the possible nets. The proposed net description language is able to represent a large number of nets. It is also capable to represent graphs that have a set of restrictions on the allowed edges.

Conteúdo

Lista de Figuras **ix**

Lista de Tabelas **x**

Lista de Abreviaturas e Siglas **xii**

1 Introdução **1**

1.1 Motivação 3

1.2 Objetivos 4

1.3 Estrutura do Texto 5

1.3.1 Notação de Conjuntos 5

2 Visão Geral e Método de Utilização **7**

2.1 Introdução 7

2.2 Componentes do Sistema 8

2.3 Método de Desenvolvimento 9

2.4 Responsabilidades dos Desenvolvedores 14

2.5 Considerações finais do capítulo 16

3 Linguagem de Representação da Meta-Rede **17**

3.1 Estrutura de redes de Petri (*RdP*'s) 17

3.2 Dinâmica de *RdP*'s 23

3.3 Meta-Rede 24

3.3.1 Representação dos Nós 25

3.3.2 Representação das Conexões 28

3.3.3 Representação dos Tipos 31

3.3.4 Representação da dinâmica 34

3.3.5 Representação da Meta-Rede e Validações 35

3.4 Exemplos de Aplicação 38

3.4.1 Representação de rede de Petri lugar-transição $(R^{L}/_{T})$ 38

3.4.2 Representação de rede de Petri arco-constante (RAC) 38

3.5 Conclusão 40

4 Interface de Acesso **43**

4.1 Inserção e deleção de nós e conexões 57

4.2 Manipulação de inscrições estáticas e dinâmicas 61

4.3 Simulação 64

4.4 Busca por elementos 65

4.5 Serialização 66

4.6 Considerações Finais 66

5 Conclusão **68**

Bibliografia **70**

Lista de Figuras

2.1 Relação entre os componentes do Sistema. 8

2.2 Definição da meta-rede e geração das classes 11

2.3 Exemplo de algoritmo utilizando a interface de acesso. 13

2.4 Exemplo de algoritmo utilizando diretamente uma implementação de armazenamento. 15

3.1 Representação gráfica de classes de cores (hexágonos) em uma rede RAC. . 29

3.2 Procedimento de validação da rede (Parte I). 36

3.3 Procedimento de validação da rede (Parte II). 37

4.1 Diagrama de estados para a rede. 56

4.2 Interface de acesso - inserção e deleção de nós e conexões; mudança de estado. 58

4.3 Exemplo de $R^{L}/_{T}$ 60

4.4 Interface de acesso - inserção e deleção de Inscrições. 61

Lista de Tabelas

1.2 Operadores sobre conjuntos. 6

1.1 Conjuntos numéricos e básicos. 6

3.1 Gramática da linguagem de representação de nós na meta-rede (formato Notação de Backus Naur("Backus Naur Form") (*BNF*)). 26

3.2 Representação de *lugar* em uma $R^{L}/_{T}$. 26

3.3 Representação de *lugar* em uma rede com capacidade. 28

3.4 Representação de *classe de cor* em uma *RAC*. 28

3.5 Gramática da linguagem de representação de conexões na meta-rede (formato *BNF*). 30

3.6 Representação de *arco* em uma rede com dois tipos de transição. 31

3.7 Gramática da linguagem de representação de tipos compostos na meta-rede (formato *BNF*). 31

3.8 Gramática da l inguagem de representação de expressões na meta-rede (formato *BNF*). 33

3.9 Gramática da linguagem de representação da dinâmica (formato *BNF*). . . 35

3.10 Gramática da linguagem de representação da meta-rede (formato *BNF*). . 35

3.11 Representação de $R^{L}/_{T}$. 39

3.12 Representação de *RAC*. 42

4.1 Conceito *Assignable*. 45

4.2 Conceito *Default Constructible*. 45

4.3 Conceito *Equality Comparable*. 46

4.4 Conceito *Container*. 48

4.5 Conceito *Associative Container*. 50

4.6 Conceito *Trivial Iterator*. 51

4.7 Conceito *Input Iterator*. 52

4.8 Conceito *Output Iterator*. 53

4.9 Conceito *Forward Iterator*. 54

4.10 Tipos expostos após a compilação da $R^{L}/_{T}$ (Cf. tabela 3.11). 54

4.11 Árvore de nomes de tipos expostos em uma rede. 55

4.12 Tipos expostos na RDPLT. 55

4.13 Métodos válidos em cada estado da rede. 57

4.14 Tentativa de inserção de conexão inválida. 58

4.15 Mensagem de erro produzida ao inserir conexão inválida. 58

4.16 Código para criação de uma $R^{L}/_{T}$. 60

4.17 Código para criação das inscrições em uma $R^{L}/_{T}$. 62

4.18 Conceito *IteratorPair*. 63

4.19 Código proposto para acesso a inscrições múltiplas. 64

Lista de Abreviaturas e Siglas

RdP	rede de Petri
$R^{L}/_{T}$	rede de Petri lugar-transição (Cf. Valk (2003b))
$R^{C}/_{E}$	rede de Petri condição-evento(Cf. Miyagi (1996))
R_{ST}	rede de Petri estocástica (Cf. Haas (2002, p. 17))
R^{R}_{ST}	rede de Petri estocástica restrita (Cf. Haas (2002, p. 64))
RAC	rede de Petri arco-constante (Cf. Girault e Valk (2003, p. 43))
RCL	rede de Petri colorida (Cf. Girault e Valk (2003, p. 45) e Lakos (1995))
BNF	Notação de Backus Naur("Backus Naur Form" ; cf. Neto (1987) e Donnely e Stallman (2005))
NS	Nassi-Shneiderman (Cf. Nassi e Shneiderman (1973))

1 Introdução

As redes de Petri (*RdP*'s) são grafos marcados bipartidos. Desde sua publicação original (PETRI, 1962 apud GIRAULT; VALK, 2003) elas têm encontrado diversas aplicações na modelagem de sistemas a eventos discretos.

Valk (2003b, p. 10-14) apresentou cinco princípios das *RdP*'s:

Dualidade Nas *RdP*'s existem dois conjuntos disjuntos de elementos: *elementos-P* e *elementos-T*. Entidades de um sistema, interpretadas como elementos passivos, são representadas na rede como *elementos-P*. Entidades de um sistema, interpretadas como elementos ativos, são representadas na rede como *elementos-T*.

Localidade O comportamento de uma transição (*elemento-T*) depende da sua *localidade*, que é definida como a totalidade de seus elementos de entrada e saída (pré e pós condições, lugares de entrada e saída, ...) juntamente com o elemento em si.

Paralelismo Transições que tem localidades disjuntas ocorrem de maneira independente (em paralelo).

Representação Gráfica *Elementos-P* são representados por símbolos gráficos arredondados (círculos,elipses,...). *Elementos-T* são representados por símbolos gráficos "pontudos" (retângulos, barras, ...). Arcos conectam cada *elemento-T* à sua localidade, que é um conjunto de *elementos-P*. Ainda podem existir inscrições, como por exemplo nomes, marcas, expressões, guardas.

Representação Algébrica Para cada representação gráfica existe uma representação

algébrica equivalente. Ela contém o conjunto de lugares, transições, arcos e informações adicionais, como por exemplo inscrições.

A representação algébrica faz com que sejam possíveis a criação de métodos formais de análise.

Uma aplicação clássica[1] é a modelagem de redes de comunicação. Uma vez que estas redes possuem facilidade na representação de múltiplas tarefas, bem como na representação do estado local do sistema. As redes de Petri podem ainda ser utilizadas para a especificação de protocolos de comunicação, resultando em representações suscintas e clara destes. (PETRI, 1962 apud GIRAULT; VALK, 2003).

Diversos artigos atuais versam sobre a modelagem, análise e síntese de softwares através do uso de redes de Petri como representação do fluxo de controle do mesmo. Elas se mostram especialmente úteis para a modelagem de sistemas multi-agentes e de programas paralelos (MOLDT; WIENBERG, 1997; XU et al., 2002; LOMAZOVA, 2002; USHER; JACKSON, 1998; GIRAULT; VALK, 2003; CARDOSO et al., 2004; GONÇALVES; BITTENCOURT, 2004), bem na representação de sistemas especialistas (WU; LEE, 1997).

Encontram uso, ainda, na modelagem e análise de fluxo de negócios ("workflow management"), logística (controle de movimentação de cargas, logística de produção, etc). (AALST; GRAAF, 2003; AALST; HEE, 1996).

As redes de Petri encontram também aplicações em sistemas de manufatura. Tais sistemas podem ser divididos em: subsistema físico (robôs, motores, esteiras, instalações) e subsistema de controle. (EZPELETA, 2003).

No subsistema físico, elas podem ser utilizadas para encontrar gargalos na produção, detectar a presença de condições que possam levar a "deadlocks", simular o tempo de utilização de máquinas, determinar a probabilidade da ocorrência de eventos na produção, etc. (NAKAMOTO, 2002; MIYAGI, 1996; HAAS, 2002; GIRAULT; VALK, 2003; DAVID; ALLA, 1994).

[1] Esta foi a aplicação original das redes de Petri.

Outros usos interessantes são encontrados no subsistema de controle, onde elas podem ser vistas tanto em técnicas de análise[2] quanto em aplicações na sua síntese. (FREY, 2000; FELDMANN et al., 1999b; FELDMANN et al., 1999a; HOLLOWAY; KROGH; GIUA, 1997; ARMELLINI; ALVES; COSTA, 2002).

Cabe ainda ressaltar que as redes de Petri encontram usos fora da engenharia, como por exemplo na representação de processos bioquímicos. (PINNEY; WESTHEAD; MCCONKEY, 2003).

Conforme demonstrado acima, as *RdP*'s encontram usos diversos na engenharia e nas ciências. No entanto sua representação computacional não é trivial.

1.1 Motivação

A manipulação de uma rede de Petri (*RdP*) em um computador requer um amplo conhecimento de estruturas de dados e de seus algoritmos. Cormen et al. (2001) apresentam uma discussão detalhada sobre estruturas e algoritmos.

Este conhecimento não é comum aos pesquisadores de métodos de modelagem em *RdP*'s. Da mesma maneira, não é comum aos pesquisadores de estururas de dados um conhecimento aprofundado de *RdP*. Desta maneira, a construção de ferramentas computacionais que utilizem *RdP* não é uma tarefa simples.

Existem diversas técnicas que utilizam *RdP*'s desenvolvidas pela academia que encontram usos no meio industrial, entretanto um número muito maior destas não os encontra.

Conjectura-se que uma barreira à utilização de determinada técnica no meio industrial é a falta de ferramenta computacional que a suporte, uma vez que estas técnicas podem possuir etapas tediosas que poderiam ser executadas por um computador.

Deve-se notar, entretanto, que a inexistência de uma ferramenta computacional é apenas um dos fatores para a não aceitação destas técnicas. Outros fatores *podem* ser: falta de divulgação da técnica, resistência de equipes de projeto a alterarem o seu *modus*

[2]Muitas vezes com aplicações análogas ao subsistema físico.

operandi, etc.

Não obstante esses outros fatores, o desenvolvimento de ferramentas computacionais para o processamento de redes de Petri é um primeiro passo para a maior aceitação do uso de redes de Petri no meio industrial. Além disso, a disponibilidade de tais ferramentas também poderá permitir um rápido desenvolvimento de protótipos de programas que trabalhem com os métodos propostos pela academia.

Nos últimos anos, o reuso de código tornou-se um "mantra" da engenharia de computação (PRESSMAN, 2001; GAMMA; HELM; JOHNSON, 1995; SOMMERVILLE, 2001; STROUSTRUP, 1992). Atualmente pode-se encontrar facilmente bibliotecas de código livre ("open-source")(RAYMOND, 2000) na Internet com os mais diversos fins, o que reduz, em muito, o tempo necessário para a programação, além de evitar a repetição de trabalho entre diferentes grupos (já que o código para as tarefas básicas precisa ser gerado apenas uma vez).

A existência de uma biblioteca que encapsulasse o armazenamento das *RdP* da forma de acesso a elas poderia reduzir a dificuldade do pesquisador de *RdP*'s em criar softwares que as utilizem.

No entanto os resultados de uma pesquisa neste sentido foram infrutíferos. A maior parte da literatura encontrada sobre o tema versa sobre a representação de grafos, entretanto não foi encontrada nenhuma literatura versando sobre a representação do comportamento dinâmico das redes. O material sobre armazenamento de estados de rede também é escasso e antigo, principalmente no aspecto de reuso e modularidade de código, bem como na utilização de modelos orientados a objetos.

1.2 Objetivos

Desta maneira, o objetivo deste trabalho é demonstrar a viabilidade de um sistema computacional que permita a representação e manipulação computacional de *RdP*'s.

Este sistema, por hipótese, deve apresentar uma interface ao pesquisador de *RdP*'s

semelhante com a que ele está acostumado a utilizar. Em outras palavras, o pesquisador deve ser exposto apenas a termos e construções presentes na literatura de *RdP*'s, evitando-se ao máximo a utilização de termos advindos de estruturas de dados.

Pretende-se ainda que o sistema possa representar *qualquer RdP* e não apenas aquelas que pertençam a uma determinada classe de redes.

1.3 Estrutura do Texto

Uma visão geral dos componentes do sistema, juntamente com um método de utilização do mesmo, será mostrada no capítulo 2.

A *linguagem de representação da meta-rede* é utilizada para a geração automática de classes que encapsulam a *interface de armazenamento* na *interface de acesso*. Ela é apresentada em maiores detalhes no capítulo 3.

A *interface de acesso* é utilizada pelo desenvolvedor para escrever os algoritmos pertinentes ao seu programa enquanto que a *interface de armazenamento* é utilizada pelo desenvolvedor de biblioteca para a implementação de métodos de armazenamento de *conexões*, *inscrições estáticas* e *inscrições dinâmicas*. A interface de acesso é apresentada no capítulo 4 bem como considerações sobre a interface de armazenamento.

Algums trabalhos futuros e as conclusões deste trabalho são apresentadas no capítulo 5.

1.3.1 Notação de Conjuntos

Os conjuntos numéricos básicos do texto são apresentados segundo a tabela 1.1.

Um conjunto pode ser representado por uma lista, entre colchetes, separada por vírgulas contendo seus elementos. Exemplo:

$$C = \{1,2,3\}$$

operação	descrição
$A \cup B$	união de A com B.
$A \cap B$	intersecção de A com B.
$A \setminus B$	conjunto contendo os elementos de A que não existem em B.
A^*	conjunto de todos os subconjuntos de A.
$A^\times$	conjunto que contém todos os elementos dos elementos de A
$e \in A$	e é elemento de A

Onde A e B são conjuntos.

Tabela 1.2: Operadores sobre conjuntos.

Não existe conceito de ordem em um conjunto. Desta maneira $\{1,2,3\} = \{3,2,1\}$.

símbolo	descrição
$\varnothing$	conjunto vazio
$\mathbb{B}$	conjunto dos valores booleanos $\{T, F\}$
$\mathbb{N}$	conjunto dos números naturais.
$\mathbb{I}$	conjunto dos números inteiros.
$\mathbb{R}$	conjunto dos números reais.

Tabela 1.1: Conjuntos numéricos e básicos.

Os operadores sobre conjuntos utilizadas no texto são apresentadas na tabela 1.2. Observe que o operador $A^\times$ só faz sentido quando A é um conjunto de conjuntos.

2 Visão Geral e Método de Utilização

2.1 Introdução

O sistema proposto não tem por finalidade a utilização por um "usuário", mas sim por desenvolvedores. Desta maneira é importante fazer a distinção entre os desenvolvedores que efetivamente implementarão o sistema e aqueles que o utilizarão para a construção de seus programas. Propõe-se então a seguinte nomenclatura:

> **Definição 1:** Desenvolvedor do programa *é o utilizador final do sistema proposto. Normalmente esta pessoa será um pesquisador de RdP's que deseja implementar computacionalmente algo relacionado com estas. No texto, quando não especificado, o termo* desenvolvedor *será utilizado somente no sentido de* desenvolvedor do programa.

> **Definição 2:** Desenvolvedor da biblioteca *é o programador que irá efetivamente transformar os requisitos deste trabalho em uma ferramenta computacional. Tal pessoa deve possuir conhecimentos avançados de programação e profudo conhecimento de RdP's.*

Este capítulo tem por finalidade permitir ao leitor uma visão ampla do sistema proposto.

