

João Paulo Mestre Pinheiro Ramos e Barros

Modularidade em Redes de Petri

Dissertação apresentada para obtenção do
Grau de Doutor em Engenharia Electrotécnica, especialidade de Sistemas Digitais,
pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa, Julho de 2006

Título: Modularidade em Redes de Petri

Autor: João Paulo Mestre Pinheiro Ramos e Barros

Tese de Doutoramento em Engenharia Electrotécnica, Especialidade de Sistemas Digitais,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Orientador: Professor Doutor Luís Filipe dos Santos Gomes

Candidatura a Doutoramento homologada pelo Conselho Científico da Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa em 19 de Dezembro de 2001

Conclusão: Julho de 2006

© 2006, João Paulo Mestre Pinheiro Ramos e Barros

João Paulo Mestre Pinheiro Ramos e Barros

Modularidade em Redes de Petri

Dissertação apresentada para obtenção do
Grau de Doutor em Engenharia Electrotécnica, especialidade de Sistemas Digitais,
pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa, Julho de 2006

*Para minha mãe e
em memória de meu pai (1936-1984)
e de meu irmão (1962-1999)*

Agradecimentos

*If you love somebody
If you love someone, set them free*
– Sting

Ao longo deste doutoramento consegui manter contacto com vários dos meus colegas ao mesmo tempo que tive o privilégio de conhecer muitos outros e de reencontrar quem há muito não via. A todos tento aqui agradecer. Que me perdoem os esquecidos.

Este trabalho é mais um resultado de uma colaboração, com mais de doze anos, com o Professor Doutor Luís Gomes. Naturalmente, ao longo deste tempo ensinou-me muito mais do que qualquer outro meu professor ou colega, tendo há muito ultrapassado qualquer desses estatutos. Por isso, pelas preciosas e numerosas sugestões e também pela paciência, disponibilidade e atenção que sempre demonstrou para com todos os meus trabalhos o meu muito obrigado.

Algumas instituições com que tive de lidar têm felizmente rostos. Um agradecimento muito particular para o João Paulo Trindade e para a Adelaide Tareco, que souberam compreender o meu abandono do barco, e obviamente também para a Isabel Sofia Brito que nele aceitou entrar por minha causa. Todos foram suficientemente generosos e sensatos para não me considerarem indispensável. Outro agradecimento vai para o Ex.^{mo} Sr. Presidente do Instituto Politécnico de Beja, o Professor José Luis Ramalho, que também demonstrou compreender a importância de permitir dispensas de serviço para doutoramento.

A todos os meus colegas na ESTIG que de alguma forma se tenham, justamente, sentido prejudicados pela minha ausência, e também a todos os que de alguma forma me apoiaram ou gastaram algum do seu escasso tempo a responder aos meus pedidos de ajuda: a Isabel Sofia Brito leu e comentou várias versões preliminares de vários artigos que fui escrevendo, a Maria João Ramos apontou-me vários erros de inglês num *paper* que teve a gentileza de ler, a Teresa Monteiro esclareceu-me as dúvidas ”matemáticas” que lhe coloquei.

De todos os contactos internacionais que tive oportunidade de realizar, o mais importante deveu-se ao Professor Kurt Jensen — actual *chair* do *Steering Committee* da *International Conference On Application and Theory of Petri Nets and Other Models of Concurrency*. A ele devo um agradecimento especial pelo convite que gentilmente me endereçou, e que me muito me hon-

rou, para uma estada junto do seu grupo (*CPN Group*) no Departamento de Informática da Universidade de Aarhus (também conhecido por *DAIMI*), na Dinamarca. Durante essa estada tive a oportunidade de falar pessoalmente com vários membros da sua equipa. Entre eles quero agradecer de forma particular ao Sören Christensen, ao Jens Jørgensen, ao Michael Westergaard e à Lisa Wells, pela afável disponibilidade que sempre demonstraram para me ouvir e pelas seus fundamentados esclarecimentos e opiniões. Embora curta, por razões familiares, a minha estada foi extremamente proveitosa a nível pessoal e profissional. Dela resultaram, em especial, duas publicações em co-autoria com Jens Jørgensen a quem estou especialmente grato pela disponibilidade demonstrada.

Devo outro agradecimento ao Dr. Daniel Moldt. Na sequência do convite do Professor Kurt Jensen para a estada em Aarhus, o Dr. Daniel Moldt desafiou-me para uma visita ao seu grupo em Hamburgo (o *Theoretische Grundlagen der Informatik*). Novamente a seu convite, aí efectuei uma apresentação do meu trabalho junto do seu grupo. Durante essa curta estada, o Dr. Moldt apresentou-me e explicou-me o essencial da sua tese de doutoramento escrita em alemão, que contém resultados ainda únicos na literatura relativamente a redes de Petri coloridas, objectos e *Statecharts*. Um agradecimento também para o Michel Duvigneau e para o Lawrence Cabac que me tiraram todas as dúvidas que eu tinha sobre a ferramenta RENEW e para o professor Rüdiger Valk pelo simpático acolhimento.

Quero também agradecer às instituições que apoiaram o meu doutoramento: ao Ministério da Educação Português através do programa PRODEP, ao Instituto Politécnico de Beja e respectiva Escola Superior de Tecnologia e Gestão, ao Instituto de Desenvolvimento de Novas Tecnologias (UNINOVA), à Universidade de Aarhus na Dinamarca e à Instituição Calouste Gulbenkian.

Os seguintes autores enviaram-me as publicações que lhes solicitei, e ainda mais algumas: Rüdiger Valk, Achilles D. Kameas, Fiorella de Cindio e Ramavarapu S. Sreenivas.

Um agradecimento também a todos os colegas com os quais interagi nos vários eventos internacionais em que participei: pela camaradagem e também porque muitos deles contribuíram, ainda que involutariamente, com ideias e sugestões. Entre eles quero destacar a Laure Petrucci, a Ana Karla Medeiros, o Gonzálo Génova e o Kyller Costa Gorgônio.

Um obrigado à Anikó Costa pela preciosa ajuda logística em várias ocasiões. E outro obrigado especial para o Rui Pais pela realização de um primeiro protótipo de uma das ferramentas propostas nesta dissertação.

Quase por fim, um obrigado a todos os que contribuíram para que o doutoramento não interferisse com a minha Educação e em especial à Elsa, ao Toucinho, e, novamente, à Isabel e ao João Paulo pela companhia (mais virtual do que presencial) e também por me manterem a par das novidades da Escola, mesmo quando não eram simpáticas.

Para a Joana e o André dois grandes beijinhos do pai mais refilão e mais ausente do que o costume. E o maior agradecimento de todos para a Guida, minha mulher e minha melhor amiga, por tudo e também por ter tornado isto possível.

Sumário

Desde o final da década de 60 que se debate a importância da utilização dos denominados métodos formais na Engenharia de Sistemas, em particular na Engenharia de Software. As redes de Petri constituem um método formal bem conhecido e ao qual são reconhecidas vantagens particulares, comparativamente a outras linguagens formais. No entanto, continuam a ser uma linguagem relativamente pouco conhecida fora da respectiva comunidade. Tal deve-se também ao facto de não existir uma quantidade suficiente de técnicas que permitam a sua fácil apreensão e utilização prática por parte de engenheiros sem formação ou experiência especial. Esta dissertação apresenta dois conjuntos de contribuições originais para contrariar esta situação:

1. Uma técnica, duas linguagens a ela associadas, e um tipo de diagramas, denominados *diagramas de adição*, para a especificação de composições e modificações de modelos em qualquer classe de redes de Petri.
2. Um conjunto de idiomas para a criação de modelos em redes de Petri construídos de acordo e com suporte para os principais conceitos do desenvolvimento orientado pelos objectos.

O primeiro conjunto de contribuições é de aplicação geral a qualquer classe de redes de Petri definível na *Petri Net Markup Language* (PNML). Baseia-se no conceito de fusão de nós e define duas operações denominadas *adição* e *subtracção*.

O segundo conjunto de contribuições permite a criação de modelos orientados pelos objectos, utilizando redes de Petri coloridas. Tal permite também uma melhor integração entre os modelos em redes de Petri e as metodologias de desenvolvimento de software orientadas pelos objectos. Em particular, permite especificar os principais conceitos do desenvolvimento orientado pelos objectos em redes de Petri. Finalmente, os diagramas de adição oferecem uma clara relação visual entre redes de Petri e diagramas de classes.

Abstract

Since the late 1960's, there has been a debate about the importance of using the so-called formal methods in Systems Engineering, and especially in Software Engineering. Petri nets are a well-known formal method, which has recognised specific advantages over other formal languages. Yet, Petri nets are still not well-known outside the respective community. One reason for this situation is the lack of techniques allowing engineers, with no especial training or experience, to construct Petri net models.

To counter this situation, this thesis presents two sets of original contributions:

1. One technique, two associated languages, and a diagram type named *addition diagram* for the specification of model compositions and modifications using any Petri net class.
2. A set of idioms allowing the creation of Petri net models, which are able to support the main concepts of object-oriented development.

The first set of contributions is applicable to any Petri net class that can be specified using the Petri Net Markup Language (PNML). It is based on the node fusion concept and defines two operations on nets, named *addition* and *subtraction*.

The second set of contributions allows the creation of object-oriented coloured Petri net models. It offers a better integration between Petri net models and object-oriented methodologies. In particular, it allows the specification of the main object-oriented concepts using a Petri net. Finally, the addition diagrams establish a clear visual relation between Petri nets and class diagrams.

Índice de Matérias

Página de rosto	iii
Agradecimentos	vii
Sumário	xi
Abstract	xiii
Índice de Matérias	xv
Índice de Figuras	xxi
Índice de Listagens	xxvii
Simbologia Utilizada	xxxix

1 Introdução — Utilização de Redes de Petri	1
1.1 Porquê as Redes de Petri?	2
1.2 Construção de Sistemas Correctos	4
1.3 Redes de Petri e Pragmatismo	6
1.3.1 Simplicidade	7
1.3.2 Legibilidade e Adaptabilidade ao Domínio de Aplicação	8

1.3.3	Precisão	8
1.3.4	Suporte Computacional	9
1.3.5	Escalibilidade	9
1.4	Teses Propostas e Contribuições Originais para a sua Defesa	10
1.5	Sobre o Conteúdo e Estrutura da Dissertação	11
1.6	Sobre os Conhecimentos que se Assume o Leitor Possuir	16
1.7	Linguagem e Convenções Utilizadas	18
2	Estruturação e Composição nas Redes de Petri	21
2.1	Estrutura e Abstracção em Redes de Petri	23
2.2	Uma Classificação para os Mecanismos de Estruturação e Composição	24
2.2.1	Composição	26
	Fusão (Composição Externa)	26
	Dobragem (Composição Interna)	29
	Fusão de Nós na Dobragem Baseada nas Marcas	33
2.2.2	Refinamento e Abstracção	36
	Macros	37
	Invocações	41
2.3	Conclusões	46
<hr/>		
I	Composição e Modificação de Modelos em Redes de Petri	47
3	Definição Modular de Modelos	49
3.1	Adição e Subtracção de Modelos	50
3.1.1	Definição Operacional de Redes	51

3.1.2	Construção Operacional de Modelos – a Perspectiva Ascendente	53
3.1.3	Suporte para Estruturação Hierárquica – a Perspectiva Descendente	61
3.1.4	Uma Proposta para a Modularização em Redes de Petri	66
3.2	Conjuntos de Fusão e Colapsos	70
3.3	Operações	73
3.4	Instâncias de Redes e Vectores de Instâncias	76
3.5	Extensão das Operações a Outras Classes de Redes	79
3.5.1	Definições Preliminares	80
3.5.2	Operações em Redes de Petri Anotadas	82
3.5.3	Exemplos de Adições em Redes de Petri Anotadas	86
	Adição de Canais Síncronos	86
	Adição de Árbitro	89
4	Especificação Textual de Redes de Petri	91
4.1	A linguagem PN_{TEXT}	92
4.1.1	Definições em Extensão	92
4.1.2	Definições em Compreensão	93
4.2	Formatos Baseados em XML	93
4.2.1	A linguagem PNML	94
4.2.2	A linguagem <i>Operational PNML</i>	96
	Instâncias de Redes e Vectores de Nós	96
	Conjuntos de Fusão	97
	Adição e Subtracção em OPNML	98
4.3	Um Tradutor de OPNML para PNML	102

II	Redes de Petri no Desenvolvimento Orientado pelos Objectos	107
5	Redes de Petri e Objectos	109
5.1	Redes de Petri Coloridas Componíveis	112
5.1.1	Definição das Redes de Petri Componíveis	113
5.1.2	Adição nas Redes de Petri Coloridas Componíveis	119
5.1.3	Das Redes de Petri Coloridas Componíveis às Redes de Petri Coloridas	120
5.2	Redes de Petri e Modelos Orientados pelos Objectos	126
5.2.1	Redes Canónicas e Adição de Módulos	127
5.2.2	Classificação	133
5.2.3	Atributos	134
5.2.4	Pedidos	134
	Pedidos Assíncronos	135
	Pedidos Síncronos	135
5.2.5	Composição	137
5.2.6	Generalização	139
	Herança na Modelação com Redes de Petri Coloridas Componíveis	140
	Herança <i>versus</i> Composição	141
5.2.7	Classes Genéricas, Navegabilidade, Visibilidade e Invocação Múltipla	143
6	Redes de Petri e Objectos — Dois Exemplos	145
6.1	Grupos de sincronismo para as CPN Tools	146
6.1.1	Especificação de Eventos <i>SEND</i> e <i>RECV</i>	146
6.1.2	Tradução dos Grupos de Sincronismo	147
6.2	Um Exemplo de Tradução de Grupos de Sincronismo na <i>CPN Tools</i>	150
6.2.1	Rede Colorida Hierárquica Equivalente	156
	Classes Iniciais Após Transformação	156

Páginas Pedido	158
6.3 Controlo de Elevadores	162
6.3.1 O Sistema a Modelar	162
6.3.2 Modelo em Rede de Petri Colorida Hierárquica	164
Anotações	164
Comportamento dos Elevadores	165
6.3.3 Redes de Petri na Especificação de Requisitos e no Desenho	169
6.3.4 Modelo em Rede de Petri Colorida Componível	170
Class Initiator	171
Classe Cage	171
Classe LocationIndicator	173
Classe CageButtonAllocation	173
Classe CageButton	175
Classe FloorButtonAllocation	176
Classe FloorButton	176
Classe Door e classe Motor	177
6.3.5 Diagrama de Adição	178
<hr/>	
7 Conclusão	181
7.1 Sobre o que foi feito	181
7.2 Trabalho Futuro	183
7.2.1 Trabalho Futuro já Planeado	183
7.2.2 Outros Trabalhos Futuros	184
A Gramática da Linguagem PN_{TEXT}	187

B Gramática da Linguagem <i>Operational PNML</i>	193
C Redes de Petri para Controladores Digitais	199
C.1 Uma Classe de Redes para Controladores Digitais	200
C.1.1 Um Modelo para o Controlo de Sistemas Conduzidos por Eventos Discretos	201
C.1.2 Redes de Petri Input/Output	202
C.1.3 Especificação em PNML	206
C.2 Sobre a Execução de Modelos	206
C.3 Um Ambiente para Desenvolvimento Utilizando Redes de Petri	208
C.4 Um Controlador para um Parque de Estacionamento	212
D Protótipo para Tradução de Especificações OPNML	215
D.1 Código Fonte do Protótipo <code>opnm12pnml</code>	215
D.2 Exemplos de Modelos em Operational PNML	234
D.2.1 Canal Síncrono (<i>vide</i> Secção 3.5.3 na pág. 86)	234
D.2.2 Adição de Árbitro (<i>vide</i> Secção 3.5.3 na pág. 89)	238
D.2.3 Modelo <code>ParkAI0</code> (<i>vide</i> Secção C.4 na pág. 212)	247
E Gramática Relax NG para Redes de Petri Input-Output	259
F Protótipo para Tradução de Grupos de Sincronismo	265
<hr/>	
Bibliografia	294
Glossário Português-Inglês	295
Glossário Inglês-Português	297
Sobre as citações	299

Índice de Figuras

2.1	Uma classificação dos mecanismos de estruturação para redes de Petri.	25
2.2	a) Uma fila <i>primeiro-a-entrar primeiro-a-sair</i> num parque de estacionamento, b) modelo para uma área ocupada e c) para uma área livre.	28
2.3	Composição, através da fusão de transições, das redes em a) no modelo b).	29
2.4	Dobragem de modelos: a) baseada em marcas e b) baseada em nós.	30
2.5	Composição de três instâncias de uma mesma rede, através da fusão de transições.	33
2.6	Duas transições em duas redes distintas, comunicando por um canal síncrono (ch).	34
2.7	Uma rede equivalente à da Fig. 2.6.	34
2.8	Nomenclatura utilizada na estruturação hierárquica de redes.	37
2.9	Duas formas de especificar a ligação entre um macronó e a respectiva subpágina: a) a nível do arco; b) por fusão de nós.	39
2.10	Submodelos para o parque de estacionamento.	40
2.11	Modelo plano resultante do refinamento do modelo na Fig. 2.5.	41
2.12	Refinamento do modelo de um parque de estacionamento.	41
2.13	Rede Passage para modelação da passagem entre zonas de estacionamento, com detecção da inversão de marcha.	44
2.14	A rede Car	45
2.15	A rede Alarm	46
3.1	Redes Enter , ParkingArea , Leave e Passage	53
3.2	Rede ParkA	54

3.3	Diagrama de adição para a rede ParkA	55
3.4	Modelo resultante da subtração do modelo Enter ao modelo ParkA (Fig. 3.2).	56
3.5	Diagrama de adição para a subtração do modelo Enter ao modelo ParkA (Fig. 3.2).	56
3.6	Diagrama de adição alternativo para o modelo ParkExit	57
3.7	Parque com duas entradas.	58
3.8	Diagrama de adição para o modelo do parque com duas entradas (E2Park).	58
3.9	Diagrama da Fig. 3.8 em que se omitiu a visualização do corpo do modelo ParkExit	59
3.10	Três visualizações para o diagrama de adição do modelo ParkAPassage2	60
3.11	Modelo E3Park2Exit2	62
3.12	Diagrama de adição do modelo E3Park2Exit2	63
3.13	Diagrama de adição do modelo E3Park2Exit2 com ocultação de informação.	64
3.14	Diagrama de adição para a rede ParkA (igual à Fig. 3.3).	64
3.15	Diagrama de adição para a rede ParkA utilizando macronós.	66
3.16	Entrecortamento de sistemas estruturados em módulos (<i>in</i> [Czarnecki e Eisenecker, 2000]).	66
3.17	Diagrama de adição do modelo E3Park2Exit2Counter com ocultação de informação.	68
3.18	Diagrama de adição do modelo E3Park2Exit2Counter com separação de <i>concerns</i>	69
3.19	Modelo para especificação de alternância. Omite-se a marcação.	69
3.20	Diagrama de adição do modelo E3Park2Exit2CounterAlt com separação de <i>concerns</i>	70
3.21	Duas redes a compor de duas formas distintas.	77
3.22	Resultado da adição utilizando dois conjuntos de fusão (R2cf).	77
3.23	Resultado da adição utilizando um conjunto de fusão (R1cf).	77
3.24	Redes Phil e Fork	78
3.25	A representação gráfica do modelo Dinner5	79
3.26	Um exemplo de adição de duas redes anotadas contendo pesos associados aos arcos e marcações associadas aos lugares.	84
3.27	(a) Um árbitro; (b) um modelo base; (c) modelos final resultante da adição de (a) e (b).	90

4.1	Estrutura do tradutor <code>opnml2pnml</code>	103
5.1	a)Um pedido síncrono com parâmetros, e b) a respectiva transição pedido com igual semântica.	124
5.2	Dois idiomas para classes canónicas num rede de Petri colorida componível canónica: (a) sem destruição de objectos e (b) com destruição de objectos.	127
5.3	Módulos para leitura e escrita dos atributos dos objectos.	129
5.4	Classe canónica (abreviada) com destruição, leitura e escrita.	129
5.5	Dois idiomas para modelação do comportamento para cada uma classes canónicas na Fig. 5.2: (a) sem destruição de objectos e (b) com destruição de objectos.	130
5.6	Dois ciclos de vida especificados por módulos adicionáveis a classes canónicas.	132
5.7	Rede canónica com um ciclo de vida com 4 estados.	133
5.8	Sincronização dupla para uma invocação não-atómica.	136
5.9	Sincronização dupla para invocação de uma operação de classe.	137
5.10	a) Composição e b) agregação.	138
5.11	(a) Duas redes de Petri coloridas componíveis canónicas utilizando o idioma na Fig. 5.5b e (b) nova rede resultante de uma relação de herança entre A e B.	144
6.1	Página para a <code>ClassA</code>	148
6.2	Página para a <code>ClassB</code>	148
6.3	<code>ClassA</code> após transformação.	148
6.4	<code>ClassB</code> após transformação.	149
6.5	Página pedido para pedido <code>req1</code> (página <code>req1_classA_t1_classB_t4</code>).	149
6.6	Página pedido para pedido <code>req2</code> (página <code>req2_classB_t6_classA_t3</code>).	150
6.7	Diagrama de classes para o sistema <code>DBReadWrite</code>	151
6.8	Diagrama de adição do modelo do sistema <code>DBReadWrite</code>	152
6.9	Class <code>Writer</code>	153
6.10	Classe <code>Producer</code>	153
6.11	Classe <code>Reader</code>	155

6.12 Classe DB.	155
6.13 Classe <code>ConsumerA</code>	155
6.14 Classe <code>ConsumerB</code>	156
6.15 Diagrama de adição do modelo equivalente, sem grupos de sincronismo, do sistema <i>DBReadWrite</i> . 157	
6.16 Classe <code>Writer</code> após transformação. Deve ser comparada com o modelo inicial na Fig. 6.9. 158	
6.17 Classe <code>Producer</code> após transformação. Deve ser comparada com o modelo inicial na Fig. 6.10.	158
6.18 Classe <code>Reader</code> após transformação. Deve ser comparada com o modelo inicial na Fig. 6.11. 159	
6.19 Classe DB após transformação. Deve ser comparada com o modelo inicial na Fig. 6.12. . . 159	
6.20 Classe <code>ConsumerA</code> após transformação. Deve ser comparada com o modelo inicial na Fig. 6.13.	159
6.21 Classe <code>ConsumerB</code> após transformação. Deve ser comparada com o modelo inicial na Fig. 6.14.	160
6.22 Página pedido <code>reqWrite</code>	160
6.23 Página pedido <code>reqRead</code>	160
6.24 Página pedido <code>reqGet</code>	161
6.25 Página pedido <code>reqPut</code> para classe <code>ConsumerA</code>	161
6.26 Página pedido <code>reqPut</code> para classe <code>ConsumerB</code>	161
6.27 Página Do Cage Cycle.	166
6.28 Página Requests.	168
6.29 Página UpDown.	168
6.30 Diagrama de adição do controlador de elevadores utilizando uma rede de Petri colorida hierárquica.	169
6.31 Diagrama de classes do controlador de elevadores.	172
6.32 Classe <code>Initiator</code>	172
6.33 Classe de sistema <code>Cage</code>	174
6.34 Classe de sistema <code>LocationIndicator</code>	175

6.35	Classe de sistema CageButtonAllocation	175
6.36	Classe de sistema CageButton	176
6.37	Classe de sistema FloorButtonAllocation	177
6.38	Classe de sistema FloorButton	177
6.39	Classe de sistema Door	178
6.40	Classe de sistema Motor	178
6.41	Diagrama de adição do controlador de elevadores utilizando uma rede de Petri colorida componível.	179
C.1	A arquitectura do ambiente de desenvolvimento.	210
C.2	Três redes de Petri <i>input-output</i> : EnterIO , ParkingAreaIO e LeaveIO	212
C.3	Modelo ParkAIO	213
D.1	Representação gráfica obtida automaticamente a partir do modelo na Listagem D.9 e correspondente ao modelo na Fig. 2.7 da pág. 34. Como anotações, foram utilizados os atributos name dos nós, em lugar dos respectivos identificadores.	239
D.2	Representação gráfica obtida automaticamente a partir do modelo na Listagem D.13 e correspondente ao modelo na Fig. 3.27c da pág. 90. Como anotações, foram utilizados os atributos name dos nós, em lugar dos respectivos identificadores.	247
D.3	Representação gráfica obtida automaticamente a partir do modelo na Listagem D.18 e correspondente ao modelo na Fig. C.3 da pág. 213. Como anotações, foram utilizados os atributos name dos nós, em lugar dos respectivos identificadores.	257

Índice de Listagens

3.1	Estrutura de transformação para o tipo de anotação <i>ColorSimple</i>	88
4.1	Código PNML para o modelo Enter na Fig. 3.1 da pág. 53.	94
4.2	Sintaxe dos elementos transitionFusion e placeFusion na gramática da linguagem Operational PNML.	97
4.3	Sintaxe dos elementos placeFusionVector e transitionFusionVector na gramática da linguagem Operational PNML.	98
4.4	Definição da rede ParkA (Fig. 3.2) em OPNML.	99
4.5	Definição da rede ParkExit a partir de uma rede ParkA já definida em OPNML e de uma rede Enter em PNML.	100
4.6	Definição da rede ParkAPassage2 a partir de uma rede ParkA já definida em OPNML e de duas instâncias da rede Passage , também definida em PNML.	101
4.7	Classe NetFusion do módulo principal (opnml2pnml.rb) à qual são adicionadas as funções de transformação, para as anotações nas redes.	103
4.8	Classe NodeFusion do módulo principal (opnml2pnml.rb) à qual são adicionadas as funções de transformação para as anotações nos nós.	103
4.9	Classe IPairsFusion à qual são adicionadas as funções de transformação para os arcos entre nós interface.	104
A.1	Exemplos de instruções permitidas pela linguagem <i>PN_{TEXT}</i>	187
A.2	Gramática da linguagem <i>PN_{TEXT}</i> em JavaCC.	188
B.1	Gramática da linguagem <i>Operational PNML</i>	193
D.1	Código fonte do componente principal (opnml2pnml.rb) do protótipo opnml2pnml	215
D.2	Código fonte para a estrutura de transformação para redes de Petri lugar-transição, em particular para o PNTD ptNetb	227

D.3	Código fonte para uma estrutura de transformação ColorSimple referida na Secção 3.5.3 (pág. 86).	230
D.4	Código fonte para a estrutura de transformação ptTest referida na Secção 3.5.3 (pág. 89).	231
D.5	Código fonte para a estrutura de transformação ptio referida na Secção C.4 (pág. 212).	232
D.6	Especificação PNML para o modelo Producer na Fig. 2.6 da pág. 34.	234
D.7	Especificação PNML para o modelo Consumer na Fig. 2.6 da pág. 34.	235
D.8	Especificação OPNML para o modelo na Fig. 2.7 da pág. 34.	236
D.9	Especificação PNML produzida pelo protótipo opnml2pnml a partir da especificação na Listagem D.8 e da estrutura de transformação na Listagem D.3.	237
D.10	Especificação PNML para o modelo Arbiter na Fig. 3.27a da pág. 90.	238
D.11	Especificação PNML para o modelo Par na Fig. 3.27b da pág. 90.	240
D.12	Especificação OPNML para o modelo na Fig. 3.27c da pág. 90.	243
D.13	Especificação PNML produzida pelo protótipo opnml2pnml a partir da especificação na Listagem D.12 e da estrutura de transformação na Listagem D.4 na pág. 231.	243
D.14	Especificação PNML para o modelo EnterIO na Fig. C.2 da pág. 212.	247
D.15	Especificação PNML para o modelo ParkingAreaIO na Fig. C.2 da pág. 212.	249
D.16	Especificação PNML para o modelo LeaveIO na Fig. C.2 da pág. 212.	250
D.17	Especificação OPNML para o modelo ParkAIIO na Fig. C.3 da pág. 213.	252
D.18	Especificação PNML produzida pelo protótipo opnml2pnml a partir da especificação na Listagem D.17 e da estrutura de transformação na Listagem D.5 na pág. 232.	252
E.1	Gramática para redes de Petri <i>input-output</i> em Relax NG.	259
F.1	Código iniciador do protótipo ccpn2hcpn .	267
F.2	Classe CPNet (protótipo ccpn2hcpn).	267
F.3	Classe Page (protótipo ccpn2hcpn).	271
F.4	Classe Transition (protótipo ccpn2hcpn).	272
F.5	Classe Place (protótipo ccpn2hcpn).	276
F.6	Classe FusionElement (protótipo ccpn2hcpn).	278

F.7	Classe <code>Expression</code> (protótipo <code>ccpn2hcpn</code>).	278
F.8	Classe <code>Request</code> e classe <code>RequestKey</code> (protótipo <code>ccpn2hcpn</code>).	279
F.9	Classe <code>IdGenerator</code> (protótipo <code>ccpn2hcpn</code>).	279
F.10	Método <code>Error</code> (protótipo <code>ccpn2hcpn</code>).	280

Simbologia Utilizada

Parte da simbologia utilizada é usual na teoria de conjuntos e como tal dispensa-se a sua descrição. Por essa razão, apresentam-se apenas as notações menos comuns, ou específicas deste trabalho, e que podem por essa razão suscitar dúvidas quanto à sua interpretação. Quando existem, indicam-se as definições directamente relacionadas. Os livros [Arbib et al., 1981] e [Eccles, 1997] constituíram os auxiliares para a simbologia não original.

$\mathcal{P}(C)$ O conjunto de todos os sub-conjuntos do conjunto C ou conjunto potência (*powerset* em inglês).

$x \mapsto y$ Existe uma função que aplica x em y . Sendo f essa função, teremos $y = f(x)$.

$f : A \rightharpoonup B$ f é uma função parcial que aplica um subconjunto de A no conjunto B : $\text{dom}(f) \subseteq A$.

\perp Valor fora do domínio de uma função parcial. Uma função parcial $f : A \rightharpoonup B$, pode ser vista como uma função (total) $f^\perp : A \rightarrow B \cup \{\perp\}$ definida pela regra

$$f^\perp(a) = \begin{cases} f(a) & \text{se } a \in \text{dom}(f) \\ \perp & \text{se } a \notin \text{dom}(f) \end{cases}$$

$f(A)$ com $f : A \rightharpoonup B$ $f(A) = \bigcup_{a \in \text{dom}(f)} f(a)$.

A^*, A^+, A^n $A^* = \bigcup_{n \geq 0} A^n$ e $A^+ = \bigcup_{n > 0} A^n$, em que $A = \{e_1, \dots, e_n\}$, $A^0 = \{()\}$, $A^1 = \{(e_1) \mid \{e_1\} \subseteq A\}$, $A^2 = \{(e_1, e_2) \mid \{e_1, e_2\} \subseteq A\}$, \dots , $A^n = \{(e_1, \dots, e_n) \mid \{e_1, \dots, e_n\} \subseteq A\}$.

$/x_1/\dots/x_n/$	Conjunto de fusão (<i>vide</i> Def. 3.4 na pág. 71).
$/x_1/\dots/x_n/\rightarrow r$	Conjunto de fusão nominativo. Os nós x_1, \dots, x_n são fundidos num nó r (<i>vide</i> Def. 3.6 na pág. 71).
$R(CF_1, \dots, CF_n)$	Colapsagem da rede R pelo colapso (CF_1, \dots, CF_n) (<i>vide</i> Def. 3.16 na pág. 74).
$(R_A + R_B)(CI)$	Adição das redes R_A e R_B pelo colapso de interface CI (<i>vide</i> Def. 3.20 na pág. 75).
$(R_A - R_B)(CI)(CR)$	Subtração da rede R_B à rede R_A pelo colapso de interface CI e colapso de remoção CR (<i>vide</i> Def. 3.21 na pág. 75).
$e^S_{ped} = e^R$	Os eventos e^S e e^R são compatíveis, ou seja, têm igual pedido associado (<i>vide</i> Def. 5.2 na pág. 115).
$e^S \rightleftharpoons e^R$	Os eventos compatíveis e^S e e^R são comunicantes, e o primeiro invoca o segundo. (<i>vide</i> Def. 5.8 na pág. 119).
$Var(exp)$	A função Var devolve o conjunto das variáveis presentes na expressão algébrica exp .
$\langle var = val \rangle$	Vínculo da variável var pelo valor val . Por vezes denominado apenas vínculo.
$exp \langle var = val \rangle$	Valor da expressão exp após aplicação do vínculo $\langle var = val \rangle$. Tipicamente, a variável var ocorre na expressão exp : $var \in Var(exp)$.
$exp \langle V_1, \dots, V_n \rangle$	Valor da expressão exp após aplicação dos vínculos ou conjuntos de vínculos V_1, \dots, V_n .

Capítulo 1

Introdução — Utilização de Redes de Petri

Jácome era um inventor de coisas impossíveis: tinta invisível, formigas mecânicas, pássaros a vapor, sapatos voadores, aparelhos de produzir espirros. Não se podia dizer dele que não tinha imaginação – tinha e de sobra. Não se podia dizer que não fosse trabalhador – Jácome trabalhava o dia inteiro. O problema era que nada do que ele inventava parecia ter utilidade.

– José Eduardo Agualusa in *Estranhões, Bizarrocos e outros seres sem exemplo*

O documento que acabámos por utilizar mais foi a rede de Petri de Erna. Ela mostrava como todas as peças do puzzle estavam relacionadas e como eram obrigadas a interagir. (...) Um dos meus colegas, Jut Kodner, observou que o diagrama era uma melhor especificação do que a própria especificação.

– Tom DeMarco sobre a sua experiência num projecto de software e hardware na Bell Telephone Laboratories entre 1963 e 1964.

Todos os ramos da engenharia utilizam muitas outras linguagens para além das linguagens naturais. Essas linguagens servem muitos fins distintos, nomeadamente para documentar e comunicar vários tipos de informação e para especificar vários tipos de sistemas e processos que se pretendem estudar ou implementar. É usual dividir estas linguagens em textuais e gráficas, dependendo da ênfase que cada uma dá à utilização de texto ou de diagramas, respectivamente. A maioria das notações matemáticas e das linguagens de programação de computadores são exemplos de linguagens textuais. Apesar de existirem argumentos conhecidos contra as linguagens gráficas, em particular os enunciados por Dijkstra [Dijkstra, 1993], este tipo de linguagens é largamente utilizado e reconhecido como extremamente útil. A história da *Unified Modeling Language* (UML) [OMG, 2003], da *System Description Language* (SDL) [Doldi, 2003; SDL Fo-

rum Society, 2005] e dos diagramas de sequências de mensagens (*Message Sequence Diagrams* ou *MSC*) [Genest et al., 2004; SDL Forum Society, 2005] prova-o de forma clara¹. Aparentemente, as linguagens gráficas são frequentemente vistas como inerentemente menos precisas, mas tal não é verdade: uma linguagem textual pode não ser precisa e uma linguagem gráfica pode sê-lo [Harel e Rumpe, 2004]. Neste último caso os **estadogramas** de Harel (*statecharts*) [Harel, 1987, 1988; Harel e Naamad, 1996] e as redes de Petri [Petri, 1962] constituem exemplos particularmente significativos.

Esta dissertação propõe novas formas de utilizar as **redes de Petri**, uma linguagem gráfica conhecida desde a década de 1960. Dada a sua grande versatilidade e elevado grau de abstracção, a definição inicial das redes de Petri foi dando origem a um conjunto de linguagens gráficas mais específicas aplicáveis na modelação do comportamento de numerosos tipos de sistemas, correspondentes a novas áreas de aplicação. Tal deveu-se fundamentalmente a duas características das redes de Petri:

1. Baseiam-se na modelação explícita de conceitos extremamente genéricos e abstractos: estados e acções.
2. Suportam de forma ainda legível, diferentes e variados tipos de extensões capazes de as adaptar à modelação de diferentes tipos de sistema.

Apesar desta versatilidade das redes de Petri, e sem prejuízo da eventual aplicação a outras áreas do conhecimento, as técnicas que se propõem neste trabalho centram-se na modelação de sistemas que se pretendam implementar em software. Assim, a dissertação enquadra-se predominantemente na **engenharia de software**, uma área onde existem numerosas outras linguagens gráficas. Entre elas, refiram-se, a título exemplificativo, a representação gráfica de máquinas de estado, os fluxogramas, a *Unified Modeling Language* (UML), os diagramas de sequências de mensagens, os estadogramas, a *System Description Language* (SDL) e muitas outras (*vide*, por exemplo, os seguintes artigos de *survey* [Wieringa, 1998; Sgroi et al., 2000; Gomes et al., 2005]). Perante este cenário justifica-se a pergunta: porquê as redes de Petri?

1.1 Porquê as Redes de Petri?

Tal como para qualquer outra linguagem, a justificação última da utilização das redes de Petri num qualquer projecto de software deverá resultar das vantagens relativas que apresenta para esse projecto concreto quando comparada com outras linguagens ou outras técnicas. Nesse sentido, as redes de Petri devem ser vistas como apenas mais uma das linguagens que um

¹É também interessante notar que as linguagens gráficas podem ser geradas a partir de uma descrição textual: por exemplo, em [Spinellis, 2003] e [Fowler, 2004] são apresentados argumentos a favor da utilização de diagramas de classe (UML) definidos de forma textual.

engenheiro de software deve conhecer de forma a alargar o seu leque de escolhas, podendo eventualmente decidir pela sua utilização quando considerar ser essa a opção adequada.

As vantagens na utilização das redes de Petri resultam das suas características fundamentais, as quais podem mostrar-se particularmente adequadas na construção de uma grande variedade de modelos de comportamento. Estas características são listadas em seguida com base na classificação proposta por Rüdiger Valk [Valk, 2003]:

Dualidade estado-transição As redes de Petri contêm dois conjuntos disjuntos de elementos: **lugares** e **transições**. As entidades do mundo real ou, para utilizar a terminologia de Michael Jackson [Jackson, 2002], do "domínio do problema" que se pretendem interpretar como objectos passivos são modeladas por lugares. Nestes podemos incluir os estados, recursos, condições, canais, *buffers*, operandos, etc.. As entidades activas (acções, eventos, execução de instruções, operações, mudanças de estado, envio de mensagens, etc.) são modeladas por transições. Desta forma é dado um tratamento igualitário aos conceitos de estado e transição. Para além dos lugares e transições, apenas existem os arcos que modelam as dependências entre ambos.

Efeito local das transições O efeito de cada transição é puramente local, ou seja, apenas afecta a própria transição e os lugares a que se encontra directamente ligada. Este conjunto de nós é aqui designado por **localidade da transição**.

Concorrência implícita do modelo As transições com localidades disjuntas (sem nenhum lugar ligado a mais do que uma dessas transições) podem **ocorrer** (também se dirá **disparar**) independentemente.

Representação gráfica Os lugares são representados por círculos ou elipses. As transições são representadas por barras, paralelepípedos ou quadrados. Os arcos ligam as transições aos restantes elementos na sua localidade. Estes são sempre lugares. É também possível adicionar anotações (textuais) a esta representação gráfica.

Representação algébrica As redes de Petri de baixo-nível prestam-se a uma representação algébrica simples: para cada representação gráfica existe uma representação algébrica que contém igual informação, nomeadamente quais os lugares, transições, arcos e anotações.

A escolha das redes de Petri como objecto deste trabalho tem por base o objectivo geral de contribuir para que as redes de Petri sejam mais reconhecidas e utilizadas por engenheiros sem formação específica. Conforme lembrado num artigo de Stuart Shapiro [Shapiro, 1997], já em 1976, Ronald Jeffries dizia numa carta à revista SIGPLAN notices:

"necessitamos de abordagens ao desenho as quais, nas mãos de meros mortais, resultem em programas que funcionem." [Jeffries, 1976].

Esta dissertação pretende ser um passo no sentido das redes de Petri serem de facto incluídas nesse conjunto de abordagens. Provavelmente, tal só será conseguido totalmente se as redes de Petri forem finalmente escolhidas pelo consórcio Object Management Group (OMG) como uma das linguagens constituintes da Unified Modeling Language (UML) [OMG, 2003] ou, pelo menos, da emergente *Systems Modeling Language* [SysML Partners, 2005]. Só dessa forma será possível vencer um conjunto de resistências à utilização de redes de Petri. Estas resistências são provavelmente diversas, mas as seguintes parecem ser as mais importantes:

- A identificação das redes de Petri com um método formal o que embora correcto afasta de forma imediata os engenheiros de software que não reconhecem utilidade prática nos denominados métodos formais [Bowen, 2003].
- O desconhecimento muito generalizado das redes de Petri. Na experiência do autor, grande parte dos alunos de engenharia e mesmo de recém licenciados em Portugal e no estrangeiro desconhece totalmente a existência de redes de Petri ou, mais frequentemente, conhece o nome mas identifica-o apenas com redes de Petri de baixo-nível.

Perante esta situação *de facto*, e reconhecendo o autor especial valor nas redes de Petri, esta dissertação apresenta propostas concretas que visam generalizar e simplificar a aplicação de redes de Petri na especificação de sistemas.

No âmbito do objectivo geral da construção de sistemas correctos, a secção seguinte enquadra a utilização dos métodos e técnicas respectivos numa perspectiva pragmática que recomenda a utilização quer de métodos formais quer de métodos informais ou semiformais. Seguidamente, argumenta-se que as redes de Petri oferecem um bom suporte para esta perspectiva. Apontam-se também as formas pelas quais as redes de Petri podem cumprir os requisitos desejáveis a uma linguagem ou método que se pretenda utilizável pelo maior número possível de profissionais.

1.2 Construção de Sistemas Correctos

A qualidade do software continua a constituir um enorme problema. Ao mesmo tempo, a quantidade de equipamentos que contêm software continua a crescer, ao ponto destes poderem ser vistos como praticamente ubíquos na sociedade moderna. Tal traduz-se no facto da qualidade do software afectar não apenas os tradicionais sistemas críticos, como por exemplo equipamento militar ou médico, mas também vários dispositivos que são utilizados no dia a dia. Nestes incluem-se variados tipos de sistemas embutidos, tipicamente constituídos por software e hardware. Estes cobrem todas as actividades correntes, desde as ligadas a actividades profissionais às praticadas nos tempos livres.