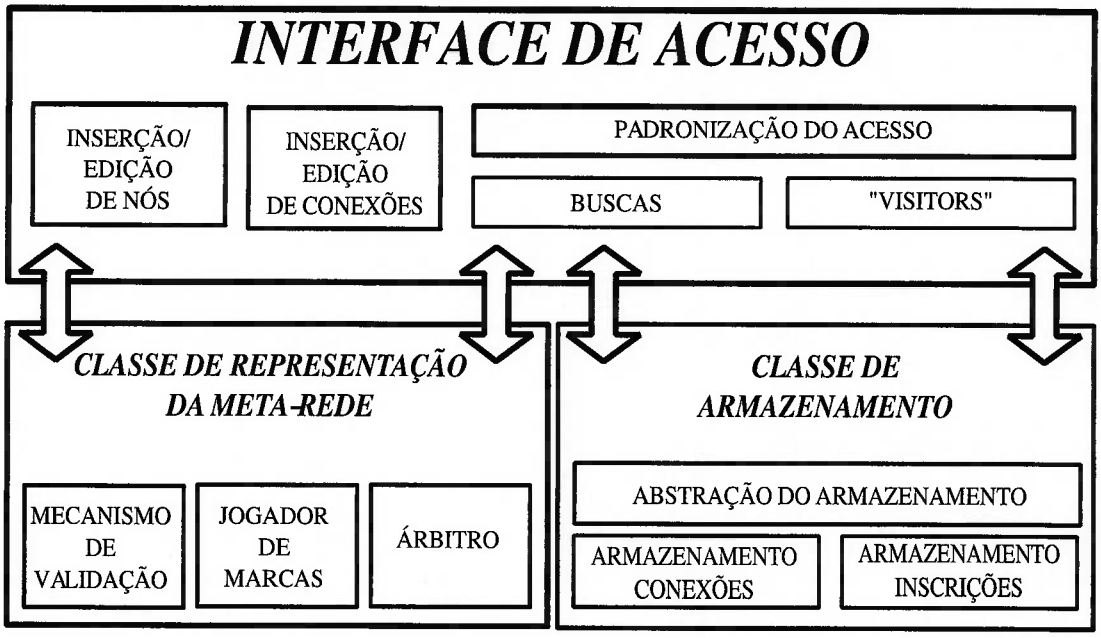

Figura 2.1: Relação entre os componentes do Sistema.

Inicialmente serão mostrados o encadeamento lógico entre os diferentes componentes do sistema, suas responsabilidade e interconexões.

A seguir discutir-se-á o método que o *desenvolvedor do programa* deve utilizar para cumprir seu objetivo. A cada etapa deste método os artefatos por ele produzidos e os aspectos da ferramenta aqui proposta utilizados serão enumerados.

Em seguida, serão mostradas as diferentes responsabilidades dos desenvolvedores de programa e biblioteca.

Concluir-se-á com a apresentação da estrutura do restante do texto.

2.2 Componentes do Sistema

O sistema computacional proposto é composto das seguintes partes:

1. Linguagem de representação da meta-rede
2. Classe de representação da meta-rede
3. Interface de acesso

4. Classes de armazenamento

Cabe aqui uma definição para o termo meta-rede:

> **Definição 3:** Meta-Rede *é a descrição de um* tipo *de RdP. Seja m uma meta-rede e p uma RdP. Se p puder ser representada por m é dito que m* atribui *p e que p é um* valor *de m ou ainda que p é atribuida por m.*

O objetivo da *Linguagem de representação da meta-rede* é a representação formal de uma meta-rede. Esta representação deve permitir verificar se uma determinada rede é atribuida por uma determinida meta-rede. A esta verificação dá-se o nome de *validação*[1] da rede.

A linguagem deve também permitir a geração automática de jogadores de marcas e de um árbitro. Considera-se estes elementos tão fundamentais para uma determinida rede quanto suas características estruturais.

Esta linguagem é compilada na *classe de representação da meta-rede.* Esta classe apresenta como interface para o desenvolvedor a *interface de acesso* (veja figura 2.1). O objetivo desta interface é padronizar a inserção, busca e edição de componentes de uma *RdP* bem como permitir a padronização de algoritmos de análise sobre esta.

A parte responsável pelo armazenamento efetivo da rede é a classe de armazenamento. Esta classe cuida do efetivo armazenamento computacional da rede.

2.3 Método de Desenvolvimento

Propõe-se a utilização do sistema de acordo com as seguintes etapas:

1. Escolha ou definição da meta-rede:

[1] Não confudir com validação do modelo.

Nesta etapa o desenvolvedor deve descrever a meta-rede que ele deseja utilizar através da *linguagem de descrição de meta-rede*, produzindo então a *descrição da meta-rede.*

Esta descrição pode ser armazenada para reuso futuro. Se já existir uma descrição de meta-rede que contemple os objetivos do desenvolvedor ele não é obrigado a repetir este passo.

2. Compilação para as classes:

   A *linguagem de descrição de meta-rede* é uma linguagem formal e permite que uma determinada descrição seja transformada automaticamente em uma *classe* para o armazenamento de *RdP*'s.

   Desta maneira a *linguagem de descrição de meta-rede* permite a descrição não só da meta-rede como também de uma classe que, ao ser instanciada, pode representar uma *RdP* qualquer atribuida por esta meta-rede.

   Para esta classe definem-se duas interfaces diferentes: a *interface de armazenamento* e a *interface de acesso.* O desenvolvedor do programa, a priori, não precisa se preocupar com a interface de armazenamento, e sim com a interface de acesso.

3. Seleção do método de armazenamento:

   Sua única preocupação com relação ao armazenamento é qual tipo por ele desejado. Diferentes formas de armazenamento, como por exemplo listas ligadas, matrizes, bancos de dados relacionais; possuem diferentes características.

   Observe que o desenvolvedor não deverá preocupar-se com aspectos da implementação deste armazenamento, mas sim em escolher dentro dos tipos disponíveis aquele que mais se adapta à sua necessidade. Separar o método de armazenamento da interface de acesso permite que a mudança de um determinado tipo de armazenamento para outro seja feita de forma transparente.

   Esta escolha não é crítica nesta etapa do projeto. Desta maneira o desenvolvedor pode simplesmente escolher uma implementação padrão para o armazenamento,

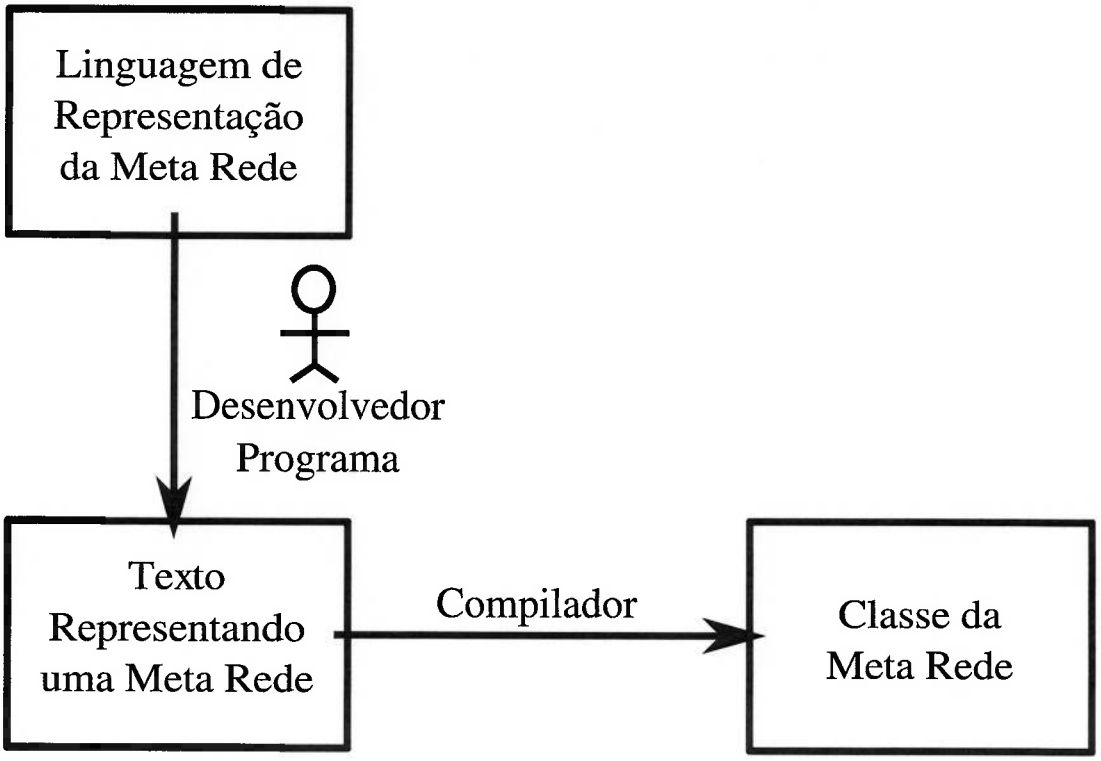

Figura 2.2: Definição da meta-rede e geração das classes

alterando-a posteriormente caso surja a necessidade. Esta alteração não implicará em alterações do código por ele produzido.

4. Escolha ou definição dos algoritmos:

   Nesta etapa o desenvolvedor deve procurar nos algoritmos disponíveis aqueles pertinentes ao seu objetivo. É provável que nem todos os algoritmos necessários sejam encontrados. O desenvolvedor deve, então, implementar quaisquer que faltem.

   Estes algoritmos devem utilizar a *interface de acesso* e não a *interface de armazenamento.*

5. Criação das interfaces específicas:

   O sistema aqui proposto, sozinho, não tem utilidade para um usuário final. É necessário portanto que o desenvolvedor encapsule os algoritmos e as classes aqui discutidas com uma interface que permita que o usuário tenha acesso a elas.

   Não é do escopo deste trabalho tratar do desenvolvimento de tais interfaces.

6. Testes do programa:

O desenvolvedor deve realizar testes em seu programa para garantir que os resultados obtidos condigam com os resultados esperados.

É importante ressaltar que esta não é uma etapa que deve ser realizada em seqüência às outras, mas sim em paralelo.

7. Determinação dos algoritmos críticos:

   Apenas quando o desenvolvedor estiver satisfeito com as funcionalidades de seu programa, ele deve preocupar-se com aspectos de eficiência na execução do mesmo.

   A primeira providência que pode ser tomada neste sentido é um estudo das diferentes classes de armazenamento disponíveis: cada uma delas apresenta diferentes características de complexidade e uso de memória. A escolha da classe adequada ao tipo de programa desejado pode alterar drasticamente seu tempo de processamento.

   O desenvolvedor deve então procurar, caso a escolha de uma boa classe de armazenamento não seja suficiente para atender a performance desejada, quais os algoritmos que consomem a maior parte do tempo de processamento. Tal busca pode ser feita através da realização de experimentos onde serão medidos os tempos gastos em cada um dos algoritmos.

   > **Definição 4:** Algoritmos críticos *são os algoritmos onde o programa do desenvolvedor gasta a maior parte do tempo de processamento.*

8. Otimizações:

   Uma vez determinado quais os algoritmos críticos o desenvolvedor procura melhorar a sua implementação.

   Para tal, pode ser necessário que ele utilize a interface de armazenamento diretamente para isto: o que é desaconcelhável uma vez que ao se utilizar a interface de armazenamento, o algoritmo fica dependende desta.

   A figura 2.3 mostra um exemplo de algoritmo utilizando diagramas Nassi-Shneiderman (*NS*). A parte *crítica* deste algoritmo é a determinação dos nós adjacentes.

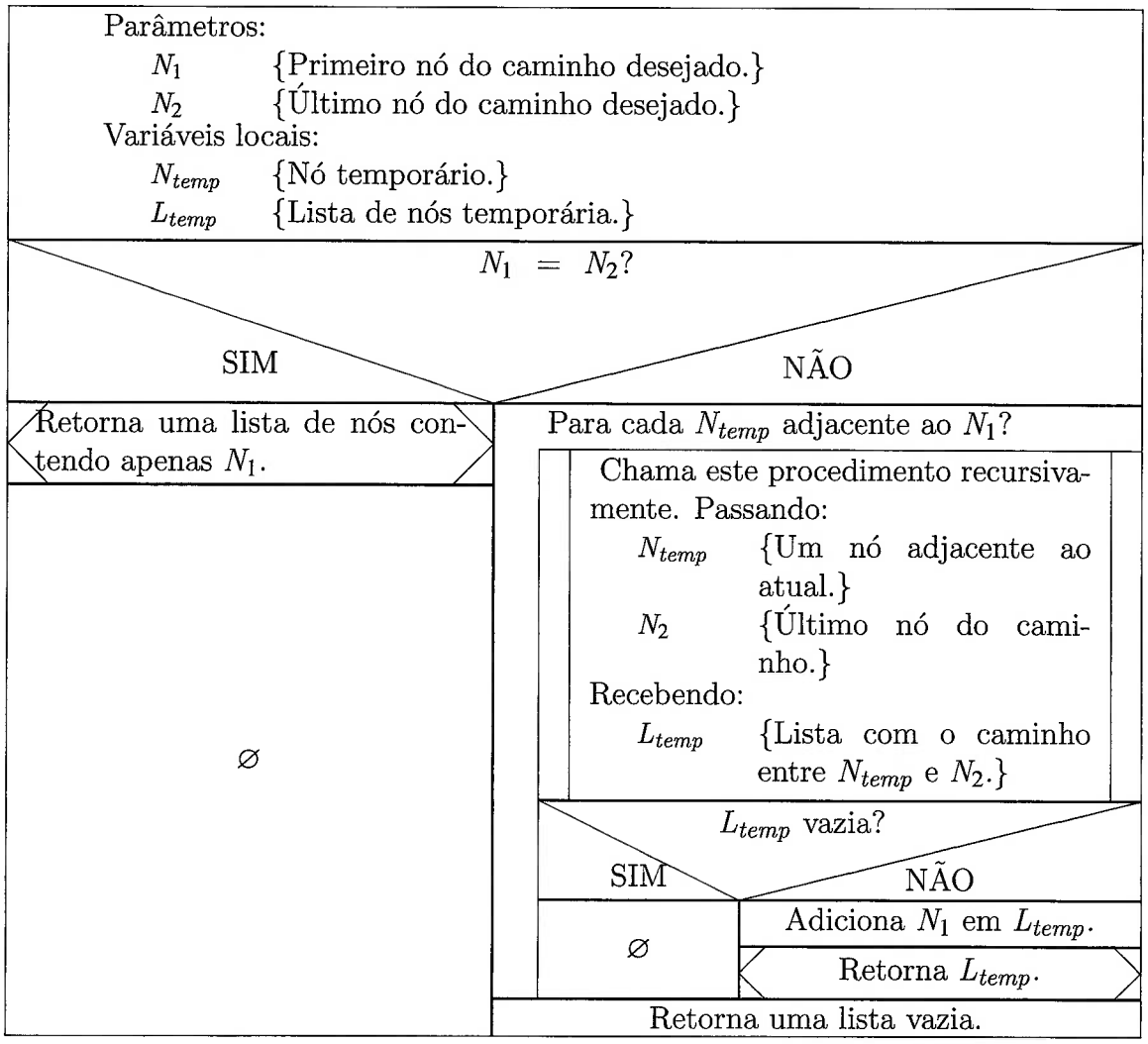

Figura 2.3: Exemplo de algoritmo utilizando a interface de acesso.

O algoritmo foi descrito naquela figura em linguagem natural. A interface de acesso, quando escrita na linguagem de programação, possuirá a mesma expressividade mostrada nas frases da figura.

Na figura 2.4 vemos as diferenças na implementação deste algoritmo utilizando diretamente uma implementação do armazenamento. Neste algoritmo assume-se que a interface de armazenamento utilizada é uma lista ligada (CORMEN et al., 2001, pg. 204-209) terminada por *NULL*.

Observe que, nesta forma, o algoritmo possui seu comportamento descrito em um nível mais baixo que na mostrada anteriormente. Sendo desta maneira dependente da implementação de armazenamento utilizada: uma mudança nesta implementação leva à necessidade de se reimplementar o algoritmo.

A implementação na interface de armazenamento é mais eficiente[2]: usa-se uma chamada a menos de função dentro da recursão. Esta implementação, no entanto, é mais díficil de ser compreendida, uma vez que exige o conhecimento de listas ligadas, e não apenas do conceito de conexão.

2.4 Responsabilidades dos Desenvolvedores

Observe que, segundo o método proposto, o desenvolvedor do programa apenas será exposto à interface de armazenamento se houver necessidade de otimização de algoritmos.

Conjectura-se que tal necessidade não será comum, uma vez que a utilização da interface de acesso, na maioria dos casos, não implicará em um aumento de complexidade algorítmica; seu custo será devido a um aumento de indireção. Uma discussão sobre as situações em que podem ocorrer aumento de complexidade algorítmica está presente no capítulo 4.

De maneira semelhante, o desenvolvedor de biblioteca não estará preocupado diretamente com detalhes da utilização da interface de acesso. Esta preocupação surgirá

[2]Esta implementação recursiva não é *a mais* eficiente: é possível melhorá-la transformando esta recursão na utilização de uma pilha.

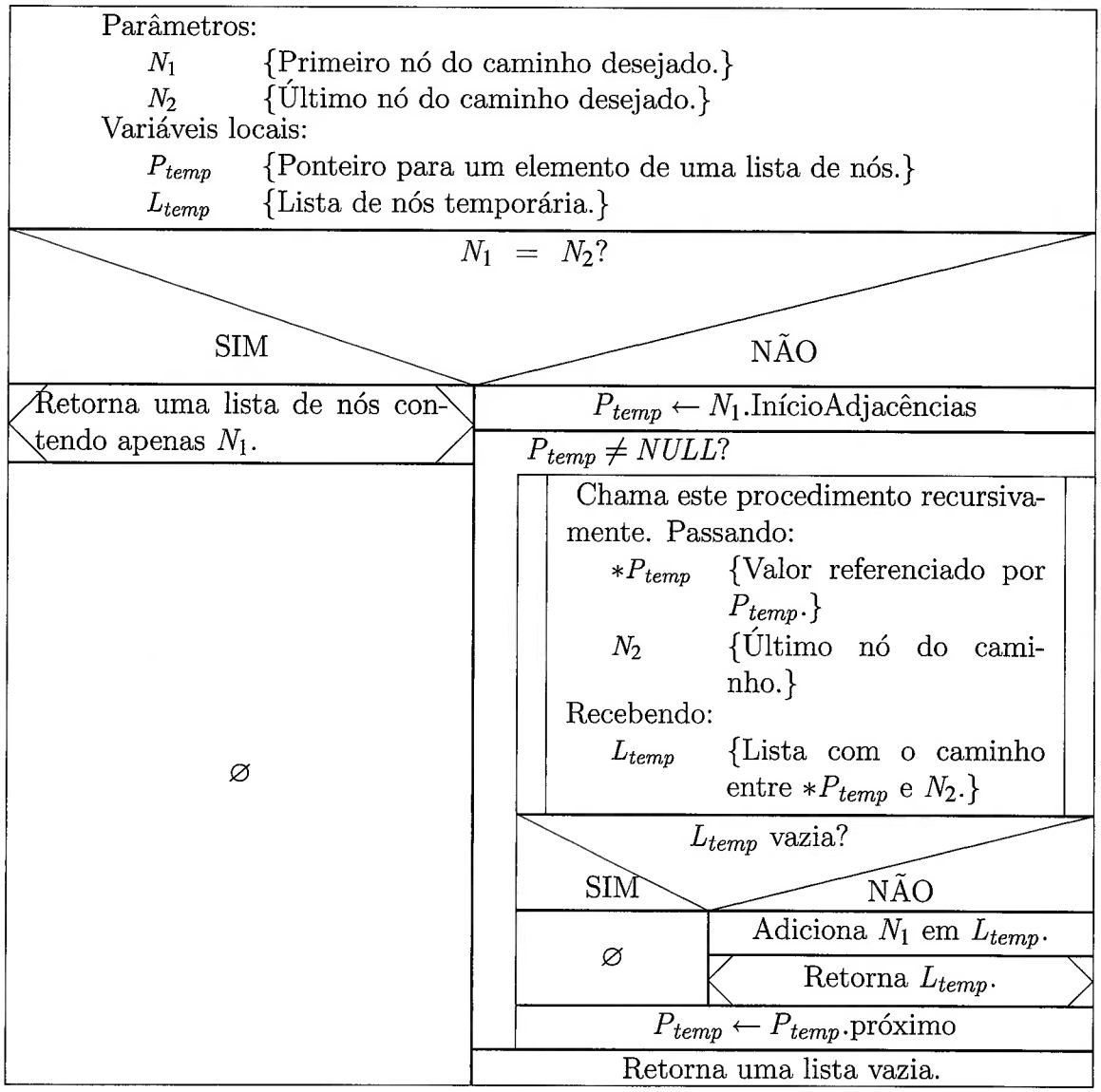

Figura 2.4: Exemplo de algoritmo utilizando diretamente uma implementação de armazenamento.

somente se este estiver modificando a ligação entre a interface de acesso e a interface de armazenamento.

A situação em que tal desenvolvedor criará uma nova forma de armazenamento para a rede é muito mais comum do que a situação em que ele terá que alterar aquela conexão.

Aconselha-se escrever novas meta-redes ao desenvolvedor de programa; tarefa que não será comum ao desenvolvedor de biblioteca.

Podemos então separar os aspectos deste trabalho entre os relevantes ao desenvolvedor de programas e o desenvolvedor de biblioteca.

Desenvolvedor de Programa :

- Linguagem de Descrição da Meta-Rede
- Interface de Acesso
- Técnicas de Construção de Algoritmos

Desenvolvedor de Biblioteca :

- Interface de Armazenamento de Inscrições Dinâmicas
- Interface de Armazenamento de Inscrições Estáticas
- Interface de Armazenamento de Conexões
- Técnicas de Construção de Algoritmos

2.5 Considerações finais do capítulo

O método apresentado permite a construção de programas baseados em *RdP*'s sem que o desenvolvedor precise conhecer profundamente os métodos utilizados para seu armazenamento. Tal conhecimento serviria apenas para tornar a implementação mais *eficiente* do ponto de vista computacional, entretanto ele não é necessário para a construção do programa.

3 Linguagem de Representação da Meta-Rede

Nesta capítulo será apresentada uma linguagem computacional para representação de meta-redes e uma nova definição para redes de Petri.

O método utilizado para o desenvolvimento desta linguagem foi o estudo de diversas definições clássicas de *RdP*'s. Algumas destas definições serão mostradas neste capítulo, seguidas de uma análise de suas características.

Dividiu-se as definições e análises em duas partes: componentes estruturais e componentes dinâmicos das redes.

> **Definição 5:** Componente estrutural *é qualquer componente da rede que não pode se alterar quando da mudança de estado da mesma.*

> **Definição 6:** Componente dinâmico *é qualquer componente da rede que* pode *se alterar quando da mudança de estado.*

Após esta análise, será apresentada a linguagem. Seguida de exemplos de redes nela representadas: as mesmas redes que foram definidas anteriormente.

3.1 Estrutura de *RdP*'s

Conforme a definição 3 (pg. 9), uma meta-rede é a descrição de um tipo de *RdP*. Esta descrição é normalmente encontrada na forma matemática. Nesta seção serão apre-

sentadas algumas definições para *RdP* encontradas na literatura. Estas definições foram utilizadas na análise que se seguirá.

Petri (apud VALK, 2003b, p. 14) definiu uma seguinte base[1] para todos os modelos de *RdP*. A seguinte definição é uma tradução deste texto:

Definição 7: *Uma rede é uma tupla* $N = \langle P, T, F \rangle$ *onde:*

- P *é um conjunto de* lugares*;*
- T *é um conjunto de* transições*, disjunto de* P*;*
- F *é um conjunto de pares ordenados* $F \subset (P \times T) \cup (T \times P)$*; cada um destes pares é denominado* arco.

Se P *e* T *forem finitos, então* N *é dita finita.*

Valk (2003a, p. 41) utiliza a base de PETRI, 1996, loc. cit. para apresentar duas definições para $R^{L}/_{T}$. A seguinte é uma variação destas:

Definição 8: *Uma rede de Petri lugar-transição* $(R^{L}/_{T})$ *é definida por uma tupla* $\mathcal{N} = \langle P, T, F, W \rangle$ *onde :*

- P *é um conjunto de* lugares*;*
- T *é um conjunto de* transições*, disjunto de* P*;*
- F *é um conjunto de pares ordenados* $F \subset (P \times T) \cup (T \times P)$*; cada um destes pares é denominado* arco*;*
- $W : F \rightarrow \mathbb{N} \setminus \{0\}$ *é uma função, denominada* função peso.

[1] Todas as definições desta secção foram modificadas de suas versões originais para melhor se adequarem a esta base.

Observe que esta definição utiliza o mesmo *modelo* de definição que a anterior, extendendo-o apenas com a inclusão da função peso.

A próxima definição é uma compilação sobre o trabalho de HAAS, 2002, passim, trabalho este onde ele define uma rede de Petri estocástica restrita (R_{ST}^{R}). A definição foi trabalhada a fim de aproximá-la do formato da definição 7 [2].

Definição 9: *Uma rede de Petri estocástica restrita* (R_{ST}^{R}) *é definida por uma tupla* $\mathcal{N} = \langle P, T, F, I, T_p \rangle$ *onde :*

- P *é um conjunto de* lugares*;*
- T *é um conjunto de* transições*, disjunto de* P*, tal que* $T = T^I \cup T^T$ *e* $T^I \cap T^T = \varnothing$*;* T^I *é um conjunto de* transições imediatas *e* T^T *é um conjunto de transições temporizadas*
- F *é um conjunto de pares ordenados* $F \subset (P \times T) \cup (T \times P)$*; cada um destes pares é denominado* arco*; os arcos* $(P \times T)$ *são chamados* arcos de entrada *e os arcos (T times P),* arcos de saída*.*
- I *é um conjunto de pares ordenados* $I \subset (P \times T)$*; cada um destes pares é denominado* arco inibidor*;*
- $T_p : T^T \rightarrow \mathscr{F}_d$ *é uma função; onde* $\mathscr{F}_d$ *é o conjunto de todas as distribuições de tempo permitidas; Cada uma das relações* $T^T \rightarrow \mathscr{F}_d$ *é chamada* função de ajuste de relógio*.*
- $\tilde{P} : (T^* \times P^*) \rightarrow [0,1]\mathbb{R}$ *é chamada* função de probabilidades*.*

Observe que as noções de tempo e de probabilidade não estão prensentes na definição 7, entretanto podem ser facilmente representadas em um modelo semelhante de definição.

Uma variante da definição de Valk (2003a, p. 43) para RAC é apresentada a seguir:

[2] Além disto a notação para alguns conjuntos foi alterada a fim de manter a consistência com este texto.

Definição 10: *Uma rede de Petri arco-constante (RAC) é definida por uma tupla* $\mathcal{N} = \langle P, T, F, \mathscr{C}, c_d, c_a \rangle$ *onde:*

- P *é um conjunto de* lugares*;*
- T *é um conjunto de* transições*, disjunto de* P*;*
- $\mathscr{C}$ *é um conjunto de* tipos*, denominadados* classes de cores*; cada classe de cor é um conjunto de* constantes*; o conjunto que contém todas as* constantes *é denominado* $\mathscr{C}^{\times}$*;*
- $c_d : P \rightarrow \mathscr{C}$ *é denominada* mapeamento do domínio de cores*;*
- F *é um conjunto de pares ordenados* $F \subset (P \times T) \cup (T \times P)$*; cada um destes pares é denominado* arco.
- $c_a : F \rightarrow (\mathbb{N} \times \mathscr{C}^{\times})^*$

Nesta rede há a introdução do conceito de *tipo*, ou seu sinônimo *classe*. Cabe aqui uma definição formal deste conceito:

Definição 11: Tipo *é um conjunto. Dizemos que uma determinada variável é de determinado tipo (ou* pertence *a determinada classe) se e somente se todos os valores que esta variável pode assumir forem elementos do* tipo*. A mesma nomenclatura pode ser utilizada para funções: dizemos que uma função é de determinado tipo se e somente se o contra-domínio desta função for este tipo.*

Apresenta-se agora uma tradução da definição de rede de Petri colorida (*RCL*) de Lakos (1995), novamente a notação foi alterada para manter a consistência com este texto:

Definição 12: *Uma rede de Petri colorida (RCL) é uma tupla* $\mathcal{N} = \langle P, T, F, \mathscr{C}, c_d, G, E \rangle$ *onde:*

- P *é um conjunto de* lugares*;*
- T *é um conjunto de* transições*, disjunto de* P*;*
- $\mathscr{C}$ *é um conjunto de* tipos*, denominadados* classes de cores*; cada classe de cor é um conjunto de* constantes*; o conjunto que contém todas as* constantes *é denominado* $\mathscr{C}^{\times}$*;*
- $c_d : P \rightarrow \mathscr{C}$ *é denominada* mapeamento do domínio de cores*;*
- F *é um conjunto de pares ordenados* $F \subset (P \times T) \cup (T \times P)$*; cada um destes pares é denominado* arco*;*
- $G : T \rightarrow expr$ *é a função de* guarda*;*
- $E : F \rightarrow expr$ *é a função de* expressão dos arcos*;*

Observe que nesta definição o conceito de expressão (*expr*) é deixado em aberto. Sendo assim ela está incompleta. Para a definição completa das *RCL* veja o trabalho de Lakos (1995) ou de Girault e Valk (2003).

Ao observar as definições acima percebe-se a presença de algums elementos principais:

1. Dois conjuntos disjuntos (P, T), algumas vezes subdivididos em outros subconjuntos (Cf. definição 9).
2. Um conjunto de pares ordenados F (*arcos*) de elementos dos conjuntos P e T de modo que (p_1, p_2) e (t_1, t_2) não são elementos válidos de F para quaisquer $p_n \in P$ e quaisquer $t_n \in T$.

   Em certos trabalhos(GIRAULT; VALK, 2003) esse conjunto de pares ordenados é representado por matrizes. Chamadas *matriz de incidência direta* e *matriz de incidência reversa.*

   Algumas definições utilizam outros conjuntos de pares ordenados com esta característica, como por exemplo, a utilização de arcos inibidores nas R^R_{ST}.

3. Funções, por exemplo a função *peso* da definição 8, que ligam elementos da rede a valores. Os valores válidos são bem definidos através de *tipos.*

   Estas funções podem ser substituidas por um *conjunto de inscrições*, sendo que cada relação entre um elemento do domínio da função com sua imagem será chamada de *inscrição.* Observe que este conjunto de inscrições é um subconjunto do produto cartesiano do domínio e do contra-domínio da função.

   A utilização do conceito de multiplicidade deixa a definição das relações possíveis mais simples, uma vez que elimina a necessidade de utilização de conjuntos do tipo A^* como contra-domínio das funções.

   > **Definição 13:** Multiplicidade *define quantas instancias de um tipo A podem estar associadas com uma instância de um tipo B. (LARMAN, 1998)*

4. Tipos: Algumas redes permitem que seus elementos tenham a eles associados inscrições, conforme discutido no item anterior. Estas inscrições, entretanto, não podem assumir valores arbitrários.

   É necessária, então, a utilização de alguma construção que permita delimitar os valores permitidos para tais inscrições. Este é o objetivo dos tipos e expressões.

   Conforme definido anterirormente, um tipo é um conjunto de valores que um determinida variável pode assumir. Para este caso específico podemos definir tipo como o conjunto de valores que um inscrição pode assumir.

   Se o número destes valores for limitado, podemos representar o tipo como uma lista simples de valores. Caso este número seja ilimitado esta representação não é possível.

   Para representarmos tipos ilimitados propõe-se a utilização de gramáticas da seguinte maneira: o conjunto das palavras gerado por determinada gramática é um tipo.

Observe que, tanto as R_{ST}^{R} quanto as RCL apresentam elementos que não podem ser representados por inscrições nos elementos. São eles, respectivamente:

1. Função de probabilidade
2. Classe de cores

3.2 Dinâmica de *RdP*'s

As *RdP*'s são ferramentas para a modelagem de sistemas dinâmicos, sendo assim elas devem, de alguma maneira, representar os diferentes *estados* destes sistemas.

Isso é feito classificamente através do conceito de *marcação.*

Definição 14: *Em uma* $R^{L}/_{T}$ *a* marcação *é uma função* $M : P \rightarrow \mathbb{N}$.