O conjunto de práticas, metodologias, teorias e até ferramentas utilizadas no desenvolvimento de software costuma ser catalogado sob a designação lata de **engenharia de software**, um termo

com origem numa famosa conferência, organizada pela Organização do Tratado do Atlântico Norte (OTAN²), e que teve lugar em Garmisch na Alemanha em 1968 [s.a., 1969]:

”A expressão ‘engenharia de software’ foi deliberadamente escolhida como provocatória, por implicar a necessidade da manufactura de software ser baseada no tipo de fundamentos teóricos que são tradicionais em ramos estabelecidos da engenharia.” [s.a., 1969].

Por outras palavras, a engenharia de software pode ser definida de uma forma semelhante a outra engenharia e como tal corresponde à aplicação de conhecimento científico para o bem-comum. Uma definição muito mais recente e particularmente interessante por distinguir *Computer Science* (o termo anglo-saxónico para Informática) de engenharia de software, pode ser encontrada na última proposta da *Joint Task Force for Computing Curricula 2005* para currículos de cursos superiores de Informática e de Computadores [The Joint Task Force for Computing Curricula 2005, 2005]:

Enquanto que a Informática³ (tal como outras ciências) foca a criação de novos conhecimentos, a engenharia de software (tal como outras engenharias) foca métodos rigorosos para desenhar e construir coisas que fazem o que são supostas fazer de forma confiável.” [The Joint Task Force for Computing Curricula 2005, 2005].

Os numerosos métodos e técnicas que têm sido, e continuam a ser, desenvolvidos de forma a contribuir para a qualidade do software fazem por isso parte da engenharia de software. Tal como nos outros ramos da engenharia, estes métodos têm por objectivo a construção de sistemas correctos, ou por outras palavras, de sistemas que implementam todos os requisitos para eles definidos e que funcionam sem falhas ao longo do seu tempo de vida.

No entanto, conforme frequentemente apontado (e.g. [Reinhold Plösch, 2004]), muitos destes métodos não são utilizados pelos típicos engenheiros de software, em projectos típicos⁴. As razões apontadas, também por Plösch, são duas: (1) o engenheiro típico não possui as capacidades matemáticas exigidas por esses métodos; (2) o escopo de aplicação do método ou técnica é demasiado limitado.

Com aparente início na segunda conferência da Organização do Tratado do Atlântico Norte sobre Engenharia de Software realizada em Roma em 1969, mas especialmente entre o início da década de setenta e o final da década de oitenta, foi visível uma acesa discussão entre defensores da utilização de métodos matemáticos no desenvolvimento de software, e defensores da impossibilidade de aplicação efectiva desses métodos. O artigo de Stuart Shapiro [Shapiro, 1997] faz

²Em inglês: *NATO*.

³No original: *computer science*.

⁴Plösch utiliza o termo “average engineer”.

um resumo muito interessante deste "conflito" entre cientistas e engenheiros, ainda latente nos dias de hoje:

"Claramente, porém, a posição que as pessoas tomaram no que respeitava à questão teste *versus* verificação formal era, pelo menos parcialmente, uma função de como elas próprias se viam. Quem se considera um cientista pode desenvolver uma visão muito diferente de quem se considera um engenheiro. Por vezes, a posição que escolhemos determina a nossa opinião⁵." [Shapiro, 1997].

Embora já em 1978 Parnas surja a defender a necessidade de utilização quer da verificação formal quer do teste [Parnas, 1978], apenas no início da década de noventa tal parece tornar-se a opinião mais aceite. Surgem então vários autores que referem a necessidade de utilizar quer métodos formais, quer métodos informais (ou semi-formais), sublinhando a necessidade de integração entre ambos (e.g. [Leveson, 1990] e [Gerhart, 1990]).

Na secção seguinte argumenta-se que as redes de Petri se enquadram, com grande versatilidade, nesta perspectiva pragmática da engenharia de software que hoje parece dominante.

1.3 Redes de Petri e Pragmatismo

Tal como todos os denominados métodos formais, as redes de Petri constituem um exemplo claro de um dos métodos disponíveis que, tipicamente, não são utilizados por um engenheiro típico seja em que projecto for. As razões parecem ser exactamente as duas apontadas por Plösch. Com efeito, as redes de Petri são quase sempre apresentadas como um "objecto matemático" ou uma linguagem formal com um pequeno escopo de aplicabilidade.

Ora a verdade é que as já apontadas características fundamentais das redes de Petri são úteis, quer quando o objectivo é a verificação formal de sistemas, quer quando o objectivo é a construção de modelos executáveis que permitam testar ou simular sistemas de forma não exaustiva.

Essas características permitem a criação de modelos legíveis, expressivos e com uma semântica precisa. Tal é útil para qualquer modelo e constitui uma vantagem clara relativamente a outras linguagens gráficas menos precisas e portanto necessariamente mais problemáticas em termos da sua efectiva compreensão. Um exemplo claro é o dos diagramas de actividades da linguagem UML. Já na sua segunda versão continuam a apresentar uma semântica ambígua que motivou já tentativas de a precisar, nomeadamente utilizando redes de Petri (e.g. [Barros e Gomes, 2003a; Störrle, 2003; Störrle e Hausmann, 2005]).

A tese de doutoramento de Carl Adam Petri [Petri, 1962] tem a data de 1962. As redes de

⁵No original: *Where you sit sometimes determines where you stand.*

Petri são portanto praticamente contemporâneas das linguagens de programação de alto-nível. Curiosamente, o primeiro departamento universitário de Informática⁶ surge também em 1962 [Rice e Rosen, 2004]. Desde essa data as redes de Petri foram estudadas e estendidas com novos conceitos, como, por exemplo, arcos com diferentes semânticas, modelação de tempo e modelação de estruturas de dados e respectivo processamento. Mas, infelizmente, as redes de Petri continuam a ser frequentemente vistas como uma extensão às máquinas de estados, sem capacidade de composição ou estruturação.

Vejamos, então, de que forma as redes de Petri cumprem os requisitos desejáveis para uma linguagem formal: simplicidade, legibilidade e adaptabilidade ao domínio da aplicação, precisão, suporte computacional e escalabilidade. Estes requisitos incluem os apontados em [Reinhold Plösch, 2004] (simplicidade, precisão, generalidade e expressividade) e também os desejáveis noutros métodos, técnicas e ferramentas computacionais para Engenharia de Software. Em particular, correspondem também a características desejáveis das linguagens de programação⁷.

1.3.1 Simplicidade

As redes de Petri podem ser tão sofisticadas quanto se queira. De facto as redes de Petri de baixo-nível apresentam sintaxes e semânticas muito simples. Tal também implica a frequente necessidade de construir modelos de grandes dimensões. Mas, dada a facilidade com que se podem estender, as redes de Petri adaptam-se facilmente à modelação de vários tipos de sistema. Em resumo, a complexidade da sintaxe e semântica das redes de Petri pode ser facilmente adaptada à complexidade do sistema a modelar.

Tal é especialmente verdadeiro quando se encaram as redes de Petri como uma linguagem de especificação e simulação e não como um método formal que permita determinados tipos específicos de verificação. Caso se pretenda utilizar as redes de Petri para verificação então deve optar-se por uma classe de redes bem conhecida que garanta a verificação, com uma complexidade aceitável, das propriedades que interessem demonstrar. Caso se pretenda utilizar as redes de Petri para especificação e simulação existe uma muito maior liberdade relativamente às extensões a utilizar. Esta dissertação enquadra-se neste segundo tipo de utilização. Também neste caso, as características fundamentais são ainda extremamente úteis pois servem de base à definição de uma linguagem precisa e fundamentada por conceitos bem estudados.

⁶Pelo menos nos Estados Unidos da América.

⁷Por exemplo, em [Sebesta, 1996] são apontadas a *legibilidade* que inclui a *simplicidade* e a *ortogonalidade*; a *reliability* garantida pela precisão, suporte computacional e escalabilidade; e o *custo* que é uma função de todas estas características.

1.3.2 Legibilidade e Adaptabilidade ao Domínio de Aplicação

A adaptabilidade, atrás sublinhada, permite a adaptação das redes de Petri a domínios específicos. Com efeito, as redes de Petri têm sido utilizadas com sucesso em vários domínios de aplicação, desde os sistemas *workflow* (e.g. [van der Aalst e van Hee, 2002]) à modelação de hardware (e.g. [Yakovlev et al., 2000]), passando pela especificação de software, sistemas de manufactura e sistemas de telecomunicação (e.g. [Jensen, 1997c; Girault e Valk, 2003]).

Outro domínio de aplicação é sem dúvida o ensino. Aí as redes de Petri são particularmente úteis para o ensino dos conceitos fundamentais de sistemas concorrentes [Barros, 2002], e de sistemas distribuídos e protocolos de comunicação [Christensen e Jørgensen, 2004]. Também no ensino do desenho de sistemas concorrentes em hardware têm sido utilizadas redes de Petri [Gomes e Costa, 2004]. O artigo [Christensen e Jørgensen, 2004] também se debruça sobre o ensino das próprias redes de Petri e foi possível encontrar um outro artigo que apresenta uma utilização de modelos de redes de Petri no ensino [Berthelot e Petrucci, 2001].

Em resumo, em várias áreas de aplicação, as redes de Petri já demonstraram serem capazes de diminuir o fosso de que falava Pamela Zave em 1996:

”O fosso conceptual entre os domínios de aplicação e a Matemática deve ser ultrapassado através da construção de modelos matemáticos dos domínios de aplicação. Dentro de um modelo apropriado, a linguagem formal é alargada de forma a incluir o vocabulário e as relações no domínio. Por outro lado, a ausência de modelos apropriados, constitui um grande obstáculo à utilização de métodos formais num domínio de aplicação.” Pamela Zave *in* [Hall et al., 1996].

1.3.3 Precisão

Existe uma enorme quantidade de classes de redes de Petri já propostas. Mas curiosamente, e provavelmente por uma questão cultural que de alguma forma enquadra a comunidade das redes de Petri, esses autores sentem quase sempre a necessidade de definir a sintaxe e semântica da nova classe de redes de forma precisa. Portanto, mesmo quando as extensões impedem a aplicação de determinados métodos de análise, mesmo quando surgem extensões mais ou menos originais, é frequente encontrar uma preocupação com a definição precisa da sintaxe e semântica dessas propostas. Tipicamente, temos uma linguagem gráfica com as características fundamentais das redes de Petri, também definida de forma precisa.

Essas mesmas características fundamentais e esta tradicional precisão na definição das classes de redes, tem também motivado a utilização de redes de Petri na formalização de linguagens semiformais e sem especificações precisas. Vejam-se por exemplo os trabalhos de Bernardi *et al.* [Bernardi et al., 2002] e de Baresi e Pezzè [Baresi e Pezzè, 2001]. Em resumo, pode-se afirmar

que as redes de Petri oferecem, por natureza e por razões históricas e culturais, um suporte versátil para a definição de linguagens precisas.

1.3.4 Suporte Computacional

Parece existir uma certa unanimidade quanto à necessidade de ferramentas computacionais na utilização de métodos formais. Em especial de ferramentas que sejam apelativas e utilizáveis por engenheiros em domínios de aplicação específicos. Por exemplo, C. Michael Holloway e Ricky W. Butler referem esse facto de forma explícita em [Hall et al., 1996]. Também no domínio das várias classes de redes de Petri tal facto é muito óbvio: as classes de redes de Petri que dispõem de ferramentas consideradas boas e com suporte efectivo são as que apresentam maior popularidade e (aparentemente) maior quantidade de utilizadores.

Talvez devido à conveniência e vantagens oferecidas pela representação gráfica das redes de Petri, existem numerosas ferramentas computacionais que utilizam linguagens baseadas em redes de Petri e, desde a década de noventa, existe uma lista actualizada dessas ferramentas [s.a., 2005d].

Nessa lista é possível verificar que existem vários protótipos realizados em meio académico e que aparentemente se encontram descontinuados, mas também algumas ferramentas comerciais que utilizam linguagens baseadas em redes de Petri, bem como excelentes aplicações de livre utilização. Existe também um esforço particularmente significativo no sentido de popularizar as redes de Petri, nomeadamente as redes de Petri coloridas. Nesse esforço deve ser sublinhada a contribuição do Professor Kurt Jensen e do seu grupo, nomeadamente através do contínuo desenvolvimento, e apoio à comunidade de utilizadores de uma ferramenta computacional denominada CPN Tools [s.a., 2004b] que sucedeu a outra do mesmo grupo, o Design-CPN [Jensen, s.d.; s.a., 2004c]. Estas ferramentas são disponibilizadas a custo zero sendo apenas necessária a respectiva licença de utilização que não impõe qualquer restrição significativa.

O RENEW é outra ferramenta de livre utilização particularmente importante e próxima dos temas desta dissertação. Oferece uma interface gráfica e suporta várias classes de redes, em especial uma variante, de alto-nível, denominada **redes de referência** que será apresentada no Capítulo 2.

1.3.5 Escalibilidade

Todas as redes de Petri, mesmo as de baixo-nível, apresentam algum grau de escalibilidade. A primeira parte desta dissertação propõe uma forma genérica de suporte à estruturação e modificação de modelos com o objectivo de oferecer uma maior escalibilidade das classes de redes que a utilizem. Em particular, as redes de Petri de alto-nível são especialmente escaláveis, tendo já sido argumentado que o são mais do que os estadogramas [Jørgensen e Christensen,

2002]: regra geral, a quantidade de lugares numa rede de Petri cresce de forma linear com o crescimento da rede, enquanto que no caso das máquinas de estado, os estados tendem a crescer muito mais rapidamente. Naturalmente, os mecanismos de estruturação e composição das redes constituem peça fundamental para esta realidade.

1.4 Teses Propostas e Contribuições Originais para a sua Defesa

Esta dissertação tem por objectivo a defesa das duas teses abaixo especificadas. Como resultado do trabalho conducente a esse objectivo obtiveram-se sete resultados originais, que se apresentam enquadrados pelas teses que os motivaram. Para cada resultado indicam-se os capítulos em que o mesmo é apresentado.

Tese 1 — É possível e conveniente definir operações entre modelos em redes de Petri que permitam a especificação de composições, refinamentos, abstracções e modificações, considerando classes comuns de redes de Petri.

1. Uma classificação original para os vários mecanismos de estruturação e composição de modelos de redes de Petri (Capítulo 2).
2. Duas operações genéricas, que operam em modelos em redes de Petri, denominadas *adição* e *subtracção*, para composição e modificação de modelos expressos em quaisquer classes de redes de Petri (Capítulo 3).
3. Uma linguagem diagramática denominada *diagrama de adição* capaz de representar de forma gráfica as operações de adição e subtracção referidas (Capítulo 3).
4. Uma linguagem textual com uma sintaxe inspirada nas linguagens orientadas pelos objectos mais conhecidas e que permite a especificação de modelos em redes de Petri e sua composição e modificação utilizando as operações de adição e subtracção (Capítulo 4).
5. Uma linguagem em XML, denominada *Operational PNML* (OPNML), que permite a especificação de modelos em redes de Petri e sua composição e modificação utilizando as operações de adição e subtracção (Capítulo 4).

Tese 2 — As redes de Petri coloridas com fusão de transições podem ser utilizadas na criação de modelos que seguem o paradigma do desenvolvimento orientado pelos objectos, sem que tal implique a adição de semânticas e sintaxes adicionais.

6. Uma classe de redes de Petri de alto-nível, constituída por redes de Petri coloridas e um tipo particular de canais síncronos (Capítulo 5).

7. Um conjunto de idiomas que permite a especificação de modelos orientados pelos objectos utilizando redes de Petri (Capítulo 5). O Capítulo 6 apresenta dois exemplos de aplicação.

A secção seguinte apresenta o conteúdo da dissertação através de um resumo de cada capítulo e indicação das respectivas publicações associadas. Desta forma, serve também como complemento à lista de resultados originais.

1.5 Sobre o Conteúdo e Estrutura da Dissertação

A definição mais comum e resumida do que é uma metodologia para o desenvolvimento de sistemas em software parece ser a de que uma metodologia é um conjunto, ou talvez uma sequência, de métodos ou de métodos e técnicas que permitem ou contribuem para a realização do software pretendido. Neste sentido, esta dissertação não propõe qualquer nova metodologia para o desenvolvimento de sistemas em software. Em vez disso apresenta um conjunto de técnicas que podem ser utilizadas em várias metodologias ou como parte de vários métodos em que se pretendam utilizar redes de Petri. Desta forma, também não se pretende sugerir que as redes de Petri são a linguagem mais adequada para este ou aquele tipo de projecto. Assume-se que o responsável saberá escolher quais as linguagens mais convenientes e adequadas ao projecto em causa. Caso opte pelas redes de Petri encontrará então neste trabalho um conjunto de técnicas, linguagens e idiomas potencialmente úteis.

A dissertação contém duas partes. Cada uma dessas partes está directamente relacionada com uma das teses apresentadas na secção anterior. Além dos capítulos que constituem as duas partes, existem mais três capítulos: (1) este capítulo de introdução; (2) um capítulo sobre o estado da arte dos mecanismos de estruturação das redes de Petri e (3) um capítulo de conclusão.

Incluem-se também seis apêndices que complementam de diversas formas os conteúdos apresentados nas duas partes da dissertação.

A primeira parte da dissertação propõe duas operações entre modelos de redes de Petri, e duas linguagens capazes de as descrever. As operações permitem a especificação de vários tipos de composição para qualquer classe de redes de Petri. Permitem também uma estruturação e modificação de modelos baseadas em composições ortogonais. Note-se que, contrariamente a outros trabalhos que têm por base uma perspectiva de manutenção de propriedades (e.g. [Ehrig e Padberg, 2004]), as operações definidas enquadram-se noutra linha de investigação. Esta procura encontrar formas precisas e facilmente aplicáveis de transformar modelos de redes de Petri enfatizando a adição e remoção de propriedades (ou requisitos). As operações propostas podem facilmente ser suportadas por ferramentas computacionais já existentes e adaptam-se a qualquer classe de redes de Petri. Assim, fornecem o suporte para especificação de construções ou modificações de sistemas.

A segunda parte apresenta um conjunto de idiomas para a utilização de redes de Petri coloridas no desenho orientado pelos objectos e exemplifica-os através de dois exemplos.

Segue-se uma apresentação resumida de cada um dos capítulos constituintes desta dissertação com indicação de quais as publicações que contribuíram, de forma directa, para os conteúdos respectivos.

Capítulo 1 — Introdução — Utilização de Redes de Petri. Corresponde ao presente capítulo. Discute a motivação para a escolha de redes de Petri como objecto desta dissertação e especifica as teses, contributos e estrutura da dissertação.

Capítulo 2 — Estruturação e Composição nas Redes de Petri. Este capítulo apresenta o estado da arte no que respeita aos mecanismos de estruturação e composição de redes de Petri. Não são apresentadas contribuições originais, com excepção da classificação que é sugerida e utilizada como guia ao longo do capítulo. Essa classificação associa os vários mecanismos para a estruturação e composição de rede de Petri de uma forma que se pretende reveladora das relações entre esses mecanismos. É dada maior atenção às técnicas de composição que se encontram mais próximas das propostas desta dissertação, nomeadamente a fusão de lugares e a fusão de transições através de canais síncronos. É também apresentada a ferramenta RENEW dada a sua relevância relativa na comunidade e especialmente por ser, de acordo com o conhecimento do autor, a única ferramenta actualmente disponível que suporta canais síncronos.

Material publicado: Este capítulo utiliza e revê um artigo de *survey* publicado na revista IEEE Transactions on Industrial Informatics [Gomes e Barros, 2005]:

- Gomes, Luís e Barros, João Paulo. 2005. Structuring and Composability Issues in Petri Nets Modeling. *IEEE Transactions on Industrial Informatics*, 1(2):112–123.

Parte I — Composição e Modificação de Modelos em Redes de Petri

Capítulo 3 — Definição Modular de Modelos. Neste capítulo propõe-se um mecanismo genérico de suporte à composição e modificação de qualquer tipo de modelos em redes de Petri. Para tal propõem-se duas operações: a adição de redes e a subtracção de redes. Mostra-se que estas mesmas operações podem servir de suporte quer à construção de modelos quer à sua modificação. Apresenta-se também uma representação diagramática para estas operações. Em particular, mostra-se como a operação de adição pode servir de suporte à construção de redes de Petri hierárquicas.

Material publicado: Este capítulo utiliza e revê temas publicados nos seguintes quatro artigos, já publicados [Gomes et al., 2002; Gomes e Barros, 2003; Barros e Gomes, 2003b, 2004c]:

- Gomes, Luís, Barros, João Paulo, e Costa, Anikó. 2002. Petri Net Model Node Structuring Techniques for Embedded System Design. In *Proceedings of the 5th Portuguese Conference on Automatic Control (CONTROLO'2002)*, Aveiro, Portugal. Associação Portuguesa de Controlo Automatico (APCA).
- Gomes, Luís e Barros, João Paulo. 2003. On Structuring Mechanisms for Petri Nets Based System Design. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, págs. 431–438. IEEE Catalog Number: 03TH8696, 2003.
- Barros, João Paulo e Gomes, Luís. 2003b. Modifying Petri Net Models by Means of Cross-cutting Operations. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD 2003)*. IEEE Computer Society Press, 2003b.
- Barros, João Paulo e Gomes, Luís. 2004c. Net Model Composition and Modification by Net Operations: a Pragmatic Approach. In *Proceedings of the 2th IEEE International Conference on Industrial Informatics (INDIN 2004)*, 2004c.

Capítulo 4 — Especificação Textual de Redes de Petri. Neste capítulo concretizam-se as propostas apresentadas no Capítulo 3 sob a forma de duas linguagens para o suporte à especificação de redes de Petri e respectivas composições: (1) uma linguagem textual para descrição da estrutura, anotações e composição de modelos de redes de Petri, inspirada na sintaxe das linguagens de programação orientadas pelos objectos mais conhecidas; (2) uma linguagem, baseada em XML, que constitui uma extensão à linguagem de especificação PNML e que permite a composição de quaisquer classes de redes de Petri, a *Operational PNML*, para a qual é apresentado um protótipo desenvolvido no âmbito desta dissertação.

Material publicado: Este capítulo utiliza e revê temas publicados no sítio da Internet da linguagem *Operational PNML* [Barros e Gomes, 2004e] e também do seguinte artigo [Barros e Gomes, 2004f]:

- Barros, João Paulo e Gomes, Luís. 2004f. Operational PNML: Towards a PNML Support for Model Construction and Modification. In *Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets; Workshop satélite da International Conference on Application and Theory of Petri Nets 2004*, 2004f.

Parte II — Redes de Petri no Desenvolvimento Orientado pelos Objectos

Capítulo 5 — Redes de Petri e Objectos. Este capítulo define uma classe de redes de Petri denominada *redes de Petri coloridas componíveis* que permite a utilização de redes de Petri Coloridas no desenho orientado pelos objectos. Para tal, demonstra-se que a classe de redes de Petri proposta é equivalente e traduzível para redes de Petri coloridas. Este capítulo apresenta também um conjunto de idiomas capazes de exprimir, utilizando redes de Petri coloridas componíveis, os conceitos do desenho e programação orientados pelos objectos.

Material publicado: A definição das redes de Petri coloridas componíveis (*Composable Coloured Petri nets*) é uma generalização da apresentada no seguinte artigo [Barros e Gomes, 2004d]:

- Barros, João Paulo e Gomes, Luís. 2004d. On the Use of Coloured Petri Nets for Object-Oriented Design. In Cortadella, Jordi e Reisig, Wolfgang, editores, *Applications and Theory of Petri Nets 2004 25th International Conference (ICATPN 2004)*, Bolonha, Itália, June 21-25, 2004, volume 3099 of *Proceedings Series: Lecture Notes in Computer Science*, págs. 117–136. Springer. ISBN: 3-540-22236-7.

Capítulo 6 — Redes de Petri e Objectos – Dois Exemplos. Este capítulo apresenta a aplicação de redes de Petri coloridas componíveis, através de dois exemplos. O primeiro é um exemplo de pequena dimensão que foi utilizado como demonstrador de um protótipo que permite a utilização de grupos de sincronismo em redes de Petri coloridas, em particular na ferramenta computacional de referência para redes coloridas, a CPN Tools [s.a., 2004b]. O segundo exemplo, ilustra a modelação de vários conceitos do desenvolvimento orientado pelos objectos partindo de um exemplo com origem num artigo de Jørgensen [Jens Bæk Jørgensen, 2004]. O exemplo apresenta um modelo executável dos requisitos de um sistema de controlo de dois elevadores, sob a forma de uma rede de Petri colorida e hierárquica. O novo modelo que é apresentado é realizado por uma rede de Petri colorida componível que modela o "mundo da máquina", utilizando os idiomas definidos no capítulo anterior e em complemento ao modelo do artigo de Jørgensen. Esse novo modelo pode ser visto como a "objectificação" do modelo criado na fase de análise, mas também a uma mudança do modelo no "domínio do problema" para um novo modelo no "domínio da máquina" [Jackson, 2002]. Corresponde a um modelo alternativo resultante da aplicação de um paradigma diferente na construção de modelos baseados em redes de Petri coloridas.

Material publicado: Os exemplos apresentados neste capítulo foram apresentados, nos seguintes três artigos [Barros e Gomes, 2004a; Barros e Jørgensen, 2005b,a]:

- Barros, João Paulo e Gomes, Luís. 2004a. A Unidirectional Transition Fusion for Coloured Petri Nets and its Implementation for the CPNTools. In Jensen, Kurt, editor, *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools*, DAIMI PB

- 570, págs. 133–150, Aarhus, Dinamarca. Department of Computer Science, University of Aarhus. ISSN 0105-8517, disponível em <http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/CPN04proceedings.pdf>.

- Barros, João Paulo e Jørgensen, Jens Bæk. 2005b. Model Transformations for an Elevator Controller: Coloured Petri Nets in Object-Oriented Analysis and Design. In *Second International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005)*, Rennes, França.
- Barros, João Paulo e Jørgensen, Jens Bæk. 2005a. A Case Study on Coloured Petri Nets in Object-Oriented Analysis and Design. *Nordic Journal of Computing*, 12(3):229–250.

O primeiro artigo apresenta o primeiro exemplo do Capítulo 6 (Secção 6.2). O terceiro artigo é uma versão revista e aumentada do segundo. Ambos apresentam uma versão reduzida do segundo exemplo do Capítulo 6 (Secção 6.3).

Capítulo 7 — Conclusão. A dissertação conclui discutindo o trabalho realizado, bem como um conjunto de trabalhos futuros que parecem promissores. Nestes incluem-se trabalhos preliminares que embora tendo sido publicados em *workshops* internacionais, acabaram por não ser desenvolvidos, razão pela qual não foram incluídos na dissertação.

Material publicado: referem-se dois artigos que embora não tenham sido alvo de desenvolvimento posterior suficiente para constituírem parte desta dissertação, podem ainda assim servir de mote a trabalhos futuros:

- Barros, João Paulo e Gomes, Luís. 2002. Activities as Behaviour Aspects. In Kandé, Mohamed, Aldawud, Omar, Booch, Grady, e Harrison, Bill, editores, *Workshop on Aspect-Oriented Modeling with UML*, 2002. Workshop satélite da «UML» 2002 - The Unified Modeling Language 5th International Conference(UML'2002).
- Barros, João Paulo e Gomes, Luis. 2003c. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In Akkawi, Faisal, Aldawud, Omar, Booch, Grady, Clarke, Siobhán, Gray, Jeff, Harrison, Bill, Kandé, Mohamed, Stein, Dominik, Tarr, Peri, e Zakaria, Aida, editores, *The 4th AOSD Modeling With UML Workshop*, pág. 8, 2003c.

Apêndice A — Gramática da Linguagem PN_{TEXT} . Contém a gramática da linguagem PN_{TEXT} descrita no Capítulo 4.

Apêndice B — Gramática da Linguagem *Operational* $PNML$. Contém a gramática da linguagem $OPNML$ descrita no Capítulo 4.

Apêndice C — Redes de Petri para Controladores Digitais. Este apêndice apresenta uma classe de redes de Petri de baixo-nível e não-autónomas para a especificação de controladores digitais em sistemas embutidos. É também proposto um ambiente de desenvolvimento que permitirá o desenvolvimento de sistemas, nomeadamente de software (e possivelmente de hardware) utilizando redes de Petri. O ambiente inclui duas ferramentas principais: um gerador de código e um editor gráfico, com suporte para a definição de modelos utilizando as operações de adição e de subtracção definidas no Capítulo 3.

Material publicado: Uma definição preliminar da classe de redes aqui definida surge no primeiro dos seguintes artigos [Gomes et al., 2004]. A versão aqui apresentada é discutida, de forma resumida, no segundo artigo [Pais et al., 2005]. A globalidade do ambiente de desenvolvimento proposto foi descrita no terceiro artigo [Barros et al., 2004]:

- Gomes, Luís, Barros, João Paulo, e Pais, Rui. 2004. From Non-Autonomous Petri net Models to Code in Embedded Systems Design. In *Second International Workshop on Discrete-Event System Design (DESDes'04)*, Zielona Gora, Polónia.
- Pais, Rui, Barros, João Paulo, e Gomes, Luís. 2005. A Tool for Tailored Code Generation from Petri Net Models. In *Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005)*. IEEE, 2005.
- Barros, João Paulo, Gomes, Luís, Pais, Rui, e Dias, Rui. 2004. From Petri Nets to Executable Systems: an Environment for Code Generation and Analysis. In *1st International Conference on Informatics in Control, Automation and Robotics (ICINCO'2004)*, Setúbal, Portugal.

Apêndice D — Protótipo para Tradução de Especificações OPNML. Contém o código fonte do protótipo descrito no Capítulo 4.

Apêndice E — Gramática Relax NG para Redes de Petri Input-Output. Contém a gramática para a classe de redes de Petri definida no Apêndice C.

Apêndice F — Protótipo para Tradução de Grupos de Sincronismo. Contém o código fonte do protótipo descrito no Capítulo 6.

1.6 Sobre os Conhecimentos que se Assume o Leitor Possuir

Ao longo da realização deste trabalho houve a intenção de o tornar acessível a qualquer engenheiro nas áreas das engenharias informática e electrotécnica e de computadores. No entanto, como esta dissertação não é, nem pretende ser ou conter, um tutorial sobre os fundamentos dos temas que aborda foi necessário assumir alguns conhecimentos prévios da parte do leitor. A lista

seguinte indica quais os conhecimentos que o leitor deve possuir e quais as partes da dissertação em que são importantes:

1. A sintaxe e semântica de uma qualquer classe de redes de Petri de baixo-nível (Parte I e Parte II).
2. Os principais conceitos da programação orientada pelos objectos ao nível do que é necessário possuir para programar numa linguagem de programação orientada pelos objectos generalista como, por exemplo, C++ ou Java (Parte II).
3. Conhecimentos elementares de diagramas de classes da linguagem UML ou de um outro qualquer diagrama de entidades e associações (Parte II).
4. A sintaxe e semântica das redes de Petri coloridas de Jensen ou teoria de conjuntos (Parte II).

Os conhecimentos referidos no ponto 1 são facilmente obtidos em várias referências da bibliografia por exemplo [Peterson, 1977; Reisig, 1985; Murata, 1989; David, 1991; David e Alla, 1992; Silva, 1993; Zurawski e Zhou, 1994; Reisig e Rozenberg, 1998a,b; Girault e Valk, 2003].

Para os pontos 2 e 3 existe uma extensa bibliografia sobre programação e desenho orientados pelos objectos. A título de exemplo e por se tratar de uma obra particularmente significativa na área, recomenda-se o livro de Bertrand Meyer [Meyer, 1997].

Contrariamente à maioria das classes de redes de Petri, existem várias referências de grande qualidade e completude sobre as redes de Petri coloridas. Refiram-se em especial os três volumes de Kurt Jensen [Jensen, 1997c,a,b] e também um artigo mais recente [Kristensen et al., 2004], publicado no âmbito do último curso avançado sobre redes de Petri realizado em Eichstätt na Alemanha [s.a., 2004a].

Será preferível, embora não necessário, que o leitor possua conhecimentos básicos de teoria de conjuntos por esta ser utilizada nas formalizações apresentadas. No entanto, estas podem ser omitidas numa primeira leitura sem perda de continuidade pois os exemplos supõem-se suficientes para a compreensão das técnicas apresentadas. A sua leitura apenas se torna necessária se o leitor pretender esclarecer alguma eventual dúvida resultante de possível ambiguidade nos textos e exemplos associados. Qualquer livro introdutório que contemple o tema será provavelmente suficiente.

Por fim, importa sublinhar ser desejo expresso do autor que este texto não constitua apenas um trabalho interno da comunidade científica das redes de Petri e nem sequer da comunidade científica em geral. Pretende-se sim que constitua uma contribuição, embora necessariamente académica e escolástica, para uma melhor e mais informada prática aquando da utilização das redes de Petri fora do mundo académico.

1.7 Linguagem e Convenções Utilizadas

Na literatura é comum encontrar vários sentidos em torno das expressões "redes de Petri", "classe de redes de Petri", "tipo de redes de Petri" e "rede de Petri". Ao longo do presente texto, estas expressões são entendidas da seguinte forma:

redes de Petri Conjunto de linguagens formais baseadas no modelo proposto por Carl Adam Petri na sua tese de doutoramento em 1962 [Petri, 1962]; também pode designar um conjunto de modelos (vide "rede de Petri").

rede de Petri Um modelo construído utilizando uma classe de redes de Petri. Por exemplo, "uma rede de Petri com sete lugares e dez transições".

classe de redes de Petri Uma variante específica de redes de Petri, devidamente definida.

tipo de redes de Petri O mesmo que classe de redes de Petri.

Tal como na literatura, e sempre que tal não dê origem a ambiguidades, utilizar-se-á frequentemente a expressão "rede de Petri" quando em rigor se deveria escrever "modelo construído utilizando uma classe de redes de Petri". Por exemplo, escreveremos "composição de redes de Petri" quando, em rigor, deveríamos escrever "composição de modelos construídos utilizando uma classe de redes de Petri". Por outras palavras, e aproveitando a nomenclatura tão cara à comunidade da linguagem UML, utiliza-se a expressão "rede de Petri" quer para o modelo quer para o seu metamodelo. Esta simplificação é muito frequentemente utilizada na literatura.

Serão também utilizadas as seguintes convenções:

- Os termos que introduzam novos conceitos considerados particularmente relevantes são assinalados a **negrito**.
- Os identificadores utilizados em modelos ou em outro tipo de código são escritos **nesta fonte de igual espaçamento**.
- Os termos numa língua que não a portuguesa são escritos em *itálico*.
- As definições formais utilizam *esta fonte* itálica.
- Embora contrariando a prática mais comum, cada parágrafo, correspondente a um item numa lista precedida de dois pontos, é terminado com um ponto final. Tal permite a utilização de todos os sinais de pontuação, nomeadamente de outros pontos finais nesse mesmo parágrafo. É o que sucede também neste próprio parágrafo.
- As entradas na bibliografia encontram-se ordenadas pelo apelido do primeiro autor. Na bibliografia utiliza-se *s.a.* (abreviatura de *sem autor*) quando não foi possível determinar

qual o autor da publicação em causa. Nalguns casos em que o "autor" corresponde a um grupo ou organização identificado, evitou-se a utilização da abreviatura, preferindo-se o nome desse grupo ou organização. Desta forma conseguiu-se melhorar a informação de algumas citações.

Capítulo 2

Estruturação e Composição nas Redes de Petri

*–Esta história começa pelo fim. Mas não acaba no princípio.
Acaba também no fim... embora noutro sítio do fim...*

– Manuel António Pina

in Uma História Que Começa Pelo Fim

Neste capítulo, após uma breve discussão sobre a escassa utilização de redes de Petri, apresenta-se o estado da arte dos mecanismos de estruturação e composição de redes de Petri. Estes correspondem a diversos tipos de abstracção que permitem a efectiva utilização prática de redes de Petri para a modelação de sistemas. A apresentação é enquadrada por uma classificação original dos diversos mecanismos.

As redes de Petri são normalmente consideradas uma linguagem formal com grande potencial de aplicação no domínio da engenharia. Tal deve-se especialmente ao facto da sua representação gráfica permitir uma fácil e intuitiva especificação de vários conceitos, de grande importância e complexidade, típicos de sistemas distribuídos ou concorrentes. Em particular, as redes de Petri permitem uma especificação graficamente explícita de sincronização, paralelismo, atomicidade, troca de mensagens e memória partilhada. Estas características estão já bem documentadas em publicações de carácter mais académico. No entanto, só na última década surgiram publicações mais dirigidas à aplicação prática das redes de Petri e denotando uma real preocupação com a sua utilização para a especificação de sistemas em diversas áreas de aplicação. Os livros de Kurt Jensen [Jensen, 1997c,a,b] foram, muito provavelmente, os mais significativos no sentido da efectiva divulgação das reais potencialidades de aplicação prática das redes de Petri.

Mais recentemente, têm sido editados vários livros que apresentam a utilização de redes de Petri de uma forma também prática [Girault e Valk, 2003; Ehrig et al., 2003]. Num formato mais pedagógico, os livros resultantes dos cursos avançados sobre redes de Petri¹ e, em especial, os dois últimos [Reisig e Rozenberg, 1998a,b; Desel et al., 2004], constituem importantes fontes de informação sobre a teoria e aplicação das redes de Petri. Os cursos foram ambos realizados na Alemanha, respectivamente em Outubro de 1996 em Dahgstell [s.a., 1996] e em Setembro de 2003 em Eichstätt [s.a., 2004a].

No entanto, estes livros são ainda limitados no seu público-alvo. Todos eles são editados pela mesma editora em colecções cujos destinatários principais são investigadores ou académicos. Tal contrasta de forma flagrante com a profusão de livros sobre a *Unified Modeling Language* (UML)[OMG, 2003], a maior parte dos quais se destina a engenheiros ou programadores profissionais. Tal facto, constitui uma indicação segura da fraca divulgação e utilização das redes de Petri quando comparadas à linguagem UML que inclui várias linguagens gráficas distintas. É por isso fácil constatar que o potencial de aplicação das redes de Petri não é ainda reconhecido por uma larga maioria de engenheiros, quer devido a um desconhecimento da existência das mesmas, quer a um conhecimento incompleto e muitas vezes errado, das reais capacidades das redes de Petri.

Vários trabalhos académicos têm tentado capitalizar a popularidade da linguagem UML como forma de divulgar e valorizar as capacidades das redes de Petri. Vejam-se, por exemplo, os já referidos trabalhos de Bernardi *et al.* [Bernardi et al., 2002] e de Baresi e Pezzè [Baresi e Pezzè, 2001]. No entanto, o seu eventual efeito não é visível fora da restrita comunidade de investigadores que conhecem as redes de Petri e nelas reconhecem interesse.

As redes de Petri têm outro problema no que respeita à sua utilização: são um **método formal**. Isto significa que transportam consigo o fardo correspondente. Muitos artigos têm sido escritos sobre a razão da fraca ou inexistente utilização de métodos formais na indústria (e.g. [s.a., 2003a; Sharpe, 2004]), mas para além dos problemas de aceitação comuns a todos os métodos formais, persiste a ideia errada de que as redes de Petri não são adequadas à criação de modelos de grandes dimensões. Neste aspecto, as redes de Petri sofreriam do mesmo problema de muitos outros métodos formais que, na prática, não se conseguem aplicar a modelos de grandes dimensões². Esta ideia advém do facto, facilmente verificável, de muito frequentemente as redes de Petri continuarem a ser vistas única e exclusivamente como um modelo de baixo nível sem possibilidades significativas de estruturação. Ora, pelo menos desde o início da década de 1990, tal não corresponde à realidade. A secção seguinte discute este facto e a Secção 2.2 apresenta uma classificação original para as várias formas de estruturar e abstrair modelos em redes de Petri.

¹*Advanced Courses on Petri Nets.*

²Em inglês utiliza-se normalmente a expressão: 'do not scale up' para formalismos que não são aplicáveis na criação de modelos de grandes dimensões.

2.1 Estrutura e Abstracção em Redes de Petri

Conforme já referido, o nome redes de Petri é frequentemente visto como equivalente a redes de Petri lugar-transição [Reisig, 1985]. Provavelmente, esta situação deve-se ao facto de durante bastante tempo tal ter correspondido à realidade: as redes de Petri eram um formalismo que facilmente obrigava à construção de modelos demasiado extensos para poderem ser utilizados. A possibilidade de uma representação gráfica, ajudou a reforçar esta ideia. De facto, o problema da "explosão do modelo", como daqui em diante o iremos designar, é mais evidente nas linguagens gráficas (ou visuais). Na respectiva comunidade, tal é conhecido pelo **limite de Deutsch**. Segundo o criador da expressão (Fred Lakin), Peter Deutsch fez o seguinte comentário em resposta a uma apresentação sobre linguagens visuais:

"Bem, isso está tudo muito bem, mas o problema com as linguagens de programação visuais é o de não podermos ter mais de 50 primitivas gráficas ao mesmo tempo no ecrã. Como é que vão conseguir escrever um sistema operativo?" [Lakin, 1998].

Apenas em 1981 se definiram alternativas de mais alto nível às redes lugar-transição, as redes de Petri predicado-transição de Genrich e Lautenbach [Genrich e Lautenbach, 1981] e as redes de Petri coloridas de Jensen [Jensen, 1981]. Ainda em 1989, o artigo de Huber, Jensen e Shapiro [Huber et al., 1989] afirmava que na literatura quase não existiam trabalhos sobre hierarquias em redes de Petri. Desde aí, esta situação tem mudado devido a numerosas propostas de mecanismos de estruturação hierárquica e de composição de redes de Petri. Vejam-se, em particular, os artigos de *survey* de Brauer *et al.* [Brauer et al., 1991] e de Bernardinello e De Cindio [Bernardinello e Fiorella De Cindio, 1992], ainda muito centrados nas redes de Petri de baixo-nível. Mais interessantes, para uma utilização prática das redes de Petri e para a sua aplicação na construção de modelos de grandes dimensões, foram os artigos de Søren Christensen *et al.* introduzindo os **canais síncronos** [Christensen e Hansen, 1992] e as redes de Petri modulares [Christensen e Petrucci, 1992]. Praticamente em simultâneo com estes desenvolvimentos, surgiram propostas de integração de conceitos da programação orientada pelos objectos nas redes de Petri. De entre estas propostas, as principais encontram-se compiladas num volume publicado em 2001 e ainda muito actual [Agha et al., 2001].