Algumas *RdP*'s podem apresentar o conceito de maneira ainda mais simples, como por exemplo as redes de Petri condição-evento ($R^{C}/_{E}$'s) (MIYAGI, 1996) que apresentam a marcação como um valor booleano associado a cada condição (lugar).

Como a marcação é um elemento dinâmico, ela se altera com a ocorrência de mudança de estado no sistema.

Em uma R_{ST}^{R}, o conceito de estado não pode ser dado apenas através de uma função com os lugares da rede como domínio. É necessário relacionar as transições temporizadas com o valor de seus relógios (HAAS, 2002).

Definição 15: *O estado em de uma* R_{ST}^{R} *é dado por:*

- $M : P \rightarrow \mathbb{N}$;
- $C_{lk} : T^{T} \rightarrow Tempo$;

Tanto as *RAC* quanto as *RCL* definem seu estado através da marcação dos lugares. Cada lugar tem a ele associado um multiconjunto de constantes, sendo que todas elas são pertencentes à classe de cor do lugar.

A única diferença entre a marcação de uma rede e as inscrições estáticas é seu caráter dinâmico: sendo assim os problemas para a sua representação são semelhantes. Desta maneira, de modo análogo às inscrições estáticas, faz-se a seguinte definição:

> **Definição 16:** Inscrição dinâmica *é uma relação entre um elemento da rede e um* valor. *Este valor pode variar dinâmicamente com a evolução da rede.*

Será então mais interessante uma discussão sobre as diferenças entre as inscrições dinâmicas e as inscrições estáticas em esferas diferentes das de sua representação. O aspecto dinâmico exige que, quando do seu armazenamento computacional, as inscrições estáticas e dinâmicas possam utilizar estruturas diferentes. Assim é necessário que, de alguma forma, seja possível a indicação do caráter estático ou dinâmico das inscrições na descrição de uma meta-rede para uso computacional.

Nas definições clássicas de redes de Petri esta diferença é normalmente deixada implícita quando da definição da evolução dos estados.

3.3 Meta-Rede

O *princípio da dualidade* nas *RdP*'s, explicado no capítulo 1 garante a existência de dois conjuntos de nós disjuntos, denominados usualmente P e T.

Entretanto, conforme mostrado nas definições anteriores, algumas *RdP*'s podem ter estes conjuntos particionados em subconjuntos. Em redes temporizadas, por exemplo, as transições são particionadas em transições imediatas e transições temporizadas.

Embora não seja usual, nada impede a criação de uma rede que tenha um terceiro (ou quarto, etc) tipo de elemento, com restrições diferentes ao tipo de conexão a ele permitido.

A linguagem de representação aqui apresentada para meta-redes norteia-se pelos seguintes principios:

- Permitir a validação de uma rede frente à meta-rede;
- Permitir o particionamento da rede em um número arbitrário de tipos de nós.
- Permitir a especificação de qualquer tipo de conexão entre os nós.
- Permitir a representação das redes apresentadas neste capítulo.

Na representação da meta-rede será permitido atribuir nomes aos diversos elementos da rede (nós,conexões, tipos, ...). Nestes nomes não é permitido a utilização do caracter "_". Ele é reservado para a utilização pelo sistema.

3.3.1 Representação dos Nós

Considere os nós de uma *RdP* de maneira individual, i. e., sem nenhuma conexão. Podemos então definí-lo da seguinte maneira:

> **Definição 17:** *Um nó é um conjunto de multi-conjuntos. Cada um destes multi-conjuntos é chamado* inscrição *do nó. Os elementos permitidos à estas* inscrições *serão chamados* tipos.

Observe que aqui não foi feita nenhuma consideração sobre quais conexões são permitidas ao nó. Atente também que está definição não tenta explicar o que é um nó na rede.

Propõe-se então a gramática da tabela 3.1 para a representação de nós. Utilizando esta gramática a representação de *lugar* em uma $R^{L}/_{T}$ fica conforme a tabela 3.2.

Cabe uma explicação sobre a semântica de cada frase nesta representação:

Node é a declaração do nó em si. É seguida de um *texto* que representa o nome do nó.

Este texto deve ser único como nome de nós, conexões e tipos.

« node »	→		BEGIN NODE « texto »
		&	« inscrições »
		&	« validação »
		&	END NODE
« inscrições »	→		« inscrição »
		|	« inscrições » \n « inscrição »
« inscrição »	→		« tipos » « multiplicidade » « nome » « dinâmica »
« multiplicidade »	→		« inteiro »
		|	« inteiro » - « inteiro »
		|	« inteiro » - *
		|	*
		|	« multiplicidade » , « multiplicidade »
« dinâmica »	→		static
		|	dinamic
« tipos »	→		« tipo »
		|	« tipos »,« tipo »
« validação »	→		*NULL* | BEGIN VALIDATION
		&	« python_func »
		&	END VALIDATION

Tabela 3.1: Gramática da linguagem de representação de nós na meta-rede (formato *BNF*).

Inscrições é uma lista com as inscrições que podem ser associadas ao nó.

Inscrição é a especificação de uma determinada inscrição

Tipo , especifica o tipo dos elementos que serão guardados na inscrição.

Multiplicidade , especifica o número de elementos que podem ser guardadados nesta inscrição.

- $*$: indica que a inscrição pode guardar qualquer número de elementos, inclusive

```
BEGIN NODE Lugar
    Inteiro 1 marcação dynamic
END NODE
```

Tabela 3.2: Representação de *lugar* em uma $R^{L}/_{T}$.

zero;

- Número inteiro: especifica que a inscrição deve guardar exatamente o número de elementos especificado;
- $n - m$: onde n e m são números inteiros. A inscrição pode guardar qualquer número de elementos no intervalo definido por n e m. Ela pode guardar inclusive n ou m elementos;
- $n - *$: onde n é um número inteiro. A inscrição pode guardar um número de elementos maior ou igual a n;
- Conjuntos dos anteriores separados por vírgulas: quaisquer uma das especificações é válida. Assim $0, 2 - 3$ indica que a inscrição pode não guardar elemento nenhum ou então guardar de 2 a 3 elementos.

Dinâmica , indica se a inscrição é dinâmica ("dynamic") ou estática ("static").

Validação , é uma função, escrita na linguagem *Python* (ROSSUM, 2005). Esta função recebe como parâmetro uma representação do elemento e deve retornar *true* se o elemento for válido. Caso contrário deve retornar uma "string" contendo uma mensagem de erro explicando o porquê o elemento não é válido.

Não é necessária escrever regras de validação para verificar se o valor de alguma inscrição pertence ao seu tipo. Este campo só é necessário quando existir a necessidade da utilização do valor de mais de uma inscrição ou de inscrições em elementos conexos para a validação do nó.

Um exemplo desta situação é nas redes com capacidade. Nestas redes a marcação nunca pode ser superior ao valor de capacidade dos lugares. A especificação de um lugar para uma rede deste tipo pode ser vista na tabela 3.3.

Uma maior discussão sobre validação pode ser encontrada na seção 3.3.5.

As informações que não podem ser representadas em inscrições (cf. seção 3.1) serão então representadas como nós. Isto é desejável pois elas são conectadas a diversos nós. Observe que este tipo de nó não é nem T nem P, mas sim uma nova categoria de nó.

```
BEGIN NODE Lugar
  Integer[0-*]     1 marcação   dynamic
  Integer[1-*]     1 capacidade static
  BEGIN VALIDATION
    if node.capacidade < node.marcaçao:
      return "Lugar com marcação superior à capacidade."
    return True
  END VALIDATION
END NODE
```

Tabela 3.3: Representação de *lugar* em uma rede com capacidade.

```
BEGIN NODE ClasseCor
    String  *  constantes  static
END NODE
```

Tabela 3.4: Representação de *classe de cor* em uma RAC.

A figura 3.1 mostra graficamente esta idéia. Esta figura é baseada na figura 3.3 de Girault e Valk (2003, p. 31). As classes de cores são $cars = \{a, b\}$; $starter = \{s\}$; $ready = \{rsa, rsb\}$ e $start = \{ssa, ssb\}$.

Desta maneira, a representação de classe de cor, assumindo um número finito de cores, em uma RAC ficaria conforme a tabela 3.4.

3.3.2 Representação das Conexões

A gramática para a definição de conexões é apresentada na tabela 3.5. A semântica das inscrições e da validação é exatamente a mesma dos nós. Cabe então uma discussão da semântica dos outros elementos.

con é a declaração da conexão. É seguida por um texto que a identifica. Valem as mesmas regras do que o texto para nós.

formato descreve a conexão. Cada conexão deve ser feita obrigatoriamente entre dois

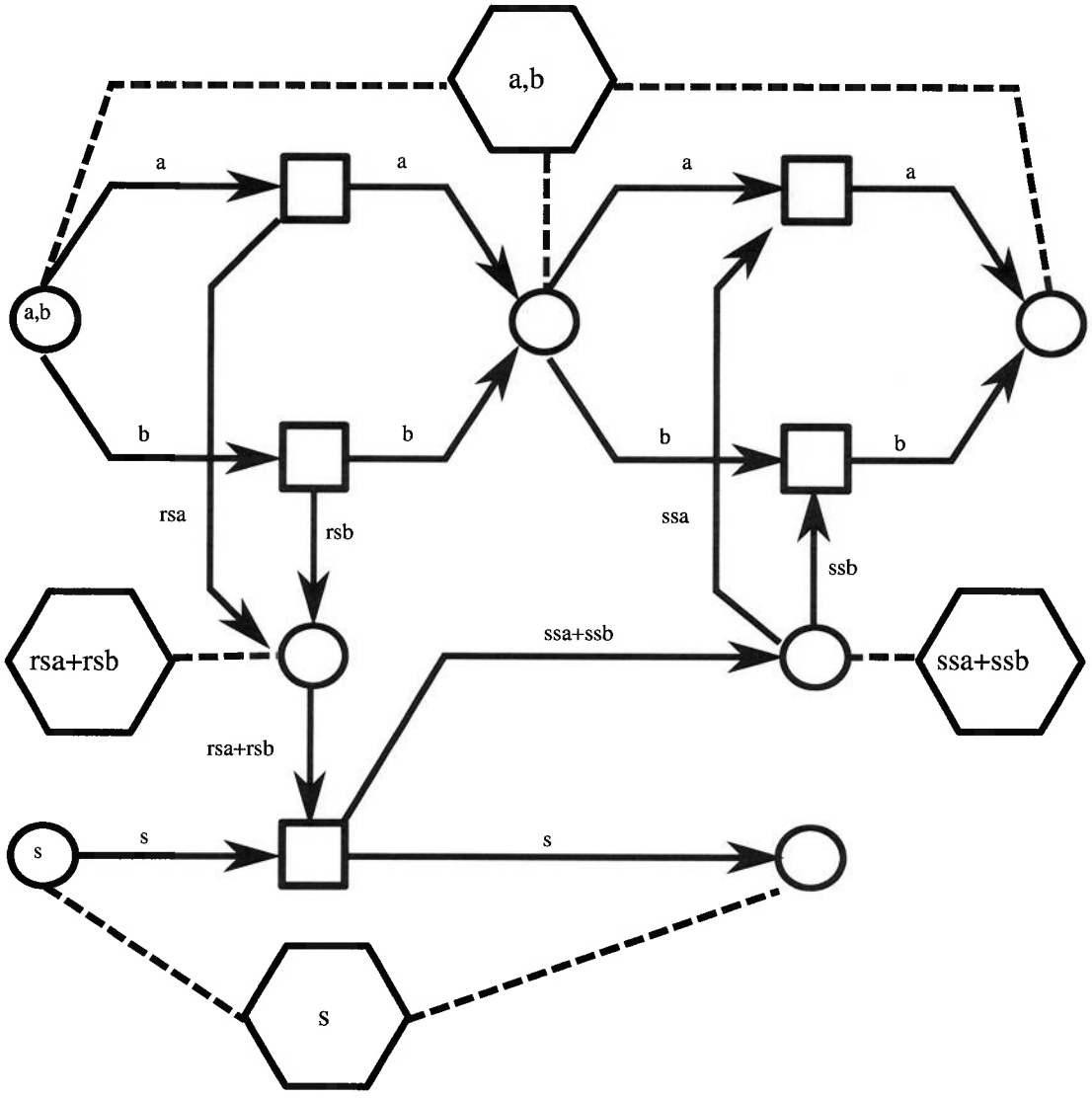

Figura 3.1: Representação gráfica de classes de cores (hexágonos) em uma rede RAC.

« con »	→		BEGIN CON « texto » « formato »
		&	« inscrições »
		&	« validação »
		&	END CON
« formato »	→		« direcionado » (« elemento »,«elemento»)
« elemento »	→		« nós »,[« multiplicidade »]
« nós »	→		« nó »
		|	{« nó* »}
« nó* »	→		« nó »
		|	« nó »,« nó* »
« multiplicidade »	→		« inteiro »
		|	« inteiro » - « inteiro »
		|	« inteiro » - *
		|	*
		|	« multiplicidade » , « multiplicidade »
« direcionado »	→		direct | undirect

Tabela 3.5: Gramática da linguagem de representação de conexões na meta-rede (formato *BNF*).

elementos.

direcionado indica se a conexão deve ser direcional ou não. Caso seja direcional utiliza-se a palavra chave "direct", caso contrario "undirect".

elemento indica quais os nós que podem fazer parte deste elemento e quantas conexões podem ser feitas para cada nó.

A tabela 3.6 mostra um exemplo de arco para uma rede com dois tipos diferentes de transições: *TransT* e *TransP*. Esta definição permite arcos ligando tanto um elemento *TransT* a um elemento *Lugar* quanto um elemento *TransP* a um elemento *Lugar*. A multiplicidade de [*] para as transições indica que cada transição da rede pode estar conectada de zero a infinitos arcos; o mesmo é válido para a mulitplexidade de [*] para os Lugares.

BEGIN CON arco direct ({TransT,TransP},[*],Lugar,[*]) END CON

Tabela 3.6: Representação de *arco* em uma rede com dois tipos de transição.

Vale ressaltar que a multiplicidade aqui apresentada é relacionada apenas à quantidade de conexões, do tipo que esta sendo definido, que cada elemento pode possuir. Cada conexão deve ser realizada entre exatamente dois elementos.

3.3.3 Representação dos Tipos

« tipo composto »	→	BEGIN TYPE « texto » COMPOSITE
	&	« inscrições »
	&	« validação »
	&	END TYPE

Tabela 3.7: Gramática da linguagem de representação de tipos compostos na meta-rede (formato *BNF*).

Os *tipos* tem como objetivo especificar o contra-domínio das inscrições.

Definem-se os seguintes tipos básicos:

Integer Número inteiro de tamanho arbitrário. *Pode* ter seu intervalo definido da seguinte maneira:

- Integer$[n-m]$: valor $\geq n$ e $\leq m$;
- Integer$[n-*]$: valor $\geq n$.

String Seqüência de tamanho arbitrário de caracteres. O número de caracteres *pode* ser definido utilizando as sequintes construções:

- String$[n]$: exatamente n caracteres;
- String$[n-m]$: qualquer número de caracteres entre n e m, inclusive;
- String$[n-*]$: um número de caracteres maior ou igual a n.

Real Número binário ou decimal de ponto flutuante. O intervalo de representação deve ser definido com as seguintes construções:

- Real[!n-m!] : valor $> n$ e $< m$;
- Real[n-m!] : valor $\geq n$ e $< m$;
- Real[!n-m] : valor $> n$ e $\leq m$;
- Real[n-m] : valor $\geq n$ e $\leq m$.

Para a definição completa de números binários com ponto flutuante outros aspectos devem ser considerados como o tamanho da mantissa e do expoente e o tratamento de excessões matemáticas. Estes aspectos, entretanto fogem do foco deste trabalho. Recomenda-se as normas (IEEE Task P754, 1985) e (IEEE, 1987) para um tratamento mais aprofundado deste assunto.

Permite-se a criação de tipos compostos utilizando a linguagem da tabela 3.7. Nesta tabela as inscrições e regras de validação são as mesmas utilizadas nas conexões e nos nós.

Algumas *RdP* como por exemplo a *RCL* apresentam em sua definição a utilização de inscrições que contenham *expressões*. Estas expressões podem ser descritas como linguagens dependentes de contexto: este contexto é normalmente a localidade da inscrição.

A validação deste tipo de inscrição não é uma tarefa simples, uma vez que sua sintaxe deve ser formalmente descrita. A formalização desta descrição não foi concluída neste trabalho, serão agora apresentados alguns resultados preliminares neste sentido.