Esta adição de mecanismos de estruturação, por um lado, e de características das linguagens orientadas pelos objectos, por outro, deu origem a uma enorme variedade de classes (ou tipos) de redes de Petri. Na verdade, a unificação das várias classes de redes de Petri tornou-se um importante tópico de investigação como pode ser comprovado pela publicação do livro [Ehrig et al., 2001]. Essa mesma obra inicia-se com um artigo onde os autores se interrogam sobre o que é uma rede de Petri [Jörg Desel, 2001], procurando deixar claro quais as suas características fundamentais.

Para esta enorme diversidade contribuiu também a variedade de mecanismos de estruturação propostos. Dada esta enorme diversidade de redes de Petri, e especialmente por ser esse o tema fulcral deste trabalho, apresenta-se aqui o estado da arte no que diz respeito aos mecanismos de estruturação e composição de modelos baseados em redes de Petri. Para tal, começa-se por apresentar uma classificação dos vários mecanismos de estruturação e composição de redes de Petri que se encontram na literatura. Estes mecanismos correspondem a diversos tipos de abstracção que permitem a efectiva utilização prática de redes de Petri para a modelação de sistemas.

Por fim, importa também notar que existem numerosas ferramentas computacionais que exploram as redes de Petri nas suas variadas vertentes, e existem também numerosas classes de redes de Petri muitas das quais suportadas por ferramentas computacionais (veja-se a base de dados no sítio "Petri nets World" [s.a., 2005d]). Nesta dissertação, optou-se por referir apenas duas ferramentas — a *CPN Tools* [s.a., 2004b] e a *RENEW* [Kummer et al., 2004a] — por se considerar serem estas as duas mais importantes para a utilização de redes de Petri na modelação de sistemas e por implementarem um conjunto extremamente relevante de mecanismos de composição e estruturação. Visto esta dissertação se centrar nos mecanismos de modularidade, composição e estruturação, outras ferramentas, como as especialmente vocacionadas para a verificação de sistemas (por exemplo a ferramenta Maria [Mäkelä, 2004]) ou suportando classes de redes de Petri muito específicas não são referidas.

Segue-se a classificação proposta.

2.2 Uma Classificação para os Mecanismos de Estruturação e Composição

Apesar de existirem, há já bastantes anos, vários bons textos de tutoria sobre redes de Petri (por exemplo [Peterson, 1977; Reisig, 1985; Murata, 1989; David, 1991; David e Alla, 1992; Silva, 1993; Zurawski e Zhou, 1994; Reisig e Rozenberg, 1998a,b; Girault e Valk, 2003]), nenhum dá uma atenção significativa às várias formas de estruturar e compor modelos de redes de Petri. Provavelmente, tal deve-se ao facto destas técnicas de composição não serem significativas do ponto de vista teórico pois, tipicamente, nada acrescentam ao poder de modelação das redes. No entanto, são claramente da máxima importância prática pois só através de alguma forma de modularização se torna possível a utilização de redes de Petri para a modelação de sistemas reais. Mesmo a forma mais simples de estruturação hierárquica pode melhorar de forma extrema a legibilidade de um modelo.

Num primeiro nível, classificam-se os mecanismos de estruturação de redes de Petri da mesma forma que é usual encontrar na literatura, de carácter mais teórico, sobre redes de Petri: **composição**, **refinamento** e **abstracção**.

A composição corresponde à interligação entre vários modelos. Esta interligação é tipicamente feita através da fusão entre dois ou mais lugares, ou entre duas ou mais transições. Por essa razão, os vários modelos podem ser vistos como módulos do sistema total que se pretende modelar. Comparando, de forma um pouco simplista, com a terminologia utilizada nas linguagens de programação, estes módulos correspondem a blocos, módulos ou classes, conforme o tipo de linguagens em causa. Ainda em termos de linguagens de programação, o refinamento e abstracção correspondem a macros, sub-rotinas ou procedimentos.

Segue-se uma proposta de classificação para os vários mecanismos de estruturação de redes de Petri que podem ser encontrados na literatura. A classificação apresenta-se na Fig. 2.1 e inicia-se pelos dois tipos fundamentais de estruturação já referidos: composição e refinamento/abstracção.

Para além da fusão de lugares e transições, também a dobragem³ das redes de alto-nível, é enquadrada num tipo de composição. Inclui-se ainda um mecanismo de estruturação menos conhecido baseado no conceito de *vectores de nós*: a dobragem baseada nos nós.

O refinamento/abstracção inclui a estruturação baseada em macros e também a criação dinâmica de módulos de redes. Esta última forma inclui as classes de redes de Petri inspiradas nas linguagens de programação orientadas pelos objectos.

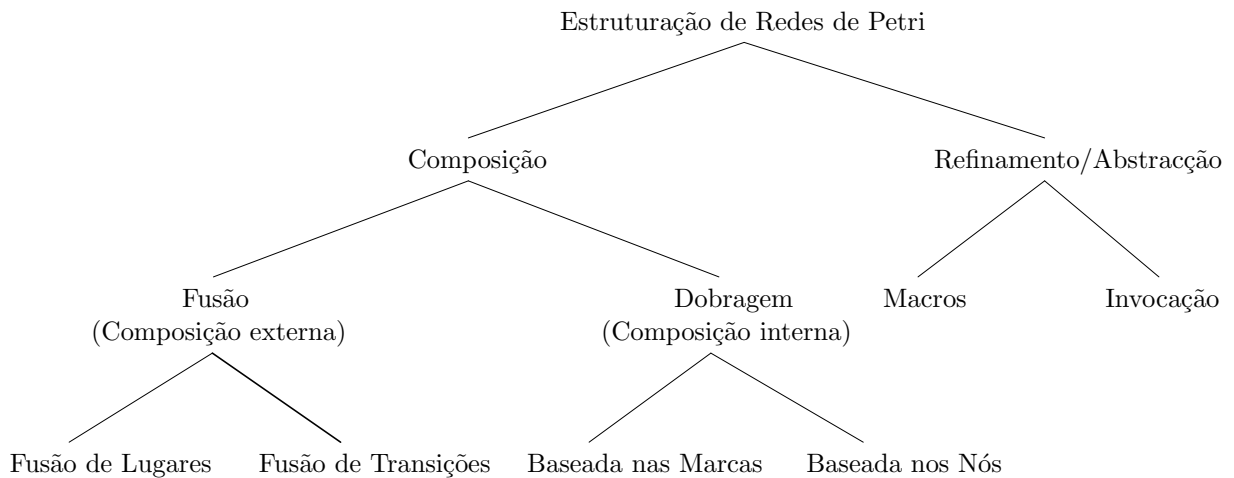


Figura 2.1: Uma classificação dos mecanismos de estruturação para redes de Petri.

Seguidamente apresentamos estes vários mecanismos de estruturação, com base na literatura conhecida. Como exemplo unificador, utilizamos a modelação de um parque de estacionamento, um sistema semelhante ao clássico sistema produtor-consumidor.

³Em inglês: *folding*.

2.2.1 Composição

O significado mais comum para a composição de redes é o da fusão de duas redes de Petri numa única, através da fusão de um ou mais nós. Tal constitui um claro suporte à construção de modelos de forma ascendente⁴ o que também permite a reutilização de módulos já existentes. Aqui generaliza-se esta ideia de composição de modelos considerando dois tipos de composição: a **externa** e a **interna**.

A composição externa corresponde ao conceito de **fusão** entre nós de redes (ou páginas) distintas, ou seja, ao caso já descrito em que duas, ou mais, redes são utilizadas para criar uma única rede. Pode ser vista como a aplicação mais directa, no âmbito dos modelos de redes de Petri, da decomposição dos sistemas em módulos, algo cuja importância é reconhecida e sublinhada pelo menos desde o artigo de Parnas em 1972 [Parnas, 1972].

A composição interna corresponde ao conceito de **dobragem**⁵. Por outras palavras, a dobragem é a composição de uma rede com ela própria: as repetições de idênticos padrões estruturais são como que dobradas sobre si próprias de forma a fundir lugares, arcos e transições. Conforme exemplificado mais adiante, esta fusão tem como consequência a necessidade de tornar mais complexas quer as marcas nos lugares, quer as anotações nos arcos e transições.

Fusão (Composição Externa)

Numa primeira aproximação, e por permitir um drástico encurtamento do comprimento dos arcos, a fusão de nós pode ser vista e utilizada como uma forma de simplificar a representação gráfica da rede de Petri. No entanto, ao basear-se na dualidade fundamental das redes de Petri, a fusão de nós — sob a forma de fusão de lugares ou de transições — constitui também a forma mais intuitiva e graficamente explícita de compor redes de Petri. Em particular, a fusão de nós oferece um suporte imediato para uma **composição horizontal** das redes. Esta designação advém de, visualmente, as redes serem "colocadas" lado a lado para seguidamente serem "coladas" umas às outras através dos seus lugares, das suas transições ou de ambos. As várias redes a serem compostas podem ser vistas como peças de um *puzzle* em que cada peça pode conter um desenho completo mas também contém partes que a podem ligar a outras peças de forma a criar um desenho maior.

A dualidade característica das redes de Petri leva-nos à utilização de dois tipos de fusão: a fusão de transições e a fusão de lugares. O primeiro tem uma clara correspondência na comunicação síncrona típica das álgebras de processos [Hoare, 1985; Milner, 1995]. Refira-se aqui o livro de Best, Devillers e Koutny [Best et al., 2001] por se tratar de um trabalho particularmente significativo que estuda com grande detalhe as relações entre as redes de Petri e as álgebras de

⁴Em inglês: *bottom-up*.

⁵Em inglês: *folding*.

processos. A dissertação de doutoramento de Twan Basten [Basten, 1998] é outra referência significativa para o estudo dessa relação.

A fusão de lugares pode ser vista como a partilha de um recurso (por exemplo, memória) permitindo a comunicação assíncrona entre processos. Assim, as redes de Petri contêm na sua dualidade os dois tipos de comunicação: síncrona e assíncrona. Tal facto, permite uma grande versatilidade às redes de Petri. Por exemplo, conforme apontado por Juan, Tsai e Murata [Eric Y. T. Juan et al., 1996, 1998], a comunicação síncrona entre processos não pode ser utilizada de forma natural para a análise de propriedades baseadas nos estados. Por essa razão esses trabalhos utilizam fusão de lugares. O mesmo sucede com um trabalho de Valmari sobre a verificação de propriedades [Valmari, 1994]. Também a proposta em [Chen et al., 1993], referida como um exemplo de fusão de lugares em [Eric Y. T. Juan et al., 1996], utiliza lugares adicionais para ligar as redes.

Apesar da óbvia complementaridade entre fusão de lugares e fusão de transições, algumas propostas de classes de redes de Petri apenas utilizam uma delas. O exemplo mais significativo é provavelmente o das próprias redes de Petri coloridas hierárquicas de Jensen [Jensen, 1997c,a,b] que utilizam a fusão de lugares mas não a fusão de transições. Na verdade, a fusão de lugares é utilizada de duas formas distintas: uma como conveniência gráfica, de forma a aumentar a legibilidade dos modelos (conforme referido anteriormente), e outra como suporte à estruturação hierárquica de redes. Neste último caso, a fusão de lugares é utilizada como suporte à implementação de macrotransições, também denominadas **transições de substituição**⁶. A Secção 3.1.3 na página 61 discute este tema no contexto das operações propostas nesse capítulo.

É interessante notar que um importante trabalho inicial sobre hierarquias em redes de Petri coloridas [Huber et al., 1989] propôs, também, a fusão de transições e a sua utilização como suporte aos macrolugares, aí denominados **lugares de substituição**⁷. No entanto, por razões de ordem prática ao nível da implementação, não foram incluídas na respectiva ferramenta computacional de referência: o Design-CPN [s.a., 2004c]. Como consequência deste facto, a fusão de transições também não surge na definição das redes de Petri coloridas hierárquicas. A CPN Tools [s.a., 2004b], que é a actual ferramenta computacional de referência para redes de Petri coloridas e hierárquicas também não disponibiliza fusão de transições.

Provavelmente devido à forte relação com a composição nas álgebras de processos, alguns trabalhos utilizam a fusão de transições mas não a fusão de lugares: vejam-se, por exemplo, as OBJA-nets de Battiston, de Cindio e Mauri [Battiston et al., 1988] e o trabalho de Notomi e Murata sobre análise hierárquica de redes de Petri [Notomi e Murata, 1994].

Existem também propostas que propõem a utilização dos dois tipos de fusão. No contexto desta dissertação, é esta a atitude mais interessante, dada a generalização e versatilidade de

⁶Em inglês: *substitution transitions*.

⁷Em inglês: *substitution places*.

composições e estruturações que possibilita. Também no contexto desta dissertação, é particularmente importante a proposta de Christensen e Hansen que introduziu uma forma generalizada de fusão de transições no contexto das redes de Petri coloridas [Christensen e Hansen, 1992]: os **canais síncronos**. Para além de melhorarem significativamente o poder de modelação das redes de Petri coloridas (especialmente em termos da sua utilização prática), os canais síncronos, se considerados no âmbito das redes de Petri modulares, vêm facilitar a modularização da análise baseada no espaço de estados [Christensen e Petrucci, 1995, 2000]. Finalmente, é interessante notar que contrariamente à proposta inicial de Huber *et al.* [Huber et al., 1989] as redes de Petri modulares não obrigam à disjunção entre as várias fusões de transições.

A Figura 2.2 introduz o sistema que irá ser utilizado neste capítulo com fins ilustrativos. Trata-se de um modelo de um sistema *primeiro-a-entrar primeiro-a-sair* (uma fila *first-in first-out* ou *FIFO*), sob a forma de um parque de estacionamento para automóveis. Considera-se que cada área de estacionamento comporta um automóvel e que o parque contém três áreas. A Figura 2.2a ilustra a disposição do parque de estacionamento e as Figuras 2.2b e 2.2c mostram, respectivamente, as redes de baixo nível associadas a cada área do parque quando ocupada por um automóvel (Figura 2.2b) ou quando livre (Figura 2.2c).

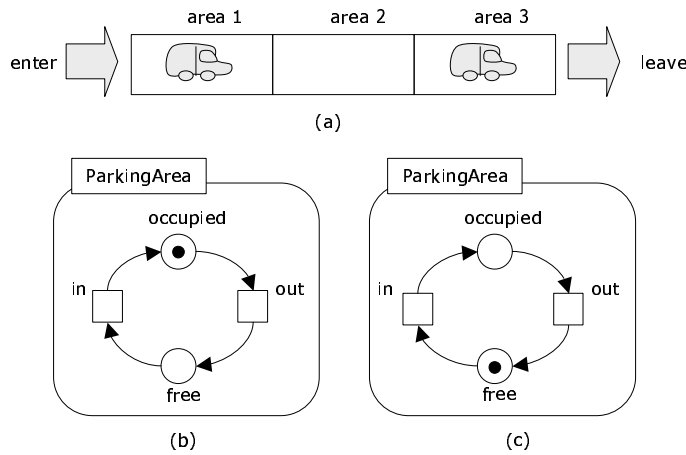


Figura 2.2: a) Uma fila *primeiro-a-entrar primeiro-a-sair* num parque de estacionamento, b) modelo para uma área ocupada e c) para uma área livre.

O sistema completo pode ser obtido pela replicação dos modelos nas Figuras 2.2b e 2.2c, conforme necessário, seguida da composição dos mesmos através da fusão de transições. O modelo para o sistema na Figura 2.2a (apresentado na Figura 2.3b) é o resultado da composição de duas instâncias da rede na Figura 2.2b e uma instância da rede da Figura 2.2c. Conforme ilustrado pela Figura 2.3b, as transições *outA* and *inB* são fundidas na transição *moveAtoB*, as transições *outB* e *inC* são fundidas na transição *moveBtoC*, a transição *inA* é renomeada *enter*, e a transição *outC* é renomeada *leave*.

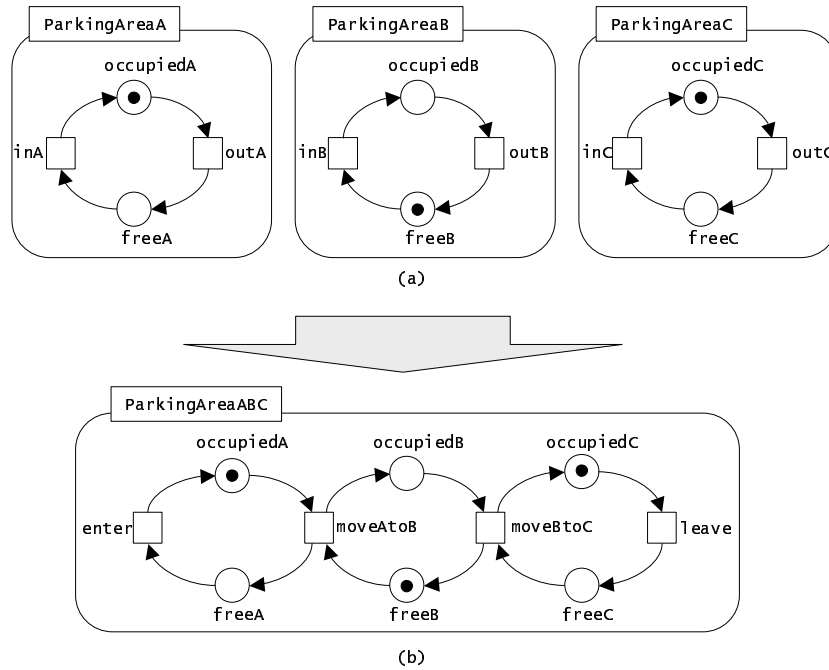


Figura 2.3: Composição, através da fusão de transições, das redes em a) no modelo b).

Dobragem (Composição Interna)

Conforme já referido, classifica-se aqui a dobragem como uma forma de composição. A dobragem oferece uma simplificação da representação gráfica do modelo através da codificação, sob a forma de anotações, de simetrias estruturais. Constitui por isso um exemplo claro de simplificação gráfica através do aumento da complexidade das anotações textuais.

Consideram-se dois tipos de dobragem:

- No primeiro a abstracção resultante é suportada pelas marcas; mais especificamente, a complexidade gráfica transfere-se para as marcas aumentando a complexidade destas, tornando-as de "alto nível". Naturalmente, tal implica um aumento de complexidade também nas anotações associadas aos arcos. Esta dobragem é a que caracteriza as redes de Petri de alto nível [Jensen e Rozenberg, 1991]. Denominamo-la **dobragem baseada nas marcas**.
- A segunda forma de abstracção é denominada **dobragem baseada em nós** e, tal como o seu nome indica, é suportada pelos próprios nós [Gomes, 1997; Gomes et al., 2002]: mais especificamente, um nó passa a representar um conjunto de nós através da adição de uma anotação que permite identificar qual a marcação de cada um dos nós por ele representado. Desta forma, cada nó mantém a sua identidade. Por esta razão falaremos em **condensação de nós** como forma de distinguir esta abreviatura da fusão de nós. Estas anotações baseiam-se em **instâncias de nós** que são agrupadas em **vectores de**

nós. Tal implica anotações mais complexas para os arcos de forma a que estes possam também condensar vários arcos. Naturalmente, os nós condensados podem também ser transições ou outros tipos de nós que porventura existam na classe de redes em causa.

Dobragem Baseada nas Marcas. A dobragem baseada em marcas é, por definição, a característica fundamental das redes de Petri de alto-nível [Jensen e Rozenberg, 1991]. Como as marcas passam a poder conter qualquer tipo de dados, tal obriga a uma maior complexidade das anotações associadas aos arcos e, por consequência, às transições. Cada transição passa a ter associada uma expressão booleana, denominada guarda, que pode devolver um valor verdade ou falso. Os arcos passam também a ter expressões algébricas associadas, capazes de efectuar transformação de dados. Por esta razão, e para além da efectiva redução da dimensão gráfica dos modelos, as redes de Petri de alto nível permitem a modelação de dados e a sua transformação.

A Figura 2.4a mostra o resultado da aplicação da dobragem baseada em marcas ao modelo da Figura 2.3b: os lugares que indicam a ocupação da área (*occupiedA*, *occupiedB* e *occupiedC*) estão fundidos num único lugar *occupied*. O mesmo sucede com os lugares que especificam a área como livre (*free*). Esta fusão foi feita dada a sua semelhança semântica. De forma idêntica, as transições intermédias (*moveAtoB* e *moveBtoC*) foram também fundidas numa única transição *move*. As transições inicial e final (*enter* e *leave*) mantiveram-se dado desempenharem um papel diferente de todas as restantes. Devido às fusões dos nós, os arcos foram também fundidos. Tal foi possível graças à utilização de expressões associadas aos arcos, expressões estas que possibilitam a modelação dos vários arcos.

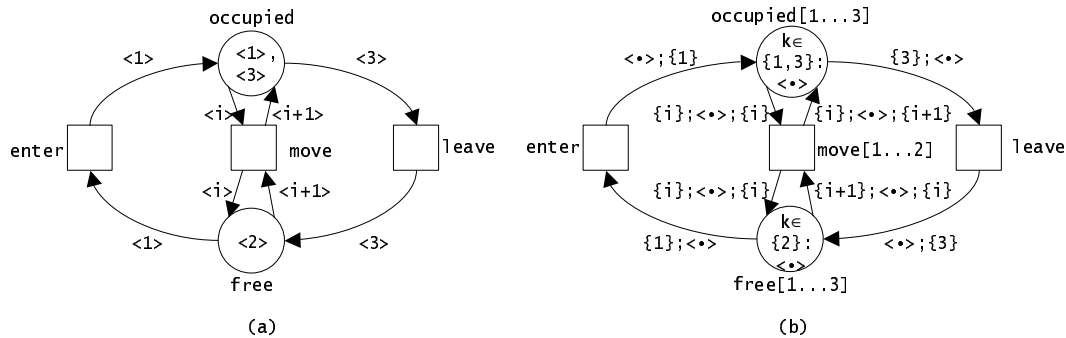


Figura 2.4: Dobragem de modelos: a) baseada em marcas e b) baseada em nós.

Com base na literatura é possível identificar duas classes principais de redes de Petri de alto nível: as redes de Petri predicado-transição [Genrich, 1987; He, 1996] e as redes de Petri coloridas [Jensen, 1996, 1997a,b,c].

As **redes de Petri predicado-transição** foram a primeira classe de redes de Petri definida sem nenhuma aplicação prática específica em mente. Conforme referido por Jensen em [Jensen, 1997a], comparadas com as redes de Petri coloridas as redes de Petri predicado-transição podem ser vistas como um dialecto, ligeiramente diferente, da mesma linguagem. No entanto, prova-

velmente devido à existência de muito mais ferramentas para redes de Petri coloridas, as redes de Petri predicado-transição tornaram-se muito menos conhecidas. Ainda assim, continuam a constituir um suporte muito utilizado para o desenvolvimento de trabalho teórico. Como exemplos particularmente relevantes, refiram-se os trabalhos de Xudong He [He, 2004] que também cobrem as **redes predicado-transição hierárquicas** (e.g. [He, 1996]).

As **redes de Petri coloridas** são claramente as mais populares. Tal pode ser atestado pela quantidade de publicações e ferramentas computacionais nelas baseadas, em especial as ferramentas *CPN Tools* [s.a., 2004b] e *RENEW* [Kummer et al., 2004a].

A ferramenta *CPN Tools* suporta redes de Petri coloridas e hierárquicas. Estas estendem as capacidades de estruturação das rede de Petri coloridas com lugares de fusão e macro transições, denominadas **transições de substituição**⁸.

A ferramenta *RENEW* [Kummer et al., 2004b,a] estende as redes de Petri coloridas de diversas formas, nomeadamente através da adição de diversos novos tipos de arco, de uma versão alto-nível da fusão de transições e, muito especialmente, tornando as redes de Petri coloridas o formalismo base das redes de Petri objecto⁹ [Valk, 2004], cuja principal característica é a possibilidade de as marcas poderem também ser redes de Petri. O resultado destas extensões é uma nova classe de redes de Petri de alto-nível utilizada na ferramenta e denominada **rede de Petri referência**¹⁰ [Kummer, 2002].

Dobragem Baseada nos Nós. A **dobragem baseada nos nós** é um mecanismo de estruturação complementar que também pode permitir a construção de modelos mais compactos. Baseia-se no conceito de **vector de nós**, introduzido na dissertação de doutoramento de Luís Gomes [Gomes, 1997], bem como em [Gomes et al., 2002]. Este conceito foi, posteriormente, generalizado a páginas de redes em [Gomes e Barros, 2003] e [Barros e Gomes, 2004c]. Esta última generalização corresponde à utilização de instâncias de redes e de **vectores de instâncias de redes**. Por esta razão, e tal como em [Barros e Gomes, 2004c], quer os vectores de nós quer os vectores de instâncias de rede são aqui referidos por **vectores de componentes**.

A dobragem baseada em nós baseia-se na associação de um factor de multiplicidade a todos os nós da rede de Petri. Nesse sentido, cada nó pode ser visto como a representação de um vector de nós idênticos. Como exemplo, e para clarificar a notação, veja-se a Fig. 2.4b na pág. 30. Utiliza-se *occupied[1..3]* para representar três lugares identificados por *occupied[1]*, *occupied[2]*, e *occupied[3]*. Estes correspondem, respectivamente, aos lugares *occupiedA*, *occupiedB*, e *occupiedC* da Fig. 2.3 na pág. 29. Para outros nós no exemplo, utiliza-se uma notação vectorial semelhante.

⁸Em inglês: *substitution transition*.

⁹Em inglês: *Object Petri nets*.

¹⁰Em inglês: *Reference nets*.

O conjunto de procedimentos necessários à **desdobragem**¹¹ de uma dobragem baseada em nós foram apresentados em [Gomes, 1997; Gomes et al., 2002; Gomes e Barros, 2003], e consideram as formas de desdobrar os seguintes elementos:

- Os arcos que interligam nós com um atributo vectorial (o factor de multiplicidade).
- As inscrições nesses mesmos arcos.
- As marcações iniciais dos vectores de lugares.
- As guardas associadas a vectores de transições.
- Os eventos associados a vectores de transição (dado que a classe de rede de Petri aí apresentada contém partes não-autónomas).

Conforme demonstrado em [Gomes, 1997], quer o modelo na Fig. 2.4a, que utiliza uma rede de Petri colorida, quer o modelo na Fig. 2.4b, que utiliza vectores de nós, têm comportamento equivalente. Na verdade, em ambos os modelos se utilizam facilidades gráficas que permitem diminuir a dimensão dos mesmos sem modificar o seu comportamento: ambos permitem a obtenção de modelos de baixo-nível com comportamento equivalente aos modelos dobrados. Neste sentido, podemos afirmar que a dobragem baseada em marcas e a dobragem baseada em nós têm igual poder de modelação. A dobragem baseada em nós pode ser obtida através da adição de um novo atributo de cor, tal como ilustrado na Fig. 2.4a.

A dobragem baseada em nós, juntamente com os mecanismos de estruturação que serão discutidos na secção seguinte, pode constituir um mecanismo útil para a reutilização de módulos em diferentes contextos, dado que a notação vectorial não obriga à modificação dos componentes replicados e posteriormente compostos (vide Fig. 2.5). Tal é especialmente verdadeiro quando a classe de redes a ser utilizada é de baixo nível: a dobragem por nós permite uma modelação de alto-nível utilizando um modelo de baixo-nível. Por fim, importa notar que a generalização para "vectores" de várias dimensões é imediata.

Tal como referido, a notação vectorial pode ser aplicada ao conceito de instância de rede, originando o conceito de vector de instâncias de rede. Ao utilizar mais do que uma instância de uma rede de Petri iremos utilizar o nome da instância como um prefixo para os nomes de todos os identificadores da instância [Gomes e Barros, 2003; Barros e Gomes, 2004c]. Por sua vez, os nomes das instâncias distinguem-se pelo índice entre parêntesis rectos. No exemplo apresentado, utilizamos três instâncias da mesma rede, correspondentes a três zonas de estacionamento vazias. Consequentemente, também definimos um vector de instâncias de rede. A Fig. 2.5 apresenta uma composição destas três instâncias. Como se verá, a primeira parte desta dissertação, propõe uma forma de especificar, de uma forma textual e também por isso especialmente compacta, a composição ilustrada na Fig. 2.5 para qualquer quantidade de instâncias.

¹¹Em inglês: *unfolding*.

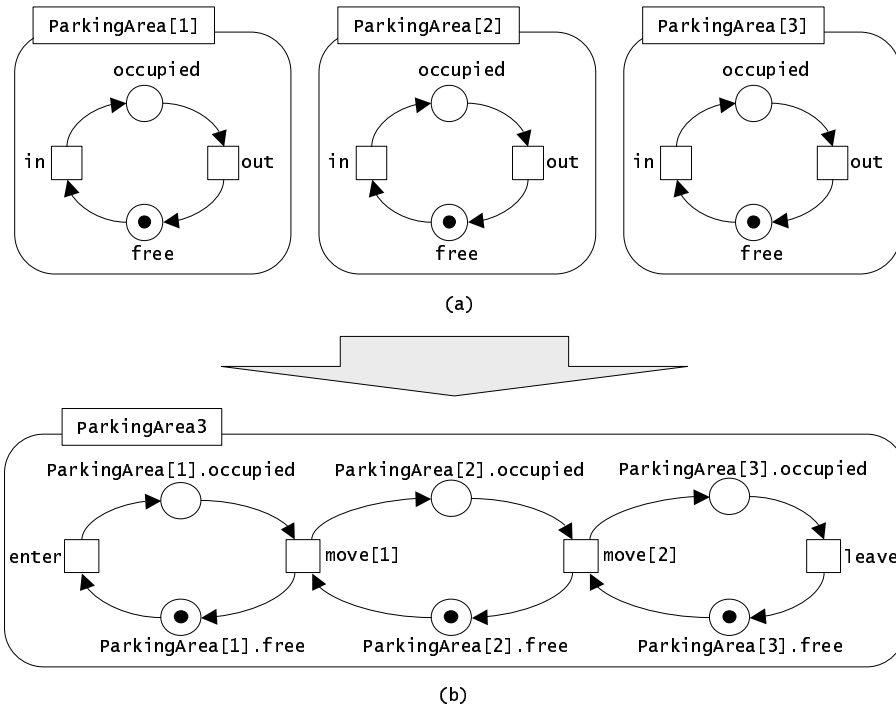


Figura 2.5: Composição de três instâncias de uma mesma rede, através da fusão de transições.

Fusão de Nós na Dobragem Baseada nas Marcas

Conforme já referido, a dobragem baseada em marcas é frequentemente combinada com a fusão de nós. O caso mais conhecido é o das redes de Petri coloridas hierárquicas. Estas recorrem à fusão de lugares em três situações distintas: (1) como suporte ao refinamento de transições; (2) como conveniência gráfica para evitar arcos demasiado longos; (3) como forma de compor diferentes páginas. Em todas estas situações, os lugares fundidos têm de ter o mesmo tipo de dados associado (o mesmo "conjunto de cores"¹²) e apenas a marcação de um desses lugares é tomada em conta. Neste sentido não existe fusão de marcações. Como também não existem outras anotações associadas aos lugares, a fusão de lugares em redes de Petri coloridas hierárquicas resume-se à fusão dos nós.

As redes de Petri coloridas hierárquicas não suportam a fusão de transições. Como as transições têm várias anotações associadas e são os elementos activos do modelo, a sua fusão deve considerar esta maior riqueza conceptual. É o que sucede com os **canais síncronos**¹³, propostos em 1992 por Christensen e Hansen [Christensen e Hansen, 1992], em que a fusão de transições é vista não apenas como uma forma de compor, de forma síncrona, várias páginas, mas principalmente como uma forma de comunicação síncrona entre as várias páginas do modelo. Dada a existência de marcas de alto nível, as transições com canais associados podem ser utilizadas para enviar dados entre si. Tal permite a construção de modelos mais compactos e legíveis.

¹²Em inglês: *colour set*.

¹³Em inglês: *synchronous channel*.

Na referida proposta inicial de Christensen e Hansen, os canais são totalmente simétricos: não é possível identificar emissores ou receptores, quer a nível do controlo quer a nível da transmissão de dados. As transições que partilham um canal síncrono podem disparar desde que cada uma delas possa disparar isoladamente e a respectiva expressão associada ao canal devolva o mesmo valor em ambas as transições. A Fig. 2.6 mostra um exemplo de duas transições (**send** e **recv**) ligadas por um mesmo canal (**ch**). A transição **send** tem associada a expressão de canal **x**, e a transição **recv** tem associada a expressão de canal **y**.

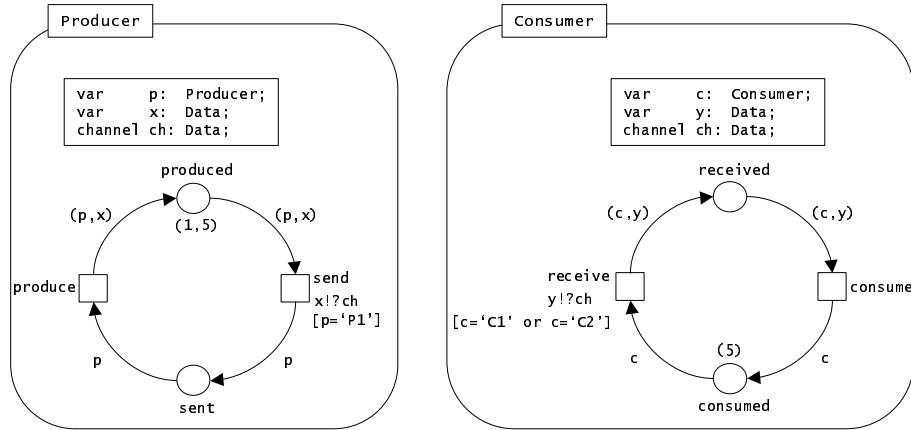


Figura 2.6: Duas transições em duas redes distintas, comunicando por um canal síncrono (**ch**).

A Fig. 2.7 mostra a rede equivalente às da Fig. 2.6: as transições que partilham o mesmo canal são fundidas e a transição resultante tem por guarda a conjunção das guardas das várias transições com o teste de igualdade entre as expressões associadas ao canal (no exemplo da Fig. 2.7 $[x=y \text{ and } p='P1' \text{ and } (c='C1' \text{ or } c='C2')]$).

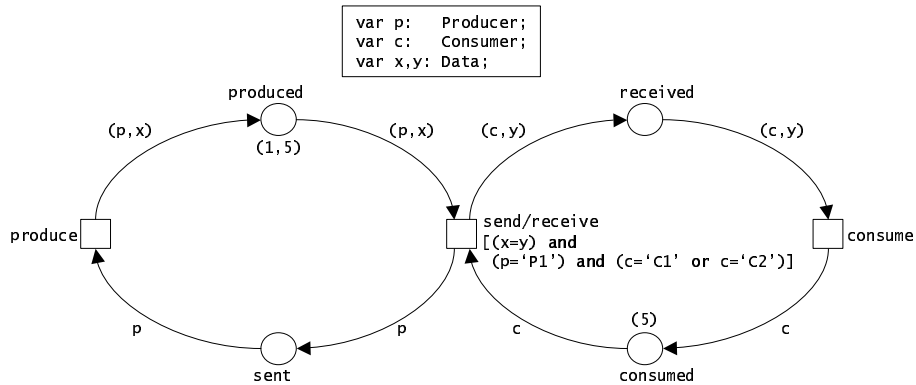


Figura 2.7: Uma rede equivalente à da Fig. 2.6.

A expressão do canal pode ser generalizada a uma sequência de expressões. Tal aproxima os canais da invocação de subprogramas. No entanto, comparando com as semânticas usuais para invocação de subprogramas, ainda encontramos duas diferenças importantes:

1. As expressões associadas ao canal, em cada transição, são avaliadas separadamente; tal

implica a não existência de passagem de parâmetros; como tal, não há especificação de sentidos para a comunicação de dados.

2. Não há sentido de invocação.

Ambas as diferenças resultam da natureza totalmente simétrica dos canais. Esta simetria tem a vantagem de permitir uma total independência entre as várias redes (ou páginas) dado que os canais nunca originam novas dependências entre as respectivas transições. Estas dependências surgem apenas quando todas as redes são unificadas num único modelo.

Ainda assim, é possível adaptar a semântica dos canais síncronos de forma a aproximá-la da invocação de subprogramas. Tal foi proposto por Kummer [Kummer, 2002] e implementado na ferramenta computacional RENEW [Kummer et al., 2004b,a], a qual será novamente referida no final deste capítulo. Kummer propôs a atribuição de uma direcção de invocação: tal obriga cada transição receptora a esperar pela habilitação de uma transição emissora, para poder disparar. A comunicação de dados contínua não dirigida. Por exemplo, uma transição pode passar um valor para outra transição, e receber outro valor dessa transição sem que o sentido da transmissão dos dados seja explicitamente especificado. Conforme referido em [Kummer, 2002], a semântica é semelhante à dos parâmetros nos predicados da linguagem de programação Prolog [Clocksin e Mellish, 1994].

In [Barros e Gomes, 2004d], no contexto da utilização de redes de Petri coloridas na modelação orientada pelos objectos, foi proposta a especificação do sentido de invocação e do sentido de envio e recepção de dados. A proposta pode resumir-se nos seguintes dois pontos:

1. Cada declaração de um canal é qualificada como emissora ou receptora; as respectivas transições são denominadas transição emissora e transição receptora.
2. Cada parâmetro é qualificado com **IN**, **OUT**, ou **INOUT**; a utilização destes qualificadores tem de satisfazer as seguintes regras:
 - Um parâmetro **IN** tem obrigatoriamente de ser vinculado por uma transição emissora. Tipicamente, esse parâmetro é utilizado em um ou mais dos arcos de saída da transição receptora: a transição receptora "utiliza" o parâmetro recebido.
 - Um parâmetro **OUT** tem obrigatoriamente de ser vinculado por uma transição receptora. Tipicamente, esse parâmetro é utilizado em um ou mais dos arcos de saída da transição emissora: a transição emissora "utiliza" o parâmetro recebido.
 - Um parâmetro **INOUT** tem obrigatoriamente de ser vinculado por uma transição receptora e também pela respectiva transição receptora. Tipicamente, esse parâmetro é utilizado em um ou mais dos arcos de saída quer da transição emissora, quer da transição receptora.

O ponto 2 dá aos canais a semântica usual das linguagens de programação para a passagem de parâmetros. Juntamente com o sentido de invocação, no ponto 1, os canais tornam-se aptos à modelação da invocação de subprogramas. Esta possibilidade é apresentada de forma mais desenvolvida no Capítulo 5.

2.2.2 Refinamento e Abstracção

No contexto da literaturas sobre redes de Petri, o **refinamento**¹⁴ corresponde a uma construção **descendente**¹⁵ dos modelos dos sistemas. Ainda em 1974, Parnas alertou para o facto da designação *outside-in* ser claramente melhor [Parnas, 1974]. Infelizmente, e apesar dessa designação ser muito mais exacta, não foi a que se tornou popular. Curiosamente, também a proposta de Wirth, para o desenvolvimento de programas, tem um nome mais preciso: "refinamento por passos"¹⁶[Wirth, 1971].

A **abstracção**¹⁷ é o nome dado ao mecanismo inverso e corresponde a uma abordagem **ascendente**¹⁸. Embora no contexto de uma classe específica de redes de Petri (as redes de Petri predicado-transição), o artigo de Xudong He e John Lee [He e Lee, 1991] apresenta, de forma muito completa, ambos os conceitos.

O refinamento adiciona uma rede, aqui denominada **sub-rede** (ou **subpágina**), dentro de um nó de outra rede, aqui denominada **super-rede** (ou **superpágina**). A abstracção é o procedimento inverso: uma rede é substituída por um nó.

Tal como a dobragem de redes, a estruturação hierárquica de redes permite a manipulação e compreensão de modelos de grandes dimensões. A estruturação hierárquica permite também, de forma simples e directa, uma rápida modificação do nível de abstracção. Tal resulta da redução do modelo ser conseguida através da ocultação de partes deste (de subpáginas que podem facilmente ser visualizadas visto continuarem presentes). Já no caso da dobragem, especialmente no caso da dobragem baseada nas marcas, o modelo encontra-se "fundido" com a abstracção: não é possível obter, de forma directa, diferentes níveis de abstracção a partir do modelo. Tal resulta da utilização intrusiva de anotações não gráficas. Estas substituem parte do modelo gráfico e dada a sua diferente natureza, não permitem a visualização directa de um modelo gráfico equivalente.

Felizmente, a ortogonalidade entre a dobragem de redes e a estruturação hierárquica, permite a aplicação simultânea de ambos os mecanismos. Tal facto, tem sido frequentemente aproveitado para melhorar as capacidades de modelação prática das redes. O exemplo mais conhecido é,

¹⁴Em inglês: *refinement*.

¹⁵Em inglês: *top-down*.

¹⁶Em inglês: *stepwise refinement*.

¹⁷Em inglês: *abstraction*.

¹⁸Em inglês: *bottom-up*.

novamente, o das redes de Petri coloridas e hierárquicas de Jensen [Jensen, 1997c,a,b].

Também a composição por fusão se pode facilmente relacionar com a estruturação hierárquica a qual, como veremos, é tipicamente suportada por fusões entre nós na superpágina e nós na subpágina.

Um par refinamento/abstracção pode ser classificado em uma de duas formas: estática ou dinâmica. Comparando novamente com a área de linguagens de programação, a forma estática corresponde ao conceito de **macro**; a forma dinâmica corresponde à chamada de um **subprograma**. Ambas as formas são apresentadas nas subsecções seguintes.