Estudou-se a utilização de uma versão modificada da *BNF* para a representação das inscrições. Esta representação permitiria a geração automática de analizadores sintáticos baseados em autômatos de pilha.

A única diferença entre esta versão modificada e a versão original da *BNF* é a inclusão de terminais representando a localidade da inscrição. Desta maneira seriam permitidos, por exemplo, símbolos dos seguintes tipos na descrição do formato destas inscrições:

$(« nome do elemento ») Representa qualquer elemento conectado ao elemento da inscrição. Este elemento deve ser do tipo « nome do elemento ».

$(« nome do elemento ».Inscrições) Representa o nome de qualquer inscrição pertencente a um elemento do tipo « nome do elemento » conectado ao elemento considerado.

$(« nome de elemento ».« nome da inscrição ») Representa o valor de uma inscrição (« nome da inscrição »). Esta inscrição deve ser de um elemento do tipo « nome do elemento » que deve estar conectado ao elemento com a inscrição original.

$(con (« nome da conexão »).« nome de elemento ».« nome da inscrição ») Tem funcionamento semelhante ao do exemplo anterior, entretanto só são válidos os elementos conectados através de uma conexão do tipo « nome da conexão ».

$(con(« nome da conexão »).« nome da inscrição») Refere-se ao valor de uma inscrição ligada a uma conexão do tipo « nome da conexão ».

Esta lista não tenta ser exaustiva nem formal a ponto de esgotar o assunto. Considera-se uma melhor definição deste ponto como trabalho futuro.

Esta linguagem de representação seria associada a um tipo utilizando a gramática apresentada na tabela 3.8.

« tipo composto »	→	BEGIN TYPE « texto » LANGUAGE
	&	« descrição da linguagem »
	&	END TYPE

Tabela 3.8: Gramática da l inguagem de representação de expressões na meta-rede (formato *BNF*).

Vale ressaltar que não é *necessária* a criação de um mecanismo de representação de expressões para permitir a validação destas. A construção de validação em *Python* é suficiente para isto. Entretanto sua utilização para este fim vai de encontro com a hipótese de que deve-se apresentar ao desenvolvedor uma interface semelhante àquela que ele esta

acostumado. Uma vez que para a utilização deste recurso o desenvolvedor teria que criar o analizador sintático manualmente.

3.3.4 Representação da dinâmica

As regras de evolução de marcas em uma *RdP* são consideradas elementos básicos da mesma e, desta forma, devem ser especificados na meta-rede. Esta especificação deve ser suficiente para a geração automática de um jogador de marcas e de um árbitro de marcas.

Desta maneira divide-se a dinâmica em duas partes:

Disparo é o conjunto de procedimentos que deve ser aplicado a determinado *nó* da rede que provocam mudança de estado na rede.

Habilitação é um conjunto de regras para a verificação da possibilidade de *disparo* de um *nó*.

> **Definição 18:** *Um* nó *da rede é dito* habilitado *se e somente se sua regra de habilitação retornar um valor verdadeiro.*

O árbitro de marcas produzido deverá funcionar de maneira cíclica:

1. No início de cada ciclo o árbitro varre todos os nós da rede para os quais existem regras de habilitação, criando uma lista com todos os nós habilitados. A ordem destes nós é arbitrária.
2. Um nó é escolhido de forma aleatória da lista. Sua habilitação é verificada novamente e, caso continue habilitado, ele é disparado. Ele é então removido da lista. Este procedimento é repetido até que a lista se encontre vazia.
3. Inicia-se um novo ciclo.

Cabe ao desenvolvedor especificar então as regras de disparo e habilitação. Isto é feito utilizando a gramática da tabela 3.9. As regras de habilitação são novamente uma função em *"pyhton"* que retorna verdadeiro se o elemento estiver habilitado. As regras de disparo são um programa em *"pyhton"* que deve alterar a localidade do nó disparado.

Qualquer tipo nó da rede pode ter associado a ele uma regra de disparo e habilitação, mas caso uma esteja presente a outra também deve necessariamnete estar.

« dinâmica »	→	BEGIN DYNAMIC « tipo de nó » \n
	&	BEGIN HABIL \n « python func » \n END HABIL \n
	&	BEGIN FIRE \n « python func » \n END FIRE \n
	&	END DYNAMIC

Tabela 3.9: Gramática da linguagem de representação da dinâmica (formato *BNF*).

3.3.5 Representação da Meta-Rede e Validações

A meta-rede é representada então como um conjunto dos elementos anteriores: nós, conexões e tipos, conforme a gramática da tabela 3.10.

É possível ainda descrever uma regra de validação para a rede como um todo.

« meta-rede »	→	BEGIN NET « texto »
	&	« elementos »
	&	« dinâmicas »
	&	« validação »
	&	END NET
« elementos »	→	« node »
	|	« con »
	|	« tipos »
	|	« elementos » \n « elementos »
« dinâmicas »	→	« dinâmica »
	|	« dinâmicas » \n « dinâmicas »

Tabela 3.10: Gramática da linguagem de representação da meta-rede (formato *BNF*).

As regras de validação são checadas nas seguintes ocasiões:

1. Validação da Rede

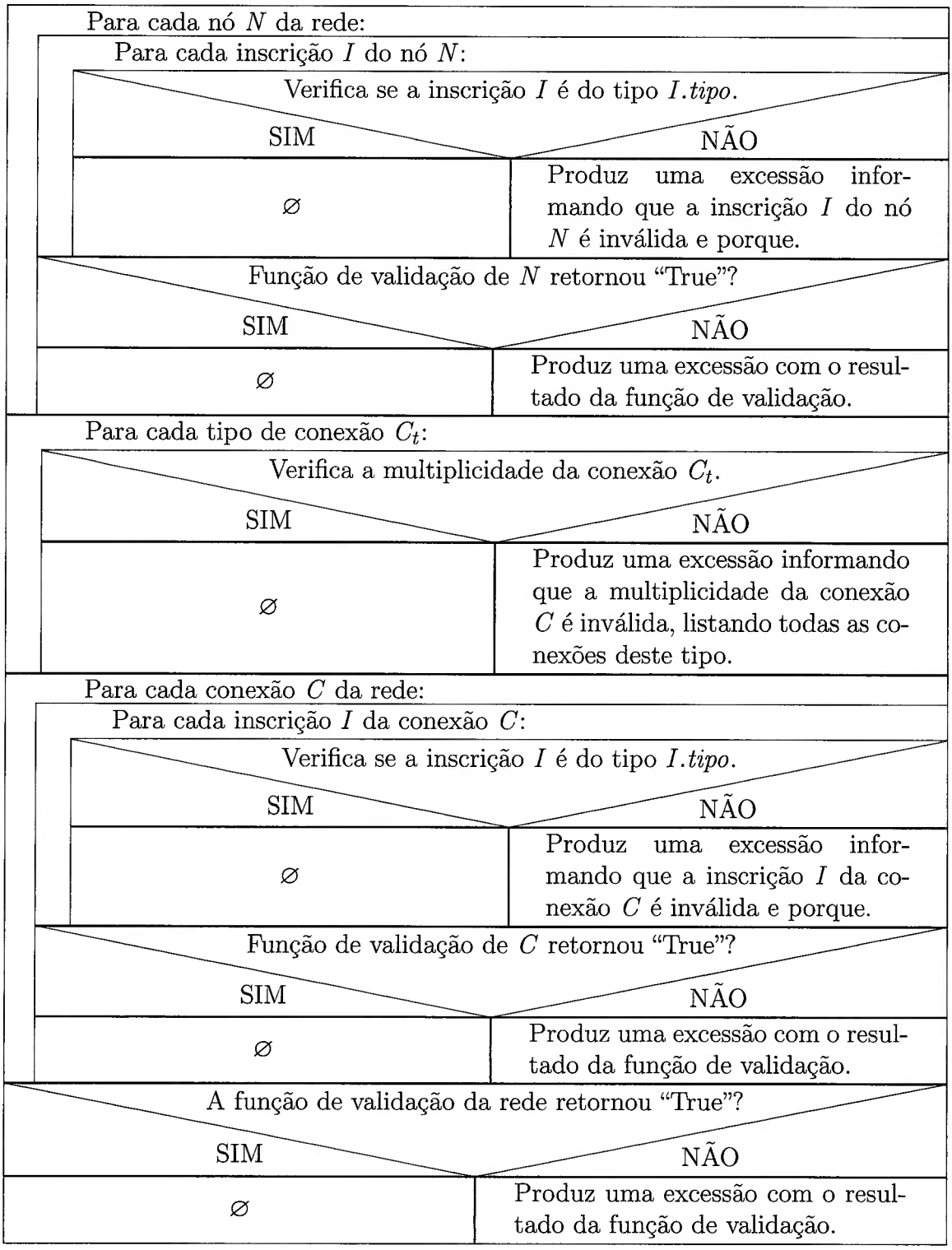

Figura 3.2: Procedimento de validação da rede (Parte I).

2. Validação de Tipo Composto

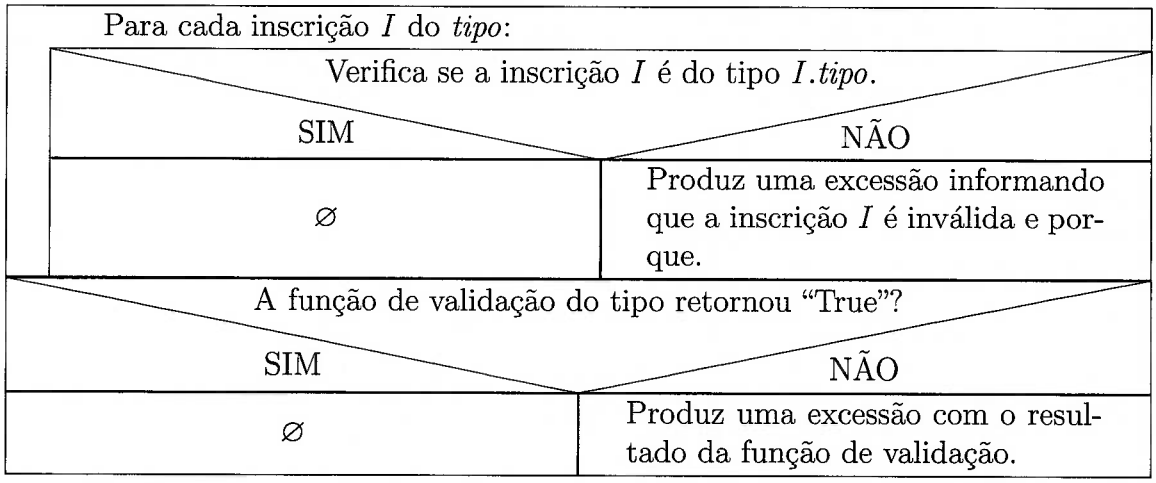

Figura 3.3: Procedimento de validação da rede (Parte II).

1. Ao inserir uma conexão na rede.

   É verificado se a conexão foi feita de acordo com a multiplicidade especificada e se conecta os tipos de nós especificados. Caso haja algum problema, uma *excessão* é produzida, com informações suficientes para identificar o erro.

   Só serão verificados os erros de multiplicidade por excesso de conexões. Isto deve ser feito uma vez que a rede é construida elemento por elemento, ou seja, não é possível determinar se ela já esta pronta, ou não, após a inclusão de uma conexão.

   Observe que a função de validação da conexão e as inscrições não são verificadas nesta etapa uma vez que as inscrições podem mudar e a regra de validação pode exigir que mais elementos estejam na rede que não os dois ligados à conexão.

2. Ao chamar o método de validação.

   O procedimento de validação segue de acordo com as figuras 3.2 e 3.3.

   As mensagens das excessões devem ser claras o suficiente para a identificação do elemento que torna a rede inválida e porque. No caso da inclusão de tipos de linguagem, sua validação deve ser feita pelo analisador sintático.

3.4 Exemplos de Aplicação

3.4.1 Representação de $R^L/_T$

A tabela 3.11 apresenta uma representação de $R^L/_T$ utilizando a meta-linguagem aqui descrita.

Observe que os *lugares* e *transições* foram modelados como nós e os *arcos* como conexões. Cada *lugar* tem uma única inscrição, chamada *marcação* atribuída. Esta inscrição é um número *inteiro* positivo.

A linha 06 indica que os *arcos* são direcionais e ligam apenas *lugares* a *transições* e que podem existir um número arbitrário de *arcos* nesta rede, inclusive múltiplos arcos ligando o mesmo lugar a uma mesma transição. Para evitar-se isto pode-se inserir uma regra de validação.

Nesta rede os únicos *nós* que podem ser disparados são as *transições*. A verificação de habilitação é feita entre as linhas 11 e 15.

A construção `node.con['Arco']` retorna uma lista contendo todas as conexões do tipo `Arco` do nó considerado. A direção de cada conexão pode ser verificada através da construção `c.dir`.

A regra de disparo esta entre as linhas 18 e 22.

3.4.2 Representação de RAC

A representação de uma RAC é mais extensa que a de uma $R^L/_T$ e pode ser vista na tabela 3.12.

Observe que na linha 04 a construção `node.con['ClasseAtrib']` retorna apenas uma conexão e não uma lista de conexões. Isto ocorre porque a multiplicidade da conexão `ClasseAtrib`, definida na linha 42, é de exatamente uma conexão por lugar.

As regras de validação do nó `Lugar` verificam se a marcação inicial do lugar esta de

```
1  BEGIN NET RDPLT
2    BEGIN NODE Lugar
3      Integer[0-*] 1 marcacao dynamic
4    END NODE
5    BEGIN NODE Transicao
6    END NODE
7    BEGIN CON Arco direct (Lugar,[*],Transicao,[*])
8      Integer[1-*] 1 multiplicidade static
9    END CON
10   BEGIN DYNAMIC Transicao
11     BEGIN HABIL
12       for c in node.con['Arco']:
13         if (c.dir=='in'):
14           if (c.multiplicidade > c.Lugar.marcacao):
15             return False
16           return True
17     END HABIL
18     BEGIN FIRE
19       for c in node.con['Arco']:
20         if (c.dir=='in'):
21           c.Lugar.marcacao -= c.multiplicidade
22         else:
23           c.Lugar.marcacao += c.multiplicidade
24     END FIRE
25   END DYNAMIC
26 END NET
```

Tabela 3.11: Representação de $R^{L}/_{T}$.

acordo com a classe de cor do mesmo. A validação dos arcos é feita de maneira semelhante.

3.5 Conclusão

Uma vez que a linguagem aqui apresentada possui um amplo poder de representação de *RdP*'s define-se, para este trabalho, uma *RdP* como sendo qualquer objeto que possa ser descrito na linguagem aqui apresentada.

A descrição das funções em *Python* foi propositalmente feita de forma informal. Um maior detalhamento será apresentado no capítulo 4.

A linguagem aqui mostrada é suficiente para descrever todos os tipos de rede mostrados no início deste capítulo. A representação de expressões, embora possível, poderia ser melhorada através da criação de uma linguagem formal para representá-las.

Através desta representação é possível a geração automática de classes de objetos que encapsulem o armazenamento e manipulação de *RdP*'s. Estas classes serão capazes de validar uma instância de rede e de realizar a simulação desta.

```
1  BEGIN NET ACN
2    BEGIN NODE Lugar
3      CteInteiro * marcacao dynamic
4      BEGIN VALIDATION
5        ctesValidas=node.con['ClasseAtrib'].
6          ClasseCor.constantes
7        for cte in ctesValidas:
8          if cte not in node.marcacao.nome:
9            novaMarca=CteInteiro()
10           novaMarca.multiplecidade=0
11           novaMarca.nome=cte
12           node.marcacao.append(novaMarca)
13       listTemp=list()
14       for nome in node.marcacao.nome:
15         if nome in listTemp:
16           return "Duas marcas com mesmo nome"
17         else:
18           listTemp.append(nome)
19       if nome not in ctesValidas:
20         return "Marcacao do No invalida"
21       return True
22     END VALIDATION
23   END NODE
24   BEGIN NODE Transicao
25   END NODE
26   BEGIN NODE ClasseCor
27     String * constantes static
28   END NODE ClasseCor
29   BEGIN TYPE CteInteiro COMPOSITE
30     Integer 1 multiplicidade
31     String  1 nome
32   END TYPE
33   BEGIN CON Arco direct (Lugar,[*],Transicao,[*])
34     CteInteiro * cor static
35     BEGIN VALIDATION
36       cores_validas=con.Lugar.con['ClasseAtrib'].
37         ClasseCor.constantes
38       for cte in con.cor:
39         if cte.nome not in cores_validas:
40           return "Cor " + cte.nome + " invalida."
41       return True
42     END VALIDATION
```

continua...

```
43    END CON
44    BEGIN CON ClasseAtrib undirect (Lugar,[1],ClasseCor,[*])
45    END CON
46    BEGIN DYNAMIC Transicao
47      BEGIN HABIL
48        for c in node.con['Arco']:
49          if (c.dir=='in'):
50            for insc in c.cor:
51              numMarcas = c.Lugar.marcacao.
52                find(nome=insc.nome).multiplicidade
53              if insc.multiplicidade > numMarcas:
54                return False
55        return True
56      END HABIL
57      BEGIN FIRE
58        for c in node.con['Arco']:
59          for insc in c.cor:
60            curMarcacao=c.Lugar.marcacao.find(nome=insc.nome)
61            if (c.dir=='in'):
62              curMarcacao -= insc.multiplicidade
63            else:
64              curMarcacao += insc.multiplicidade
65      END FIRE
66    END DYNAMIC
67  END NET
```

conclusão.