Macros

O grafo representativo de um modelo em redes de Petri tem dois tipos de nós. Como tal, a forma provavelmente mais intuitiva para decompor modelos consiste na substituição de um nó por uma sub-rede: o nó na super-rede contém outra rede. Neste tipo de decomposição é comum falar-se em **página** em lugar de rede; da mesma forma, utiliza-se **subpágina** e **superpágina** para sub-rede e super-rede, respectivamente. Ao longo da dissertação utilizar-se-á, por vezes, esta nomenclatura para referir as redes constituídas por outras redes. A Fig. 2.8 ilustra, resumidamente, esta nomenclatura.

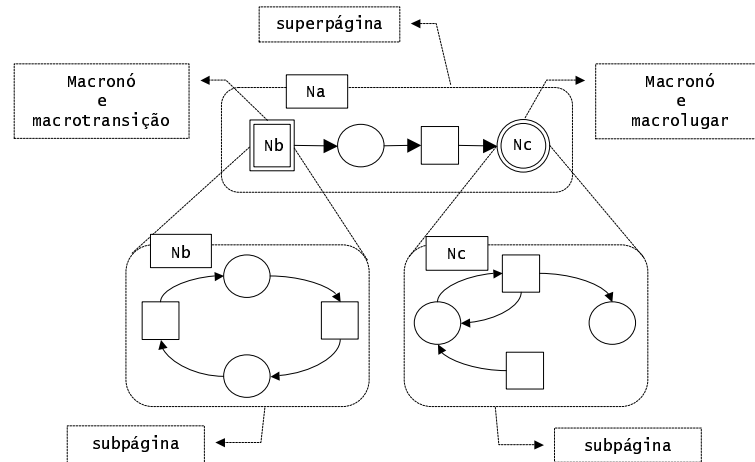


Figura 2.8: Nomenclatura utilizada na estruturação hierárquica de redes.

Os nós que representam a subpágina na superpágina são denominados **macronós** e as subpáginas são então denominadas **macros**. Naturalmente, estas também podem conter macronós e desta forma suportam a especificação de modelos hierárquicos. Assim sendo, a utilização de macros permite um mecanismo de estruturação vertical estático que possibilita uma estrutura em árvore, garantindo-se a não existência de dependências circulares. Obtemos "redes dentro de redes" através da utilização de novos tipos de nós. Mais especificamente obtemos "**nós como redes**". A mesma rede pode ser replicada e cada instância sua pode ser posteriormente vista como uma

macro.

Os macronós podem tomar três formas: **macrotransições**, **macrolugares** e **macroblocos**. Distinguem-se entre si pelos vários tipos de nós a que se podem ligar na superpágina: as macrotransições (propostas nos trabalhos iniciais de Carl Adam Petri [Petri, 1962]) podem apenas ligar-se a lugares ou a macrolugares; os macrolugares (propostos em trabalhos subsequentes, nomeadamente em [Silva, 1985]) podem apenas ligar-se a transições ou a macrotransições; os macroblocos (propostos em [Gomes, 1997]) podem ligar-se quer a lugares ou macrolugares, quer a transições ou macrotransições.

Os macrolugares e as macrotransições são utilizados por numerosas ferramentas e métodos de análise pois oferecem uma dualidade semelhante à que é característica das redes de Petri. Ainda assim, importa notar que esta semelhança é apenas sintáctica: dado, no contexto da superpágina, possuírem comportamento interno, nem os macrolugares se comportam como lugares, nem as macrotransições se comportam como transições.

Alguns autores não permitem a interligação entre macronós (e.g. [Huber et al., 1989]). No entanto, de um ponto de vista prático, valorizando a aplicabilidade do modelo, essa possibilidade de interligação pode ser extremamente útil tendo sido já apresentada e defendida em [Gomes et al., 2002] e [Gomes e Barros, 2003].

A ligação entre o macronó e os nós na página em que está inserido pode ser feita de duas formas: (1) a nível do arco; (2) a nível do nó complementar. A Fig. 2.9 ilustra ambas as formas para o caso de uma macrotransição. Com as devidas adaptações os mesmos métodos podem ser aplicados para um macrolugar ou macronó. No primeiro caso (Fig. 2.9a), para cada arco ligado ao macronó, especifica-se qual o nó na subpágina a que se encontra ligado. Corresponde a uma utilização implícita da fusão de transições: cada arco na superpágina é visto como ligado a uma transição que é fundida com as transições t_1 e t_2 na subpágina. No segundo caso (Fig. 2.9b), para cada nó ligado ao macronó, especifica-se qual o nó na subpágina com o qual este é fundido: a fusão de nós serve de suporte à estruturação hierárquica de forma explícita. Esta forma é a preferida, em particular quando se consideram redes de Petri coloridas ou de alto-nível, dado basear-se num conceito de colagem ou sobreposição de nós e páginas que, provavelmente, é mais intuitivo e legível para a maior parte dos utilizadores. Os nós na subpágina que são fundidos com nós na superpágina são, por vezes (por exemplo em [He e Lee, 1991]), denominados **nós fronteira** ou apenas **fronteira**. No exemplo na Fig. 2.9b, os nós fronteira são os lugares p_3 e p_4 .

Neste sentido, o conceito de composição de modelos, por fusão de nós, constitui o suporte para a representação hierárquica de modelos e, portanto para o refinamento e abstracção de modelos.

Este mapeamento, entre cada um dos nós ligados ao macronó e cada um dos nós na fronteira da respectiva sub-rede, pode ser entendido e definido como um conjunto de fusões de pares de

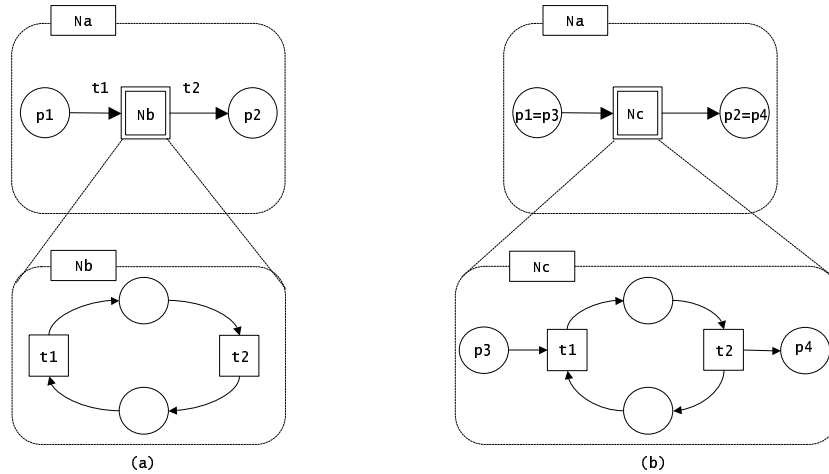


Figura 2.9: Duas formas de especificar a ligação entre um macronó e a respectiva subpágina: a) a nível do arco; b) por fusão de nós.

nós. No entanto, é também possível considerar que apenas os nós na superpágina, ou apenas os nós na subpágina, são nós "reais", sendo os outros vistos como nós "referência". Por exemplo, no caso das redes de Petri coloridas e hierárquicas, os lugares na superpágina são denominados *sockets places*, e os correspondentes lugares na subpágina **lugares porto**¹⁹.

Desde muito cedo, os macronós foram considerados úteis mesmo em redes de alto nível [Huber et al., 1989]. No entanto, apenas a macrotransição foi implementada nas ferramentas computacionais de referência para redes de Petri coloridas (Design-CPN [s.a., 2004c] e CPN Tools [s.a., 2004b]), sob o nome de **transição de substituição**²⁰.

O conceito de macrolugar tem sido bastante utilizado no suporte a métodos de redução e em algumas áreas de aplicação, como, por exemplo, o projecto de hardware e os sistemas de manufatura [Silva, 1985].

Os macroblocos foram apresentados em [Gomes, 1997; Gomes et al., 2002; Gomes e Barros, 2003] e correspondem a ignorar, ao nível sintáctico, a dualidade lugar-transição: um macrobloco é representado na superpágina, por um novo tipo de nó que se liga a pelo menos um lugar ou a um macrolugar com semântica de lugar e também a, pelo menos, uma transição ou macrotransição com semântica de transição. Tal como sucede com os macrolugares e as macrotransições, apenas os modelos sem macroblocos podem ser executados: os macronós não têm semântica própria pelo que têm sempre de ser substituídos pelas respectivas subpáginas para que a superpágina seja executável. Neste sentido, os macronós não violam a dualidade intrínseca das redes de Petri, e tal é verdadeiro também para os macroblocos.

¹⁹Em inglês: *port places*.

²⁰Em inglês: *substitution transition*.

Exemplo. Quando se utilizam macros, é necessário obter o modelo plano correspondente, ou seja, um modelo com igual comportamento mas em que todos os macronós foram iterativamente substituídos pelas subpáginas respectivas. Esta substituição é intuitiva, apenas os detalhes dependem da forma utilizada para especificar a ligação entre cada macronó e a respectiva subpágina: basicamente, cada macronó é removido e, em seu lugar, é inserida uma instância da subpágina a ele associada; os nós fronteira, na subpágina, são ligados aos respectivos nós na superpágina.

Seguidamente, apresenta-se um modelo que utiliza o conceito de macronós de várias formas. A Fig. 2.10 apresenta as quatro redes que serão utilizadas. Estas redes devem ser vistas como geradoras de instâncias de páginas que podem ser utilizadas para construir modelos. Pretendemos modelar o controlador de um parque de estacionamento com uma entrada (**Enter**), uma saída (**Leave**), três áreas de estacionamento (**ParkingArea** agora com seis lugares livres) e passagens entre estas áreas (**Move**). As transições e os lugares são interpretados como estando associados a estados de detectores de passagem. Em particular, as transições **firstUp**, **secondUp**, **firstDown** e **secondDown** do modelo **Move** da Fig. 2.10 estão associadas à activação e desactivação de sensores de presença que servem para detectar a passagem de carros entre a zona 1 e a zona 2.

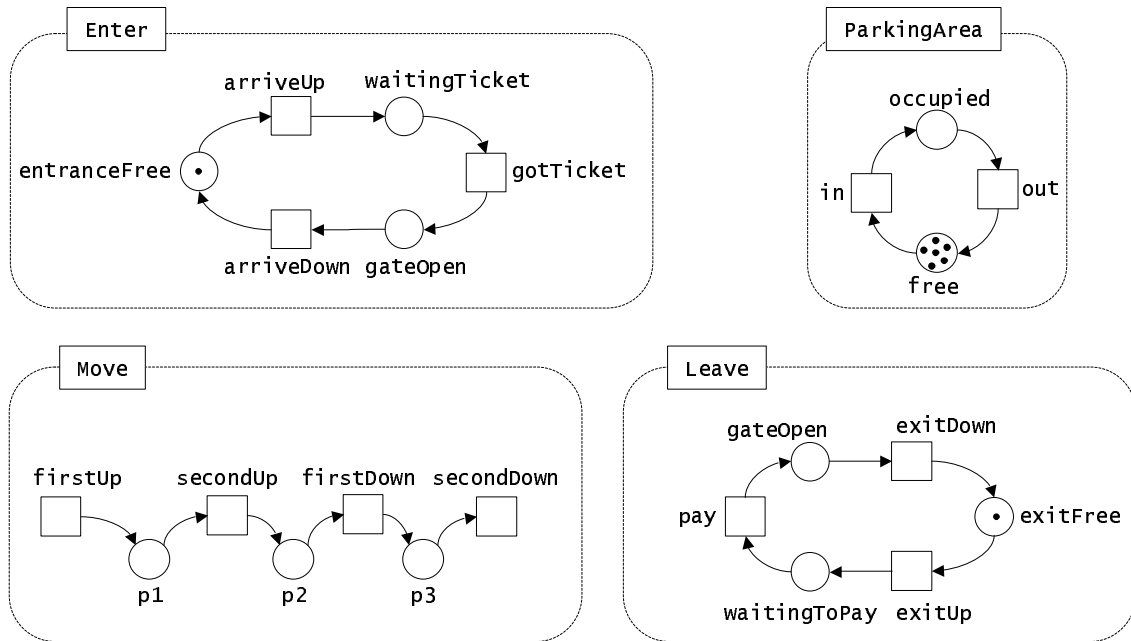


Figura 2.10: Submodelos para o parque de estacionamento.

Tomando como ponto de partida o modelo da Fig. 2.5b na pág. 33 representando o modelo de um parque de estacionamento com três zonas, dispostas em sequência, procedemos ao refinamento das transições fundidas, nomeadamente de **move[1]** e **move[2]**, por duas instâncias da rede **Move**, bem como das transições **enter** e **leave** pelas redes **Enter** e **Leave**, respectivamente. A Fig. 2.11 apresenta o modelo plano resultante. Tipicamente, este modelo plano não será gerado na sua forma gráfica, dada a sua grande dimensão. No entanto, é o modelo utilizado para a execução, para a análise e, possivelmente, como base para a geração de código. A Fig. 2.12

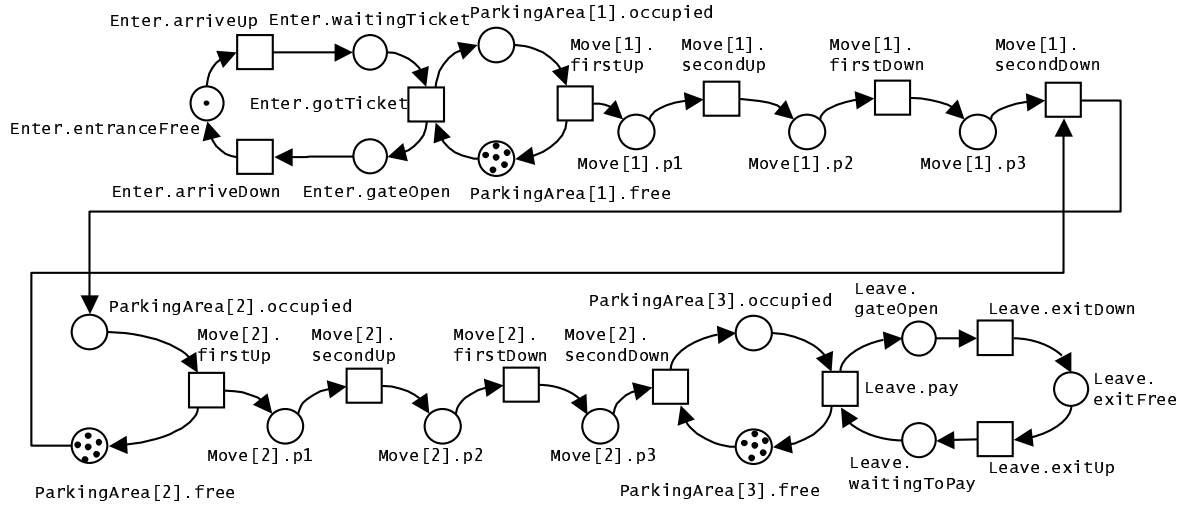


Figura 2.11: Modelo plano resultante do refinamento do modelo na Fig. 2.5.

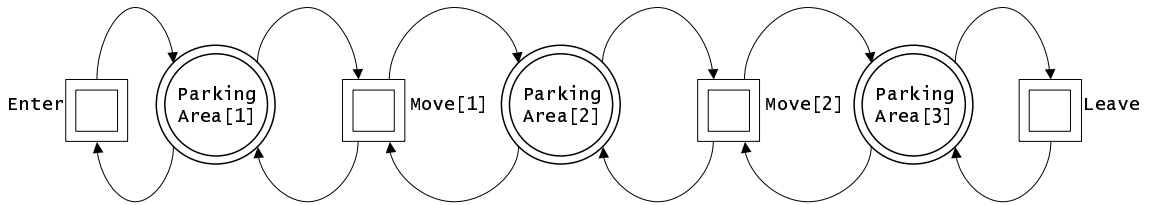


Figura 2.12: Refinamento do modelo de um parque de estacionamento.

apresenta o mesmo modelo, sob a forma de uma superpágina numa estruturação hierárquica, utilizando macronós para representar as várias subpáginas que o constituem. É utilizada uma linha dupla para representar os macronós.

Invocações

As secções anteriores apresentaram a composição estática e a estruturação hierárquica que ela torna possível. Naturalmente, é também possível modelar composições dinâmicas entre páginas. Também se utilizam nós mas neste caso não se fala de macronós mas sim de **nós de invocação**²¹ [Huber et al., 1989]. Estes não contêm subpáginas de forma permanente. Em vez disso, as subpáginas são criadas dinamicamente. Por esta razão, traça-se por vezes um paralelismo com as linguagens de programação: as subpáginas criadas são denominadas procedimentos, e a sua criação passa a corresponder a uma invocação de procedimentos. Na realidade, este tipo de estruturação de modelos raramente é utilizado, provavelmente por modificar, de forma radical, a semântica das redes de Petri que se baseia numa estrutura imutável. Ainda assim, algumas classes de redes de Petri suportam este tipo de estruturação dinâmica: por exemplo, para além das redes de Petri objecto de Lakos e Valk, também as Thorn Petri nets [S. Schöf et al., 1995] suportam a criação dinâmica de redes.

²¹Em inglês: *invocation nodes*.

Dado o comportamento activo implícito na criação de subpáginas, é frequente associar-se esta criação de páginas a transições: quando uma transição de invocação dispara, uma nova página é criada e passa a ser executada em paralelo com o modelo existente. Embora também possamos considerar a utilização dos lugares como criadores de novas subpáginas, esta não é uma solução intuitiva: tal só se pode basear na criação (e destruição) de marcas no lugar e mesmo essas alterações são provocadas por transições. Como tal, parece preferível atribuir essa responsabilidade às transições de uma forma directa, em lugar de uma forma indirecta baseada na criação e destruição de marcas. No entanto, dado que as transições criam essas marcas, é muito intuitivo associar cada marca criada a uma nova subpágina. Esta atitude de modelação encontra-se já extremamente próxima dos conceitos do desenvolvimento orientado pelos objectos com origem na linguagem Simula na década de sessenta, e posteriormente popularizados por numerosas linguagens de programação. Na última década esses mesmo conceitos, tornaram-se populares mesmo a nível do desenho²² de sistemas .

Com base em trabalhos anteriores sobre **redes dentro de redes**²³, Valk propôs as **redes de Petri objecto** [Valk, 1996, 2004]²⁴. Intuitivamente, as redes de Petri objecto oferecem uma composição dinâmica e "interna": tal como nas tradicionais bonecas russas, as *matrioscas*, obtemos redes dentro de redes. Contrariamente às macros em que as subpáginas se encontram dentro dos nós, nas redes de Petri objecto, as subpáginas encontram-se dentro das marcas. Como a criação e destruição de marcas constitui parte da semântica característica das redes de Petri, as redes de Petri objecto permitem uma forma elegante de estruturar de forma dinâmica os modelos de redes de Petri.

As redes de Petri objecto admitem duas variantes na sua semântica para a criação dinâmica de subpáginas [Valk, 2004]: (1) uma semântica *baseada em valores*²⁵; (2) uma semântica *baseada em referências*²⁶. Na semântica baseada em valores, cada marca criada corresponde a uma nova subpágina. Esta subpágina é destruída quando a marca é removida. Na semântica baseada em referências, cada marca corresponde a uma referência para uma nova subpágina. Tal significa que uma mesma subpágina pode ser referenciada por várias marcas no mesmo ou em diferentes lugares. As **rede de Petri referência**²⁷ de Kummer [Kummer, 2002] utilizam a semântica baseada em referências.

Existem numerosas propostas de classes de redes de Petri baseadas em objectos e o livro [Agha et al., 2001] constitui uma colectânea muito completa deste tipo de propostas. De entre essas propostas, sublinha-se a contribuição de Lakos [Lakos, 2001] que também utiliza as marcas como subpáginas. Ainda assim a apresentação seguinte centra-se nas rede de Petri referência de Kummer devido ao suporte existente para a utilização prática das mesmas, sob a forma da

²²Em inglês: *design*.

²³Em inglês: *nets-within-nets*.

²⁴Em inglês: *Object Petri nets*.

²⁵Em inglês: *value semantics*.

²⁶Em inglês: *reference semantics*.

²⁷Em inglês: *Reference Nets*.

ferramenta computacional RENEW[Kummer et al., 2004a,b].

Redes de Petri referência. Diferentemente das redes de Petri objecto de Valk, que são definidas como redes lugar-transição interligadas por fusão de transições, as redes de Petri referência são baseadas em redes de Petri coloridas: cada nó pode ser uma referência ou um tipo de dados complexo.

A possibilidade de uma marca poder especificar uma referência para outra página é a principal característica identificadora das redes de Petri referência. As transições podem criar novas instâncias de páginas e devolvem as respectivas referências que podem ser depositadas num ou mais lugares de saída. Cada uma destas novas páginas é executada, em verdadeira concorrência, com todas as restantes páginas, incluindo a página da transição que a criou. A execução do modelo inicia-se pela página escolhida, para a qual uma instância é automaticamente criada. As outras instâncias das páginas definidas inicialmente, são criadas, directa ou indirectamente, por essa instância inicial. As várias instâncias comunicam entre si através de um tipo particular de canais síncronos. Tal como na proposta inicial de Christensen [Christensen e Hansen, 1992], a comunicação de dados continua a ser bidireccional e sem especificação do sentido de comunicação de cada expressão. No entanto, as declarações de canais são classificadas em emissoras ou receptoras: a transição com uma declaração receptora só pode disparar quando uma transição com uma declaração emissora estiver habilitada, para o mesmo canal.

Seguidamente, modelam-se algumas especificações adicionais para o parque de estacionamento, utilizando uma rede de Petri referência.

Um exemplo de rede de Petri referência. Quando as marcas podem ser redes, como sucede nas redes de Petri referência, uma interpretação frequente consiste em considerar as marcas como objectos que se deslocam de um local para outro. Estes objectos são denominados **agentes** [Moldt e Wienberg, 1997; Köhler et al., 2001]. Naturalmente, estes objectos ou agentes, devem exhibir um comportamento que interesse modelar. Caso contrário, poderão ser especificados por marcas de alto-nível tal como nas redes de Petri coloridas. Este comportamento é então modelado por uma rede. As várias redes que modelam o comportamento dos agentes, bem como a rede que modela o ambiente em que se deslocam, interagem por meio de canais síncronos.

Como exemplo, vamos considerar novamente o exemplo do controlador para o parque de estacionamento com três zonas de estacionamento (1 a 3). No entanto, vamos admitir que o movimento dos automóveis entre as zonas de estacionamento é vigiado. Assim, os automóveis podem deslocar-se entre a zona i e a zona $i+1$, ou entre a zona $i+1$ e a zona i ; podem também deslocar-se em qualquer desses sentidos e posteriormente fazer marcha atrás. Considera-se que não é permitida uma deslocação entre a zona $i+1$ e a zona i , ou regressar à zona i depois de iniciar a travessia para a zona $i+1$. Nessas situações deve soar um alarme e o sistema deve fotografar o automóvel.

A Fig. 2.13 mostra a rede **Passage**²⁸ que modela a passagem de um automóvel entre duas zonas. Corresponde a um modelo mais elaborado da rede **Move** na Fig. 2.10, agora modelada por uma rede de Petri referência tal como as suportadas pela ferramenta RENEW.

Tal como anteriormente, as marcas representam automóveis, mas agora esses automóveis são também modelados por redes, mais especificamente são referências para instâncias da rede **Car** (ilustrada na Fig. 2.14). Estas instâncias da rede **Car** comunicam com a instância da rede **Passage** através de canais síncronos. Para cada carro que inicia a travessia de uma zona para outra, é criada uma instância da rede **Car**.

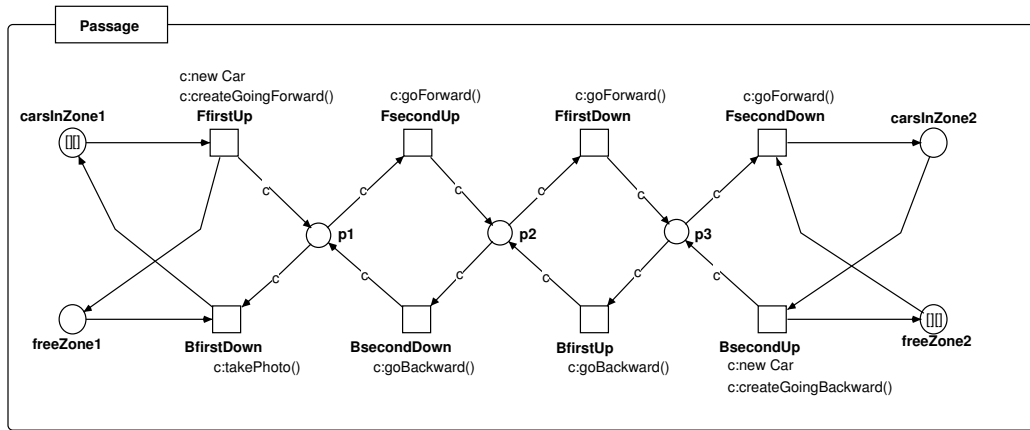


Figura 2.13: Rede **Passage** para modelação da passagem entre zonas de estacionamento, com detecção da inversão de marcha.

A rede **Passage** prevê a possibilidade de um carro fazer marcha atrás após ter iniciado a travessia entre duas zonas. Também prevê a possibilidade de um carro entrar em sentido contrário (de $i+1$ para i) através da transição **BsecondUp** no canto inferior direito da Fig. 2.13. Em ambos os casos um alarme dispara e só é desligado quando o carro regressa ao sentido de marcha permitido. Tal como na rede **Move**, os nomes das transições modelam eventos de entrada que correspondem à activação ou desactivação dos detectores de passagem. Agora, para cada activação ou desactivação de um sensor existe uma transição para a modelação de cada um dos dois sentidos possíveis de deslocamento. Estas transições distinguem-se pela utilização dos prefixos **F** e **B**, para os movimentos no sentido *forward* e no sentido *backward*, respectivamente. Se um carro regressa à zona de onde havia partido em sentido oposto ao inicial, é fotografado (*vide* transição **BfirstDown** no canto inferior-esquerdo da Fig. 2.13).

A transição **FfirstUp** no canto superior-esquerdo da Fig. 2.13 cria uma nova instância da classe **Car** (*vide* Fig. 2.14) e afecta a variável c com a respectiva referência: $c: \text{new Car}$. Através da utilização de um canal síncrono denominado **createGoingForward**, a mesma transição funde-se com a transição **createForward** na rede **Car**.

²⁸Este modelo e os seguintes foram construídos na ferramenta RENEW.

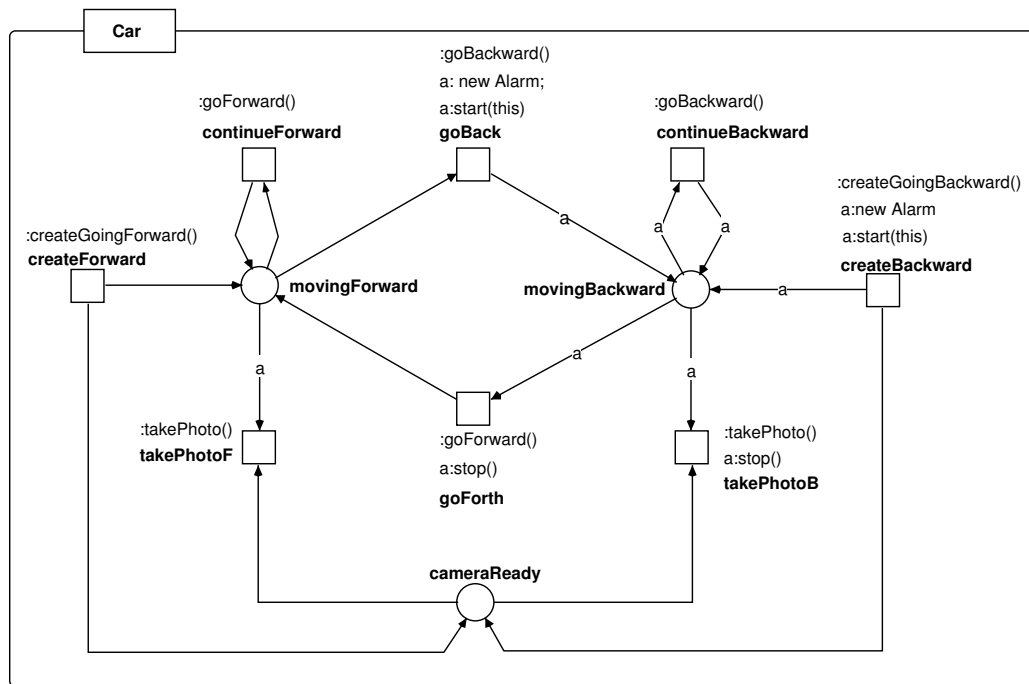


Figura 2.14: A rede Car.

A rede **Car** modela dois estados possíveis para os seus objectos que fluem ao longo da passagem: **movingForward** e **movingBackward**. No ponto de vista da rede **Passage**, isto significa que o carro ou se desloca da zona i para a zona $i+1$ ou da zona $i+1$ para a zona i , respectivamente.

Quando um carro faz marcha atrás, a transição **goBack** na Fig. 2.14 cria uma nova instância da rede **Alarm** (*vide* Fig. 2.15) e a transição **create**, com o canal síncrono **start**, é disparada. Note-se que o canal síncrono **start** tem como parâmetro a instância da rede **Car**, a qual na rede **Car** é especificado pela palavra reservada **this**.

O alarme pára quando dispara a transição **goForth** ou **takePhotoB** na Fig. 2.14. Cada uma destas transições também destrói a referência **a** para a instância da rede **Alarm**. Como não existe mais nenhuma referência, e desde que seja considerada morta (sem evoluções), a instância da rede **Alarm** pode ser removida da memória.

Algo totalmente semelhante acontece com as referências para a rede **Car** quando passam pela transição **BfirstDown** (no canto inferior esquerdo da Fig. 2.13) e pela transição **FsecondDown** (no canto superior direito da Fig. 2.13 na rede **Passage**).

Um carro que se desloca de uma zona de estacionamento i para uma zona de estacionamento $i+1$, e continua esse movimento sem inverter o sentido da sua marcha, é modelado, na rede **Passage**, pelo canal síncrono **goForward** e pelas transições **FsecondUp**, **FfirstDown** e **FsecondDown**. De forma idêntica, um carro que se desloca de uma zona de estacionamento $i+1$ para uma zona de estacionamento i , e continua esse movimento sem inverter o sentido da sua marcha, é modelado,

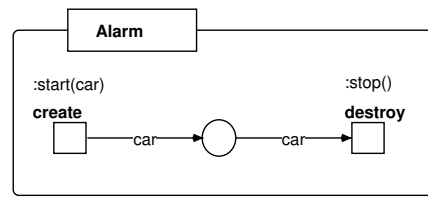


Figura 2.15: A rede Alarm.

na rede *Passage*, pelo canal síncrono *goBackward* e pelas transições *BfirstUp*, *BsecondDown* e *BfirstDown*.

O sistema de controlo tira uma fotografia a cada carro, no lugar *p1*, que desactive o detector de presença, ou seja quando dispara a transição *BfirstDown*. Tal é modelado por um canal síncrono denominado *takePhoto*, entre a transição *BfirstDown* na rede *Passage*, e as transições *takePhotoF* e *takePhotoB* na rede *Car*, uma para cada um dos dois estados possíveis do carro: *movingForward* e *movingBackward*.

2.3 Conclusões

Os mecanismos de estruturação e composição aqui descritos reflectem as principais abordagens actuais, no âmbito das redes de Petri, para combater o problema da "explosão do modelo". No entanto, todos surgem no contexto de trabalhos teóricos ou, mais frequentemente, como propostas *ad-hoc* associadas a ferramentas computacionais.

Na primeira parte desta dissertação, propõem-se duas operações básicas, denominadas adição e subtracção, para suporte não apenas à estruturação e composição de modelos mas também à modificação dos mesmos. Estas operações podem ser adaptadas a quaisquer classes comuns de redes de Petri pelo que permitem uma unificação dos mecanismos já apresentados.

Na segunda parte da dissertação, apresenta-se uma proposta para a utilização de redes de Petri no desenvolvimento orientado pelos objectos. Para tal, recorre-se a duas formas específicas de fusão de lugares e transições em redes de Petri coloridas, bem como à operação de adição descrita na primeira parte.

Parte I

Composição e Modificação de Modelos em Redes de Petri

Capítulo 3

Definição Modular de Modelos

ESTRUTURA HIERÁRQUICA é uma forma de separar concerns. Normalmente não funciona bem e é melhor algum tipo de estrutura paralela. (...) A estrutura hierárquica é a areia na qual se constroem os métodos descendentes.

– Michael Jackson, 1995

Este capítulo inicia-se com uma apresentação informal das operações de adição e subtracção de modelos em redes de Petri. Estas operações permitem um suporte operacional para a construção e modificação de modelos de forma ascendente e descendente. Segue-se uma breve discussão sobre a importância de reflectir as alterações dos requisitos em modelos já existentes, estabelecendo uma analogia com os problemas identificados na área do desenvolvimento de software orientado pelos aspectos. Nesse sentido, apresenta-se uma proposta para a modularização de modelos em redes de Petri, que suporta a adição de módulos que entrecortam¹ o modelo existente. A apresentação baseia-se no exemplo do parque de estacionamento já introduzido no Capítulo 2. Finalmente, as operações são definidas formalmente para redes não-marcadas e posteriormente generalizadas a redes de Petri anotadas.

Todos os mecanismos de composição, estruturação e abstracção de modelos em redes de Petri existentes, e resumidamente apresentados no Capítulo 2, foram motivados pela busca de melhores formas de construir modelos. Nesse sentido, todos esses métodos enfatizam ou a necessidade de construir modelos a partir de submodelos — a criação de modelos mais complexos por refinamentos sucessivos do modelo actual — ou, simplesmente, a visualização hierárquica de modelos não hierárquicos. Daqui podemos concluir que as redes de Petri já dispõem dos mecanismos

¹Em inglês: *crosscut*.

de estruturação e abstracção que é possível encontrar na generalidade das linguagens (textuais) de programação e que são desejáveis também em linguagens gráficas de programação ou especificação. No entanto, esses mecanismos são definidos de forma *ad-hoc* em cada classe de redes e não parecem motivados pela necessidade de modificar modelos já existentes.

Seguidamente, apresentam-se duas operações para a especificação de modificações em modelos em redes de Petri. Estas duas operações permitem quer a construção de modelos de uma forma descendente ou ascendente, quer a modificação desses mesmos modelos por via aditiva ou subtractiva. Correspondentemente, as operações foram denominadas **adição de redes** e **subtracção de redes**. A adição é a operação fundamental por permitir a composição de modelos. Já a subtracção é definida para permitir a remoção de adições prévias ou para remover parte do modelo, eventualmente para em seguida lhe adicionar um novo submodelo. Por exemplo, um refinamento de um nó pode ser visto como a subtracção desse nó seguida da adição da sub-rede correspondente.

Note-se que uma atitude comum na literatura sobre redes de Petri é a de se considerar que as modificações em modelos devem necessariamente manter determinadas propriedades. Esta atitude pode ser denominada **composição com preservação de propriedades**. A atitude que justifica a definição das operações de adição e subtracção tem por base uma diferente perspectiva que não substitui a primeira, antes a complementa: a da **composição de modelos para adição ou remoção de propriedades ou funcionalidades**. Visto de outra forma, a preocupação com a manutenção de propriedades em modelos é tradicional nas abordagens mais formais que exploram as características fundamentais dos modelos de forma a garantirem uma constância de determinadas propriedades mesmo após modificação do modelo. Outra atitude é a de disponibilizar formas genéricas e práticas de especificar modificações em modelos. Estas enfatizam a aplicabilidade, generalidade e legibilidade das especificações relativas a composições e modificações dos modelos. A adição e a subtracção de redes pretendem dar resposta a este segundo tipo de atitude.

Os exemplos apresentados na secção seguinte centram-se numa perspectiva ascendente. A utilização das mesmas operações numa atitude descendente é discutida na Secção 3.1.3.

3.1 Adição e Subtracção de Modelos

De um modo informal e através de vários exemplos de composição, esta secção apresenta a definição de redes utilizando operações sobre outras redes. Estas definições, quando aditivas, são claramente construções ascendentes. No entanto, são acompanhadas das respectivas representações diagramáticas denominadas **diagramas de adição**. Estes diagramas fornecem uma visão simultaneamente ascendente e descendente.

3.1.1 Definição Operacional de Redes

Existem numerosas classes de redes de Petri, cada qual com diversos tipos de anotações, nós, arcos e semânticas associados. Tal facto motivou já a criação de uma linguagem que permite a especificação dessas diferentes classes num formato textual. Essa linguagem denomina-se *Petri net Markup Language* (PNML), é baseada na linguagem de descrição de dados XML [W3C, 2005], e deverá constituir parte da norma ISO para redes de Petri de alto-nível [s.a., 2005c]. Dispõe de um sítio de suporte na Internet [s.a., 2004d] e existem vários artigos publicados [Jünger et al., 2000; Kindler e Weber, 2001; Weber e Kindler, 2003; Billington et al., 2003] dos quais o mais recente e relevante, por ser o mais actualizado e próximo da proposta a normalizar, é o publicado na conferência das redes de Petri de 2003, realizada em Eindhoven na Holanda [Billington et al., 2003].

A PNML suporta a definição de várias classes de redes de Petri com base em dois princípios fundamentais:

1. Todas as classes de redes de Petri têm lugares, transições e arcos.
2. Todas as classes de redes de Petri podem ser expressas por meio da adição de vários tipos de anotação a um, ou mais, destes três tipos de elementos. É também possível definir anotações para a própria rede.

As operações sobre redes que aqui se propõem separam totalmente os dois aspectos seguintes: (1) a especificação das composições entre redes; (2) as anotações nas próprias redes. Ou seja, não é necessário modificar a definição das redes de forma a suportar novas anotações que seriam utilizadas para especificar a composição dessas mesmas redes.

Na verdade, as operações ilustradas nesta secção devem ser vistas como uma nova forma de definir redes. Assim, e fazendo uma analogia com a teoria de conjuntos, dada uma determinada classe de redes de Petri, temos duas formas distintas de definir modelos utilizando essa classe de redes:

1. A definição **em extensão** que especifica cada um dos elementos constituintes da rede: lugares, transições, arcos, e respectivas anotações de cada um desses elementos.
2. A definição **em compreensão** (ou **operacional**) em que definimos uma rede especificando operações sobre instâncias de outras redes (as **redes operando**).

A primeira é suportada pela linguagem PNML. A segunda é a que aqui se proporá.

Uma rede definida de forma operacional pode ser utilizada como rede operando de outras especificações de redes. Desta forma, a definição operacional oferece uma forma extremamente flexível e genérica de definir redes. Em particular, oferece as seguintes três vantagens:

1. Visto não serem necessárias anotações adicionais especificando as composições, novos modelos podem ser definidos sem que sejam necessárias quaisquer modificações nas definições de redes existentes.
2. Permite uma rápida especificação de modelos de grandes dimensões que na prática nunca poderiam ser editados manualmente. Esta potencialidade é ainda mais relevante quando surgem repetições estruturais em modelos em redes de baixo-nível.
3. Suporta quer a construção modular de modelos, quer a sua posterior modificação.

A utilização de operações de forma ortogonal às anotações características da classe de redes em causa é fundamentalmente diferente da utilização de anotações adicionais na classe de redes para especificar as interconexões entre módulos. É esta ortogonalidade que está na base das três vantagens apontadas. É também ela que permite a utilização destas operações quer na construção de modelos, quer na sua modificação, mesmo quando esta afecta vários módulos.

A desvantagem que pode ser apontada à definição operacional advém da sua natureza textual. Mais concretamente, e partindo do princípio que se utilizou um editor gráfico para criar alguns dos módulos (redes), quando se recorre à especificação operacional parte da especificação do modelo total passa a estar na forma textual. A diferença face às anotações que é possível encontrar em muitas classes de redes de Petri, e em especial nas redes de Petri de alto nível, está na utilização de anotações de forma independente dos nós, arcos, ou módulos: as operações utilizam vários módulos e não privilegiam nenhum em particular. Assim sendo, a ligação entre os modelos gráficos, que definem redes em extensão, e as redes definidas operacionalmente deve ser feita através da adição de funcionalidades aos editores gráficos. Nomeadamente, um editor de modelos em redes de Petri deverá suportar os seguintes requisitos adicionais:

- Disposição² automática de redes definidas operacionalmente de forma a poderem ser visualizadas sempre que a sua dimensão o permita.
- Visualização de um modelo abstracto que especifique as ligações entre as várias redes compostas.
- Indicação, em cada rede (módulo) de quais as definições operacionais em que são utilizadas instâncias suas.

A realização do primeiro requisito implica a resolução de um problema muito complexo: o da geração automática de disposições de grafos. Felizmente existem já ferramentas computacionais capazes de obter bons resultados. Entre elas destaca-se o Graphviz [Ellson et al., 2004] devido à sua qualidade e também por se tratar de uma ferramenta *open source*. As ferramentas RENEW[Kummer et al., 2004a,b] e *Petri net Kernel* [Abdourahaman et al., 2002] também são

²Em inglês: *layout*.

capazes de dispor automaticamente modelos em redes de Petri. O segundo requisito pode ser realizado pelos diagramas de adição que serão utilizados ao longo deste capítulo e que permitem uma visualização gráfica das operações definidas. O terceiro requisito pode ser trivialmente satisfeito pelo editor de modelos.

3.1.2 Construção Operacional de Modelos – a Perspectiva Ascendente

Nesta secção retoma-se o modelo do parque de estacionamento, já apresentado no capítulo anterior, agora para exemplificar a utilização das operações sobre redes, nomeadamente a adição e subtracção de redes, bem como os diagramas de adição — propostos nesta dissertação.

Começa-se por reconsiderar as redes **Enter**, **ParkingArea** e **Leave** da Fig. 2.10 na pág. 40 e que se apresentam novamente na Fig. 3.1, agora nas suas versões não-marcadas. Utiliza-se também uma versão simplificada da rede **Passage** da Fig. 2.13.

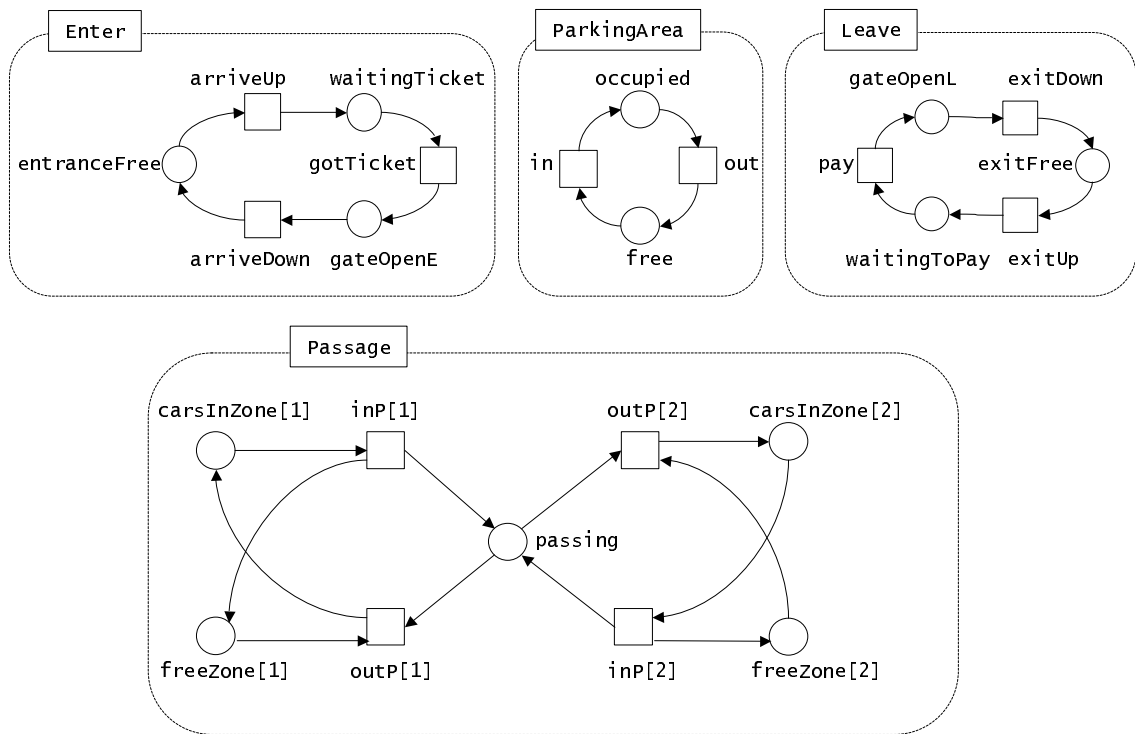


Figura 3.1: Redes **Enter**, **ParkingArea**, **Leave** e **Passage**.

Utilizando estas redes como geradoras de instâncias, a rede **ParkA** representada graficamente na Fig. 3.2 da página 54 modela um parque com uma entrada, uma área de estacionamento e uma saída. Como tal utiliza uma instância da rede **Enter**, uma instância da rede **ParkingArea** e uma instância da rede **Leave** (*vide* Fig. 3.1). Utilizando a operação de adição de redes podemos definir este novo modelo da seguinte forma:

ParkA := (Enter + ParkingArea + Leave)(/gotTicket/in/ -> in,
/out/pay/ -> out)

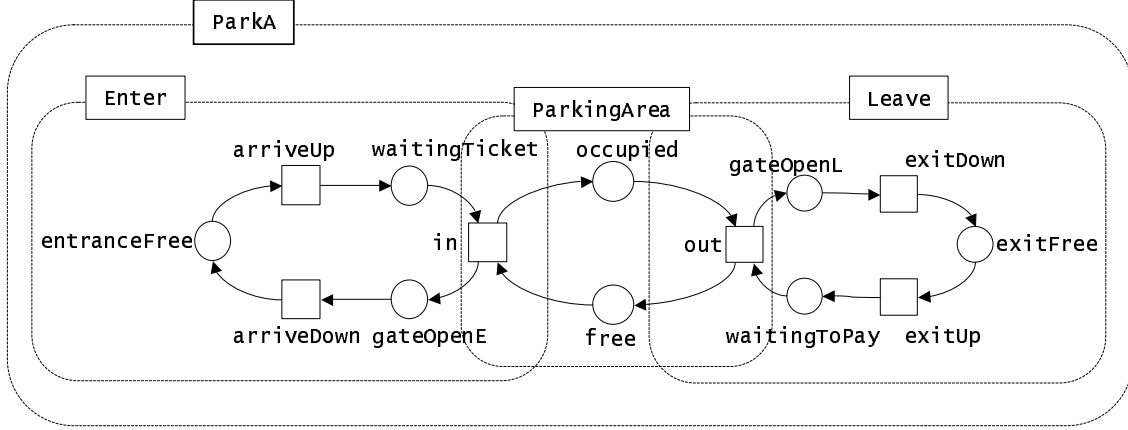


Figura 3.2: Rede ParkA.

A rede **ParkA** é obtida por composição das três redes referidas que devem ser vistas como módulos ou submodelos: a adição de redes especifica, em primeiro lugar, uma união disjunta das redes envolvidas (no exemplo **Enter + ParkingArea + Leave**) e, em segundo lugar, uma lista de pares separados por vírgulas denominada **colapso**. Cada um destes pares contém uma sublista de nós a fundir, denominada **conjunto de fusão** e o nome do **nó resultado**. O par (/sublista/ -> resultado) denomina-se **conjunto de fusão nominativo**. Na definição do modelo **ParkA** o colapso é (/gotTicket/in/ -> in, /out/pay/ -> out) que utiliza **identificadores reduzidos** para os nós nas redes adicionadas. Tal é possível por não haver ambiguidade. Caso contrário seria necessário utilizar os **identificadores completos** em que o identificador da rede (um ou mais) é utilizado como prefixo. Por exemplo, considerando a Fig. 3.2, **in**, **ParkingArea.in** e **ParkA.in** são possíveis identificadores (reduzidos) para a mesma transição **ParkA.ParkingArea.in** (o identificador completo).

A operação que define o modelo **ParkA** pode ser representada na sua forma gráfica através de um novo tipo de diagrama, introduzido nesta dissertação: o **diagrama de adição**. Este tipo de diagrama tem algumas semelhanças sintácticas com os *diagramas de estrutura*³ de Magee e Kramer [Magee e Kramer, 1999]. No entanto, os diagramas de estrutura permitem apenas a especificação de sincronizações entre acções de máquinas de estado concorrentes e não referem a possibilidade de especificação de um sentido de invocação, conforme utilizado no Capítulo 6 desta dissertação.

O diagrama de adição para a rede **ParkA** (vide Fig. 3.3) permite visualizar quer a união dos vários submodelos, quer a sua interligação por meio de fusões de nós (transições, no caso presente). As transições fundidas (*transições interface*) são assinaladas por pequenos quadrados

³No original: *structure diagrams*.

a tracejado, que podem também ser rectângulos, junto das respectivas instâncias de rede. De forma semelhante, para os lugares (*lugares interface*) utilizar-se-ão circunferências, que podem também ser elipses.

Em resumo, um diagrama de adição representa as instâncias das redes e os seus nós interface. Estes são interligados por nós resultado.

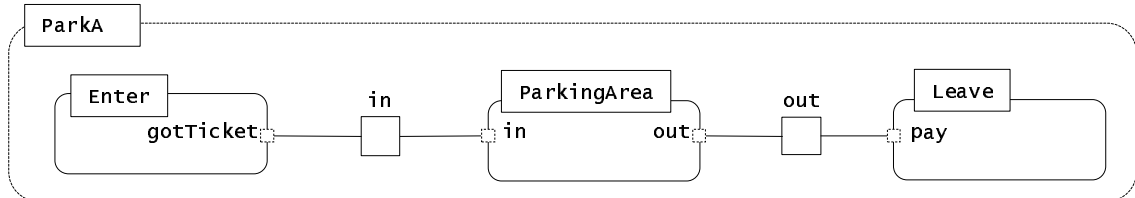


Figura 3.3: Diagrama de adição para a rede ParkA.

Por último, a utilização do símbolo $:=$ justifica-se pela analogia entre a definição operacional de modelos e a afectação de valores a variáveis nas linguagens de programação imperativas, como por exemplo a linguagem PASCAL e a linguagem ADA. Tal também permite a especificação de definições operacionais em qualquer editor de texto, por oposição a um eventual uso de caracteres que não constem dos primeiros 128 elementos da tabela ASCII. Também por esta razão, utilizam-se os símbolos \rightarrow em lugar de um único símbolo. Tal permite evitar a existência de uma sintaxe utilizada na definição da linguagem e nos subsequentes textos que a utilizem, e de uma outra sintaxe para as ferramentas computacionais⁴.

A operação de subtracção de redes permite a remoção de parte de um modelo: é definida como uma adição de redes seguida de uma remoção de nós. Estes nós são especificados por um segundo colapso em que os nós resultado são removidos juntamente com os arcos a eles ligados. Este segundo colapso é denominado **colapso de remoção**. Por exemplo, pode-se definir um novo modelo ParkExit (Fig. 3.4), que corresponde ao resultado da subtracção da rede Enter ao modelo ParkA:

```
ParkExit := (ParkA - Enter)
  (/ParkA.in/Enter.gotTicket/ -> in)
  (/ParkA.entranceFree/Enter.entranceFree/,
   /ParkA.arriveUp/Enter.arriveUp/,
   /ParkA.arriveDown/Enter.arriveDown/,
   /ParkA.waitingTicket/Enter.waitingTicket/,
   /ParkA.gateOpenE/Enter.gateOpenE/
  )
```

Neste caso o modelo Enter é o **modelo subtrativo** ou **modelo negativo**.

A subtracção pode ser especificada em diagramas de adição. Distingue-se da adição colocando

⁴Tal sucede, por exemplo, com a linguagem CSP [Hoare, 1985; Roscoe et al., 1997].

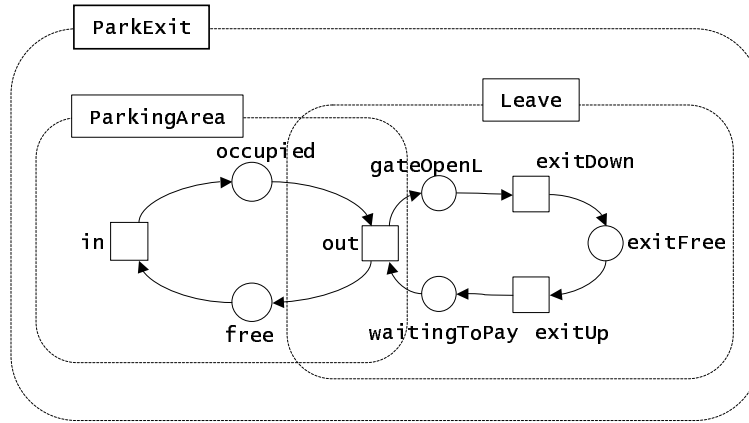


Figura 3.4: Modelo resultante da subtração do modelo **Enter** ao modelo **ParkA** (Fig. 3.2).

um \times dentro de cada nó resultado a remover. A Fig. 3.5 apresenta o diagrama de adição para a definição de **ParkExit**, por meio de uma subtração.

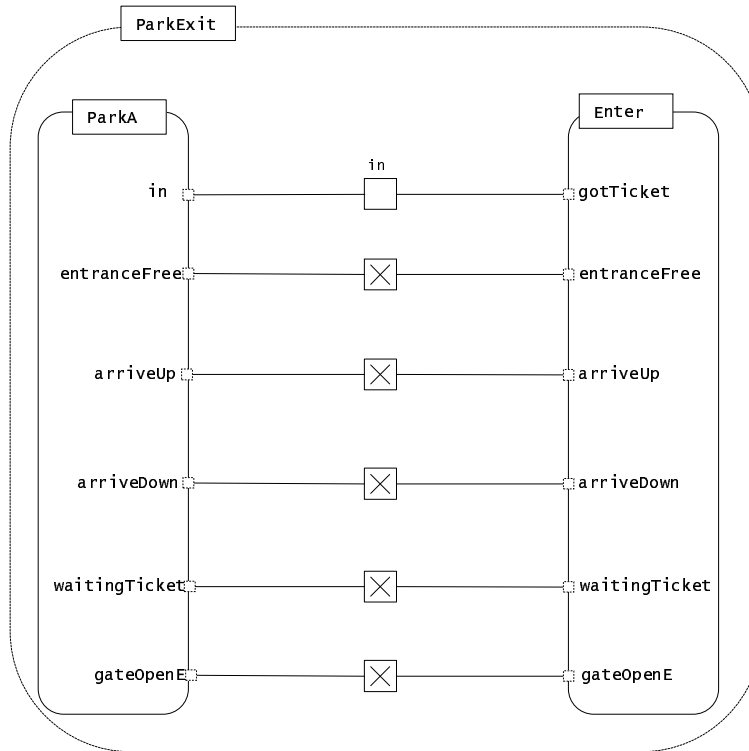


Figura 3.5: Diagrama de adição para a subtração do modelo **Enter** ao modelo **ParkA** (Fig. 3.2).

Note-se que no caso da subtração seria possível omitir a especificação dos nós interface, especificando apenas uma lista de nós a remover. Tal não é feito por duas razões:

1. Para garantir uma semelhança estrutural e semântica com a operação de adição.
2. Para forçar uma estruturação da própria operação de subtração.

O primeiro ponto é importante por garantir uniformidade e portanto uma maior simplicidade na utilização das duas operações. Por minimizar a propensão para erros de especificação, o segundo ponto torna a operação mais segura. Também garante maior legibilidade do que aquela que seria possível caso fosse necessário um maior número de operações mais primitivas para efectuar uma transformação inversa da efectuada pela adição, como por exemplo uma sequência de remoções de nós. Por oposição a uma visão alternativa que classifica a subtracção como uma operação inversa da adição, estas mesmas razões pretendem também reflectir a visão da subtracção como um caso particular da adição. Por esta mesma razão os diagramas de adição são ainda diagramas de adição mesmo quando especificam uma subtracção. Também quando a adição e subtracção são vistas como formas de modificação, faz sentido pensar unicamente em termos de adição: nesse caso adição de modificações ao modelo.

A subtracção que define o modelo `ParkExit` (pág. 55) corresponde a um caso particular de subtracção, embora muito intuitivo e portanto passível de frequente utilização: todos os nós do subtrativo (`Enter`) que não são nós interface são nós de subtracção e são fundidos com nós homónimos⁵ no aditivo (`ParkA`). Nesta situação podemos utilizar uma notação abreviada em que omitimos a lista de remoção:

```
ParkExit := (ParkA - Enter)(/in/gotTicket/ -> in)
```

Naturalmente, também poderíamos ter definido o modelo `ParkExit` de forma aditiva, ou seja ascendente:

```
ParkExit := (ParkingArea + Leave)(/out/pay/ -> out)
```

A esta definição corresponde o diagrama de adição na Fig. 3.6.

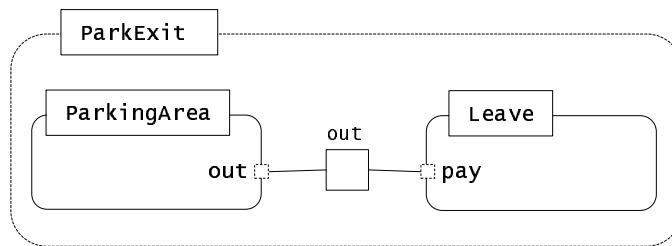


Figura 3.6: Diagrama de adição alternativo para o modelo `ParkExit`.

Considerando que se pretende modelar um sistema no qual existem duas áreas para entrada, podemos agora adicionar duas instâncias do modelo `Enter` ao modelo `ParkExit` na Fig. 3.4. Obtemos assim o modelo `E2Park` (representado graficamente na Fig. 3.7):

⁵Na forma reduzida.

```

E2Park := (Enter[1] + Enter[2] + ParkExit)
  (/Enter[1].gotTicket/in/ -> in[1],
   /Enter[2].gotTicket/in/ -> in[2]
  )

```

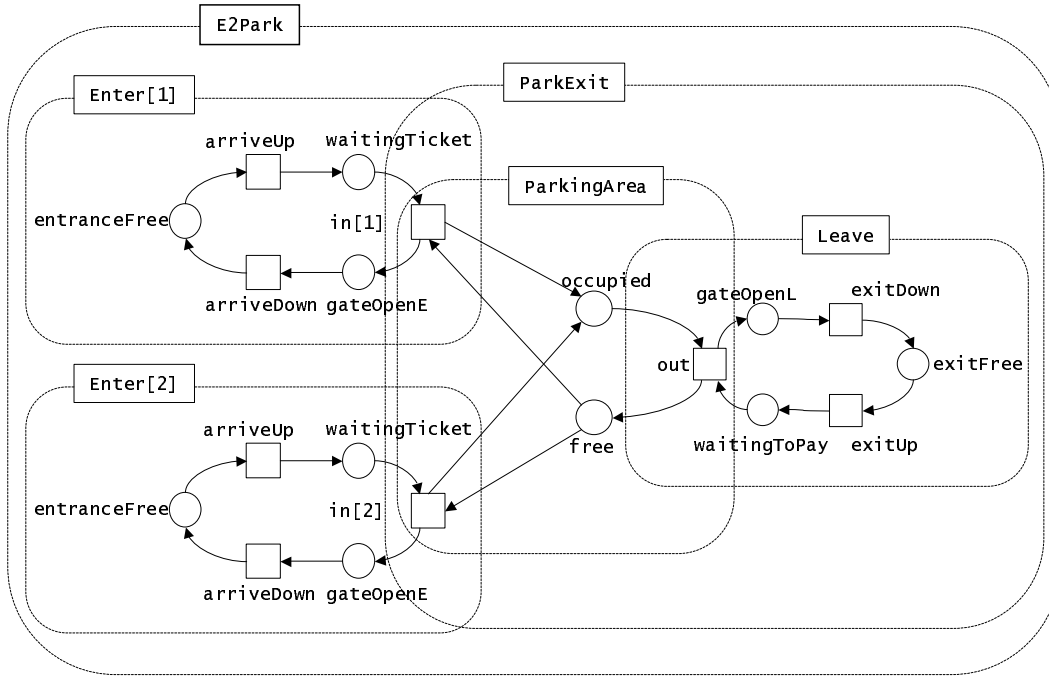


Figura 3.7: Parque com duas entradas.

As Figs. 3.8 e 3.9 apresentam duas possíveis visualizações do diagrama de adição para a rede E2Park. Na Fig. 3.9 o detalhe do modelo ParkExit foi omitido. Este maior ou menor detalhe na visualização dos diagramas de adição é uma funcionalidade que deve ser suportada por um editor gráfico.

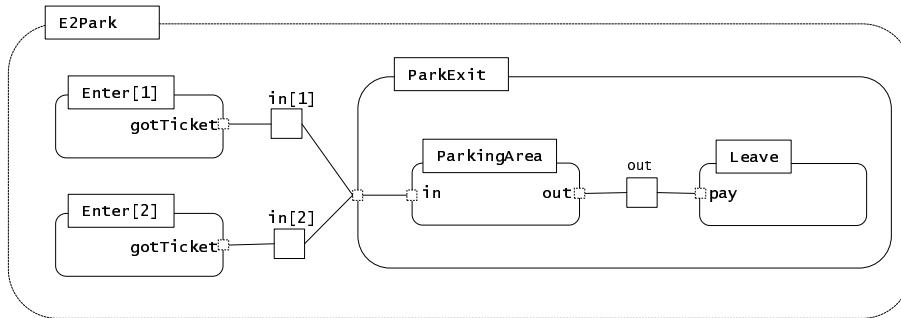


Figura 3.8: Diagrama de adição para o modelo do parque com duas entradas (E2Park).

Note-se que o identificador `in` aparece na sua forma reduzida. A forma completa seria `ParkExit.ParkingArea.in`. Mais uma vez, por não existir ambiguidade, utilizou-se a forma reduzida. São utilizadas duas instâncias do modelo `Enter` que são denotadas por `Enter[1]` e `Enter[2]`. Uma

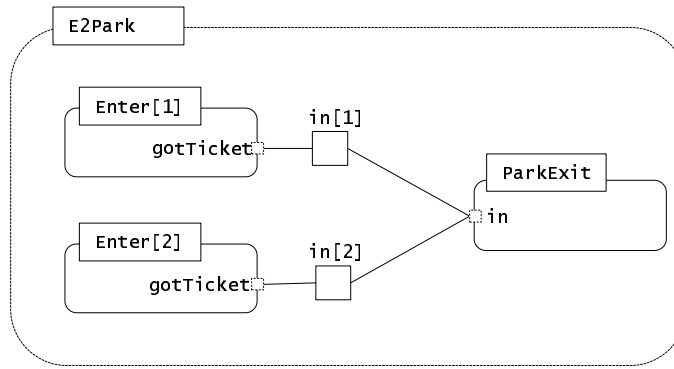


Figura 3.9: Diagrama da Fig. 3.8 em que se omitiu a visualização do corpo do modelo **ParkExit**.

notação alternativa consiste em utilizar um vector de instâncias: **Enter**[1...2]. Curiosamente, dada a simetria dos conjuntos de fusão nominativos, podemos também utilizar um vector de colapsos:

```
E2Park := (Enter[1...2] + ParkExit)
          (/Enter[i].gotTicket/in/ -> in[i])[i:1...2]
```

O vector instâncias (**Enter**[1...2]) e o vector de colapsos (**(/Enter[i].gotTicket/in/ -> in[i])[i:1...2]**) tornam esta definição de **E2Park** imediatamente generalizável a qualquer quantidade de instâncias da rede **Enter**. A especificação textual permite a especificação de qualquer quantidade de modelos adicionais os quais não necessitam de ser visualizados para serem definidos e compreendidos. Por outras palavras, as anotações textuais, aqui sob a forma da definição operacional de modelos, oferecem à estruturação e modificação de modelos algo semelhante ao que as redes de alto nível oferecem ao nível dos nós: a transferência da complexidade gráfica do modelo para anotações textuais.

Vejamos agora um exemplo adicional. Pretende-se adicionar ao modelo **E2Park** uma nova zona de estacionamento com a sua própria entrada e saída, permitindo a circulação entre as duas zonas de estacionamento. Esta nova zona corresponde ao modelo **ParkA** na pág. 53 e nas Figs. 3.2 e 3.3. Pretende-se também interligar as duas zonas de estacionamento de forma que os carros possam deslocar-se de uma zona de estacionamento para outra, através de passagens de sentido único. Especificaremos este modelo final em dois passos⁶:

1. Ligação das passagens ao modelo **ParkA**.
2. Ligação do modelo resultante ao modelo **E2Park**.

⁶Por razões ilustrativas, optou-se por esta construção em dois passos. No entanto, é perfeitamente possível uma construção num único passo utilizando uma adição com três operandos: os modelos **ParkA**, **Passage**[1...2] e **E2Park**.

Assim, em primeiro lugar, adicionamos uma instância do modelo **ParkA** a duas instâncias do modelo **Passage**:

```
ParkAPassage2 := (ParkA + Passage[1...2])
  (/ParkA.free/Passage[1].freeZone[2]/
    Passage[2].freeZone[1]/ -> free,
  /ParkA.occupied/Passage[1].carsInZone[2]/
    Passage[2].carsInZone[1]/ -> occupied
  )
```

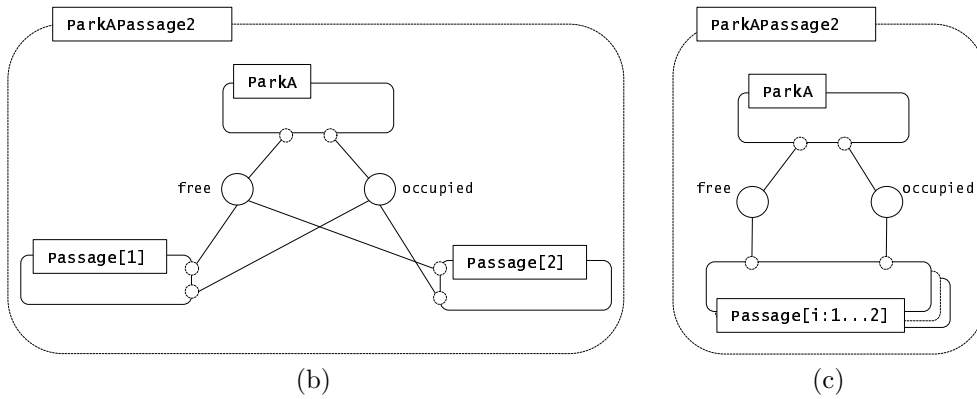
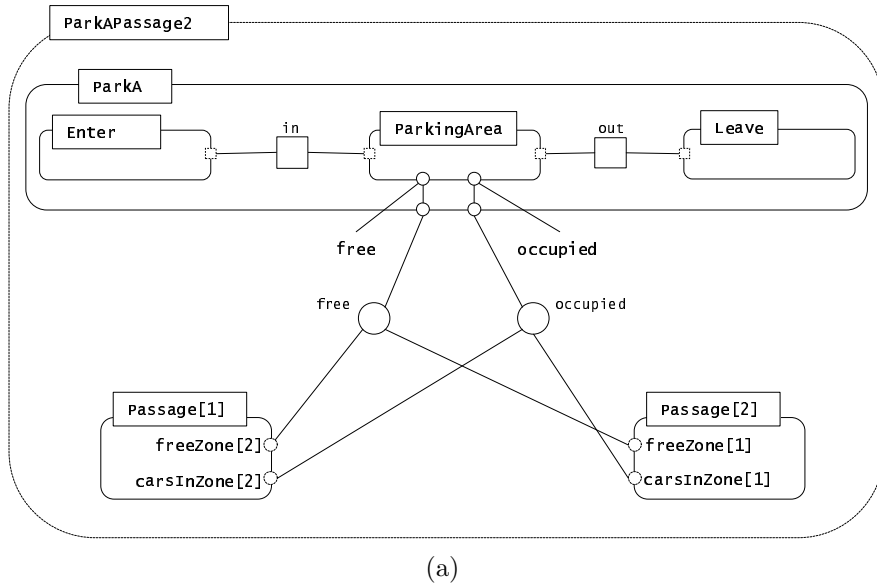


Figura 3.10: Três visualizações para o diagrama de adição do modelo **ParkAPassage2**.

A Fig. 3.10 apresenta três visualizações possíveis para o diagrama de adição do modelo **ParkAPassage2**. A Fig. 3.10c apresenta uma versão em que as várias instâncias do modelo **Passage** surgem compactadas num único nó, correspondente ao vector **Passage[1...2]**. Também as ligações entre este nó e os restantes (**free** e **occupied**) surgem compactadas, o que permite uma visualização cuja legibilidade não é prejudicada pela quantidade de instâncias do vector.

Seguidamente, adicionamos o modelo `ParkAPassage2` ao modelo `E2Park` e obtemos o modelo `E3Park2Exit2`, representado graficamente na Fig. 3.11 da pág. 62. Este modelo também exemplifica um caso no qual não é claro que a representação gráfica seja preferível à definição textual. É por isso desejável que as ferramentas computacionais disponibilizem a vista de alto nível possibilitada pelos diagramas de adição pois estes permitem visualizações com maior (Fig. 3.12) ou menor (Fig. 3.13) detalhe. Neste caso, o modelo detalhado apenas seria mostrado (gerado) na forma gráfica a pedido do utilizador.

```
E3Park2Exit2 := (ParkAPassage2 + E2Park)
    (/ParkAPassage2.Passage[1].carsInZone[1]/
     ParkAPassage2.Passage[2].carsInZone[2]/
     E2Park.occupied/ -> occupied,
    /ParkAPassage2.Passage[1].freeZone[1]/
     ParkAPassage2.Passage[2].freeZone[2]/
     E2Park.free/ -> free
    )
```

Nos exemplos apresentados, as operações foram utilizadas para a construção de um modelo: o `E3Park2Exit2` representado na Fig. 3.11. Para além da aplicabilidade à construção de modelos, as operações de adição e subtracção permitem também efectuar modificações de modelos já existentes. Estas modificações podem afectar vários módulos e podem corresponder quer ao acrescento de uma nova funcionalidade, quer à modificação de um padrão de comportamento específico devido à alteração dos requisitos que estiveram na sua origem. Um possível exemplo deste último caso seria a modificação do modelo `Passage` impondo um valor máximo na quantidade de carros que pode estar em trânsito na passagem (capacidade do lugar `passing`).

A secção seguinte discute a decomposição descendente e a Secção 3.1.4 propõe uma nova forma de estruturar modelos de redes de Petri que suporta a inserção de *concerns* que entrecortam um modelo base hierárquico.

3.1.3 Suporte para Estruturação Hierárquica – a Perspectiva Descendente

Os exemplos de aplicação de adição de redes e subtracção apresentados na Secção 3.1.2 mostram que a operação de adição permite a construção de modelos de forma ascendente. As operações apresentadas nas secções anteriores, e em especial a adição de redes, fornecem o suporte necessário quer para a estruturação hierárquica de sistemas, quer para as estruturações paralelas necessárias para uma completa especificação. Estas estruturas são tipicamente ortogonais entre si e podem corresponder aos diversos *concerns* necessários para a completa especificação do sistema. O efeito pode ser visualizado como a sobreposição de várias folhas transparentes cada

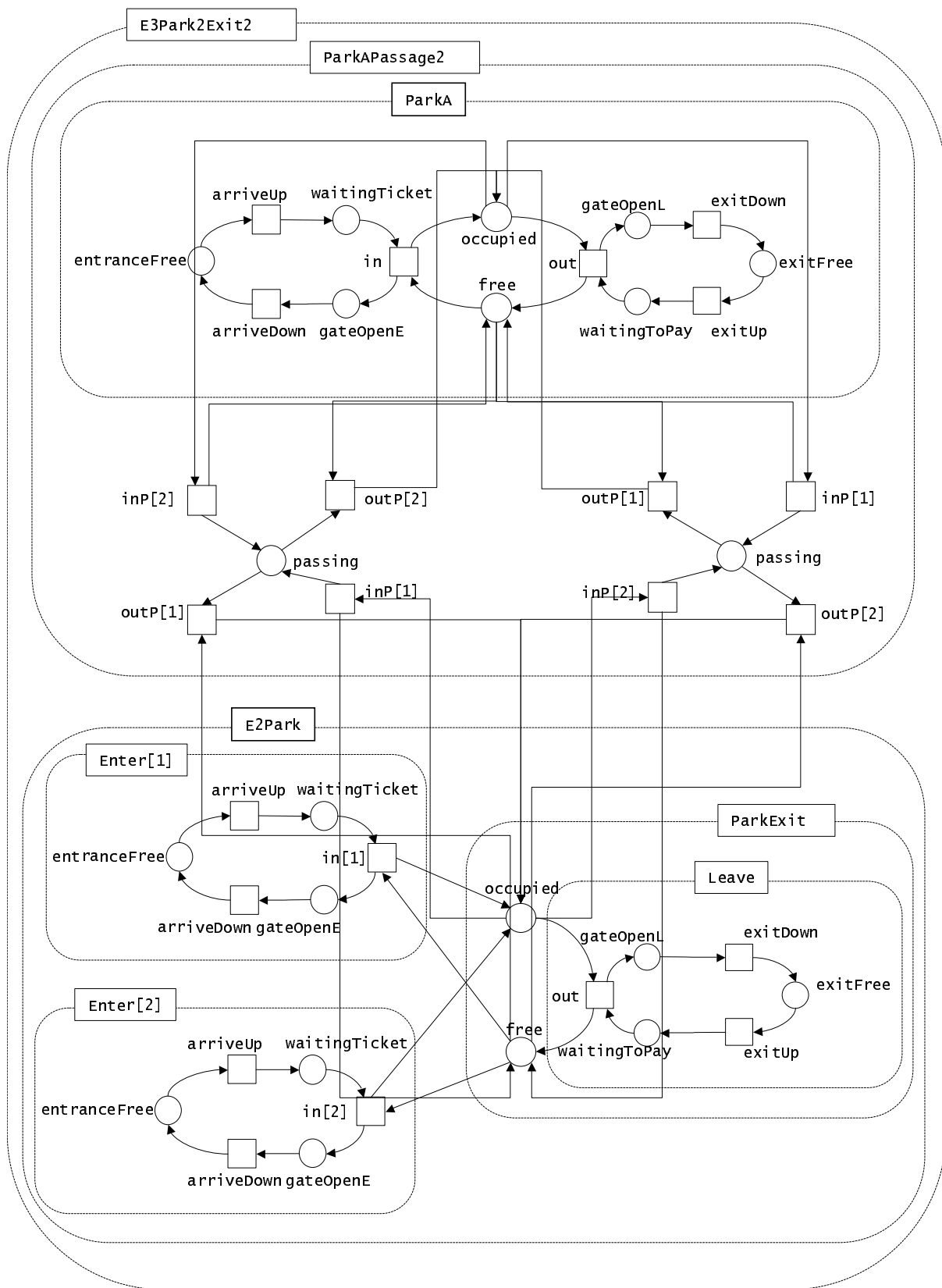


Figura 3.11: Modelo E3Park2Exit2.

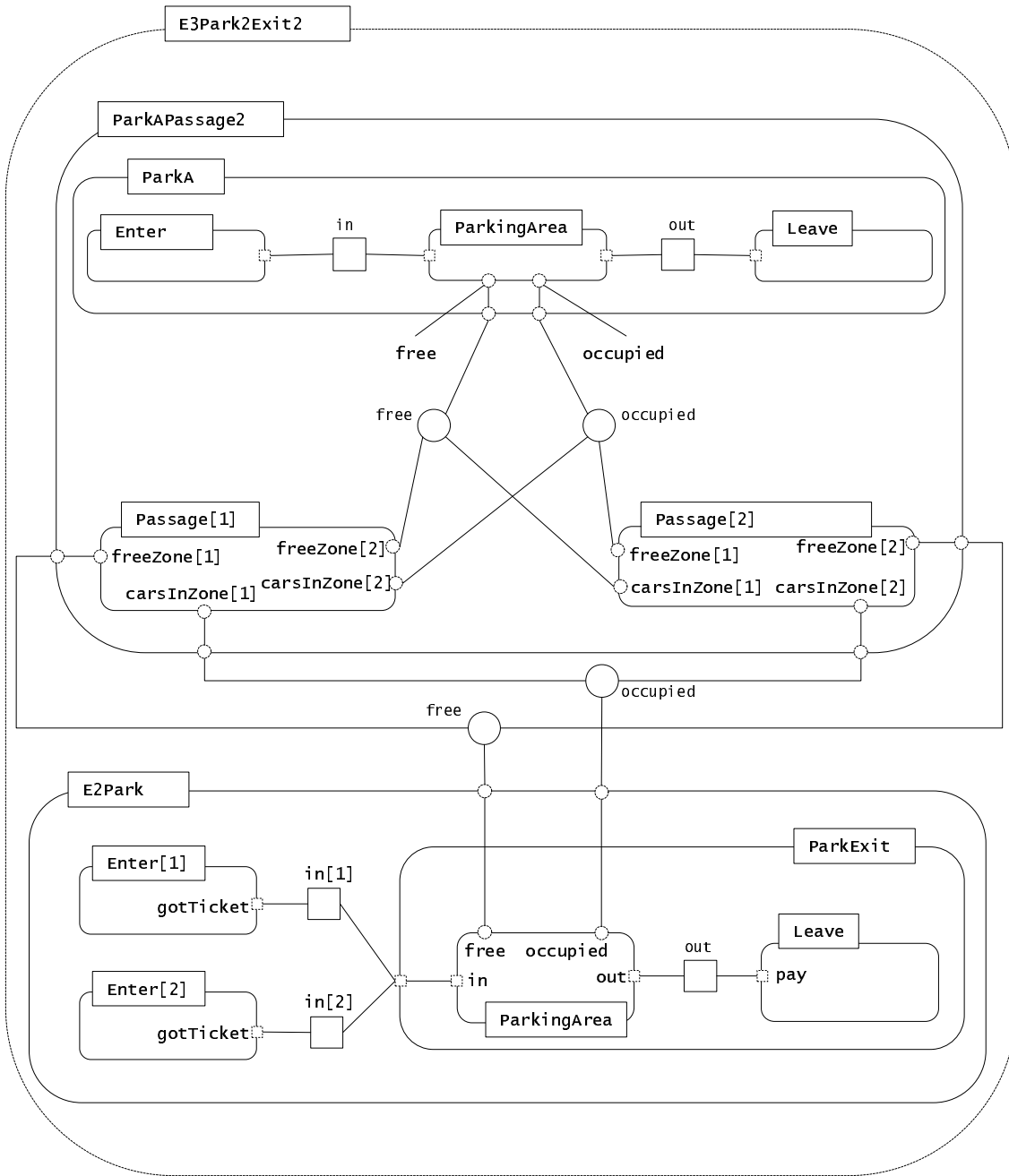


Figura 3.12: Diagrama de adição do modelo E3Park2Exit2.

qual especificando um *concern* do sistema. Estes podem exibir um maior ou menor grau de interdependência mas todos têm de estar ligados em determinados pontos de interface específicos: os locais onde as partes relevantes de cada especificação se sobrepõem. A adição de redes torna explícita a modelação destes locais: eles correspondem aos nós resultado dos conjuntos de fusão, conforme já ilustrado pelos diagramas de adição na Secção 3.1.2.

A estruturação de um sistema de forma hierárquica é de facto útil desde que nos consigamos convencer que é suficiente. Tal como referido por Jackson (*vide* citação no início deste capítulo

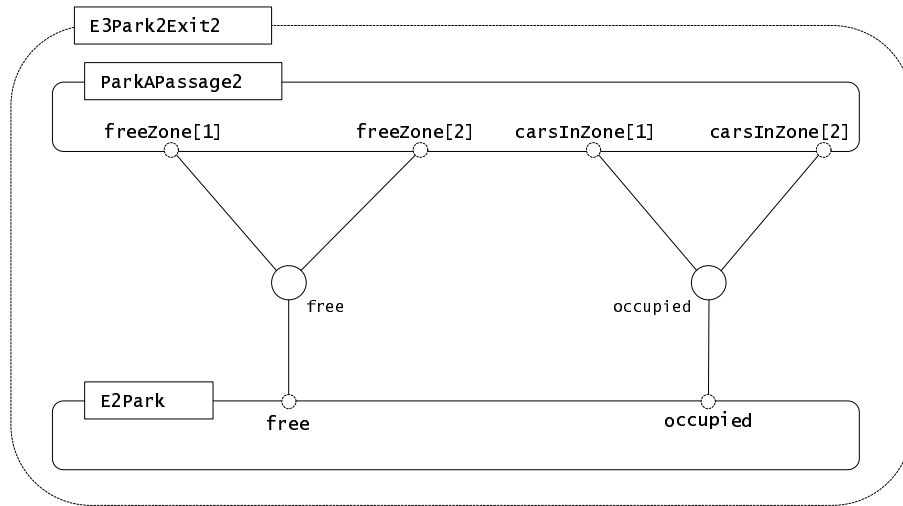


Figura 3.13: Diagrama de adição do modelo E3Park2Exit2 com ocultação de informação.

retirada de [Jackson, 1995, pág. 4]), a estrutura hierárquica é frequentemente insuficiente para reflectir as diversas relações entre os módulos. Ainda assim, uma estrutura hierárquica é muito intuitiva e constitui frequentemente um claro auxiliar na especificação de modelos de forma faseada quando se pretende caminhar do geral para o específico, de forma descendente ou, conforme notado por Parnas [Parnas, 1974], *de fora para dentro (outside-in)*.

Os diagramas de adição já apresentados podem também ser lidos de forma descendente. Tal como na construção de modelos ascendentes, os nós resultado assumem especial importância. Também na perspectiva descendente os nós resultado devem ser vistos como ligações entre dois submodelos. Por exemplo, o diagrama de adição para o modelo **ParkA** apresentado na Fig. 3.3 e repetido na Fig. 3.14, pode ser visto como o resultado da decomposição do modelo **ParkA** em três submodelos: **Enter**, **ParkingArea** e **Leave**. As ligações entre estes três modelos são definidas pelos nós **in** e **out** que interligam os nós interface de cada um dos submodelos. Note-se que a definição textual do modelo **ParkA** continua a ser aplicável, mesmo nesta perspectiva descendente.

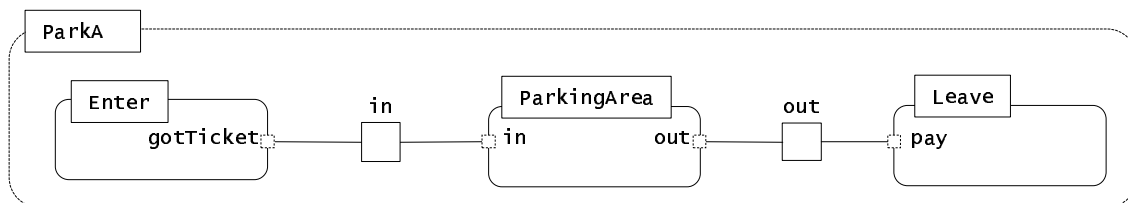


Figura 3.14: Diagrama de adição para a rede ParkA (igual à Fig. 3.3).

Por último, é significativo constatar que os diagramas de adição podem facilmente assumir a forma tradicional de decomposição hierárquica em macronós tal como exemplificada na Fig. 2.9b da pág. 39. Conforme os tipos de nós a que se encontra ligada na super-rede, cada sub-rede pode ser identificada como um macrolugar, macrotransição, ou macrobloco. Para tal basta efectuar

três simples mudanças sintáticas no diagrama:

1. Os submodelos que se interligam apenas a transições são representados como macrolugares.
2. Os submodelos que se interligam apenas a lugares são representados como macrotransições.
3. Os submodelos que se interligam a lugares e a transições são representados como macro-blocos.