Tabela 3.12: Representação de *RAC*.

4 Interface de Acesso

As interfaces dividem-se em *interfaces de acesso* e *interface de armazenamento.* Divisão feita com o intuito de desacoplar a interface utilizada para *manipular* a rede da interface utilizada para o seu *armazenamento.*

A seleção do tipo de armazenamento desejado não será feito através de composição, conforme sugerido por Gamma, Helm e Johnson (1995, p. 151) na *pattern bridge* mas sim através das *classes de política* (*policy classes*) sugeridas por Alexandrescu (2001, p. 3).

Esta técnica faz uso dos *templates* (ECKEL, 2000) e da programação genérica(MUSSER; STEPANOV, 1989; HINZE, 2004; DEHNERT; STEPANOV, 1998)[1].

Conceitos são fundamentais na programação genérica. De acordo com Siek, Lee e Lumsdaine (2002, p. 19), em tradução livre:

> Em programação genérica [...] ao invés de escrevermos as especificações de um único tipo, descrevemos uma família de tipos em que todos apresentem a mesma interface e comportamento semântico. Ao conjunto de requisitos que descreve esta interface e comportamento semântico dá-se o nome de *conceito.* Algoritmos construidos no estilo genérico são então aplicáveis a *quaisqer* tipos que satisfizerem os requisitos do algoritmo.

É importante ainda definir *exceções*:

[1] Uma descrição deste paradigma de programação e uma comparação com programação orientada a objetos pode ser encontrada no livro de Siek, Lee e Lumsdaine (2002, p. 19-40).

Definição 19: *Uma* exceção *é um objeto que descreve um erro de programa. A produção de uma exeção dentro de um método para imediatamente a sua execução e retorna o controle do programa para o método que o chamou, caso este método apresente um bloco de tratamento de exceções; caso contrário o controle é passado para o primeiro método da pilha de funções que possua este bloco. Se nenhum método possuir tal bloco o programa é finalizado e uma mensagem de erro é produzida.*

Adotar-se-á então a seguinte definição para conceito:

Definição 20: Conceito *é o conjunto de requisitos que descreve a interface e o comportamento semântico de um determinado tipo.*

A linguagem de programação escolhida para a definição das interfaces foi o C++(STROUSTRUP, 1992; ECKEL, 2000). Este trabalho utiliza os seguintes conceitos da STL(ECKEL; ALLISON, 2003; Silicon Graphics, Inc., 1994):

- Assignable (cf. tabela 4.1);
- Default Constructible (cf. tabela 4.2);
- Equality Comparable(cf. tabela 4.3);
- Container (cf. tabela 4.4);
- Associative Container (cf. tabela 4.5) ;
- Iteradores associados aos containers (cf. tabelas 4.6, 4.7, 4.8, 4.9).

Assignable
Descrição
Um tipo é *Assignable*(atribuível) se é possível copiar objetos deste tipo e atribuí-los a variáveis.
Notação

- X →Um tipo que é aderente ao conceito *Assignable.*
- x,y →Uma instância de X.

Expressões Válidas

- X(x); →Retorna uma cópia de x.
- X x(y); →Cria uma nova instância de X, x; e faz com que ela seja uma cópia de y.
- X x=y; →Identico à linha anterior.
- x=y; →Altera x para que este seja uma cópia de y.
- swap(x,y); →Faz com que os valores de x e y sejam trocados um pelo outro.

Tabela 4.1: Conceito *Assignable.*

Default Constructible
Descrição
Um tipo é *Default Constructible*(construtível por padrão) se ele possui um construtor padrão, i. e., se é permitida a construção de uma instância deste tipo sem inicializá-la para qualquer valor particular.
Notação

- X →Um tipo aderente ao conceito considerado.
- x →Uma instância de X.

Expressões Válidas

- X(); →Cria uma instância de X.
- X x; →Cria uma instância de X denominada x.

Tabela 4.2: Conceito *Default Constructible.*

Equality Comparable
Descrição
Um tipo é *equality comparable*(igualável) se instâncias deste tipo podem ser comparadas utilizando o operador == e se este operador representa uma relação de equivalência.
Notação

- X →Um tipo aderente ao conceito considerado.
- x,y,z →Instâncias de X.

Expressões Válidas

- x==y; →Retorna uma variável booleana indicando se x é igual a y.
- x!=y; →Retorna uma variável booleana indicando se x é diferente de y.

Invariantes

- Identidade →&x==&y implica x==y.
- Reflexividade →x==x é sempre verdadeiro.
- Simetria →x==y implica y==x.
- Transitividade →x==y e y==z implicam x==z.

Observações
Observe que em nenhum momento foram feitas considerações sobre a semântica da igualdade.

Tabela 4.3: Conceito *Equality Comparable.*

Container
Descrição
Um *container* é um objeto que guarda outros objetos (seus *elementos*). Cada *container* tem um tipo *iterator* associado que é utilizado para varrer seus elementos.
Os elementos em um *container* não são armazenados em nenhuma ordem definida.
Um *container* é *dono* de seus *elementos*, i. e., quando o *container* deixa de existir, seus *elementos* também deixarão de existir.
Refinamento de
Trivial Iterator.
Definições
O *tamanho* de um container é o número de elementos que ele contém. Um *container de tamanho variável* é um container que permite a inserção e deleção de elementos. Um *container de tamanho fixo* não altera seu número de elementos.
Refinamento de
Assignable.
Tipos associados

- X::value_type →O tipo dos *elementos* do containter. Deve aderir ao conceito *Assignable.*
- X::iterator →O tipo do *iterator* usado para varrer os elementos do container. Deve aderir ao conceito de *input iterator* e permitir a conversão para X::const_iterator.
- X::const_iterator →Um iterador que pode ser utilizado para ler o valor dos elementos, mas não para alterá-los.
- X::reference →Um tipo que se comporta como uma *referência* aos elementos do container.
- X::const_reference →Um tipo que se comporta como uma *referência constante* aos elementos do container, ou seja, permite a leitura dos elementos, mas não sua alteração.
- X::pointer →Um tipo que se comporta como um ponteiro aos elementos do container.
- X::diference_type →Um tipo inteiro com sinal que pode ser utilizado para representar a distância entre dois iteradores.
- X::size_type →Um tipo inteiro positivo. Este tipo deve poder representar qualquer valor não negativo representável pelo X::diference_type.

Notação

- X →Um tipo aderente ao conceito *container.*
- a,b →Instâncias de X.
- T →X::value_type

continua...

Expressões Válidas

- X(a); →Cria um novo container com uma cópia de todos os elementos de a.
- X b(a); →Cria um novo container,chamado b com uma cópia de todos os elementos de a.
- b=a; →Faz com que b seja um container com uma cópia de todos os elementos de a.
- a.~X(); →Destroi todos os elementos de a. Qualquer memória alocada para eles é liberada.
- a.begin(); →Retorna um iterador que aponta para o primeiro elemento de a.
- a.end(); →Retorna um iterador que aponta para um elemento fictício após o último elemento de a.
- a.size(); →Retorna o número de elementos de a.
- a.max_size(); →Retorna o número máximo de elementos que a pode guardar.
- a.empty(); →Retorna uma variável boolena indicando se a está vazio.
- a.swap(b); →Equivalente a swap(a,b) (cf. tabela 4.1).

Complexidade

- X(a); X b(a); b=a; a.~X(); →O(n), onde n é o número de elementos de a.
- a.begin(); a.end(); →O(1).
- a.size(); →O(n), onde n é o número de elementos de a.
- a.max_size(); a.empty(); →O(1).
- a.swap(b); →O(1).

Invariantes

- Intervalo válido →Para qualquer container, o intervalo definido entre a.begin() e a.end() é válido.
- Tamanho do intervalo →a.size() é igual a distância entre a.begin() e a.end().
- Complitude →Um algoritmo que itere no intervalo definido por a.begin() e a.end() passará por todos os elementos de a.

---

conclusão.

Tabela 4.4: Conceito *Container*.

A compilação de uma meta-rede produzirá uma classe com o nome desta meta-rede. Assim, para a meta-rede descrita na tabela 3.11, o nome desta classe será RDPLT. Esta classe será chamada classe *principal.*

Além da classe principal, serão geradas classes nomeadas de acordo com os nós, conexões, inscrições e tipos compostos da rede[2]. A estas classes dá-se o nome de classes *elementares.*

Os tipos são expostos conforme a tabela 4.11. Além disto, dentro de cada elemento, suas inscrições também terão o tipo exposto. Assim, para uma $R^L/_T$ os tipos da tabela 4.12 serão válidos.

A classe principal será sempre criada dentro de um *namespace* de mesmo nome da rede. O nome da classe também será o mesmo da rede. Veja a tabela 3.11 para um exemplo.

A interface de acesso é a união das interfaces das classes elementares, da classe principal e de classes *auxiliares.* Estas últimas são classes utilizadas para auxiliar o acesso às classes elementares e à classe principal.

A classe principal mantém o *estado* da rede (cf. figura 4.1). É importante ressaltar que, aqui, por *estado* não se entende o conjunto de inscrições estáticas da rede. O sentido é o mesmo utilizado por Gamma, Helm e Johnson (1995, p. 305) na *pattern* comportamental *state.*

Gamma, Helm e Johnson (1995) definem o objetivo da *pattern state* como sendo(em tradução livre):

> Permitir que um objeto altere seu comportamento quando seu estado inicial muda. O objeto parecerá ter mudado de classe.

Em outras palavras, este *estado* define quais métodos estarão disponíveis em um determinado instante de tempo para uma determinada instância de rede.

[2]Para um exemplo veja a tabela 4.10.

Associative Container
Descrição

Um *associative container* é um container que associa a cada um de seus elementos uma *chave*. Esta chave é utilizada para busca de seus elementos.
Refinamento de
Containter.
Tipos Associados
Os mesmos do container(cf. tabela 4.4). Mais o seguinte:

- X::key_type →O tipo da *chave*.

Notação

- X →Um tipo aderente ao conceito *associative container*.
- a →Instância de X.
- t →Instância de X::value_type.
- k →Instância de X::key_type.
- p,q →Instâncias de X::iterator.

Expressões Válidas

- X() →Cria um container vazio.
- a.erase(k) →Apaga todos os elementos associados à chave k.
- a.erase(p) →Apaga o elemento referenciado por p.
- a.erase(p,q) →Apaga os elementos dentro do intervalo definido por p,q.
- a.clear() →Apaga todos os elementos do container.
- a.find(k) →Encontra um elemento cuja chave seja k.
- a.count(k) →Retorna o número de elementos associados à chave k.
- a.equal_range(k) →Retorna um par de iteradores que definem um intervalo onde todos os elementos possuem a mesma chave k.

Tabela 4.5: Conceito *Associative Container*.

Trivial Iterator
Descrição

Um *trivial iterator* é um objeto que representa um outro objeto.
Tipos Associados

- value_type →O tipo do objeto representado pelo iterador.

Notação

- X →Um tipo que é aderente ao conceito *Trivial Iterator*.
- x,y →Uma instância de X.
- T →O tipo X::value_type.
- t →Um objeto do tipo T.

Definições

Um tipo aderente ao conceito *trivial iterator* pode ser *mutável*, ou seja, os objetos por ele referenciados podem ser alterados; por outro lado, este tipo pode ser *constante*, i. e., o valor dos objetos por ele referenciados não pode ser alterado.
Um iterador pode ter um valor *singular*, ou seja, o objeto que ele referencia é inválido. *Invalidar* um iterador significa aplicar sobre ele alguma operação que faça com que seu valor se torne singular.
Expressões Válidas

- X x →Cria um iterador de valor singular.
- *x →Se x não for singular retorna o objeto por ele referenciado. A operação é inválida se x for singular.
- *x = t →Esta operação só é válida de x for não singular e X mutável. Faz com que o objeto referenciado por x se torne uma cópia de t.
- x->m →Esta operação só é válida se x for não singular e T for um tipo com a operação t.m definida. Executa a operação m no objeto referenciado por x.

Complexidade

Todas as operações são O(1).
Invariantes

- Identidade →x==y se e somente se &*x==&*y.

Tabela 4.6: Conceito *Trivial Iterator*.

Input Iterator
Descrição

Um *input iterator* é um objeto que representa um outro objeto. Além disto ele pode ser incrementado, mudando assim o objeto que ele representa para o próximo elemento de uma seqüência.
Tipos Associados

- value_type →O tipo do objeto representado pelo iterador.
- distance_type →Um tipo inteiro com sinal que representa a distância entre dois iteradores em um intervalo.

Notação

- X →Um tipo que é aderente ao conceito *Input Iterator.*
- i,j →Uma instância de X.
- T →O tipo X::value_type.
- t →Um objeto do tipo T.

Definições

Um iterador está *além-do-fim* se ele aponta para um elemento fictício além do último elemento de um container(cf. tabela 4.4).
Um iterador é *válido* se ele é não-singular ou se ele está *além-do-fim.*
Um iterador é *incrementável* se existe um "próximo" iterador, i. e., se ++i está bem definido. Iteradores *além-do-fim* não são *incrementáveis.*
Um iterador j é *alcançável* a partir de um iterador i se, após aplicar-se a operação ++ em i um número finito de vezes i==j.
A notação [i,j) é um *intervalo* contendo todos os iteradores a partir de i até j, incluindo i mas sem incluir j.
O intervalo [i,j) é dito *válido* se i e j são iteradores válidos e j é alcançavel a partir de i.
Expressões Válidas

- ++i →Operação só é permitida se i for válido e não for *além-do-fim.* Troca o elemento referenciado pelo iterador para o próximo da lista.
- i++ →Idem.
- *i++ →Equivalente a T t=*i; i++; return t;

Complexidade

Todas as operações são O(1).

Tabela 4.7: Conceito *Input Iterator.*

Output Iterator
Descrição

Um *output iterator* é um tipo que apresenta mecanismos para guardar(mas não necessariamente acessar) uma seqüência de valores.
Refinamento de
Assignable, Default Constructible.
Notação

- X →Um tipo que é aderente ao conceito *Input Iterator.*
- i,j →Uma instância de X.

Expressões Válidas
São permitidas as expressões dos conceitos Assignable e Default Constructible juntamente com as seguintes:

- ++i →Avança o iterador. Só pode ser utilizada se um valor já tiver sido atribuído à i.
- i++ →Idem.
- *i=t →Guarda o valor de t no iterador.
- *i++=t →Equivalente a T *i=t; i++;

Complexidade

Todas as operações são O(1).
Notas
Um output iterator é utilizado para preencher um container com valores. Assim ele só pode ser incrementado após ter algum valor a ele atribuído.

Tabela 4.8: Conceito *Output Iterator.*

Forward Iterator
Descrição

Um *forward iterator* corresponde à noção de uma seqüência de valores, i. e., ao atravessar um intervalo definido por *forward iterators* passar-se-á sempre pelos mesmos elementos, na mesma ordem.
Um *forward iterator* pode ser mutável ou imutável (cf. tabela 4.6).
Refinamento de
Input Iterator, Output Iterator.
Tipos Associados
Os mesmos dos input iterators(cf. tabela 4.7).
Notação

- X →Um tipo que é aderente ao conceito *Input Iterator.*
- i,j →Uma instância de X.
- T →O tipo X::value_type.
- t →Um objeto do tipo T.