Como exemplo, a Fig. 3.15 mostra o diagrama de adição para a modelo **ParkA** utilizando macronós. Dado a ligação entre os módulos ser feita pela fusão de transições, cada módulo é representado por um macrolugar. Esta interpretação pode chocar com uma visão do módulo **Enter** como um elemento activo que como tal deveria ser representado por uma macrotransição. Tal está na linha da representação da Fig. 2.12 na pág. 41. No entanto, tal deve-se ao facto do elemento de ligação — o nó resultado — não ser representado no nível da superpágina. Na verdade esse nó não é representado dado considerar-se uma ligação directa entre as duas subpáginas. Já no diagrama de adição, a ligação fica explícita ao nível da superpágina, o que força uma determinada interpretação para cada um dos macronós correspondentes. Esta interpretação é escolhida quando se especificam quais os nós de interface. Tal origina três regras para a escolha dos nós de interface de cada módulo, cada uma delas correspondendo a uma das mudanças sintáticas atrás apresentadas:

1. Caso um módulo deva ser interpretado como macrolugar, todos os seus nós de interface devem ser transições.
2. Caso um módulo deva ser interpretado como macrotransição, todos os seus nós de interface devem ser lugares.
3. Caso um módulo deva ser interpretado como macrobloco, os seus nós de interface devem incluir pelo menos um lugar e uma transição.

Dada a generalidade da operação de adição, um determinado módulo pode assumir interpretações distintas conforme o tipo dos seus nós que são escolhidos para nós interface. Caso esta versatilidade seja considerada indesejável, tipicamente por ser preferível definir uma interface rígida, deve ser utilizado um nível adicional de especificação que permita a indicação de quais os nós interface do módulo, ou seja, qual a sua interface.

A secção seguinte propõe uma forma de complementar um modelo estruturado de forma hierárquica (o **modelo base**), que utiliza um diagrama de adição (com ou sem macronós), com um outro conjunto de modelos que introduzem modificações que entrecortam esse modelo base.

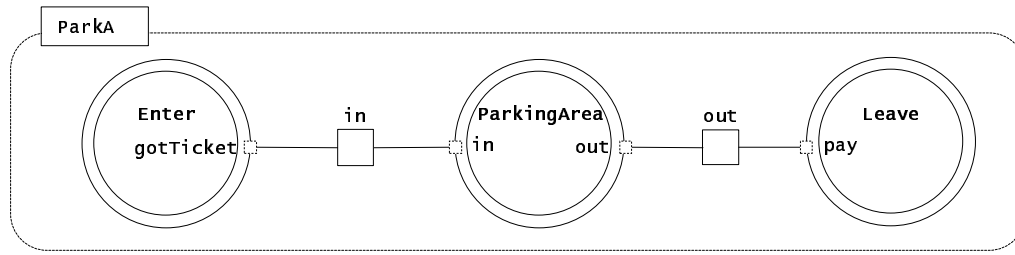


Figura 3.15: Diagrama de adição para a rede *ParkA* utilizando macronós.

3.1.4 Uma Proposta para a Modularização em Redes de Petri

As perspectivas descendente e ascendente são úteis para a construção de modelos estruturados hierarquicamente. No entanto, as estruturas resultantes da aplicação destes mecanismos podem facilmente tornar-se um obstáculo a futuras evoluções do sistema modelado que impliquem modificações no modelo. Tal é verdade sempre que os novos requisitos do sistema entrecortem os módulos em que o modelo se encontra estruturado (*vide* Fig. 3.16).

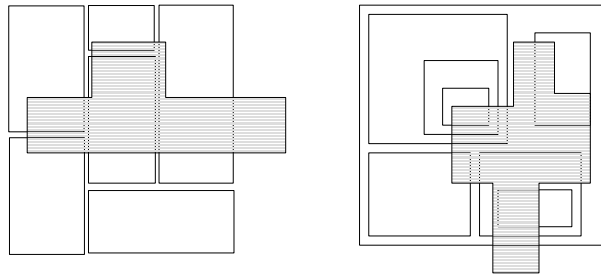


Figura 3.16: Entrecortamento de sistemas estruturados em módulos (*in* [Czarnecki e Eisenecker, 2000]).

A dificuldade em introduzir modificações em modelos estruturados segundo um determinado tipo de módulos foi identificada em [Tarr et al., 1999] onde é referida como "a tirania da decomposição dominante". Com essa designação os autores alertam para a dificuldade de introduzir modificações num modelo previamente estruturado utilizando um determinado tipo de módulos, sempre que essas modificações tenham que afectar vários desses módulos. Por outras palavras, é difícil modificar um modelo estruturado em módulos quando a modificação não pode, fácil ou intuitivamente, ser modelada por novos módulos do mesmo tipo.

Este problema é o objecto de estudo do denominado **desenvolvimento orientado pelos aspectos** [Aksit et al., 2005; Filman et al., 2005; Elrad et al., 2001] o qual foi popularizado através dos trabalhos sobre programação orientada pelos aspectos [Kiczales et al., 1997], nomeadamente através da ferramenta Aspect/J [s.a., 2005a], bem como através dos trabalhos na área da **programação generativa** (e.g. [Czarnecki e Eisenecker, 2000]). A orientação pelos aspectos inclui também a fase de desenho onde as linguagens gráficas de especificação são frequentemente utilizadas. Desde 1997, têm vindo a realizar-se várias *workshops* sobre o desenvolvimento orientado pelos aspectos e, desde o ano 2002, tem também vindo a realizar-se uma conferência

internacional sobre o tema: a *International Conference on Aspect-Oriented Software Development*. Um aspecto é um **assunto transversal**⁷. O termo *concern* parece ter sido utilizado pela primeira vez por Dijkstra embora no contexto mais vasto de uma discussão sobre a importância do pensamento científico [Dijkstra, 1974].

A referida "tirania da decomposição dominante" dificulta a introdução de modificações devidas a alterações nos requisitos iniciais, ou devidas à necessidade de modelar novos requisitos. Nesta secção apresenta-se, através de exemplos, uma proposta que possibilita quer a estruturação hierárquica de redes de Petri, quer a introdução de alterações futuras ou concomitantes com a construção do modelo hierárquico. Todos estes mecanismos são suportados pela adição e subtracção de redes.

Retomando o exemplo do parque de estacionamento, a primeira alteração é o acrescento de uma nova funcionalidade: um contador global que indica quantos carros se encontram no parque. Tal é conveniente porque no modelo **E3Park2Exit2** (Figs. 3.12 e 3.13) os carros podem mover-se entre as duas zonas.

O modelo com contador global pode ser obtido pela simples adição de um modelo **ParkingArea** "global". Este modelo não irá especificar uma nova zona de estacionamento mas apenas um contador global. Por outras palavras, mais do que uma adição à estrutura do modelo, esta alteração deve ser vista como um acrescento de funcionalidade, acrescento este que é modelado por um novo módulo composto ortogonalmente com os módulos que constituem o **modelo base** existente. Esta alteração pode também ser vista como uma zona de estacionamento virtual correspondente a todo o parque de estacionamento. A adição funde cada uma das transições **in**, **in[1]** e **in[2]** em **E3Park2Exit2** na pág. 61 (*vide* também a Fig. 3.11) com a transição **in** em **ParkingArea**; funde também cada uma das transições **out** em **E3Park2Exit2** com a transição **out** em **ParkingArea**. Omitimos a representação gráfica de **E3Park2Exit2Counter** pois dificilmente seria legível. No entanto, devido à sua capacidade de ocultar informação, o diagrama de adição continua a ser legível, mais até do que a respectiva representação textual (*vide* Fig. 3.17).

```
E3Park2Exit2Counter := (E3Park2Exit2 + ParkingArea)
  ((/E3Park2Exit2.E2Park.in[i]/
    ParkingArea.in/ -> in[i])[i:1...2],
    /E3Park2Exit2.ParkAPassage2.ParkA.in/
    ParkingArea.in/ -> in[3],
    /E3Park2Exit2.E2Park.ParkExit.out/
    ParkingArea.out/ -> out[1],
    /E3Park2Exit2.ParkAPassage2.ParkA.out/
    ParkingArea.out/ -> out[2])
```

A adição de redes permite-nos evitar a estruturação hierárquica pura e a adição do contador

⁷Em inglês: *crosscutting concern*.

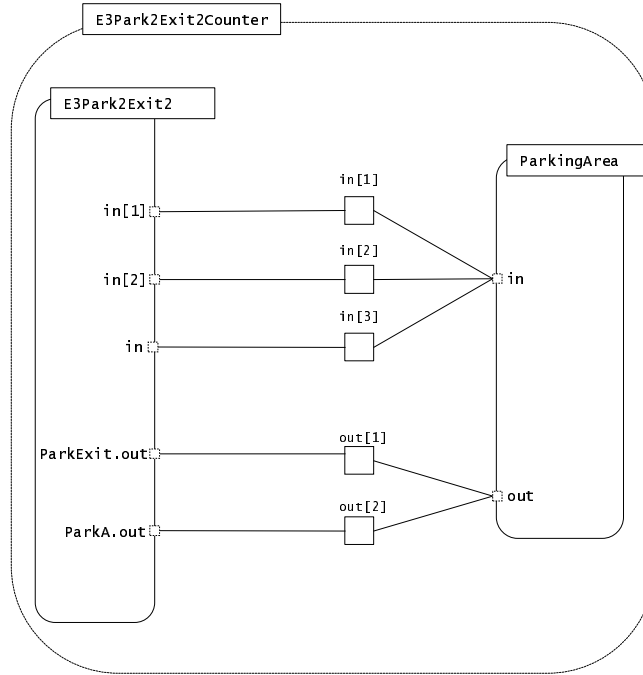


Figura 3.17: Diagrama de adição do modelo **E3Park2Exit2Counter** com ocultação de informação.

global é um exemplo em que adição pode ser vista como uma modificação ortogonal à hierarquia existente. Nesse sentido, o modelo **ParkingArea** deve continuar separado da estruturação hierárquica existente. Tal corresponde a uma divisão lógica do sistema em três partes (*vide* Fig. 3.18):

1. O sistema *Base* estruturado hierarquicamente.
2. Os *Crosscutting Concerns* que introduzem modificações no modelo existente.
3. A *Composition* que interliga os dois primeiros.

Futuros *concerns* são adicionados de forma visualmente "paralela" aos já existentes. Por exemplo, se pretendermos introduzir um modelo que force uma alternância na utilização das duas entradas do modelo **E3Park2Exit2Counter**, pertencentes ao submodelo **E2Park**, podemos fazê-lo introduzindo um novo *concern* no modelo. Tal está exemplificado pela adição do modelo **Alternate** (*vide* Fig. 3.19) ao modelo **E3Park2Exit2Counter** (*vide* Fig. 3.20).

Na forma textual, temos então uma especificação com mais um módulo que é composto com o módulo *Base* **E3Park2Exit2**. Comparativamente com **E3Park2Exit2Counter**, adicionaram-se as transições **turn[i]** na terceira linha:

```
E3Park2Exit2CounterAlt := (E3Park2Exit2 + ParkingArea + Alternate)
                        ((/E3Park2Exit2.E2Park.in[i]/
                          ParkingArea.in/turn[i]/ -> in[i])[i:1...2],
```

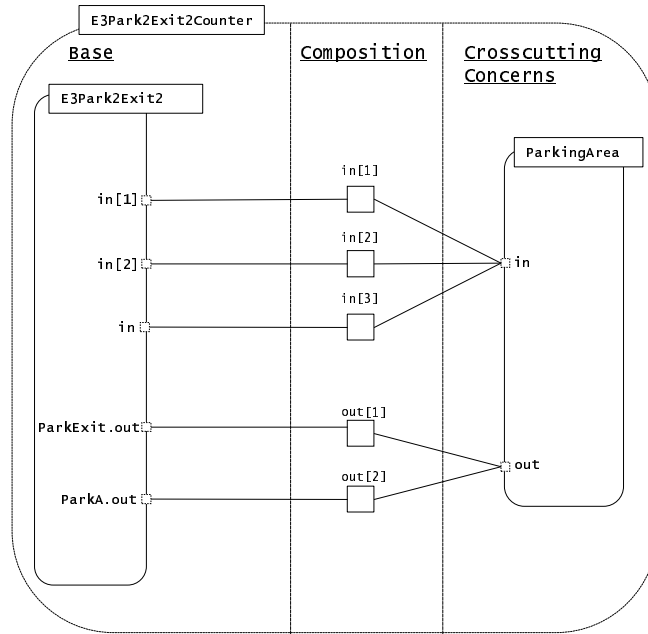


Figura 3.18: Diagrama de adição do modelo E3Park2Exit2Counter com separação de *concerns*.

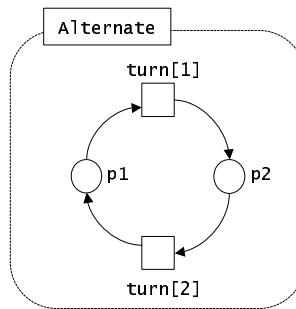


Figura 3.19: Modelo para especificação de alternância. Omite-se a marcação.

```

/E3Park2Exit2.ParkAPassage2.ParkA.in/
  ParkingArea.in/ -> in[3],
/E3Park2Exit2.E2Park.ParkExit.out/
  ParkingArea.out/ -> out[1],
/E3Park2Exit2.ParkAPassage2.ParkA.out/
  ParkingArea.out/ -> out[2] )

```

A composição de *concerns*, através da adição, baseia-se sempre numa adição "paralela" do modelo base (no exemplo E3Park2Exit2) com cada um dos vários *concerns* (no exemplo Parking Area e Alternate). Naturalmente, é também possível adicionar *concerns* sobre a composição do modelo base com *concerns* anteriores. No entanto, tal corresponde novamente a uma visão hierárquica do modelo, misturando agora módulos que constituem o modelo base com módulos que são considerados *concerns* transversais ou, pelo menos, que se pretendem compor de forma ortogonal com o modelo base. Desta forma, a adição de redes, com a possível inclusão da

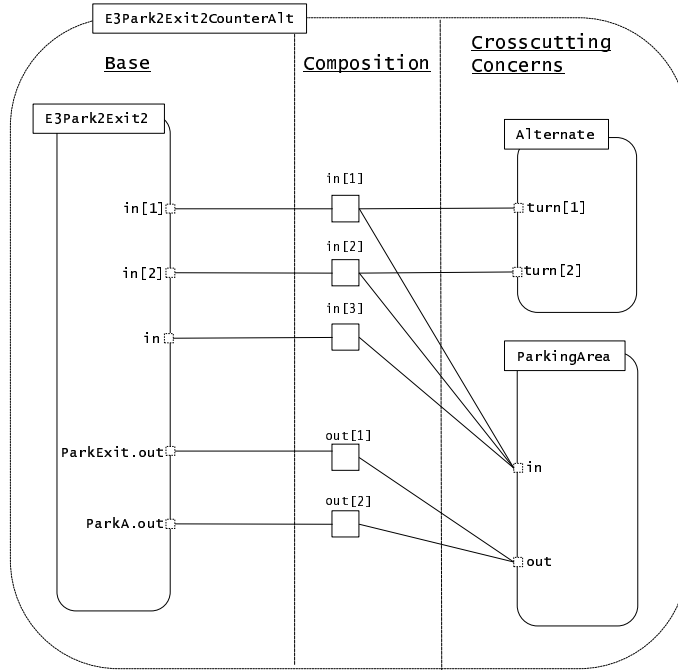


Figura 3.20: Diagrama de adição do modelo E3Park2Exit2CounterAlt com separação de *concerns*.

subtração, fornece um suporte adequado para qualquer das três atitudes de construção de modelos: (1) ascendente; (2) descendente; e (3) uma composição ortogonal baseada em *concerns* mais ou menos transversais.

As duas secções seguintes definem de forma precisa e formal, as operações de adição e subtração em redes de baixo nível não-marcadas.

3.2 Conjuntos de Fusão e Colapsos

Esta secção apresenta um conjunto de definições formais que a secção seguinte utilizará para definir, formalmente e de forma precisa, as operações de adição e subtração de redes informalmente apresentadas na secção anterior. As definições baseiam-se na utilização de redes de Petri de baixo nível, não marcadas e sem pesos associados aos arcos. A extensão para classes arbitrárias de redes de Petri, que incluem as redes de Petri marcadas, é feita na Secção 3.5.

Começa-se então por definir as redes aqui utilizadas:

Definição 3.1 (Rede de Petri): Uma rede de Petri é um terno $R = (L, T, F)$ que satisfaz os seguintes requisitos:

1. L é um conjunto finito de **lugares**.
2. T é um conjunto finito de **transições** (disjunto de L).

3. F é a relação de fluxo (correspondente aos **arcos da rede**): $F \subseteq ((L \times T) \cup (T \times L))$.

Escrever-se-á L_A em lugar de L_{R_A} para significar o conjunto de lugares da rede R_A . O mesmo é verdade para outros casos similares. Por abuso de notação considerar-se-á uma lista $LST = (e_1, \dots, e_n)$ como sendo também o conjunto dos seus elementos $\{e_1, \dots, e_n\}$. Também as seguintes notações serão consideradas como notações alternativas para uma mesma lista: $LST = (LST) = ((e_1, \dots, e_n)) = (e_1, \dots, e_n)$. Por fim, dadas duas listas $LST_A = (a_1, \dots, a_n)$ e $LST_B = (b_1, \dots, b_m)$, utilizar-se-á $(LST_A, LST_B) = ((a_1, \dots, a_n), (b_1, \dots, b_m)) = (a_1, \dots, a_n, b_1, \dots, b_m)$ para denotar a lista resultante da junção de LST_B a LST_A .

Note-se que a rede definida (Def. 3.1) não está marcada. As operações serão definidas para redes não marcadas porque a marcação é na verdade um caso particular (embora natural) de anotação dos lugares. Com efeito, é possível definir vários tipos diferentes de marcas, ou, de um modo geral, de quaisquer tipos de anotações associadas aos lugares, transições ou arcos. Por esta razão, a relação das anotações com as operações aqui apresentadas é tratada na Secção 3.5.

Conforme já exemplificado na Secção 3.1.2, e adiante definido (Def. 3.19 na pág. 75), cada nó é identificado não apenas pelo seu próprio identificador mas também pela rede de que faz parte. Por essa razão, considera-se que todos os nós e fluxos de duas redes distintas são sempre disjuntos.

Apresentam-se agora as definições de **conjunto de fusão**, **conjunto de fusão conectivo** e **conjunto de fusão nominativo**:

Definição 3.2 (Conjunto de Fusão de Transições): *Dada uma rede de Petri $R = (L, T, F)$, um conjunto de fusão de transições é um subconjunto de T cujos elementos serão fundidos numa única transição.*

Definição 3.3 (Conjunto de Fusão de Lugares): *Dada uma rede de Petri $R = (L, T, F)$, um conjunto de fusão de lugares é um subconjunto de L cujos elementos serão fundidos num único lugar.*

Definição 3.4 (Conjunto de Fusão e Notação): *Um conjunto de fusão é um conjunto de fusão de transições ou um conjunto de fusão de lugares. Um conjunto de fusão CF contendo os elementos $\{e_1, \dots, e_n\}$ é denotado por $/e_1/ \dots /e_n/$.*

Dado um conjunto de fusão $CF = /e_1/ \dots /e_n/$ utilizar-se-á $Result(CF)$ para designar o nó resultante dessa fusão.

Definição 3.5 (Conjunto de Fusão Conectivo): *Dado um conjunto de duas ou mais redes $R_i = (L_i, T_i, F_i)$ com $1 \leq i \leq 2 \leq n$, um conjunto de fusão CF diz-se **conectivo** das redes R_i se contiver, pelo menos, um nó de cada uma das redes R_i : $\forall 1 \leq i \leq 2 \leq n, (CF \cap (L_i \cup T_i)) \neq \emptyset$.*

Definição 3.6 (Conjunto de Fusão Nominativo): *Dado um conjunto de fusão $CF = /e_1/ \dots /e_n/$ podemos especificar um nome (identificador) para o nó resultado nr . O conjunto de fusão acompanhado do*

nome do nó resultado é denominado **conjunto de fusão nominativo** e é denotado $/e_1/\dots/e_n/-> nr$.

Dado um conjunto de fusão nominativo $CFN = (/e_1/\dots/e_n/-> nr)$ utilizar-se-á $Nodes(CFN) = \{e_1, \dots, e_n\}$ para denotar o conjunto de nós de fusão, e $Result(CFN) = nr$ para denotar o nó resultado.

Seguidamente, define-se um **colapso de rede** como uma lista de conjuntos de fusão numa dada rede. À semelhança do conjunto de fusão conectivo, define-se também **colapso conectivo** e ainda o **conectivo do colapso**. Os colapsos de rede serão posteriormente utilizados como suporte às definições de adição e subtração de redes. No entanto, eles podem também servir para definir outros mecanismos de estruturação como, por exemplo, a fusão de lugares nas redes de Petri coloridas [Jensen, 1997c,a,b] e mecanismos de estruturação hierárquica como os apresentados em [Gomes e Barros, 2003]. Tal utilização corresponde à aplicação da operação de colapsagem de rede a definir na secção seguinte.

Definição 3.7 (Colapso): *Seja $R = (L, T, F)$ uma rede. Uma lista de conjuntos de fusão $CO_R = (CF_1, \dots, CF_n)$ é denominada **colapso** em R se e só se $\forall 1 \leq i \leq n, Nodes(CF_i) \subseteq L \vee Nodes(CF_i) \subseteq T$.*

Note-se que a totalidade, ou parte, dos conjuntos de fusão de um colapso podem ser nominativos.

Definição 3.8 (Conectivo do Colapso): *Dado um colapso CO e duas ou mais redes $R_i = (L_i, T_i, F_i)$, com $1 \leq i \leq 2 \leq n$, o conectivo de CO , denotado $CO_{(R_1, \dots, R_n)}^C$, é um colapso contendo apenas os conjuntos de fusão conectivos das redes R_i , pertencentes a CO .*

Definição 3.9 (Colapso Conectivo): *Dado um colapso CO , este diz-se conectivo se e só se contiver, pelo menos, um conjunto de fusão conectivo: $CO \cap CO^C \neq \emptyset$.*

Definição 3.10 (Colapso de Conexão): *Um colapso CO diz-se **colapso de conexão** se e só se $CO = CO^C$.*

Dado um colapso na rede $CO = (CF_1, \dots, CF_n)$, temos $Nodes(CO) = \bigcup_{1 \leq i \leq n} Nodes(CF_i)$ e $Result(CO) = \bigcup_{1 \leq i \leq n} Result(CF_i)$.

Define-se também a parte do colapso relativa aos lugares e a relativa às transições (Def. 3.11) como uma lista de conjuntos de fusão de lugares e como uma lista de conjuntos de fusão de transições, respectivamente:

Definição 3.11 (Colapso de Lugares e Colapso de Transições): *Seja $R = (L, T, F)$ uma rede e $CO = (CF_1, \dots, CF_n)$ um colapso de R . O **colapso de lugares** associado a CO é então denotado CO^L e definido como:*

$$CO^L = (CFL_1, \dots, CFL_{np}), \text{ em que } \forall 1 \leq i \leq np \leq n, CFL_i \in CO \wedge Nodes(CFL_i) \subseteq L$$

*Correspondentemente, o **colapso de transições** associado a CO é denotado CO^T e definido como:*

$$CO^T = (CFT_1, \dots, CFT_{nt}), \text{ em que } \forall 1 \leq i \leq nt \leq n, CFT_i \in CO \wedge Nodes(CFT_i) \subseteq T$$

Seguidamente, definem-se os conceitos de **lugar colapsado** e **transição colapsada**.

Definição 3.12 (Lugares Colapsados e Transições Colapsadas): *Seja $CO = (CF_1, \dots, CF_n)$ um colapso. Define-se o conjunto dos **lugares colapsados**, denotado por L_{CO} , como o conjunto de lugares pertencentes aos conjuntos de fusão do colapso: $L_{CO} = \bigcup_{CFL \in CO^L} \text{Nodes}(CFL)$. Define-se o conjunto das **transições colapsadas**, denotado por T_{CO} , como o conjunto de transições pertencentes aos conjuntos de fusão do colapso: $T_{CO} = \bigcup_{CFT \in CO^T} \text{Nodes}(CFT)$.*

Finalmente, definem-se duas funções parciais que especificam quais os nós resultado (*Result*) que utilizaram um dado nó.

Definição 3.13 (Resultados de Lugar e Resultados de Transição): *Dada uma rede $R = (L, T, F)$ e um colapso $CO = (CF_1, \dots, CF_n)$ em R , definem-se os **resultados de lugar** RL como uma função em L :*

$$RL : L \rightarrow \mathcal{P}(\text{Result}(CO^L))$$

$$\forall l \in \text{dom}(RL), \forall r \in RL(l), \exists CFL \in CO^L, l \in \text{Nodes}(CFL) \wedge r = \text{Result}(CFL)$$

e os **resultados de transição** RT como uma função em T :

$$RT : T \rightarrow \mathcal{P}(\text{Result}(CO^T))$$

$$\forall t \in \text{dom}(RT), \forall r \in RT(t), \exists CFT \in CO^T, t \in \text{Nodes}(CFT) \wedge r = \text{Result}(CFT)$$

Na secção seguinte define-se o conjunto proposto de operações sobre as redes.

3.3 Operações

Nesta secção definem-se as operações de adição e de subtracção de redes. Antes, são definidas quatro operações auxiliares: (1) renomeação de nós; (2) remoção de nós; (3) colapsagem de rede; e (4) união disjunta de redes.

A **renomeação de nós** permite a modificação dos identificadores de um conjunto de nós. Tal é útil especialmente após outras operações de modificação de redes, por duas razões: (1) por ser necessário atribuir novas interpretações aos nós resultado, interpretações estas que devem ser reflectidas nos identificadores dos nós, (2) para simplificar os identificadores completos resultantes de operações anteriores.

Definição 3.14 (Renomeação de nós): *Seja $R = (L, T, F)$ uma rede, e S um conjunto que satisfaz os seguintes dois requisitos: (1) $S = \{a_1, \dots, a_n\}$ e (2) $S \cap (L \cup T) = \emptyset$. Uma renomeação de nós de R é denotada $R/(/a_1/b_1/, \dots, /a_n/b_n/)$, em que $\{b_1, \dots, b_n\} \subseteq (L \cup T)$, e devolve uma nova rede $R' = (L', T', F')$ em que L' , T' e F' são obtidos por substituição de b_i por a_i , em L , T e F , respectivamente.*

A **remoção de nós** limita-se a remover os nós especificados e todos os arcos que a eles se encontrem ligados. É uma operação simples mas que no caso de uma classe de redes anotadas exigirá provavelmente cuidados especiais de forma a não originar redes sintáctica ou semanticamente erradas. Um exemplo desta situação pode ocorrer, numa rede colorida, aquando da remoção de variáveis em arcos de entrada de uma transição e que são utilizadas noutras anotações.

Definição 3.15 (Remoção de Nós): *Seja $R = (L, T, F)$ uma rede e seja $S \subseteq (L \cup T)$ um conjunto de nós a serem removidos de R . A remoção de nós, denotada $R \setminus S$, devolve uma nova rede $R' = R \setminus S = (L', T', F') = (L \setminus S, T \setminus S, F \cap ((L' \times T') \cup (T' \times L')))$.*

Seguidamente, define-se uma operação primitiva que também opera sobre uma única rede: a **colapsagem de rede**. Quer a adição quer a subtração utilizam esta operação. Resumidamente, a colapsagem de uma rede aplica um colapso à rede: para cada conjunto de fusão nominativo, os seus nós são removidos e os arcos correspondentes ficam ligados aos respectivos nós resultado.

Definição 3.16 (Colapsagem de Rede): *Seja $R = (L, T, F)$ uma rede e seja $CO = (CF_1, \dots, CF_n)$ um colapso em R . A associação de CO à rede R , denotada $R(CF_1, \dots, CF_n)$, especifica uma colapsagem da rede R . Seja agora $LR = \text{Result}(CO^L)$ e $TR = \text{Result}(CO^T)$. A colapsagem da rede R devolve uma nova rede $R' = (L', T', F')$ onde:*

$$\begin{aligned} L' &= (L \setminus L_{CO}) \cup LR \\ T' &= (T \setminus T_{CO}) \cup TR \\ F' &= F \cap ((L' \times T') \cup (T' \times L')) \cup \\ &\quad \{(lr, t) \in LR \times T \mid \exists l \in L, lr \in RL(l) \wedge (l, t) \in F\} \cup \\ &\quad \{(t, lr) \in T \times LR \mid \exists l \in L, lr \in RL(l) \wedge (t, l) \in F\} \cup \\ &\quad \{(tr, l) \in TR \times L \mid \exists t \in T, tr \in RT(t) \wedge (t, l) \in F\} \cup \\ &\quad \{(l, tr) \in L \times TR \mid \exists t \in T, tr \in RT(t) \wedge (l, t) \in F\} \cup \\ &\quad \{(lr, tr) \in LR \times TR \mid \exists (l, t) \in F, lr \in RL(l) \wedge tr \in RT(t)\} \cup \\ &\quad \{(tr, lr) \in TR \times LR \mid \exists (t, l) \in F, tr \in RT(t) \wedge lr \in RL(l)\} \end{aligned}$$

Um colapso diz-se neutro quando a sua aplicação a uma rede não a modifica:

Definição 3.17 (Colapso Neutro): *Dada uma rede $R = (L, T, F)$, $CO_0 = (l_1 \rightarrow l_1, \dots, l_{nl} \rightarrow l_{nl}, t_1 \rightarrow t_1, \dots, t_{nt} \rightarrow t_{nt})$, com $\{l_1, \dots, l_{nl}\} \subseteq L$ e $\{t_1, \dots, t_{nt}\} \subseteq T$, é um **colapso neutro** de R . Nesse caso $R(CO_0) = R$.*

A união disjunta de redes (Def. 3.18) é uma operação trivial que torna duas redes numa só sem as modificar e sem as interligar.

Definição 3.18 (União Disjunta de Redes): *Dadas duas ou mais redes $R_i = (L_i, T_i, F_i)$, com $2 \leq i \leq n$, a união disjunta destas n redes, denotada $R_1 \uplus \dots \uplus R_n$, devolve uma nova rede $R' = (L_1 \cup \dots \cup L_n, T_1 \cup \dots \cup T_n, F_1 \cup \dots \cup F_n)$.*

Devido ao facto de, na maioria dos casos, as redes desconexas não se poderem influenciar entre si, a união disjunta, só por si, não é de grande utilidade prática para as classes de redes de Petri usuais — embora de grande conveniência teórica.

A definição de uma nova rede como sendo a união disjunta de redes origina uma relação de inclusão entre a nova rede e as redes unidas. Estas tornam-se sub-redes da nova super-rede. Dir-se-á também que uma rede R_B está aninhada noutra rede R_A , ou que R_B é uma sub-rede de R_A ou ainda que R_A é uma super-rede de R_B , quando $R_A \supset R_B$. Com base nesta relação definem-se os conceitos de identificador completo e identificador reduzido:

Definição 3.19 (Identificador Completo e Identificador Reduzido): *Dada uma sequência de redes aninhadas $R_1 \supset \dots \supset R_k$, todo o nó $n \in L_k \cup T_k$ tem como identificador completo uma lista $nID = (R_1, \dots, R_k, n)$. Qualquer sublista de nID que contenha n é um identificador reduzido.*

Na definição de modelos e para especificar um identificador completo ou reduzido, utiliza-se o ponto para separar elementos da lista. Tal foi já exemplificado na Secção 3.1.2. Aí foram também utilizados índices que correspondem à utilização de vectores de instâncias de redes, ou de nós. Estes são apresentados na Secção 3.4. A sintaxe utilizada coloca os índices entre parêntesis rectos.

Podemos agora definir a adição de redes, já utilizada na definição de vários modelos nas Secções 3.1.2 e 3.1.4: **ParkA** (pág. 53), **ParkExit** (pág. 55), **E2Park** (pág. 58), **ParkAPassage2** (pág. 60), **E3Park2Exit2** (pág. 61), **E3Park2Exit2Counter** (pág. 67) e **E3Park2Exit2CounterAlt** (pág. 68).

Definição 3.20 (Adição de Redes): *Dadas n redes $R_1 = (L_1, T_1, F_1), \dots, R_n = (L_n, T_n, F_n)$, e um colapso CI em $R_1 \uplus \dots \uplus R_n$, denominado **colapso de interface**, a adição das n redes, denotada $(R_1 + \dots + R_n)(CI)$, devolve uma nova rede R' definida como: $(R_1 + \dots + R_n)(CI) = R' = (R_1 \uplus \dots \uplus R_n)(CI)$.*

Da Def. 3.20 podemos concluir que para um colapso de interface neutro, vazio, ou de conectivo vazio, a adição de redes reduz-se à união disjunta, dado que as redes são unificadas mas nenhum nó é fundido. Os nós do conectivo de um colapso de interface CI são denominados **nós interface** e são denotados por $IN(CI)$. Temos, portanto: $IN(CI) = Nodes(CI^C)$. Da mesma forma, também se referem **lugares interface** e **transições interface**.

A subtracção de redes (Def. 3.21) define uma forma estruturada para remoção de parte de um modelo. É definida como uma adição de redes seguida de uma remoção de nós. Para tal utiliza um segundo colapso de rede, denominado **colapso de remoção**.

Definição 3.21 (Subtracção de Redes): *Dadas n redes $R_1 = (L_1, T_1, F_1), \dots, R_n = (L_n, T_n, F_n)$, um colapso CI em $R_1 \uplus \dots \uplus R_n$, e um colapso de conexão (CR) denominado **colapso de remoção**, satisfazendo $Nodes(CI) \cap Nodes(CR) = \emptyset$, a subtracção das redes, denotada $(R_1 - \dots - R_n)(CI)(CR)$, devolve uma nova rede R_C definida como: $(R_1 - \dots - R_n)(CI)(CR) = R_C = (R_1 + \dots + R_n)(CI, CR) \setminus Result(CR)$.*

Na subtracção, as redes são adicionadas considerando ambos os colapsos, mas seguidamente os nós resultado do colapso de remoção são removidos, bem como os arcos que a eles se encontrem ligados. Tal como na adição de redes, os arcos entre nós de interface são mantidos.

3.4 Instâncias de Redes e Vectores de Instâncias

Conforme referido nas secções anteriores, cada rede pode ser vista como um módulo que pode ser incluído numa super-rede como um tipo especial de nó. Cada rede pode também ser utilizada como uma fonte geradora de instâncias. Para suportar a utilização destas instâncias de redes define-se o conceito de **vector de instâncias** (Def. 3.22)⁸. O nome é motivado pela utilização genérica que é dada a estes vectores: são aplicados não apenas a redes mas também a nós, a conjuntos de fusão e a colapsos. Neste último caso, servem para denotar listas de colapsos sobre várias instâncias de uma mesma rede.

Definição 3.22 (Vectores de Instâncias): *Seja $Elem$ uma rede, um lugar, uma transição, um conjunto de fusão ou um colapso. Um vector de instâncias de $Elem$, denotado $Elem[b \dots e]$ com $0 \leq b \leq e$ e $\{b, e\} \subset \mathbb{N}_0$, é uma lista em que cada elemento (denotado $Elem[i]$) é associado a um número diferente no intervalo especificado ($b \dots e$): $Elem[b \dots e] = (Elem[b], \dots, Elem[e])$.*

Conforme já referido, cada instância de rede renomeia, implicitamente, cada um dos seus componentes (nós e arcos) por meio da adição do respectivo nome como prefixo. Por exemplo, dada uma rede $R = (L, T, F) = (\{l_a, l_b\}, \{t_a, t_b\}, \{(l_a, t_a), (t_a, l_b), (l_b, t_b)\})$, o vector $R[1 \dots 2]$ define as redes $R[1] = (\{R[1].l_a, R[1].l_b\}, \{R[1].t_a, R[1].t_b\}, \{(R[1].l_a, R[1].t_a), (R[1].t_a, R[1].l_b), (R[1].l_b, R[1].t_b)\})$ e $R[2] = (\{R[2].l_a, R[2].l_b\}, \{R[2].t_a, R[2].t_b\}, \{(R[2].l_a, R[2].t_a), (R[2].t_a, R[2].l_b), (R[2].l_b, R[2].t_b)\})$.

Definem-se também variáveis inteiras positivas que servem como índices de vector e que assumem, de forma iterativa, todos os valores no intervalo especificado. Por esta razão e por influência do termo idêntico utilizado na literatura técnica em língua inglesa (*iterator*), estas variáveis são designadas por **variáveis iteradoras** ou **iteradores**.

Definição 3.23 (Variável Iteradora): *Uma definição de vector de instâncias pode especificar uma variável iteradora (ou iterador). Esta variável pode tomar valores inteiros não negativos no intervalo especificado e que são utilizados como expressões ou partes de expressões cujo resultado corresponde a uma posição num vector. A definição de uma variável iteradora i numa definição de vector tem a seguinte sintaxe: $e[i : \min \dots \max]$ em que e pode ser um identificador de rede, de nó, um conjunto de fusão ou um colapso.*

Tal como os vectores de instâncias, as variáveis iteradoras são aplicáveis a identificadores de redes, de nós, a conjuntos de fusão ou a colapsos. Tal permite a especificação de semânticas

⁸Denominado *component vector* em [Barros e Gomes, 2004c].

muito distintas de uma forma compacta. Veja-se por exemplo o caso dos dois modelos na Fig. 3.21 e as respectivas adições por meio das duas expressões seguintes:

$$R2cf := (Na + Nb[0...1])(/t1/Nb[i].t2/ \rightarrow t[i])[i:0...1]$$

$$R1cf := (Na + Nb[0...1])(/t1/(Nb[i].t2)[i:0...1]/ \rightarrow t)$$

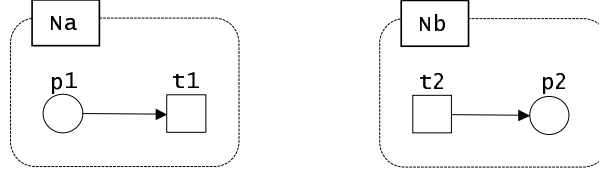


Figura 3.21: Duas redes a compor de duas formas distintas.

No primeiro caso (modelo **R2cf**), a transição **t1** é fundida (para $i = 0$) com a transição **Nb[0].t2** originando uma nova transição **t[0]**. A transição **t1** é novamente fundida mas agora (para $i = 1$) com a transição **Nb[1].t2** (outra instância do módulo **Nb**, outro conjunto de fusão e outro colapso) originando a transição **t[1]**. Obtém-se assim o modelo na Fig. 3.22.

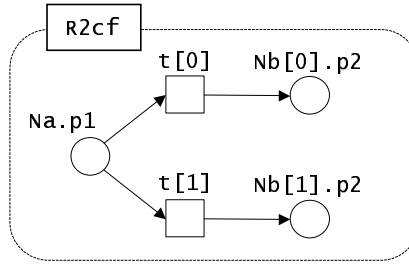


Figura 3.22: Resultado da adição utilizando dois conjuntos de fusão (**R2cf**).

No segundo caso (modelo **R1cf**), a transição **t1** é fundida com a transição **Nb[0].t2** e com a transição **Nb[1].t2** originando uma nova, e única, transição **t**. Obtemos assim o modelo na Fig. 3.23.

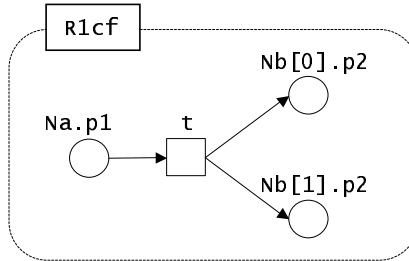


Figura 3.23: Resultado da adição utilizando um conjunto de fusão (**R1cf**).

Um Exemplo de Múltiplas Instâncias: o Problema dos Filósofos A modelação com uma rede de Petri do bem conhecido problema dos filósofos [Dijkstra, 1971] é um exemplo claro

em que são utilizadas várias instâncias de uma mesma rede⁹. Facilmente se identifica uma rede correspondente a um filósofo que corresponde também à parte activa do modelo. Os garfos são a parte passiva e cada um pode ser modelado por um lugar que deverá estar marcado no modelo inicial. Este lugar pode ser visto como uma rede trivial. Tal permite-nos compor um modelo para n filósofos partindo de duas redes: a rede **Phil** (geradora de instâncias de filósofos) e a rede **Fork** (geradora de instâncias de garfos) (*vide* Fig. 3.24). O lugar da rede **Fork** deve estar marcado de forma a representar o garfo e também para melhor justificar a sua existência dado que a rede *Phil* já disponibiliza lugares para os garfos. Como as marcas (um tipo particular de anotação) não foram ainda tratadas, o modelo surge não marcado. Conforme já referido, a Secção 3.5 apresenta uma forma generalizada de tratar as várias anotações incluindo, naturalmente, as marcações em redes de baixo nível.

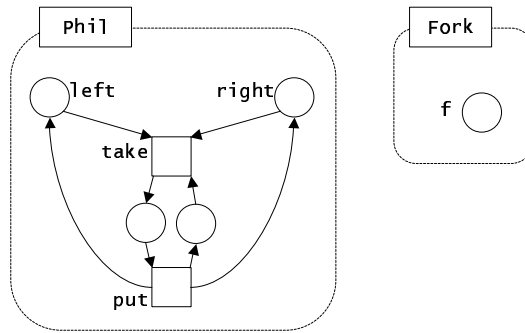


Figura 3.24: Redes Phil e Fork.

Pode então construir-se o modelo clássico, com cinco filósofos, através da adição de cinco instâncias de cada rede, o que corresponde ao modelo total aqui designado por **Dinner5**. Note-se a utilização de índices associados aos nós resultado $f[0]$ a $f[4]$.

```
Dinner5 := (Phil[0] + Phil[1] + Phil[2] + Phil[3] + Phil[4] +
            Fork[0] + Fork[1] + Fork[2] + Fork[3] + Fork[4])
(/Phil[0].right/Phil[1].left/Fork[0].f/ -> f[0],
 /Phil[1].right/Phil[2].left/Fork[1].f/ -> f[1],
 /Phil[2].right/Phil[3].left/Fork[2].f/ -> f[2],
 /Phil[3].right/Phil[4].left/Fork[3].f/ -> f[3],
 /Phil[4].right/Phil[0].left/Fork[4].f/ -> f[4])
```

Dado o modelo **Dinner5** conter apenas cinco filósofos é ainda possível representá-lo de forma gráfica (*vide* Fig. 3.25).

O facto do modelo dos filósofos ser construído a partir de várias instâncias de duas redes, ou de apenas uma se dispensarmos a definição da rede **Fork**, torna-o especialmente adequado à modelação utilizando vectores de instâncias. Tal permite uma solução compacta cujo modelo

⁹Considera-se que cada filósofo retira ambos os garfos simultaneamente.

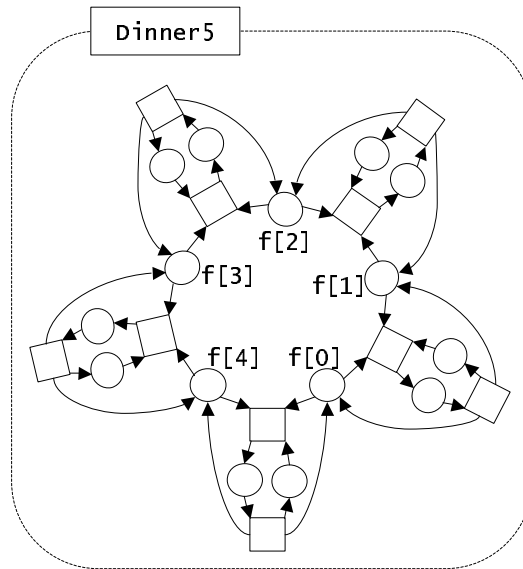


Figura 3.25: A representação gráfica do modelo Dinner5.

é imune ao aumento do sistema modelado, ou seja, ao problema da explosão do modelo. Do ponto de vista prático este facto assume enorme importância pois não obriga à edição gráfica de um modelo de maior dimensão o que, dependendo dessa dimensão, pode ser demorado ao ponto de tornar a sua construção impossível por impraticável. O modelo **Dinner** generaliza o modelo **Dinner5** para n filósofos. Para tal utiliza dois vectores de instâncias, identificadores reduzidos, um vector de colapsos com uma variável iteradora associada (i), e o operador MOD que devolve o resto da divisão inteira dos seus operandos e é aqui utilizado para especificar um contador circular de módulo n .

```
Dinner := (Phil[0...n-1] + Fork[0...n-1])
          (/Phil[i].right/Phil[(i+1) MOD n].left/Fork[i].f/ -> f[i])[i:0...n-1]
```

A secção seguinte discute a extensão das operações apresentadas a redes com uma variedade e quantidade arbitrárias de anotações associadas às redes, aos nós e aos arcos.

3.5 Extensão das Operações a Outras Classes de Redes

Nesta secção generalizam-se as operações de adição e subtracção de redes, definidas na secção anterior, a qualquer tipo de rede especificável em PNML. As definições aqui apresentadas baseiam-se nas definições nas Secções 3.2 e 3.3, pelo que também utilizam a linguagem da teoria de conjuntos. O capítulo seguinte inclui um exemplo de implementação das generalizações aqui definidas (*vide* §4.2.2). Seguem-se algumas definições e notações necessárias à definição das operações de adição e subtracção generalizadas.

3.5.1 Definições Preliminares

Na definição das anotações, e tal como no caso das redes de Petri coloridas, assume-se a existência de uma linguagem para as anotações, aqui denominada **linguagem de inscrições**¹⁰ e denotada por \mathcal{L} . Na prática, esta linguagem deve ser uma linguagem de programação pelo que se omite a sua descrição. Qualquer uma das mais conhecidas linguagens de programação imperativas ou funcionais é adequada e provavelmente muitas outras também o serão. A razão pela qual uma linguagem de descrição de dados (e.g. XML [W3C, 2005]) não é suficiente deve-se à conveniência de utilizar a mesma linguagem para especificar e avaliar as anotações (as que necessitem de o ser) e para definir transformações sobre as anotações quando procedemos a adições ou subtracções. No entanto, é possível utilizar uma linguagem para especificar e avaliar as anotações e outra para especificar transformações sobre essas anotações.

Tal como suportados pela Petri Net Markup Language (PNML), as redes (elementos XML **net**), os lugares (elementos XML **place**), as transições (elementos XML **transition**) e os arcos (elementos XML **arc**) podem ter, cada um, um conjunto de tipos de anotação associados. Esses **conjuntos de tipos de anotação**, para redes, lugares, transições e arcos, são aqui denotados, respectivamente, por TAR , TAL , TAT e TAF . Mais informalmente, são os "nomes das anotações". Considerados juntamente, os quatro conjuntos constituem um **conjunto de tipos de anotação** de uma classe de redes:

Definição 3.24 (Conjunto de Tipos de Anotação): *Um conjunto de tipos de anotação é um conjunto de quatro conjuntos $TAnot = \{TAR, TAL, TAT, TAF\}$ que satisfaz os seguintes requisitos:*

1. TAR é um conjunto finito de tipos denominados **tipos de anotação de rede**.
2. TAL é um conjunto finito de tipos denominados **tipos de anotação de lugar**.
3. TAT é um conjunto finito de tipos denominados **tipos de anotação de transição**.
4. TAF é um conjunto finito de tipos denominados **tipos de anotação de fluxo**.

Em cada um dos elementos PNML referidos, cada um dos tipos de anotação pode ser associado a um **valor de anotação** que corresponde a uma *string* numa linguagem de inscrições \mathcal{L} . Mais informalmente, são as anotações "propriamente ditas" da rede. Para um dado tipo de anotação ta , o conjunto de anotações, em \mathcal{L} , permitidas como valor para esse tipo, é denotado $A_{ta}^{\mathcal{L}}$. De forma idêntica, os conjuntos de anotações em \mathcal{L} permitidas para algum tipo de anotação de rede, lugar, transição ou arco são denotados, respectivamente, por $A_{TAR}^{\mathcal{L}}$, $A_{TAL}^{\mathcal{L}}$, $A_{TAT}^{\mathcal{L}}$ e $A_{TAF}^{\mathcal{L}}$. Assim, $A_{TAR}^{\mathcal{L}} = \bigcup_{tar \in TAR} A_{tar}^{\mathcal{L}}$ e de igual forma para $A_{TAL}^{\mathcal{L}}$, $A_{TAT}^{\mathcal{L}}$ e $A_{TAF}^{\mathcal{L}}$.

Define-se agora rede de Petri anotada.

¹⁰Em inglês: *inscription language*.

Definição 3.25 (Rede de Petri Anotada): Dada uma linguagem de inscrições \mathcal{L} , uma rede de Petri anotada é um tuplo $R = (L, T, F, TAnot, FA_R^{\mathcal{L}}, FA_L^{\mathcal{L}}, FA_T^{\mathcal{L}}, FA_F^{\mathcal{L}})$ que satisfaz os seguintes requisitos:

1. L é um conjunto finito de **lugares**.
2. T é um conjunto finito de **transições** (disjunto de L).
3. F é a relação de fluxo (correspondente aos **arcos da rede**): $F \subseteq (L \times T \cup T \times L)$.
4. $TAnot$ é um conjunto de tipos de anotação (vide Def. 3.24).
5. $FA_R^{\mathcal{L}}$ é uma **função anotação de redes** que aplica cada anotação de rede numa anotação expressa em \mathcal{L} : $FA_R^{\mathcal{L}} : TAR \rightarrow A_{TAR}^{\mathcal{L}}$. Da mesma forma, com $tar \in TAR$, define-se $FA_{tar}^{\mathcal{L}} : \{tar\} \rightarrow A_{tar}^{\mathcal{L}}$.
6. $FA_L^{\mathcal{L}}$ é uma **função anotação de lugares** que aplica cada anotação de cada lugar numa anotação expressa em \mathcal{L} : $FA_L^{\mathcal{L}} : (L \times TAL) \rightarrow A_{TAL}^{\mathcal{L}}$. Da mesma forma, com $tal \in TAL$, define-se $FA_{tal}^{\mathcal{L}} : (L \times \{tal\}) \rightarrow A_{tal}^{\mathcal{L}}$.
7. $FA_T^{\mathcal{L}}$ é uma **função anotação de transições** que aplica cada anotação de cada transição numa anotação expressa em \mathcal{L} : $FA_T^{\mathcal{L}} : (T \times TAT) \rightarrow A_{TAT}^{\mathcal{L}}$. Da mesma forma, com $tat \in TAT$, define-se $FA_{tat}^{\mathcal{L}} : (T \times \{tat\}) \rightarrow A_{tat}^{\mathcal{L}}$.
8. $FA_F^{\mathcal{L}}$ é uma **função anotação de fluxos** que aplica cada anotação de cada fluxo numa anotação expressa em \mathcal{L} : $FA_F^{\mathcal{L}} : (F \times TAF) \rightarrow A_{TAF}^{\mathcal{L}}$. Da mesma forma, com $taf \in TAF$, define-se $FA_{taf}^{\mathcal{L}} : (F \times \{taf\}) \rightarrow A_{taf}^{\mathcal{L}}$.

Com base nas Defs. 3.24 e 3.25, considera-se que uma rede de Petri anotada pode ser apresentada, na sua forma mais abreviada, como o tuplo $(L, T, F, TAnot, FA_R^{\mathcal{L}}, FA_L^{\mathcal{L}}, FA_T^{\mathcal{L}}, FA_F^{\mathcal{L}})$ ou, discriminando os elementos de $TAnot$, como o tuplo $(L, T, F, TAR, TAL, TAT, TAF, FA_R^{\mathcal{L}}, FA_L^{\mathcal{L}}, FA_T^{\mathcal{L}}, FA_F^{\mathcal{L}})$.

Por exemplo, uma rede de Petri colorida, semelhante à definida por Jensen [Jensen, 1997a], pode basear-se no seguinte conjunto de tipos de anotação: $ColorSimple = \{\{name\}, \{name, marking\}, \{name, guard, codeSegment\}, \{expression\}\}$. Com igual significado, escrever-se-á $ColorSimple = \{TAR = \{name\}, TAL = \{name, marking\}, TAT = \{name, guard, codeSegment\}, TAF = \{expression\}\}$.

Considerando o tipo de anotação de rede $ColorSimple$, na rede anotada $\textcircled{11} \xrightarrow{i} \boxed{t1} \xrightarrow{i+1} \textcircled{12}$ o tipo de anotação *expression* tem os valores i e $i+1$ para cada um dos arcos, respectivamente¹¹. Tal é expresso pela função anotação de fluxos $FA_F^{\mathcal{L}}$: $FA_F^{\mathcal{L}}((l_1, t_1), expression) = i$ e $FA_F^{\mathcal{L}}((t_1, l_2), expression) = i + 1$ ou, com igual significado, $FA_F^{\mathcal{L}} = \{(((l_1, t_1), expression), i), (((t_1, l_2), expression), i + 1)\}$. Para as restantes anotações procede-se de forma idêntica.

Define-se ainda o conceito de redes anotadas compatíveis. Duas redes anotadas são compatíveis quando têm igual conjunto de tipos de anotação. Tipicamente, tal corresponderá a duas redes de uma mesma classe, ou seja, dois modelos que utilizam a mesma classe de redes de Petri.

¹¹Poderia ser de um tipo `String`, fornecido pela linguagem \mathcal{L} , embora fosse avaliado e retornasse um valor do tipo inteiro.

Definição 3.26 (Redes de Petri Anotadas Compatíveis): *Duas redes de Petri anotadas $R_A = (L_A, T_A, F_A, T_{Anot_A}, FA_{R_A}^{\mathcal{L}}, FA_{L_A}^{\mathcal{L}}, FA_{T_A}^{\mathcal{L}}, FA_{F_A}^{\mathcal{L}})$ e $R_B = (L_B, T_B, F_B, T_{Bnot_B}, FA_{R_B}^{\mathcal{L}}, FA_{L_B}^{\mathcal{L}}, FA_{T_B}^{\mathcal{L}}, FA_{F_B}^{\mathcal{L}})$ dizem-se compatíveis se e só se $T_{Anot_A} = T_{Bnot_B}$.*

Seguidamente, generalizam-se as operações de adição e subtracção de redes às redes de Petri anotadas. O capítulo termina com dois exemplos de operações em redes anotadas.

3.5.2 Operações em Redes de Petri Anotadas

A adição e a subtracção em redes anotadas são efectuadas em cinco fases:

1. Operação gráfica conforme as Defs. 3.20 (adição) ou 3.21 (subtracção).
2. Alteração da origem ou destino dos arcos entre nós interface e outros nós, para arcos entre os respectivos nós resultado e outros nós.
3. Cálculo das anotações associadas à rede resultado a partir das respectivas anotações das redes operando.
4. Cálculo das anotações associadas aos lugares e transições resultado, a partir das respectivas anotações dos nós interface respectivos.
5. Cálculo dos arcos entre nós resultado, e das respectivas anotações, a partir das anotações dos arcos entre os nós interface respectivos.

Para a adição, dadas duas redes anotadas (Def. 3.25) e uma estrutura de transformação (Def. 3.28 na pág. 84), começa-se por efectuar a adição definida na página 75 (Def. 3.20) correspondente a uma transformação gráfica (ponto 1).

As anotações associadas aos arcos, entre nós interface e outros nós nas redes operando, não são modificadas. Como tal basta modificar a origem ou destino destes arcos para o respectivo nó resultado (ponto 2).

No ponto 3 calculam-se as anotações da própria rede resultado aplicando as funções de transformação de anotações de rede (*vide* $FT_{TAR}^{\mathcal{L}}$ na Def. 3.28 da pág. 84).

No ponto 4, calculam-se as anotações dos nós interface aplicando as funções de transformação de anotações de lugar e as funções de transformação de anotações de transição (*vide* $FT_{TAL}^{\mathcal{L}}$ e $FT_{TAT}^{\mathcal{L}}$ na Def. 3.28 da pág. 84).

Por fim, os arcos e as anotações associadas aos arcos entre nós interface também podem ser modificadas. Para tal definem-se **pares de interface**. Cada um destes é constituído por um par de conjuntos de fusão em que um ou mais nós estão ligados entre si por arcos. As funções de

transformação dos arcos (*vide* $FT_{TAF}^{\mathcal{L}}$ na Def. 3.28 da pág. 84) são então utilizadas para definir os arcos, e as respectivas anotações, entre os nós resultado de cada par de interface (ponto 5).

Para a subtracção, procede-se de forma idêntica mas, em lugar da Def. 3.20 (na pág. 75), utiliza-se a Def. 3.21 (na pág. 75) e as funções de transformação correspondentes.

A Def. 3.27 define formalmente um par de interface.

Definição 3.27 (Par de Interface): *Dada uma adição das redes R_1 a R_n , $R_{res} = (R_1 + \dots + R_n)(CI)$, ou uma subtracção $R_{res} = (R_1 - \dots - R_n)(CI)(CR)$, o conjunto de pares de interface PI é definido como:*

$$PI = \left\{ (cf_l, cf_t) \in (CI^L \times CI^T) : \right. \\ \left. \left((Nodes(cf_l) \times Nodes(cf_t)) \cup (Nodes(cf_t) \times Nodes(cf_l)) \right) \cap \left(\bigcup_{1 \leq i \leq n} F_i \right) \neq \emptyset \right\}$$

Com base na Def. 3.27, utilizar-se-ão também as seguintes notações:

- $PI^L = \{cf_l \in CI^L : \exists (cf_l, cf_t) \in PI\}$
- $PI^T = \{cf_t \in CI^T : \exists (cf_l, cf_t) \in PI\}$
- $Nodes(PI) = \bigcup_{(cf_l, cf_t) \in PI} Nodes(cf_l, cf_t)$
- $F(Nodes(PI)) = \left(\bigcup_{pi \in PI} (Nodes(pi) \times Nodes(pi)) \right) \cap \left(\bigcup_{1 \leq i \leq n} F_i \right)$
- $\forall pi \in PI, Result(pi) = Result((cf_l, cf_t)) = Result(cf_l) \cup Result(cf_t)$
- $F(Result(PI)) = (Result(PI) \times Result(PI)) \cap F_{res}$

Por exemplo, na adição $R_c := (R_a + R_b) (/p1/p5/ \rightarrow p9, /p2/p6/ \rightarrow p10, /t2/t4/ \rightarrow t5, /p4/p7/ \rightarrow p11)$, cujos modelos estão ilustrados na Fig. 3.26, temos três pares de interface:

$$PI = \{ (/p1/p5/ \rightarrow p9, /t2/t4/ \rightarrow t5), \\ (/p2/p6/ \rightarrow p10, /t2/t4/ \rightarrow t5), \\ (/p4/p7/ \rightarrow p11, /t2/t4/ \rightarrow t5) \}.$$

A colapsagem de uma rede anotada obriga à existência de uma **estrutura de transformação** e, novamente, à utilização da linguagem de inscrições. Para além de definir os valores das anotações, a linguagem de inscrições (\mathcal{L}) é utilizada para definir as transformações a efectuar nas anotações.

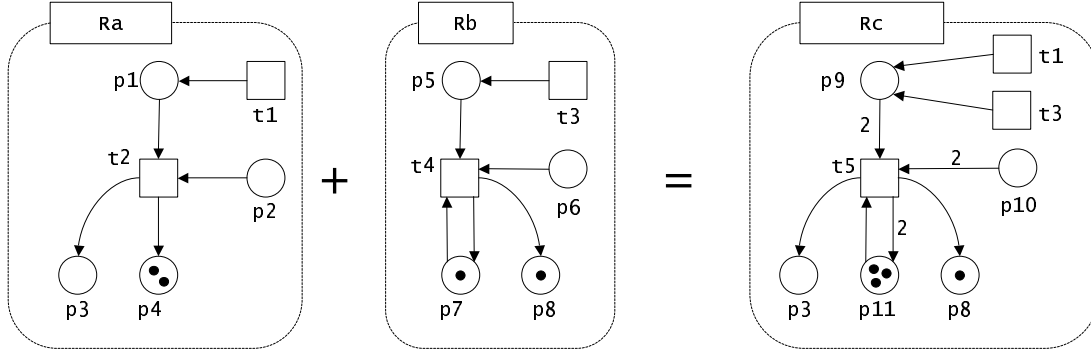


Figura 3.26: Um exemplo de adição de duas redes anotadas contendo pesos associados aos arcos e marcações associadas aos lugares.

Após aplicação da adição e subtração gráficas, já apresentadas na Secção 3.3 da pág. 73, é utilizada uma **estrutura de transformação** para efectuar a transformação das anotações. Algumas anotações podem não ser fundíveis, por não existir ou não ser desejável atribuir uma semântica à sua fusão. Por exemplo, não é claro como efectuar a fusão de transições que tenham associada uma anotação para especificação de uma prioridade de disparo. Por essa ou outra razão, essa fusão pode não ser definida, sendo considerada ilegal.

Definição 3.28 (Estrutura de Transformação): Seja \mathcal{L} uma linguagem de inscrições, $OP = \{+, -\}$ um conjunto de operações e $TAnot = (TAR, TAL, TAT, TAF)$ um conjunto de tipos de anotação. Dada uma adição das redes anotadas $(R_1 + \dots + R_n)(CI)$, ou uma subtração das redes anotadas $(R_1 - \dots - R_n)(CI)(CR)$, com PI o conjunto dos pares interface de CI e $F = \bigcup_{1 \leq i \leq n} F_i$, uma estrutura de transformação é um tuplo $ET = (FT_{TAR}^{\mathcal{L}}, FT_{TAL}^{\mathcal{L}}, FT_{TAT}^{\mathcal{L}}, FT_{TAF}^{\mathcal{L}})$ de conjuntos de funções de transformação de anotações que satisfaz os seguintes requisitos:

1. $FT_{TAR}^{\mathcal{L}}$ é o conjunto das funções de transformação (implementadas em \mathcal{L}) que aplicam subconjuntos de anotações de rede numa nova anotação de rede: $\forall tar \in TAR, \forall op \in OP, \exists ft_{(tar, op)} \in FT_{TAR}^{\mathcal{L}}, ft_{(tar, op)} : \mathcal{P}(A_{TAR}^{\mathcal{L}}) \rightarrow (A_{tar}^{\mathcal{L}} \cup \perp)$.
2. $FT_{TAL}^{\mathcal{L}}$ é o conjunto das funções de transformação (implementadas em \mathcal{L}) que aplicam subconjuntos de anotações de lugar numa nova anotação de lugar: $\forall tal \in TAL, \forall op \in OP, \exists ft_{(tal, op)} \in FT_{TAL}^{\mathcal{L}}, ft_{(tal, op)} : \mathcal{P}(A_{TAL}^{\mathcal{L}}) \rightarrow (A_{tal}^{\mathcal{L}} \cup \perp)$.
3. $FT_{TAT}^{\mathcal{L}}$ é o conjunto das funções de transformação (implementadas em \mathcal{L}) que aplicam subconjuntos de anotações de transição numa nova anotação de transição: $\forall tat \in TAT, \forall op \in OP, \exists ft_{(tat, op)} \in FT_{TAT}^{\mathcal{L}}, ft_{(tat, op)} : \mathcal{P}(A_{TAT}^{\mathcal{L}}) \rightarrow (A_{tat}^{\mathcal{L}} \cup \perp)$.
4. $FT_{TAF}^{\mathcal{L}}$ é o conjunto das funções de transformação (implementadas em \mathcal{L}) que aplicam arcos e respectivas anotações numa lista de arcos entre nós de um par de interface, e respectivas anotações:

$$\forall op \in OP, \exists ft_{op} \in FT_{TAF}^{\mathcal{L}}, ft_{op} : \mathcal{P}(F(Nodes(PI)) \times TAF \times A_{TAF}^{\mathcal{L}}) \rightarrow (\mathcal{P}(F(Result(PI)) \times TAF \times A_{TAF}^{\mathcal{L}}) \cup \perp)$$

De acordo com a Def. 3.28, cada anotação de rede é gerada a partir de todas as anotações de rede das redes operando. No caso dos lugares e transições, é definida uma função de transformação para cada anotação e para cada operação. Essa função baseia-se no conjunto de anotações dos nós presentes no conjunto de fusão respectivo. Finalmente, no caso dos arcos, é definida uma função para cada operação. Essa função baseia-se em todos os arcos, e respectivas anotações, entre os nós de cada par de interface. Devolve um novo conjunto de arcos, e respectivas anotações, entre os nós resultado do par de interface.

No caso atrás referido, para a anotação prioridade teríamos $FT_{prioridade}^{\mathcal{L}} = \perp$.

Os conjuntos de funções de transformação que compõem a estrutura de transformação são particionados conforme a quantidade e qualidade das operações que se pretendam suportar. Como estamos a considerar o conjunto de operações $OP = \{+, -\}$, consideram-se dois subconjuntos para cada conjunto de funções de transformação. Assim temos: $FT_{TAR}^{\mathcal{L}} = FT_{(TAR,+)}^{\mathcal{L}} \cup FT_{(TAR,-)}^{\mathcal{L}}$; $FT_{TAL}^{\mathcal{L}} = FT_{(TAL,+)}^{\mathcal{L}} \cup FT_{(TAL,-)}^{\mathcal{L}}$; $FT_{TAT}^{\mathcal{L}} = FT_{(TAT,+)}^{\mathcal{L}} \cup FT_{(TAT,-)}^{\mathcal{L}}$; e $FT_{TAF}^{\mathcal{L}} = FT_{(TAF,+)}^{\mathcal{L}} \cup FT_{(TAF,-)}^{\mathcal{L}}$.

Definição 3.29 (Adição de Redes Anotadas): *Sejam R_1, \dots, R_n n redes anotadas compatíveis, CI um colapso de interface entre elas, R_{total} a união disjunta das mesmas e PI o conjunto dos pares interface de CI . Seja ainda R_{res} a rede que resulta da transformação da rede R_{total} pela estrutura de transformação $ET = (FT_{TAR}^{\mathcal{L}}, FT_{TAL}^{\mathcal{L}}, FT_{TAT}^{\mathcal{L}}, FT_{TAF}^{\mathcal{L}})$ de $FA_{R_{total}}^{\mathcal{L}}, FA_{L_{total}}^{\mathcal{L}}, FA_{T_{total}}^{\mathcal{L}}$ e $FA_{F_{total}}^{\mathcal{L}}$ em $FA_{R_{res}}^{\mathcal{L}}, FA_{L_{res}}^{\mathcal{L}}, FA_{T_{res}}^{\mathcal{L}}$ e $FA_{F_{res}}^{\mathcal{L}}$, respectivamente. A rede R_{res} satisfaz os seguinte requisitos:*

1. Sendo (L_{na}, T_{na}, F_{na}) a rede não anotada resultante da adição $((L_i, T_i, F_i) + \dots + (L_n, T_n, F_n))(CI)$, teremos:

$$(a) \ L_{res} = L_{na}.$$

$$(b) \ T_{res} = T_{na}.$$

$$(c) \ F_{res} = F_{na} \cup F(FT_{(TAF,+)}^{\mathcal{L}}) \setminus (Nodes(PI) \times Nodes(PI)).$$

$$2. \ FA_{R_{res}}^{\mathcal{L}} = \left\{ (tar, ar) \in (TAR \times A_{tar}^{\mathcal{L}}) \mid \right. \\ \left. \exists ft_{(tar,+)} \in FT_{(TAR,+)}^{\mathcal{L}}, ar = ft_{(tar,+)}(FA_{R_{total}}^{\mathcal{L}}) \right\}$$

$$3. \ FA_{L_{res}}^{\mathcal{L}} = \left\{ ((l, tal), al) \in FA_{L_{total}}^{\mathcal{L}} \mid l \notin L_{CI} \right\} \cup \\ \left\{ ((l, tal), al) \in ((Result(CI^L) \times TAL) \times A_{tal}^{\mathcal{L}}) \mid \right. \\ \left. \exists cf_l \in CI^L, \exists ft_{(tal,+)} \in FT_{(TAL,+)}^{\mathcal{L}}, \right. \\ \left. l = Result(cf_l) \wedge al = ft_{(tal,+)}(FA_{L_{total}}^{\mathcal{L}}(Nodes(cf_l) \times TAL)) \right\}$$

$$4. \ FA_{T_{res}}^{\mathcal{L}} = \left\{ ((t, tat), at) \in FA_{T_{total}}^{\mathcal{L}} \mid t \notin T_{CI} \right\} \cup \\ \left\{ ((t, tat), at) \in ((Result(CI^T) \times TAT) \times A_{tat}^{\mathcal{L}}) \mid \right. \\ \left. \exists cf_t \in CI^T, \exists ft_{(tat,+)} \in FT_{(TAT,+)}^{\mathcal{L}}, \right. \\ \left. t = Result(cf_t) \wedge at = ft_{(tat,+)}(FA_{T_{total}}^{\mathcal{L}}(Nodes(cf_t) \times TAT)) \right\}$$

$$\begin{aligned}
5. \quad FA_{F_{res}}^{\mathcal{L}} &= \text{arcos sem ligação a colapsos} \\
&\left\{ (((x, y), taf), af) \in FA_{F_{total}}^{\mathcal{L}} \mid \{x, y\} \cap Nodes(PI) = \emptyset \right\} \cup \\
&\quad \text{arcos "dentro" de pares de interface} \\
&\left\{ ((fr, taf), af) \in ((F(Result(PI)) \times TAF) \times A_{TAF}^{\mathcal{L}}) \mid \right. \\
&\quad \left. \exists ft_+ \in FT_{(TAF, +)}^{\mathcal{L}}, ((fr, taf), af) \in ft_+(FA_{(F_{total} \cap F(Nodes(PI)))}^{\mathcal{L}}) \right\} \cup \\
&\quad \text{arcos entre um colapso e um nó ou entre um nó e um colapso} \\
&\left\{ (((x, y), taf), af) \in ((F_{res} \times TAF) \times A_{TAF}^{\mathcal{L}}) \mid \right. \\
&\quad \exists cf \in CI, \exists z \in Nodes(cf), \\
&\quad (x = Result(cf) \wedge y \notin Result(CI) \wedge af = FA_{F_{total}}^{\mathcal{L}}((z, y), taf)) \vee \\
&\quad \left. (x \notin Result(CI) \wedge y = Result(cf) \wedge af = FA_{F_{total}}^{\mathcal{L}}((x, z), taf)) \right\}
\end{aligned}$$

Note-se que as estruturas das redes são adicionadas tal como nas redes não anotadas, mas os arcos entre nós do mesmo par de interface são removidos (*vide* 1c). Em seu lugar irão surgir os nós resultantes da função de transformação de fluxos para a adição.

Definição 3.30 (Subtracção de Redes Anotadas): *Idêntica à adição de redes anotadas (Def. 3.29) mas utilizando a definição de subtracção gráfica (Def. 3.21), os conjuntos de funções de transformação respectivos e um colapso de remoção (CR) entre as redes R_1, \dots, R_n .*

A secção seguinte ilustra a operação de adição em redes anotadas.

3.5.3 Exemplos de Adições em Redes de Petri Anotadas

Esta secção apresenta mais dois exemplos de adições em redes anotadas. Neste caso apresentam-se as especificações completas, quer das redes operando, quer das estruturas de transformação utilizadas.

O primeiro exemplo ilustra a especificação das redes **Producer** e **Consumer** na Fig. 2.6 na pág. 34. O segundo exemplo ilustra a aplicação da adição de redes anotadas na adição de árbitros [Smith, 1993; Gomes e Steiger-Garção, 1996; Gomes, 2005].

Adição de Canais Síncronos

As redes na Fig. 2.6, na pág. 34, estão ligadas dinamicamente por um canal de sincronismo. A rede na Fig. 2.7 na pág. 34 mostra o modelo com igual comportamento mas sem o canal de sincronismo. Este modelo pode ser visto como o resultado de uma adição entre as redes anotadas **Producer** e **Consumer**. Assumindo um conjunto de tipos de anotação simplificado denominado *ColorSimple*, e considerando o valor da anotação *name* igual ao identificador da rede, estas

redes podem ser definidas da seguinte forma:

$$\begin{aligned}
ColorSimple = (& \mathbf{TA} \mathbf{R} = \{name\}, \\
& \mathbf{TAL} = \{name, initialMarking\}, \mathbf{TAT} = \{name, guard, channel\}, \mathbf{TAF} = \{arcInscription\}) \\
\\
Producer = (& \mathbf{L} = \{produced, sent\}, \\
& \mathbf{T} = \{produce, send\}, \\
& \mathbf{F} = \{(produce, produced), (produced, send), (send, sent), (sent, produce)\}, \\
& \mathbf{TAnot} = ColorSimple, \\
& \mathbf{FA}_R^C = \{(name, Producer)\}, \\
& \mathbf{FA}_L^C = \{((produced, name), produced), \\
& \quad ((produced, initialMarking), (1, 5)), \\
& \quad ((sent, name), sent), ((sent, initialMarking), \emptyset)\}, \\
& \mathbf{FA}_T^C = \{((produce, name), produce), \\
& \quad ((produce, guard), \emptyset), ((produce, channel), \emptyset), \\
& \quad ((send, name), send), ((send, guard), p = 'P1'), ((send, channel), x! ? ch)\}, \\
& \mathbf{FA}_F^C = \{(((produce, produced), arcInscription), (p, x)), \\
& \quad (((produced, send), arcInscription), (p, x)), \\
& \quad (((send, sent), arcInscription), p), \\
& \quad (((sent, produce), arcInscription), p)\}, \\
&) \\
Consumer = (& \mathbf{L} = \{received, consumed\}, \\
& \mathbf{T} = \{receive, consume\}, \\
& \mathbf{F} = \{(receive, received), (received, consume), (consume, consumed), (consumed, receive)\}, \\
& \mathbf{TAnot} = ColorSimple, \\
& \mathbf{FA}_R^C = \{(name, Consumer)\}, \\
& \mathbf{FA}_L^C = \{((received, name), received), \\
& \quad ((received, initialMarking), \emptyset), \\
& \quad ((consumed, name), consumed), ((consumed, initialMarking), (5)), \}, \\
& \mathbf{FA}_T^C = \{((receive, name), receive), \\
& \quad ((receive, guard), c = 'C1' or c = 'C2'), ((produce, channel), y! ? ch) \\
& \quad ((consume, name), consume), ((consume, guard), \emptyset), ((consume, channel), \emptyset)\}, \\
& \mathbf{FA}_F^C = \{(((receive, received), arcInscription), (c, y)), \\
& \quad (((received, consume), arcInscription), (c, y)), \\
& \quad (((consume, consumed), arcInscription), c), \\
& \quad (((consumed, receive), arcInscription), c)\} \\
&)
\end{aligned}$$

Estas duas redes anotadas podem ser adicionadas, obtendo-se a rede na Fig. 2.7. Para tal é necessário definir uma estrutura de transformação ($ET_{ColorSimple}$). Esta está definida com indicação dos nomes das funções de transformação na linguagem de inscrições utilizada no protótipo desenvolvido e que é apresentado na Secção 4.3: a linguagem de programação Ruby [Ruby Home Page, 2004]:

$$ET_{ColorSimple} = (\begin{array}{l} FT_{TAR}^{Ruby} = \{add_name\}, \\ FT_{TAL}^{Ruby} = \{add_name, add_initialMarking\}, \\ FT_{TAT}^{Ruby} = \{add_name, add_guard, add_channel\}, \\ FT_{TAF}^{Ruby} = \{add, sub\} \end{array})$$

As funções encontram-se definidas na Listagem 3.1 e permitem a adição das duas redes nas Figs. 2.6 e 2.7 (contemplando apenas adição de nós). Apresentam-se apenas as funções redefinidas para o tipo *ColorSimple*. Outras funções são "herdadas" da estrutura de transformação *ptNetB* (*vide* Listagem D.2 na pág. 227) e foram omitidas por não serem relevantes para o exemplo apresentado.

Listagem 3.1: Estrutura de transformação para o tipo de anotação *ColorSimple*.

```

1  # For opnml2pnml
  # Author: Joao Paulo Barros
  # Beja, Portugal
  # 2005/09/19

6  # Defines a transformation structure for a PNID named colorSimple
  # The net structure only overrides ptNetb.ts for addition.
  # initialMarking labels are "added" by simply choosing one of them for
  #   the result node.
  # Channel annotations are removed.
11 # Guard addition

  require 'ptNetb.ts' #defaults come from place/transition nets

  class NodeFusion
16     def add_initialMarking()
        result = @labels[0].xml.elements["text"].text
        return "<text>#{result}</text>"
      end

21     def add_channel()
        return "<text></text>"
      end

      def add_guard()
26     result = '(' + @labels[0].xml.elements["text"].text + ')'
        for i in 1..@labels.size - 1
            result = result + ' and (' + @labels[i].xml.elements["text"].text + ')'
        end
  end

```



```

31      channels = @fe.labels['channel']

      if (!channels.empty?())
        # add condition to guard due to the channels.
        # Only two channels are supported.
36      if (channels.size() != 2)
        Error("The number of channels must be 2.")
      end
      paramLists = Array.new
      channels.each do |ch|
41      m = Regexp.new('(.*)\\!\\.?.*').match(ch.elements['text'].text)
        Error("Syntax error in channel expression \"#{ch}\".") if m == nil
        paramLists.push(Regexp.last_match(1).to_s.split(', '))
      end
      if (paramLists[0].size() != paramLists[1].size())
46      Error('Mismatch in the number of channel parameters');
      end
      # Add comparisons, between parameters, to the guard
      for i in 0..(paramLists[0].size() - 1) do
        result = '(' + paramLists[0][i].strip() +
51      ' = ' + paramLists[1][i].strip() + ') and ' + result
      end
      end
      return "<text>#{result}</text>"
    end
56 end # end class NodeFusion

```

Adição de Árbitro

O exemplo aqui apresentado ilustra a adição de um árbitro a um modelo base existente. Este especifica a alternância entre dois processos. O modelo resultante (*vide* Fig. 3.27c) força a alternância entre os dois processos, desde que se considere uma semântica de disparo em que todas as transições aptas disparam, em cada passo de execução.

As redes na Fig. 3.27 têm o conjunto de tipos de anotação *ptTest* associado o qual, comparativamente com o tipo *ptNetb*, se limita a adicionar a anotação **type** aos arcos a qual permite qualificar um arco como arco de teste. Na prática considera-se que a omissão desta anotação significa um arco regular, ou seja, que não é de teste.

$$ptTest = (AR = \{name\}, TAL = \{name, initialMarking\}, TAT = \{name\}, TAF = \{arcInscription, type\})$$

Dada a sua dimensão omite-se a especificação formal das redes **Arbiter** e **Par**. No entanto, as especificações PNML constam, respectivamente, das listagens D.10 na pág. 238 e D.11 na pág. 240. Também a especificação da adição em Operational PNML (*vide* Listagem D.12 na pág. 243) e o respectivo ficheiro PNML resultante (*vide* Listagem D.13 na pág. 243) podem ser

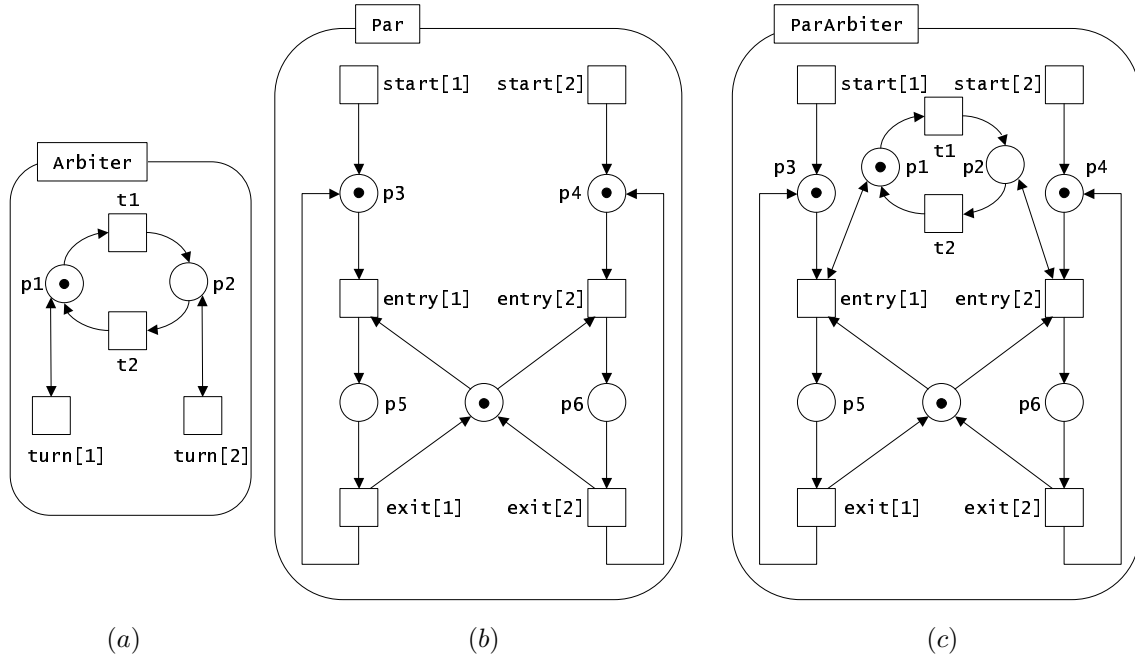


Figura 3.27: (a) Um árbitro; (b) um modelo base; (c) modelos final resultante da adição de (a) e (b).

consultados na Secção D.2.2.

Para obtenção da rede na Fig. 2.7 é necessário definir uma estrutura de transformação (ET_{ptTest}). Esta limita-se a redefinir a função de transformação para adição de arcos entre nós de cada par interface. As restantes funções são "herdadas" da definição da estrutura de transformação para redes de Petri lugar-transição, a estrutura $ptNetb$. O código, na linguagem Ruby, para a estrutura de transformação ET_{ptTest} pode ser consultado na Listagem D.4 na pág. 231.

No próximo capítulo propõem-se duas linguagens textuais para suporte às modularizações e às operações adição e subtracção.

Capítulo 4

Especificação Textual de Redes de Petri

Numa descrição gráfica temos também uma tarefa adicional: decidir como colocar os símbolos. Esta pode ser uma complicação indesejada, especialmente se estivermos convencidos de que não devem existir cruzamentos.

– Michael Jackson, 1995

Este capítulo apresenta duas formas textuais para a descrição dos mecanismos de modularização de redes apresentados no capítulo anterior: uma corresponde às expressões para definição operacional já utilizadas e é especialmente adequada à integração com ferramentas computacionais para edição interactiva de modelos em redes de Petri; a outra utiliza uma linguagem, baseada na linguagem XML, especialmente adequada para a troca de especificações de modelos entre ferramentas computacionais distintas.

No capítulo anterior propôs-se uma forma textual para a especificação de composições de redes de Petri de forma independente do tipo de redes em causa. Para que tais composições possam ser utilizadas com um máximo de conveniência e eficácia é fundamental complementá-las com uma descrição da estrutura de uma qualquer rede de Petri. Esta descrição deve também ser textual e independente do tipo de redes. Com esse objectivo, definiu-se uma linguagem para a especificação da estrutura da rede. Esta linguagem, juntamente com a utilizada no capítulo anterior, define a linguagem PN_{TEXT} apresentada na secção seguinte.

A linguagem PNML, baseada na linguagem XML e apresentada na secção 4.2.1, permite já

a definição da estrutura de uma qualquer rede de Petri. Actualmente, tudo indica que irá constituir o formato normalizado para transferência de modelos em redes de Petri [s.a., 2004d, 2005c]. Nesse sentido, o capítulo termina com a definição de uma extensão à linguagem PNML que permite a especificação, também em XML, das operações definidas no capítulo anterior.