Expressões Válidas
São permitidas as mesmas expressões dos input iterators. Entretanto a seguinte condição é garantida: se i==j então ++i == ++j.
Complexidade

Todas as operações são O(1).

Tabela 4.9: Conceito *Forward Iterator.*

```
namespace RDPLT {
  class RDPLT {
    /* ... */
    public:
      /* ... */
      typedef /* ... */ Transicao_t;
      typedef /* ... */ Lugar_t;
      typedef /* ... */ Arco_t;
  };
}
```

Tabela 4.10: Tipos expostos após a compilação da $R^{L}/_{T}$ (Cf. tabela 3.11).

<Nome da Rede>
→ <Nome do No1>_t
→ <Nome do No2>_t
→ ⋮
→ <Nome da Conexao1>_t
→ ⋮
→ <Nome do Tipo Composto1>_t
→ ⋮

Tabela 4.11: Árvore de nomes de tipos expostos em uma rede.

• RDPLT
• RDPLT::Lugar_t
• RDPLT::Lugar_t::marcacao_t
• RDPLT::Transicao_t
• RDPLT::Arco_t
• RDPLT::Arco_t::multiplicidade_t

Tabela 4.12: Tipos expostos na RDPLT.

O principal objetivo deste particionamento da interface de acesso é impedir que uma mudança estrutural na rede invalide a simulação da mesma. A transição do estado *mutável* para o estado *simulável* é feita através da chamada do método validate(). A transição só é efetivada se a rede for *válida*, i. e, se obedecer as regras de validação apresentadas na seção 3.3.5.

Quando a rede está no estado *mutável* é permitida qualquer alteração estrutural da rede como por exemplo a criação e deleção de nós e conexões, edição de inscrições estáticas, etc. Neste estado, entretanto, as funções de simulação da rede não estarão disponíveis.

No estado *simulável* não é permitida a modificação de componentes estáticos da rede, apenas a de componentes dinâmicos. Neste estado as funções de simulação ficam disponíveis ao desenvolvedor. Uma vez neste estado para retornar-se ao estado *mutável* basta a chamada do método mutate().

A tabela 4.13 apresenta os métodos válidos em cada estado da rede. A utilização de um método fora de um estado em que ele seja válido resultará no disparo de uma exceção.

Para a utilização do sistema o desenvolvedor deve em primeiro lugar carregar uma rede. Isso é feito inserindo seus elementos um a um utilizando a interface de acesso. Sugere-se a seguinte ordem de inserção dos elementos:

- Inserção dos nós e das conexões;

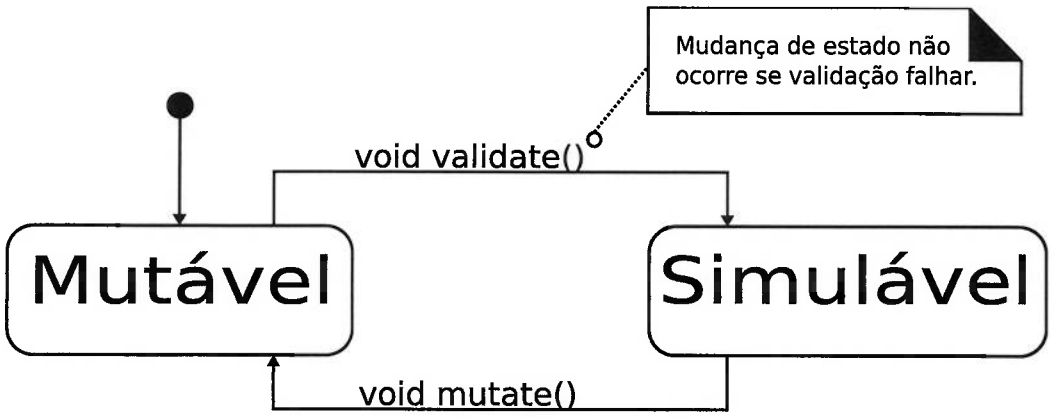

Figura 4.1: Diagrama de estados para a rede.

- Preenchimento das inscrições estáticas;
- Preenchimento das inscrições dinâmicas com o estado inicial da rede.

Os tipos dos nós e conexões seguem garantidamente o conceito de *assignable.*

4.1 Inserção e deleção de nós e conexões

A inserção e deleção de nós e conexões é feita através dos métodos apresentados na figura 4.2. Estes métodos só podem ser utilizados no estado *mutável.*

A utilização de *"templates"* para a definição desta interface faz com que seja possível uma verificação de tipos , em tempo de compilação, para o código. Assim, se em uma rede lugar transição, o desenvolvedor tentar criar uma conexão lugar-lugar inválida conforme o código abaixo da tabela 4.14 receberá um erro de compilação semelhante ao mostrado na tabela 4.15.

A possibilidade desta verificação em tempo de compilação reduz o tempo necessário para a depuração do programa. É importante ressaltar que está checagem de tipos é realizada pelo próprio compilador do C++.

Estado	
Mutável	**Simulável**
addNode	validate
delNode	mutate
addConnection	getConstInscription (Inscrições Estáticas)
delConnection	getConstInscription (Inscrições Dinâmicas)
validate	getMutableInscription (Inscrições Dinâmicas)
getMutableInscription (Inscrições Estáticas)	delInscription (Inscrições Dinâmicas)
getConstInscription (Inscrições Estáticas)	
addInscription (Inscrições Estáticas)	
delInscription (Inscrições Estáticas)	
getMutableInscription (Inscrições Dinâmicas)	
getConstInscription (Inscrições Dinâmicas)	
addInscription (Inscrições Dinâmicas)	
delInscription (Inscrições Dinâmicas)	

Tabela 4.13: Métodos válidos em cada estado da rede.

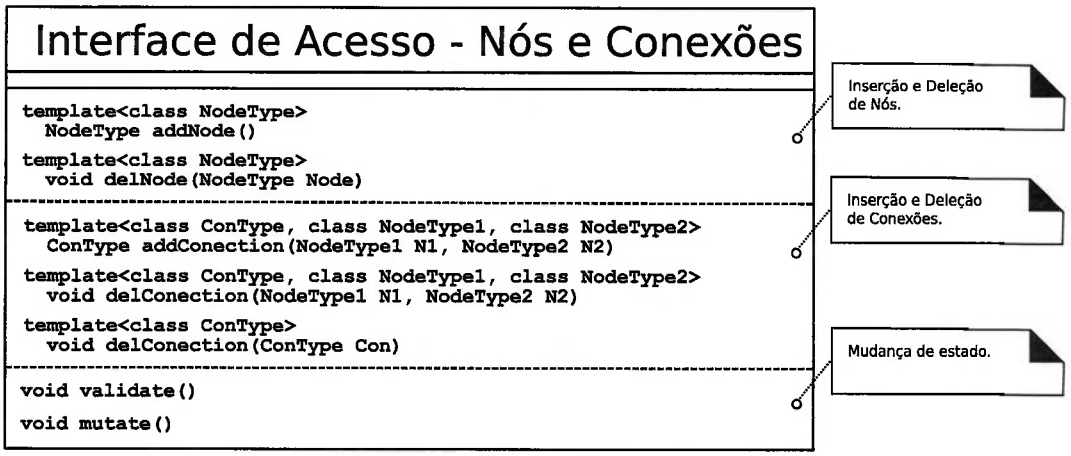

Figura 4.2: Interface de acesso - inserção e deleção de nós e conexões; mudança de estado.

```
Rede Rd;
Rd::Lugar L1,L2;
L1=Rd.addNode<Rd::Lugar>();
L2=Rd.addNode<Rd::Lugar>();
rede.addConnection<Rd::Arco>(L1,L2);
```

Tabela 4.14: Tentativa de inserção de conexão inválida.

```
/tmp/ccX0vKfy.o: In function `main':
main.cpp:(.text+0x3f): undefined reference to `Arco
InterfaceAcesso::addConnection<Arco, Lugar, Lugar>(Lugar, Lugar)'
collect2: ld returned 1 exit status
```

Tabela 4.15: Mensagem de erro produzida ao inserir conexão inválida.

Observe também que não é necessária a especificação dos três tipos (ConType,NodeType1 e NodeType2) para a função addConnection uma vez que os tipos NodeType1 e NodeType2 são deduzidos automaticamente pelo compilador.

Segue agora uma breve explicação da sintaxe e do funcionamento de cada um dos métodos:

- addNode<NodeType>()

  Adiciona um nó do tipo <NodeType> na rede. Retorna o nó.

- delNode(N1)

  Apaga um determinado nó da rede. Caso este nó não exista na rede é disparada uma exceção.

- addConection<ConType>(N1,N2)

  Adiciona uma conexão entre os nós N1 e N2 do tipo <ConType>.

- delConection<ConType>(N1,N2)

  Apaga *todas* as conexões do tipo <ConType> entre os nós N1 e N2. Caso não exista nenhuma conexão deste tipo é disparada uma exceção.

- delConection(C1)

  Apaga a conexão C1. Caso esta conexão não exista na rede é disparada uma exceção.

Por exemplo, para criar-se os nós e conexões da rede da figura 4.3 deve-se utilizar o código da tabela 4.16.

Observe que não é obrigatória a criação de variáveis para guardar um elemento recém criado, entretanto elas são úteis para referênciá-los sem a necessidade da realização de uma busca.

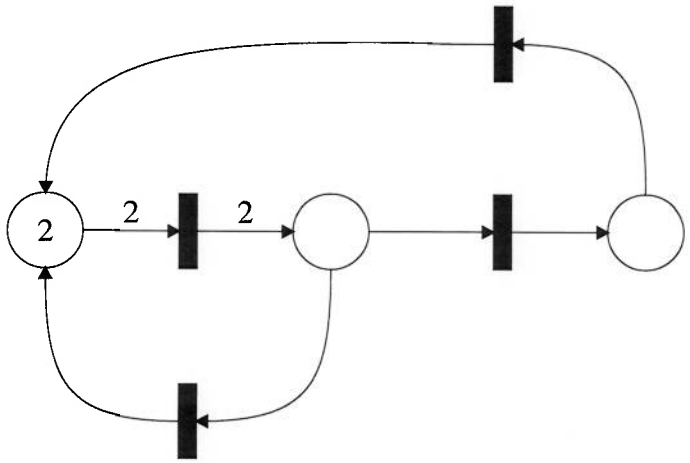

Figura 4.3: Exemplo de $R^{L}/_{T}$.

```
1   using namespace RDPLT;
2   RDPLT rede();
3   RDPLT::Lugar_t L1,L2,L3;
4   RDPLT::Transicao_t T1,T2,T3,T4;
5   RDPLT::Arcos_t A1,A2;
6   L1=rede.addNode<Lugar_t>();
7   L2=rede.addNode<Lugar_t>();
8   L3=rede.addNode<Lugar_t>();
9   T1=rede.addNode<Transicao_t>();
10  T2=rede.addNode<Transicao_t>();
11  T3=rede.addNode<Transicao_t>();
12  T4=rede.addNode<Transicao_t>();
13  A1=rede.addConection<Arco_t>(P1,T1);
14  A2=rede.addConection<Arco_t>(T1,P2);
15  rede.addConection<Arco_t>(P2,T2);
16  rede.addConection<Arco_t>(T2,P3);
17  rede.addConection<Arco_t>(P3,T4);
18  rede.addConection<Arco_t>(T4,P1);
19  rede.addConection<Arco_t>(P2,T3);
20  rede.addConection<Arco_t>(T3,P1);
```

Tabela 4.16: Código para criação de uma $R^{L}/_{T}$.

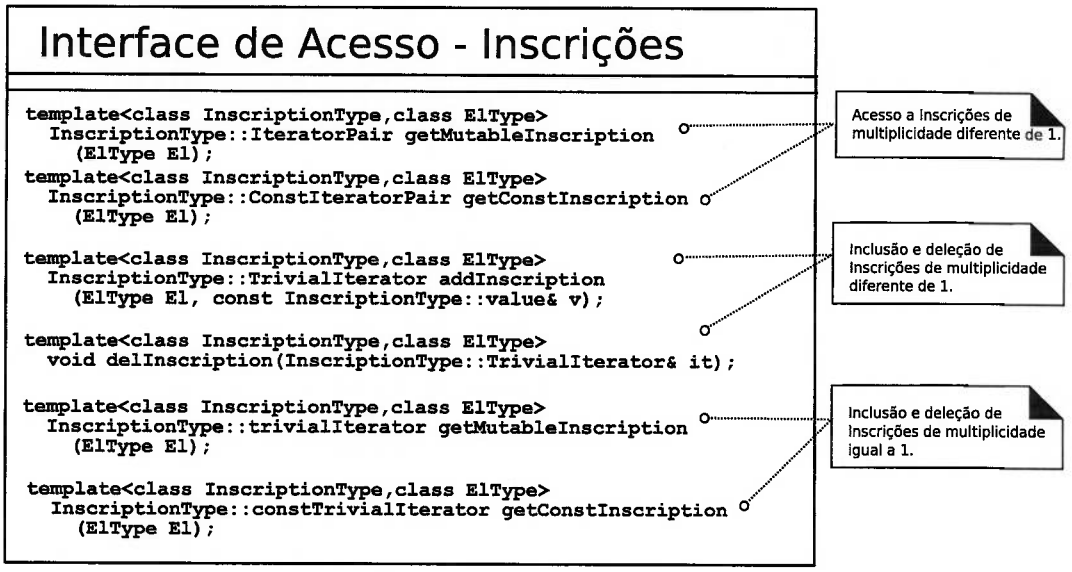

Figura 4.4: Interface de acesso - inserção e deleção de Inscrições.

4.2 Manipulação de inscrições estáticas e dinâmicas

O acesso às inscrições da rede é feito através das funções mostradas na figura 4.4. Para a criação das inscrições na $R^{L}/_{T}$ considerada deve-se então utilizar o código da tabela 4.17.

Para o entendimento desta interface deve-se primeiro compreender o conceito de *IteratorPair* (cf. tabela 4.18).

Segue então a descrição dos métodos desta seção da interface.

- getMutableInscription<InscriptionType>(NodeType E1)

  Caso esta função seja chamada para uma inscrição de multiplicidade igual a 1 ela retorna um tipo que adere ao conceito de *Trivial Iterator* que aponta para o valor da inscrição.

  Caso a função seja chamada para uma inscrição multipla, será retornado um tipo aderente ao conceito *IteratorPair*.

- getConstInscription<InscriptionType>(NodeType E1)

  Semelhante ao método anterior, entretanto os iteradores retornados por este método

```
21  typedef RDPLT::Lugar_t::marcacao_t LugarMarca;
22  typedef RDPLT::Arco_t::multiplicidade_t ArcoMult;
23  (*rede.
24    getMutableInscription<LugarMarca>(L1)) = 2;
25  (*rede.
26    getMutableInscription<LugarMarca>(L2)) = 0;
27  (*rede.
28    getMutableInscription<LugarMarca>(L3)) = 0;
29
30  (*rede.
31    getMutableInscription<ArcoMult>(A1)) = 2;
32  (*rede.
33    getMutableInscription<ArcoMult>(A2)) = 2;
34
35  rede.validate();
36
37  // Causa erro. Inscricao estatica.
38  (*rede.
39    getMutableInscription<ArcoMult>(A1)) = 3;
40
41  // OK. Inscricao dinamica.
42  (*rede.
43    getMutableInscription<LugarMarca>(L1)) = 3;
44
45  rede.mutate();
46
47  // OK. Rede mutavel.
48  (*rede.
49    getMutableInscription<ArcoMult>(A1)) = 3;
```

Tabela 4.17: Código para criação das inscrições em uma $R^{L}/_{T}$.

IteratorPair
Descrição
Um par de *forward iterators*. Eles definem um intervalo válido de elementos.
Notação

- X →Um tipo que é um modelo de Iterator Pair.
- x →Uma instância de X.

Tipos Associados

- X::Iterator →Tipo dos iteradores associados.

Expressões Válidas

- x.begin(); →Retorna o primeiro dos iteradores.
- x.end(); →Retorna o segundoo dos iteradores.

Tabela 4.18: Conceito *IteratorPair*.

são *const iterators*, ou seja, não permitem alterações no valor da inscrição.

- delInscription (it)

  Apaga a inscrição apontada pelo iterador *it*. O iterador se torna inválido após a chamada desta função. Só pode ser chamada para iteradores não constantes. Caso seja utilizada em uma inscrição de multiplicidade única, esta receberá o valor padrão.

- addInscription<InscriptionType>(N1,V1)

  Adiciona uma inscrição no nó N1 de valor V1. Este método só é válido para inscrições múltiplas. Caso o parâmetro V1 seja omitido, a inscrição será criada com o valor padrão.

Um caso de uso comum destes métodos é a verificação do valor de todos os elementos contidos em uma determinada inscrição múltipla. Isto pode ser feito através da construção proposta na tabela 4.19.

```
REDE::Lugar_t::insc_multipla_t::IteratorPair it_p;
REDE::Lugar_t::insc_multipla_t
   ::IteratorPair::iterator it;
typedef REDE::Lugar_t::insc_multipla_t imultp;
it_p = rede.getMutabaleInscription<imultp>(L1);
for(it=it_p.begin();it!=it_p.end();it++) {
  *it = ... /* Altera o valor das inscricoes */
  var = *it /* Le o valor das inscricoes */
}
```

Tabela 4.19: Código proposto para acesso a inscrições múltiplas.

Observe que nesta tabela pretende-se alterar o valor das inscrições. Caso isto não seja necessário utiliza-se o método *getConstInscrition*.

É importante ressaltar que estes iteradores atendem às especificações do conceito *ForwardIterator* (cf. tabela 4.9). Assim, todos os métodos da STL que operam sobre estes podem ser utilizados.

Assim para encontrar, dentro de uma inscrição múltipla, os elementos que atendem determinado predicado basta utilizar o algoritmo *find_if*. Este método utiliza como parâmetros dois iteradores que definem um intervalo válido e um predicado. Ele retorna o primeiro iterador alcançável a partir do início deste intervalo para o qual o predicado é verdadeiro.

Sendo assim basta utilizar o método *getMutableIsncription* para obter o par de iteradores, descrever o predicado desejado e passá-los como parâmetros.

4.3 Simulação

Uma instância só poderá ser simulada quando no estado simulável. A chamada de qualquer um dos métodos aqui descritos com a rede fora deste estado deve produzir uma exceção.

Os métodos presentes para a simulação são os seguintes:

- fireNet(); Dispara todos os nós habilitados da rede em determinado instante. Para o caso de conflito entre nós, a resolução dar-se-á de maneira randômica.
- fire(Node E1); Dispara o nó E1, produzindo alterações nas inscrições de sua localidade de acordo com a definição da meta-rede. Caso E1 não esteja habilitado neste instante este método deve disparar uma exceção.
- getState(); Retorna um objeto que guarda o estado atual da rede, i. e., o valor de todas as inscrições dinâmicas. Todas as funções de busca por elementos de rede podem ser executadas sobre este objeto.
- setState(State ST); Faz com que a rede vá para determinado estado.
- getHabList<NodeType>(); Retorna uma lista contendo todos os nós habilitados da rede de um determinado tipo.

4.4 Busca por elementos

As buscas por elementos conexos é feita usando uma destas funções:

- getConnectList<ConenctionType CT>(Elemento E1); Retorna uma lista contendo todas as conexões do tipo CT ligadas ao Elemento E1.
- getConnectedNodes<ConnectionType CT,NodeType N>(Elemento E1); Retorna uma lista contendo todos os nós do tipo N conectados ao nó E1 por uma conexão do tipo CT.

Mais uma vez o tipo retornado por estas funções é aderente ao conceito de *lista* da STL, permitindo assim que os algoritmos nesta definida sejam utilizados para, por exemplo, criar uma lista contendo apenas os elementos em que determinado predicado seja verdadeiro.

Pode ser necessário o retorno de uma lista de todos os elementos de determinado tipo. Isto é feito através da seguinte função: getNodes<Ntype>().

Para receber uma lista com todas as conexões de determinado tipo pertencentes à rede basta utilizar a função getConnection<Ntype>().

4.5 Serialização

O desenvolvedor terá a sua disposição um método que transforma a rede em uma seqüência de caracteres. Esta seqüência deve ser tal que permita a reconstrução da instância no estado atual de maneira independente do meio de armazenamento utilizado.

Da mesma maneira é necessária a existência de um método que receba esta seqüência de caracteres e construa a rede.

4.6 Considerações Finais

A interface apresentada permite a manipulação de redes de Petri. Esta interface utiliza o compilador C++ para validação dos tipos de conexões permitidas na rede.

A definição completa da interface de armazenamento depende da implementação da interface de acesso pelo desenvolvedor de biblioteca. O objetivo da classe de armazenamento é a efetiva representação na memória, disco, ou outro dispositivo computacional das estruturas de dados necessárias para a interface de acesso. Estas classes não são geradas pela compilação de uma meta-rede.

A implementação da serialização na interface de acesso permite a criação de uma cópia de uma determinada instância, armazenada por uma determinada classe de armazenamento, em uma segunda classe de armazenamento. Para isto a primeira rede é serializada e a seqüência de caractéres gerados é transmitida para a segunda.

A classe de armazenamento não precisa, em tempo de compilação, de nenhuma informação referente ao tipo de rede que nela será armazenada: esta classe deve apenas

permitir o armazenamento dos componentes básicos usados na implementação da interface de acesso.

Desta maneira, a interface de acesso comunica, em tempo de execução, à interface de armazenamento quais as tabelas e relações por necessárias a ela. Isto permite um desacoplamento entre os diferentes tipos de rede e os diferentes tipos de armazenamento.

5 Conclusão

No trabalho foram apresentados os fundamentos de um sistema computacional para manipulação e armazenamento de *RdP*'s. Este sistema permite que diferentes tipos de redes de Petri sejam manipulados através de uma interface única.

Esta interface é mais próxima ao conceito de *RdP* intuitivo de um pesquisador de *RdP*'s do que a disponível quando da utilização direta de estruturas de dados para a representação da rede. Conjectura-se que isto levará a um aumento na produtividade quando do desenvolvimento de softwares que utilizem *RdP*'s.

A interface foi construída de modo a utilizar as características de segurança de tipos da linguagem C++ para garantir a consistência da rede frente às conexões válidas entre seus elementos.

A linguagem de representação da meta-rede é capaz não só de representar as características estruturais como também as dinâmicas de uma classe de *RdP*'s. Ambas estas características são vistas como fundamentais.

Esta linguagem é capaz de representar uma ampla gama de tipos de *RdP*'s, produzindo assim uma nova definição para o termo. A representação de expressões pode, no entanto ser melhorada. Sugere-se como trabalho futuro uma tentativa de extensão à *BNF* afim de representar o contexto de um elemento da rede, uma pesquisa bibliográfica para outras formas de representação de contexto também é aconselhável.

Uma efetiva implementação desta sistema abrirá algumas possibilidades interessantes de trabalho. Pode-se, por exemplo, analisar a produtividade de equipes de trabalho quando sujeitas ao método proposto e compará-la à produtivadade de equipes sujeitas ao

método tradicional de desenvolvimento. Este estudo provavelmente produzirá refinamentos não só no método como também nas interfaces.

A interface de armazenamento só pode ser completamente definida após a implementação da interface de acesso. Sua implementação e formalização completa também são deixadas como trabalho futuro.

Bibliografia

AALST, W.; GRAAF, M. Workflow systems. In: _____. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2003. cap. 25, p. 507–540. ISBN 3-540-41217-4.

AALST, W.; HEE, K. Business process redesign: A petri-net-based approach. *Computers in Industry*, v. 29, n. 1-2, p. 15–26, 1996. Disponível em: <citeseer.ist.psu.edu-/vanderaalst96business.html>.

ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied.* [S.l.]: Addison Wesley, 2001. ISBN 0201704315.

ARMELLINI, F.; ALVES, G. R.; COSTA, J. D. da. *Desenvolvimento de plataforma de controle baseada em redes Mark Flow Graph.* 2002. Trabalho de conclusão de curso EPUSP.

CARDOSO, J. et al. Petri nets for authoring mechanism. In: *XV Simpósio Brasileiro de Informática na Educação (SBIE'2004).* Manaus: [s.n.], 2004.

CORMEN, T. H. et al. *Introduction to Algorithms, Second Edition.* Second edition. [S.l.]: The MIT Press and McGraw-Hill Book Company, 2001. ISBN 0-262-03293-7; 0-07-013151-1.

DAVID, R.; ALLA, H. Petri nets for modeling of dynamic systems: A survey. *Automatica*, v. 30, n. 2, p. 175–202, 1994. ISSN 0005-1098/94.

DEHNERT, J. C.; STEPANOV, A. A. Fundamentals of generic programming. In: JAZAYERI, M.; LOOS, R.; MUSSER, D. R. (Ed.). *Generic Programming.* Springer, 1998. (Lecture Notes in Computer Science, v. 1766), p. 1–11. ISBN 3-540-41090-2. Disponível em: <http://link.springer.de/link/service/series/0558/bibs/1766/17660001.htm>.

DONNELY, C.; STALLMAN, R. M. *Bison Manual: Using the YACC-compatible Parser Generator.* Boston, MA, USA: Free Software Foundation, 2005. ISBN 1-882114-44-2. Disponível em: <http://www.gnu.org/software/bison/manual/pdf/bison.pdf>.

ECKEL, B. *Thinking in C++: Introduction to Standard C++.* Second. Prentice Hall, 2000. ISBN 0139798099. Disponível em: <http://www.mindview.net/Books/TICPP-/ThinkingInCPP2e.html>.

ECKEL, B.; ALLISON, C. *Thinking in C++: Practical Programming.* Prentice Hall, 2003. ISBN 0130353132. Disponível em: <http://www.mindview.net/Books/TICPP-/ThinkingInCPP2e.html>.

EZPELETA, J. Flexible manufacturing systems. In: _____. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2003. cap. 24, p. 479–506. ISBN 3-540-41217-4.

FELDMANN, K. et al. Specification and implementation of logic controllers based on colored petri net models and the standard IEC 1131. II: Design and implementation. *IEEE Trans. on Control Systems Technology*, v. 7, n. 6, p. 666–674, 1999.

FELDMANN, K. et al. Specification, design, and implementation of logic controllers based on colored petri net models and the standard IEC 1131 part I: specification and design. *IEEE Trans. on Control Systems Technology*, v. 7, n. 6, p. 657–665, 1999.

FREY, G. PLC programming for hybrid systems via signal interpreted petri nets. In: *Proceedings of the 4th International Conference on Automation of Mixed Processes ADPM.* Dortmund, Germany: [s.n.], 2000. p. 189–194.

GAMMA, E.; HELM, R.; JOHNSON, R. *Design Patterns: Elements of Reusable Object-Oriented Software.* [S.l.]: Addison-Wesley, 1995. (Addison-Wesley Professional Computing Series). ISBN 0-201-63361-2.

GIRAULT, C.; VALK, R. (Ed.). *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2003. ISBN 3-540-41217-4.

GONÇALVES, E. M. N.; BITTENCOURT, G. Organização de conhecimento de sistemas inteligentes utilizando redes de petri. In: *Anais do XV Congresso Brasileiro de Automática (CBA'2004).* Centro de Convenções da UFRGS, Gramado, RS: [s.n.], 2004.

HAAS, P. J. *Stochastic Petri Nets: modelling,stability, simulation.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2002. (Springer Series in Operations Research). ISBN 0-387-95445-7.

HINZE, R. Generics for the masses. In: *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming.* New York, NY, USA: ACM Press, 2004. p. 236–243. ISBN 1-58113-905-5. Disponível em: <http://doi.acm.org/10.1145/1016850.1016882>.

HOLLOWAY, L. E.; KROGH, B. H.; GIUA, A. A survey of petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, v. 7, p. 151–190, 1997.

IEEE. *854-1987 (R1994) IEEE Standard for Radix-Independent Floating-Point Arithmetic.* New York, NY, USA: IEEE, 1987. 16 p. Revised 1994. ISBN 1-55937-859-X. Disponível em: <http://standards.ieee.org/reading/ieee/std/busarch/854-1987.pdf>.

IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* New York, NY, USA: IEEE, 1985. 20 p. ISBN 1-55937-653-8. Disponível em: <http://-/standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html>.

LAKOS, C. A. From coloured Petri nets to object Petri nets. In: *Proceedings of the Application and Theory of Petri Nets 1995.* Berlin, Germany: Springer-Verlag, 1995. v. 935, p. 278–297. Disponível em: <http://citeseer.ist.psu.edu/lakos95from.html>.

LARMAN, C. *Applying UML and patterns.* Upper Saddle River: Prentice Hall PTR, 1998. ISBN 0-13-748880-7.

LOMAZOVA, I. A. Modeling dynamic objects in distributed systems with nested petri nets. *Fundam. Inf.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 51, n. 1, p. 121–133, 2002. ISSN 0169-2968.

MIYAGI, P. E. *Controle Programável: Fundamentos do controle de sistemas a eventos discreto.* São Paulo, Brasil: Editora Edgard Blücher, 1996.

MOLDT, D.; WIENBERG, F. Multi-agent-systems based on coloured petri nets. In: *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets.* London, UK: Springer-Verlag, 1997. p. 82–101. ISBN 3-540-63139-9.

MUSSER, D. R.; STEPANOV, A. A. Generic programming. In: GIANNI, P. P. (Ed.). *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings.* Berlin, Germany / Heidelberg, Germany / London, UK / etc.: Springer Verlag, 1989. (Lecture Notes in Computer Science, v. 358), p. 13–25. ISBN 3-540-51084-2.

NAKAMOTO, F. Y. *Sistematização do Projeto do Controle de Sistemas Produtivos.* Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, 2002.

NASSI, I.; SHNEIDERMAN, B. Flowchart techniques for structured programming. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 8, n. 8, p. 12–26, 1973. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/953349.953350>.

NETO, J. J. *Introdução à compilação.* Rio de Janeiro, Brasil: LTC, 1987. ISBN 85-216-0483-1.

PETRI, C. A. *Kommunikation mit Automaten.* Tese (Doutorado) — Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

PETRI, C. A. Nets, time and space. *Theoretical Computer Science*, v. 153, n. 1–2, p. 3–48, 1996.

PINNEY, J. W.; WESTHEAD, D. R.; MCCONKEY, G. A. Petri Net representations in systems biology. *Biochem Soc Trans*, v. 31, n. Pt 6, p. 1513–1515, dez. 2003.

PRESSMAN, R. S. *Software Engineering: A Pratictioner's Approach.* [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681.

RAYMOND, E. S. *The Cathedral and the Bazaar.* 2000. Visitado em 2006-02-18. Disponível em: <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>.

ROSSUM, G. van. *Python Reference Manual.* set. 2005. Visitado em 2006-02-18. Disponível em: <http://docs.python.org/ref/ref.html>.

SIEK, J. G.; LEE, L.-Q.; LUMSDAINE, A. *The Boost Graph Library: User Guide and Reference Manual.* [S.l.]: Addison Wesley, 2002. ISBN 0201729148.

Silicon Graphics, Inc. *Standard Template Library Programmer's Guide.* 1994. Visitado em 2006-02-28. Disponível em: <http://www.sgi.com/tech/stl/>.

SOMMERVILLE, I. *Software engineering (6th ed.).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-39815-X.

STROUSTRUP, B. *The C++ Programming Language.* [S.l.]: Addison-Wesley, 1992. ISBN 0-201-53992-6.

USHER, M.; JACKSON, D. A petri net based visual programming language. In: *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'98), 11-14 October 1998, San Diego, USA.* [S.l.: s.n.], 1998. p. 107–112.

VALK, R. Basic definitions. In: _____. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2003a. cap. 4, p. 41–52. ISBN 3-540-41217-4.

VALK, R. Essential features of petri nets. In: _____. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications.* Berlin; Heidelberg; New York; Hong Kong; London; Milan; Paris; Tokyo: Springer, 2003b. cap. 2, p. 9–28. ISBN 3-540-41217-4.

WU, C.; LEE, S. Enhanced high-level petri nets with multiple colors for knowledge verification/validation of rule-based expert systems. *IEEE Transaction on systems, man and cybernetics – Part B: Cybernetics*, v. 27, n. 5, p. 760–773, out. 1997.

XU, D. et al. Modeling and verifying multi-agent behaviors using predicate/transition nets. In: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering.* New York, NY, USA: ACM Press, 2002. p. 193–200. ISBN 1-58113-556-4.