4.1 A linguagem PN_{TEXT}

Os modelos apresentados no Capítulo 3 foram todos expressos na linguagem PN_{TEXT} . Esta permite a especificação de modelos de redes de Petri numa linguagem textual. A gramática completa da linguagem PN_{TEXT} consta do Apêndice A. No mesmo apêndice encontram-se vários exemplos de expressões válidas.

Os modelos podem ser especificados em extensão, enumerando todos os elementos que os constituem, ou em compreensão, da forma operacional apresentada no capítulo anterior. As subsecções seguintes ilustram estas duas formas de especificação.

4.1.1 Definições em Extensão

Para as definições em extensão utiliza-se uma sintaxe inspirada em duas das linguagens de programação orientadas pelos objectos actualmente mais conhecidas, as linguagens C++ e Java. Por exemplo as redes anotadas R_A e R_B , que constam da Fig. 3.26 na pág. 84, podem ser especificadas pelo seguinte código:

```

1 ra := Net("Ra")
2 ra.places := Place("p1", 0) + Place("p2", 0) + Place("p3", 0) +
3           Place("p4", 2)
4 ra.transitions := Transition("t1") + Transition("t2")
5 ra.arcs := Arc(("t1", "p1"), 1) + Arc(("p1", "t2"), 1) +
6           Arc(("p2", "t2"), 1) + Arc(("t2", "p3"), 1) +
7           Arc(("t2", "p4"), 1)
8 rb := Net("Rb")
9 rb.places := Place("p5", 0) + Place("p6", 0) + Place("p7", 1) +
10            Place("p8", 1)
11 rb.transitions := Transition("t3") + Transition("t4")
12 rb.arcs := Arc(("t3", "p5"), 1) + Arc(("p5", "t4"), 1) +
13            Arc(("p6", "t4"), 1) + Arc(("p7", "t4"), 1) +
14            Arc(("t4", "p7"), 1) + Arc(("t4", "p8"), 1)

```

Tal como na linguagem PNML, considera-se que qualquer rede de Petri, independentemente da classe a que pertença, contém lugares (**Places**), transições (**Transitions**) e arcos (**Arcs**). Assim, tipicamente define-se uma rede (e.g. `ra := Net("Ra")` na linha 1) e seguidamente são-lhe atribuídos os vários lugares, transições e arcos (linhas 2 a 7 para a rede `ra`).

No caso dos lugares e transições, indica-se sempre o identificador seguido da lista de anotações utilizadas pelo tipo de rede em uso. No caso dos arcos, o identificador é o par (*origem, destino*) constituído pelos identificadores dos nós a que o arco se encontra ligado. Aquando da definição de lugares, transições e arcos, e tal como sucede com os identificadores, todos os seus atributos podem ser especificados como parâmetros do respectivo "construtor". Por exemplo, para um lugar com identificador "p1", um atributo para especificação da marcação e outro para especificação de uma capacidade máxima, uma definição possível para uma marcação de 2 e uma capacidade de 10 seria: `Place("p1", 2, 10)`.

4.1.2 Definições em Compreensão

A definição em compreensão corresponde às definições apresentadas no capítulo anterior. Como exemplo adicional apresenta-se a definição da rede `nc` resultante da adição das redes `ra` e `rb` especificadas na subsecção anterior em *PN_{TEXT}*:

```
rc := Net("Rc")
rc := (ra + rb) (/p1/p5/ -> p9, /p2/p6/ -> p10,
                /t2/t4/ -> t5, /p4/p7/ -> p11)
```

A primeira afectação serve para atribuir um identificador e (por omissão) um nome (atributo `name`) à rede `rc`. A segunda afectação cria a rede com base na operação de adição.

4.2 Formatos Baseados em XML

Esta secção inicia-se por uma breve apresentação da linguagem *Petri Net Markup Language* (PNML) [s.a., 2004d; Jüngel et al., 2000; Billington et al., 2003] baseada em XML [W3C, 2005] e que constitui uma proposta de formato normativo para a especificação de redes de Petri num formato textual. Seguidamente apresenta-se a linguagem *Operational PNML* [Barros e Gomes, 2004e,f] também baseada em XML. A linguagem *Operational PNML* apresenta algumas semelhanças sintácticas com a PNML mas permite a especificação de redes de Petri da forma operacional apresentada no Capítulo 3.

4.2.1 A linguagem PNML

A linguagem PNML permite definir uma rede de Petri *em extensão*, ou seja, enumerando todos os seus elementos constituintes. Por exemplo, para a rede **Enter** ilustrada na Fig. 3.1 da pág. 53, o código PNML correspondente é o ilustrado na Listagem 4.1. A estrutura é muito simples e baseia-se em três conjuntos de elementos: os lugares (**place**), as transições (**transition**) e os arcos (**arc**). Todos estes três conjuntos de elementos fazem parte do elemento **net**.

Listagem 4.1: Código PNML para o modelo **Enter** na Fig. 3.1 da pág. 53.

```

<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="Enter" type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
3    <place id="entranceFree">
      <initialMarking>
        <text>0</text>
      </initialMarking>
    </place>
8    <place id="waitingTicket">
      <initialMarking>
        <text>0</text>
      </initialMarking>
    </place>
13   <place id="gateOpenE">
      <initialMarking>
        <text>0</text>
      </initialMarking>
    </place>
18   <transition id="arriveUp">
      <name>
        <text>arriveUp</text>
      </name>
    </transition>
23   <transition id="arriveDown">
      <name>
        <text>arriveDown</text>
      </name>
    </transition>
28   <transition id="gotTicket">
      <name>
        <text>gotTicket</text>
      </name>
    </transition>
33   <arc id="a1" source="entranceFree" target="arriveUp">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
38   <arc id="a2" source="arriveUp" target="waitingTicket">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
43   <arc id="a3" source="waitingTicket" target="gotTicket">
      <inscription>
        <text>1</text>
      </inscription>
  </net>
</pnml>

```

```

48      </arc>
      <arc id="a4" source="gotTicket" target="gateOpenE">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
53      <arc id="a5" source="gateOpenE" target="arriveDown">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
58      <arc id="a6" source="arriveDown" target="entranceFree">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
63    </net>
  </pnml>

```

A linguagem PNML prevê a definição de parte da sintaxe de tipos de redes de Petri, nomeadamente das anotações possíveis nos lugares, transições, arcos e redes. Estas sintaxes são definidas num formato próprio em ficheiros com a extensão **pntd** e são denominadas *Petri Net Type Definitions* ou **PNTD**. Nos exemplos apresentados utiliza-se o PNTD para redes de Petri lugar-transição por estar disponível [s.a., 2004d] e ser suficiente para expressar as redes não marcadas.

Apesar da linguagem PNML o permitir, não é aqui utilizada nenhuma informação gráfica (por exemplo o elemento PNML **graphics**) por tal ser irrelevante para o que se pretende ilustrar.

Actualmente existem duas propostas baseadas em PNML — mais precisamente em **basic PNML** — para especificar modelos estruturados em vários módulos [Kindler e Weber, 2001]:

1. O **Structured PNML** em que cada rede é uma página e cada página pode referenciar outras páginas através da utilização de **lugares referência**, **transições referência**, ou ambos.
2. O **Modular PNML** em que cada rede é vista como um módulo que contém uma *interface* e uma *implementação*.

O PNML estruturado constitui uma forma simples de particionar uma rede de grandes dimensões em vários submodelos que se referenciam entre si: cada nó numa determinada página pode ser uma referência para outro nó noutra página. Dessa forma, temos *nós* e *nós referência*. O modelo resultante funde cada nó referência com um nó real resultando num único nó. O utilizador continua a ser obrigado a ter todo o modelo presente, ou seja, o modelo resultante das referências entre as várias páginas.

Embora traduzível para PNML estruturado, o PNML modular é bastante diferente dado que permite a criação de várias instâncias independentes de uma determinada rede, vista portanto como um módulo. Neste sentido, o PNML modular permite a reutilização de módulos em diferentes contextos, possivelmente ao mesmo tempo, através da utilização de várias instâncias. Para além disso, o utilizador não é obrigado a conhecer os contextos em que o módulo será utilizado, apenas a funcionalidade do módulo é relevante. Para tal, cada módulo especifica uma interface, constituída por *nós de importação*, que na realidade são referências para nós no exterior (em outros módulos), e *nós de exportação*, que são nós com os quais os nós de importação de outros módulos se podem fundir.

A definição de modelos na forma operacional apresentada no capítulo anterior constitui uma forma alternativa de compor modelos que também suporta, como casos particulares, os tipos de estruturação presentes no PNML estruturado e no PNML modular. Comparativamente com estes, apresenta três vantagens significativas:

1. Possibilidade de compor um número arbitrário de instâncias de forma compacta permitindo assim a construção de modelos de grandes dimensões de forma simples e rápida.
2. Suporte para as perspectivas descendente, ascendente e entrecortada.
3. Visualização facilitada graças à completa integração com os diagramas de adição.

Assim sendo, e dada a indiscutível importância da linguagem PNML, é também conveniente dispor de uma forma de interligar as definições operacionais com a linguagem PNML. A secção seguinte, apresenta uma proposta de uma linguagem que complementa a linguagem PNML com a possibilidade de definir, de forma operacional, modelos de redes de Petri. A linguagem denomina-se *Operational PNML* e tal como a linguagem PNML é baseada em XML.

4.2.2 A linguagem *Operational PNML*

Nesta secção apresenta-se a linguagem *Operational PNML* (OPNML) através de exemplos. Estes são parte dos exemplos apresentados no capítulo anterior, nomeadamente na secção 3.1.2. Tal permite uma mais fácil comparação com as notações já apresentadas, nomeadamente a linguagem *PNTEXT* e os diagramas de adição.

Instâncias de Redes e Vectors de Nós

Qualquer rede, previamente disponível em PNML, pode ser utilizada por uma especificação OPNML para, a partir dela, serem geradas uma ou mais instâncias de rede. Note-se que uma

especificação OPNML pode gerar uma ou várias especificações PNML, pelo que as especificações OPNML podem utilizar o resultado de outras especificações em OPNML.

Os índices associados aos identificadores de instâncias de redes são considerados parte do identificador, ou seja, não são especificados separadamente. Neste caso, tal como em PN_{TEXT} , os índices surgem sempre entre parêntesis rectos.

Conjuntos de Fusão

Em OPNML distinguem-se explicitamente, por meio de diferentes elementos XML, os conjuntos de fusão de lugares e os conjuntos de fusão de transições. Aos primeiros corresponde o elemento `placeFusion`, aos segundos o elemento `transitionFusion`. Cada nó de um conjunto de fusão é identificado através da indicação da instância de rede, da qual faz parte, e do identificador do nó. Os ficheiros PNML que contêm cada uma das redes dos nós fundidos são identificados por possuírem como nome o identificador da rede. Por exemplo, a rede **Enter** deve estar especificada em PNML no ficheiro **Enter.pnml**.

A Listagem 4.2 apresenta a parte da gramática da linguagem OPNML que especifica a sintaxe para definição de conjuntos de fusão. À imagem da linguagem PNML, a gramática da linguagem OPNML foi definida em Relax NG [s.a., 2003c]. Esta linguagem encontra-se actualmente no estágio final de normalização [s.a., 2003b].

Listagem 4.2: Sintaxe dos elementos `transitionFusion` e `placeFusion` na gramática da linguagem Operational PNML.

```

205 <a:documentation>#####</a:documentation>
    <a:documentation>##### Fusion Sets #####</a:documentation>
    <a:documentation>#####</a:documentation>
    <define name="transitionFusion.element">
        <element name="transitionFusion">
210     <ref name="transitionFusion.content"/>
        </element>
    </define>
    <define name="placeFusion.element">
        <element name="placeFusion">
215     <ref name="placeFusion.content"/>
        </element>
    </define>
    <define name="transitionFusion.content">
        <attribute name="nodeID">
220     <data type="token"/>
        </attribute>
        <oneOrMore>
            <ref name="transitionRef"/>
        </oneOrMore>
225 </define>
    <define name="placeFusion.content">
        <attribute name="nodeID">
            <data type="token"/>
        </attribute>
    </define>

```

```

230    <oneOrMore>
        <ref name="placeRef"/>
    </oneOrMore>
</define>

```

Os vectores de nós são especificados através de outro par de elementos: o `placeFusionVector` e o `transitionFusionVector`. Estes elementos admitem a especificação de um iterador e de valores mínimo e máximo (*vide* Listagem 4.3) que juntos definem quer a quantidade de instâncias, quer a sequência de valores possíveis para os índices das instâncias.

Listagem 4.3: Sintaxe dos elementos `placeFusionVector` e `transitionFusionVector` na gramática da linguagem Operational PNML.

```

<define name="placeFusionVector.element">
235    <element name="placeFusionVector">
        <attribute name="iterator">
            <data type="NMIOKEN"/>
        </attribute>
        <attribute name="first">
240            <data type="nonNegativeInteger"/>
        </attribute>
        <attribute name="last">
            <data type="nonNegativeInteger"/>
        </attribute>
245    <ref name="placeFusion.content"/>
    </element>
</define>
<define name="transitionFusionVector.element">
    <element name="transitionFusionVector">
250        <attribute name="iterator">
            <data type="NMIOKEN"/>
        </attribute>
        <attribute name="first">
            <data type="nonNegativeInteger"/>
255        </attribute>
        <attribute name="last">
            <data type="nonNegativeInteger"/>
        </attribute>
        <ref name="transitionFusion.content"/>
260    </element>
    </define>

```

A totalidade da gramática da linguagem OPNML pode ser consultada no Apêndice B. Seguidamente apresentam-se alguns dos exemplos da secção 3.1.2 expressos em OPNML. Conjuntamente, estes exemplos apresentam todas as características da linguagem.

Adição e Subtracção em OPNML

A definição de adições ou subtracções de redes em OPNML baseia-se nas seguintes características:

- Pelo menos uma rede tem de estar definida em basic PNML, ou seja, *em extensão*.
- Toda a definição de uma rede em OPNML gera uma definição expressa em basic PNML que pode ser utilizada como rede operando noutra definição de rede.
- As operações, que definem uma nova rede, operam sobre um conjunto especificado de instâncias de rede: as *redes operando*.
- Podem ser definidos vectores de redes, com base em redes já especificadas em PNML ou resultantes de especificações em OPNML. Estes vectores de instâncias podem então ser utilizados na definição, em OPNML, de novas redes.
- As instâncias de redes são identificadas tal como em PN_{TEXT} , ou seja, pelo identificador da rede original seguido do índice entre parêntesis rectos.
- Os nós em instâncias de redes são identificados pelo identificador da instância da rede e pelo identificador do nó original.
- Cada ficheiro OPNML pode definir várias operações sobre redes; estas operações podem utilizar as redes, expressas em basic PNML, resultantes de outras operações no mesmo ficheiro; tal significa que as operações num dado ficheiro OPNML se encontram ordenadas parcialmente.
- De entre as operações especificadas num ficheiro OPNML, apenas uma devolve a rede definida por esse ficheiro; tal é especificado atribuindo o valor **return** ao identificador de uma, e só uma, das redes definida em OPNML.

A Listagem 4.4 exemplifica a definição de uma rede em OPNML através da operação de adição. Corresponde à definição do modelo **ParkA**, já apresentada na pág. 53 (juntamente com o respectivo diagrama de adição na Fig. 3.3).

Listagem 4.4: Definição da rede **ParkA** (Fig. 3.2) em OPNML.

```

<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="ParkA"
    type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
4    <operations>
      <addition result="return">
        <net netID="Enter"/>
        <net netID="ParkingArea"/>
        <net netID="Leave"/>
9      <merge>
        <transitionFusion nodeID="in">
          <transition netID="Enter" nodeID="gotTicket"/>
          <transition netID="ParkingArea" nodeID="in"/>
        </transitionFusion>
14      <transitionFusion nodeID="out">
        <transition netID="ParkingArea" nodeID="out"/>
        <transition netID="Leave" nodeID="pay"/>
      </transitionFusion>
    </merge>
  </net>
</pnml>

```

```

19      </addition>
      </operations>
    </net>
  </pnml>

```

Os dois primeiros elementos são idênticos aos utilizados em PNML: o elemento mais externo (**pnml**) contém apenas uma especificação de um espaço de nomes¹ em XML (atributo **xmlns**). O elemento seguinte, **net**, contém um identificador (**id**) e a indicação de qual o PNTD utilizado pelas redes operando — o mesmo a considerar para as redes resultantes da especificação. O restante conteúdo é muito diferente de um ficheiro PNML. Em particular, identificam-se as seguintes diferenças:

- O interior do elemento **net** contém única e exclusivamente um elemento denominado **operations**. Este contém conjuntos de elementos **addition**, **subtraction** ou ambos. Na definição OPNML do modelo **ParkA** este contém uma única operação de adição. Esta é especificada pelo elemento **addition** o qual contém uma lista de elementos **net**, que identificam as redes operando, e um elemento **merge**, que contém a definição dos conjuntos de fusão entre as várias redes operando.
- Todas as operações têm um atributo denominado **result**. Este atributo especifica o identificador da rede resultante. No interior do elemento **operations** apenas uma operação deve atribuir o valor **return** ao seu atributo **result**. Essa operação define a rede. A definição da rede **ParkA** apenas contém uma operação pelo que o valor do respectivo atributo é **return**.

O modelo **ParkExit**, definido na pág. 55, é um exemplo em que se utiliza a subtração de redes. A Listagem 4.5 apresenta o código OPNML correspondente.

Listagem 4.5: Definição da rede **ParkExit** a partir de uma rede **ParkA** já definida em OPNML e de uma rede **Enter** em PNML.

```

<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="ParkExit"
3    type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
    <operations>
      <subtraction result="return">
        <net netID="ParkA"/>
        <net netID="Enter" sign="minus"/>
8      <merge>
        <transitionFusion nodeID="in">
          <transition netID="ParkA" nodeID="in"/>
          <transition netID="Enter" nodeID="gotTicket"/>
        </transitionFusion>
13    </merge>
    <removal>
      <transitionFusion nodeID="n1">
        <transition netID="ParkA" nodeID="entranceFree"/>

```

¹Em inglês: *namespace*.

```

18      <transition netID="Enter" nodeID="entranceFree"/>
    </transitionFusion>
    <transitionFusion nodeID="n2">
      <transition netID="ParkA" nodeID="arriveUp"/>
      <transition netID="Enter" nodeID="arriveUp"/>
    </transitionFusion>
23    <transitionFusion nodeID="n3">
      <transition netID="ParkA" nodeID="arriveDown"/>
      <transition netID="Enter" nodeID="arriveDown"/>
    </transitionFusion>
    <transitionFusion nodeID="n4">
28      <transition netID="ParkA" nodeID="waitingTicket"/>
      <transition netID="Enter" nodeID="waitingTicket"/>
    </transitionFusion>
    <transitionFusion nodeID="n5">
      <transition netID="ParkA" nodeID="gateOpenE"/>
33      <transition netID="Enter" nodeID="gateOpenE"/>
    </transitionFusion>
    </removal>
  </subtraction>
</operations>
38 </net>
</pnml>

```

Comparativamente com o código OPNML para uma adição, na subtracção utiliza-se um elemento **subtraction** que para além de todas as características do elemento **addition** contém um elemento **removal**. Este corresponde ao colapso de remoção da subtracção. O modelo que é subtraído (o *modelo substractivo* ou *modelo negativo*) é assinalado atribuindo o valor **minus** ao atributo **sign**. Neste exemplo, tal sucede com o módulo **Enter**.

Finalmente, apresenta-se um exemplo que utiliza vectores de instâncias de redes e nós indexados. A Listagem 4.6 apresenta o código OPNML para a definição do modelo **ParkAPassage2** já definido na pág. 60. Note-se a utilização de elementos **placeFusion**.

Listagem 4.6: Definição da rede **ParkAPassage2** a partir de uma rede **ParkA** já definida em OPNML e de duas instâncias da rede **Passage**, também definida em PNML.

```

1 <pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="ParkAPassage2"
    type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
    <operations>
      <addition result="return">
6        <net netID="ParkA"/>
        <netVector netID="Passage" first="1" last="2"/>
        <merge>
          <placeFusion nodeID="free">
            <place netID="ParkA" nodeID="free"/>
11          <place netID="Passage[1]" nodeID="freeZone[2]"/>
          <place netID="Passage[2]" nodeID="freeZone[1]"/>
        </placeFusion>
        <placeFusion nodeID="occupied">
          <place netID="ParkA" nodeID="occupied"/>
16          <place netID="Passage[1]" nodeID="carsInZone[2]"/>
          <place netID="Passage[2]" nodeID="carsInZone[1]"/>
        </placeFusion>
      </addition>
    </operations>
  </net>
</pnml>

```

```

21      </merge>
      </addition>
      </operations>
    </net>
  </pnml>

```

A secção seguinte apresenta um protótipo capaz de transformar modelos em OPNML num ou mais modelos em PNML.

4.3 Um Tradutor de OPNML para PNML

Como prova de conceito, foi implementada uma ferramenta computacional, denominada `opnml2pnml`, capaz de traduzir definições de modelos em OPNML para uma ou várias definições em PNML. Mais especificamente, a ferramenta efectua adições e subtracções sobre redes anotadas e como tal implementa as extensões definidas na Secção 3.5 (pág. 79).

O tradutor foi implementado na linguagem de programação Ruby [Ruby Home Page, 2004] e é constituído por um módulo principal e pelos vários módulos para suporte a estruturas de transformação (*vide* Def. 3.28) na linguagem de inscrições — neste caso também a linguagem Ruby [Ruby Home Page, 2004]. Como prova de conceito, foram implementados quatro módulos que definem, cada um, uma estrutura de transformação: (1) uma para redes de Petri lugar-transição (`ptNetb.ts.rb`); (2) uma para fusão de canais síncronos num tipo simplificado de redes coloridas (`colorSimple.ts.rb`); (3) outra para redes lugar-transição estendidas com arcos de teste (`ptTest.ts.rb`) e (4) uma para uma classe de redes de Petri não-autónomas apresentada no Apêndice C (`ptio.ts.rb`). O módulo `colorSimple` serve apenas para demonstrar a fusão de transições apresentada nas Figs. 2.6 e 2.7 a qual ilustra a semântica dos canais síncronos, pelo que não corresponde a nenhuma classe de redes específica e documentada. A Fig. 4.1 ilustra os cinco módulos realizados e as respectivas dependências sob a forma de um diagrama de componentes da linguagem UML [OMG, 2003]: o principal, correspondente ao ficheiro `opnml2pnml.rb`; e as quatro estruturas de transformação, `ptNetb.ts.rb`, `colorSimple.ts.rb`, `ptTest.ts.rb` e `ptio.ts.rb`. O módulo principal lê o ficheiro com a especificação OPNML e, com base no atributo `type` do elemento `net`, determina qual a estrutura de transformação que deve ser utilizada para efectuar as operações especificadas.

As estruturas de transformação devem definir as funções de transformação necessárias para as operações e modelos que se pretendem manipular. Conforme referido na Def. 3.28 na pág. 84, uma estrutura de transformação completa deve definir as seguintes funções:

- Uma função de transformação para cada anotação da rede. Estas funções constituem o conjunto FT_{TAR}^C na Def. 3.28 e são definidas como adições à classe `NetFusion` que faz parte do módulo principal (*vide* Listagem 4.7).

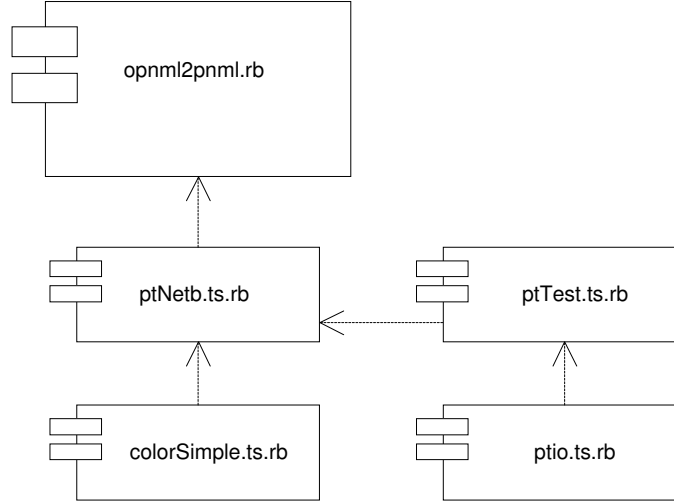


Figura 4.1: Estrutura do tradutor `opnml2pnml`.

- Uma função de transformação para cada anotação de nó. Estas funções constituem os conjuntos $FT_{TAL}^{\mathcal{L}}$ e $FT_{TAF}^{\mathcal{L}}$ na Def. 3.28 e são definidas como adições à classe `NodeFusion` que faz parte do módulo principal (*vide* Listagem 4.8).
- Uma função para adicionar arcos e outra para subtrair arcos entre cada par de interface, incluindo as respectivas anotações. Estas funções constituem o conjunto $FT_{TAF}^{\mathcal{L}}$ na Def. 3.28 e são definidas como adições à classe `IPairsFusion` que faz parte do módulo principal (*vide* Listagem 4.9).

Listagem 4.7: Classe `NetFusion` do módulo principal (`opnml2pnml.rb`) à qual são adicionadas as funções de transformação, para as anotações nas redes.

```
# Defines the transformations to be applied to
598 # the several fused net labels.
# Each label "lab" can have a method named "add_lab" and
# another named "sub_lab".
class NetFusion
  def initialize(label, netLevelLabelList)
603     @label = label
        @labels = netLevelLabelList
  end
  #calls method with @label name
  def addsub(as)
608     return eval(as + @label)
  end
end
```

Listagem 4.8: Classe `NodeFusion` do módulo principal (`opnml2pnml.rb`) à qual são adicionadas as funções de transformação para as anotações nos nós.

```
# Defines the transformations to be applied to
616 # the several fused labels.
# Each label "lab" can have a method named "add_lab" and
# another named "sub_lab". The former is applied to interface places
# (merge part) in net addition. The latter is applied to interface places
```

```

# (merge part) in net subtraction.
621 # NetFusion objects are created in methods Addition#collapse and
# Subtraction#collapse .
class NodeFusion
  def initialize(label, fe)
    @label = label
626    @labels = Array.new
    @fe = fe
    @nodesToFuse = fe.toFuse
    @nodesToFuse.each_value do |nodeData|
      labelXML = nodeData.xml.elements[label]
631    @labels.push(Label.new(labelXML, nodeData.sign)) if labelXML
    end
  end
end

  def addsub(as)
636    return eval(as + @label)
  end
end # end class NodeFusion

```

Listagem 4.9: Classe `IPairsFusion` à qual são adicionadas as funções de transformação para os arcos entre nós interface.

```

# Defines the transformations to be applied to
661 # the arcs between interface nodes (the IPair objects).
# A method named "add" and another named "sub_lab" must be defined.
# The former is applied to interface pairs
# (merge part) in net addition. The latter is applied to interface pairs
# (merge part) in net subtraction.
666 # IPairsFusion objects are created in methods Addition#processIPairs and
# Subtraction#processIPairs.
class IPairsFusion
  def initialize(iPairsList, totalNet)
    @iPairsList = iPairsList
671    @iPairsList.list.each do |ip|
      puts 'Interface Pair (result place, result transition) = (' +
        ip.placeFE.final.to_s + ', ' + ip.transitionFE.final.to_s + ')'
    end
    @net = totalNet.elements["pnml"].elements["net"]
676  end

  def deleteArc(arc)
    #auxiliary function used in add() and sub() implementations
    @net.elements.each do |e|
681      @net.delete_element(e) if (e.name() == "arc") && (e.attributes['id'] == arc.id)
    end
  end
end # class IPairsFusion

```

O código relativo aos cinco módulos na Fig. 4.1 consta da Secção D.1 a qual se inicia na pág. 215.

É importante sublinhar que a implementação das estruturas de transformação é sempre necessária, quer se utilize uma especificação em OPNML quer uma especificação em PN_{TEXT} .

Em resumo, as operações sobre redes definidas no Capítulo 3 são sempre definidas em dois níveis:

1. A operação gráfica a nível de redes não marcadas e que afecta a estrutura da rede.
2. A modificação das anotações das redes, nós e arcos.

O primeiro nível pode ser especificado em *PNTEXT* ou em OPNML. O segundo nível é especificado por uma linguagem de programação que implemente uma estrutura de transformação. Conforme já apresentado, o protótipo desenvolvido suporta especificações em OPNML e utiliza estruturas de transformação definidas em Ruby.

No Apêndice D é apresentado o código fonte do tradutor. O protótipo foi testado nos exemplos que se apresentaram neste capítulo e que ilustram as várias possibilidades da linguagem OPNML. Na Secção D.2 apresentam-se os seguintes exemplos adicionais em que o tradutor foi testado, testes mais extensos justificar-se-ão quando suportados por um ambiente de desenvolvimento como, por o exemplo, o descrito no Apêndice C:

- Tradução para um modelo equivalente sem canal síncrono. Para tal especificaram-se os modelos **Producer** e **Consumer**, da Secção "Adição de Canais Síncronos" na pág. 86, e efectuou-se a sua tradução para o modelo equivalente sem o canal síncrono. A Secção D.2.1, na pág. 234, contém as especificações em PNML dos modelos **Producer** e **Consumer**, bem como, a especificação OPNML que especifica a tradução para o modelo equivalente sem canais síncronos. Apresenta-se também o modelo resultante em PNML. Nesta tradução, utilizou-se a estrutura de transformação **ColorSimple** que se apresenta na Listagem D.3 da pág. 230.
- Adição de um árbitro numa rede com arcos de teste. Para tal especificaram-se os modelos **Arbiter** e **Par**, na pág. 89 da Secção 3.5.3, e efectuou-se a respectiva operação de adição. A Secção D.2.2, na pág. D.2.2, contém as especificações em PNML dos modelos **Arbiter** e **Par**, bem como, a especificação OPNML que especifica a adição de ambos. Apresenta-se também o modelo resultante em PNML e uma sua representação gráfica gerada automaticamente. Na tradução, utilizou-se a estrutura de transformação **ptTest** que se apresenta na Listagem D.4 da pág. 231.
- Adição de modelos de redes de uma nova classe de redes de Petri não-autónomas. Esta classe é definida no Apêndice C. Para além da estrutura de transformação respectiva (**ptio.ts.rb**), apresentada na Listagem D.5 da pág. 232, foi definida uma gramática para esta nova classe de redes de Petri. Esta gramática é compatível com a linguagem PNML e constitui o Apêndice E. Como teste, definiu-se a transformação das redes na Fig. C.2 da pág. 212. O resultado é a rede na Fig. C.3 da pág. 213. A Secção D.2.3 apresenta as especificações em PNML das redes adicionadas (**EnterIO**, **ParkingAreaIO** e **LeaveIO**), a especificação da sua adição e o modelo, em PNML, resultante. Para este apresenta-se também uma representação gráfica gerada automaticamente.

É intenção do autor manter na Internet a informação mais relevante relativa à linguagem OPNML [Barros e Gomes, 2004e]. Em particular foi já iniciado o desenvolvimento de um projecto que irá utilizar a linguagem OPNML e os conceitos associados, nomeadamente as estruturas de transformação associadas e sua associação a classes de redes. Nesse sentido, no Apêndice C, que apresenta a nova classe de redes de Petri não-autónomas referida, apresenta-se igualmente uma proposta preliminar de um ambiente de desenvolvimento baseado em redes de Petri que deverá dar suporte às linguagens PNML, OPNML e PN_{TEXT} .

Parte II

Redes de Petri no Desenvolvimento Orientado pelos Objectos

Capítulo 5

Redes de Petri e Objectos

Estou agora convencido que a ocultação de informação, hoje em dia frequentemente incorporada na programação orientada pelos objectos, constitui a única forma de aumentar o nível do desenho de software

– Frederick P. Brooks, Jr., 1995

Neste capítulo propõe-se um conjunto de idiomas que permitem a utilização de redes de Petri de alto-nível no desenho orientado pelos objectos. Para tal define-se uma nova classe de redes de Petri fortemente baseadas nas redes de Petri coloridas e utilizando também a adição de redes definida no Capítulo 3. Estas redes, denominadas redes de Petri coloridas componíveis, são formalmente equivalentes a uma única rede de Petri colorida. Apresenta-se também um conjunto de idiomas para a modelação dos principais conceitos do desenvolvimento orientado pelos objectos.

Este capítulo e o seguinte debruçam-se sobre a aplicabilidade das redes de Petri no contexto já bem conhecido do desenvolvimento orientado pelos objectos. No artigo original sobre estadogramas [Harel, 1987], Harel reconhece que as redes de Petri são "uma das mais bem conhecidas e melhor compreendidas soluções" para a descrição do comportamento de sistemas reactivos complexos. No entanto, aponta como defeito a ausência de um mecanismo satisfatório de decomposição hierárquica. No Capítulo 2 apresentaram-se já os vários mecanismos para a decomposição hierárquica de redes de Petri, e os capítulos 3 e 4 mostraram que é possível definir um mecanismo genérico, porque independente do tipo de redes de Petri, para a composição de redes. Este permite a adição de estruturação a qualquer tipo de rede de Petri. Nesse sentido, é um mecanismo particularmente útil aquando do desenvolvimento de tipos específicos de redes de Petri: os respectivos modelos tornam-se modulares, de uma forma genérica e independente

do tipo de redes em causa. Juntamente com a desejável e expectável utilização generalizada da linguagem PNML, tal permite uma maior uniformidade sintáctica nas definições de redes de Petri. Também permite uma maior uniformidade na definição da semântica das composições, dado que estas podem ser definidas num quadro genérico constituído por adições e subtracções definidas de forma também ela estruturada.

Para além de um melhor suporte para a estruturação de redes de Petri, outra forma de aumentar a aplicabilidade das redes de Petri consiste em aproximar os tipos já existentes das reais necessidades dos engenheiros. Este capítulo apresenta uma contribuição neste sentido. Neste capítulo e no seguinte, veremos outra forma pela qual as redes de Petri se podem tornar mais aplicáveis. Esta é constituída por duas ideias fundamentais:

1. Utilização de uma classe de redes de Petri de alto-nível bem conhecida, bem estudada e com boas ferramentas computacionais disponíveis.
2. Utilização de uma metodologia orientada pelos objectos.

A classe de redes de Petri de alto-nível mais conhecida e com melhores ferramentas de suporte é claramente a das redes de Petri coloridas de Jensen [Jensen, 1997c,a,b]. Para estas redes existe uma ferramenta de referência há mais de catorze anos, o Design-CPN [s.a., 2004c]. Nos últimos anos surgiu uma nova ferramenta computacional para redes de Petri coloridas, também com origem no trabalho de Jensen e do seu grupo na Universidade de Aarhus, em Aarhus na Dinamarca: a CPN Tools [s.a., 2004b]. As redes de Petri coloridas têm uma outra característica que as distingue de numerosas outras classes de redes de Petri que constantemente vão surgindo na literatura: são indiscutivelmente redes de Petri pois podem ser traduzidas para redes de Petri lugar-transição (*vide* [Jensen, 1997a]). Na verdade, como se pode facilmente verificar pela literatura publicada e também através da consulta da base de dados sobre ferramentas de redes de Petri [s.a., 2005d], a quantidade de linguagens apresentadas como redes de Petri continua a crescer¹.

A utilização de uma metodologia orientada pelos objectos é aqui considerada de grande importância para a efectiva e frequente utilização das redes de Petri. Com efeito, sem uma ligação bem definida entre redes de Petri e as linguagens e conceitos utilizados no desenvolvimento orientado pelos objectos, será muito difícil tornar as redes de Petri conhecidas e utilizadas fora da comunidade, ainda restrita e relativamente pouco conhecida, dos utilizadores de redes de Petri.

Esta constatação tem sido frequente, especialmente nos últimos quinze anos. Como consequência deste facto, vários trabalhos têm procurado juntar num único formalismo, as redes de Petri e

¹Tal facto constituiu, muito provavelmente, a motivação para que Desel e Juhás tenham sentido a necessidade de escrever um artigo no qual discutem o que é uma rede de Petri [Jörg Desel, 2001]. Esse artigo é o primeiro de um livro sobre a questão da unificação das várias classes de redes de Petri [Ehrig et al., 2001].

os conceitos do desenvolvimento orientado pelos objectos. Existe já um *survey* que no essencial se mantém actualizado, constituído por uma colecção de artigos sobre as principais abordagens à ligação entre redes de Petri e desenvolvimento orientado pelos objectos [Agha et al., 2001]. As abordagens que é possível encontrar nesse *survey*, e noutros artigos na literatura, podem dividir-se em dois grupos:

1. Linguagens baseadas em redes de Petri que adicionam novos conceitos, sintaxes e semânticas às redes de Petri, ainda que possibilitando uma tradução (teórica mas pouco prática) para redes de Petri coloridas.
2. Linguagens baseadas em redes de Petri que adicionam novos conceitos, sintaxes e semânticas às redes de Petri sem a preocupação de manter uma ligação clara quer com as redes de Petri coloridas, quer com as redes de Petri de baixo-nível. Estas abordagens afastam-se, por vezes de forma muito significativa, das sintaxes e semânticas características das redes de Petri.

No primeiro grupo, os trabalhos que se mantêm traduzíveis para as redes de Petri coloridas, sublinham-se as *object Petri nets* de Lakos [Lakos, 1995], e as *object coloured Petri nets* de Maier e Moldt [Maier e Moldt, 2001].

No segundo grupo, destacam-se aqui dois trabalhos por se encontrarem mais divulgados: (1) o formalismo *CO-OPN/2* de Biberstein et al. [Biberstein et al., 2001], baseado no trabalho inicial de Buchs e Guelfi [Buchs e Guelfi, 1991] e que se apresenta como uma solução para simplificar todo o ciclo de desenvolvimento de software; (2) as *reference nets* de Kummer [Kummer, 2002], suportadas pela ferramenta RENEW[Kummer et al., 2004b,a] que estendem as redes de Petri coloridas com criação dinâmica de redes e vários tipos de arcos. Ainda neste grupo, existem trabalhos que visam a obtenção de modelos para domínios de aplicação específicos (e.g. sistemas embutidos [Machado e Fernandes, 2001]), ou que enfatizam a combinação de uma linguagem de programação com conceitos oriundos das redes de Petri (e.g. [Köster et al., 2001]).

Contrariamente a ambas as atitudes, propõe-se aqui uma terceira via que pode ser vista como uma radicalização da motivação por trás do primeiro grupo de trabalhos: **utilizar uma variante das redes de Petri coloridas no desenvolvimento orientado pelos objectos, minimizando a adição de novas sintaxes e mantendo a possibilidade de traduzir esses modelos para modelos em redes de Petri coloridas "puras", de forma simples e intuitiva**. Para tal, em lugar de extensões ou sintaxes e semânticas relativamente sofisticadas, propõe-se um conjunto de idiomas e padrões de desenho que suportados pela adição de redes (apresentada no Capítulo 3) e por uma variante dos canais síncronos de Christensen e Hansen [Christensen e Hansen, 1992] permitem a utilização de redes de Petri coloridas no desenho de modelos orientados pelos objectos. Ou seja, a variante proposta para as redes de Petri coloridas, denominada **redes de Petri coloridas componíveis** é suficientemente próxima das redes de Petri coloridas para poder ainda ser vista como um conjunto de redes coloridas ligadas por me-

canismos de composição gráfica. Neste sentido, podem e devem ser vistas como um passo mais no sentido indicado pelas redes de Petri coloridas hierárquicas: uma conveniência gráfica para facilitar a construção de modelos com redes de Petri coloridas.

No entanto, contrariamente às rede de Petri coloridas e hierárquicas, que favorecem um paradigma correspondente ao da programação estruturada, as redes de Petri coloridas componíveis têm por objectivo principal o encurtar da separação entre as redes de Petri e as metodologias orientadas pelos objectos, em particular as baseadas na *Unified Modeling Language* (UML) [OMG, 2003]. Mais especificamente, as redes de Petri coloridas componíveis constituem uma alternativa à modelação do comportamento de objectos por outras linguagens gráficas presentes na UML, em especial os estadogramas. Tal é conseguido sem prejuízo das características fundamentais das redes de Petri coloridas. Por outras palavras, procurou-se minimizar as modificações à sintaxe das redes de Petri coloridas, quer na sua vertente textual, quer na sua vertente gráfica. Procurou-se também manter a semântica tão próxima quanto possível da das redes de Petri coloridas. Assim, existe uma diferença fundamental relativamente a outras propostas para modelação de conceitos do desenvolvimento orientado pelos objectos utilizando redes de Petri as quais se afastam significativamente, ou em termos sintácticos (e.g. [Lakos, 1995]) ou em termos semânticos (e.g. [Kummer et al., 2004b,a]), das redes de Petri coloridas. A secção seguinte define as redes de Petri coloridas componíveis.

5.1 Redes de Petri Coloridas Componíveis

A maior parte do que esperamos ganhar com a programação orientada pelos objectos deriva na verdade do primeiro passo, o encapsulamento de módulos, mais a ideia de bibliotecas pré-construídas de módulos ou classes que são desenhadas e testadas para serem reutilizadas.

– Frederick P. Brooks, Jr., 1995

Nesta secção define-se uma nova classe de redes de Petri denominada **redes de Petri coloridas componíveis**. Estas redes diferem das propostas conhecidas para "redes de Petri com objectos", nos seguintes pontos:

- Evitam toda e qualquer modificação à notação gráfica das redes de Petri coloridas.
- Implicam adições mínimas às notações textuais das redes de Petri coloridas.
- Utilizam unicamente conceitos bem conhecidos no âmbito das redes de Petri coloridas e aplicam-nos na modelação dos principais conceitos do desenvolvimento orientado pelos objectos.

- São traduzíveis para redes de Petri coloridas.

No que respeita à semântica, mostrar-se-á que as redes de Petri coloridas componíveis constituem representações de mais alto-nível para redes de Petri coloridas equivalentes e, por consequência, também para redes lugar-transição.

É interessante notar que neste aspecto, as redes de Petri coloridas componíveis têm claras semelhanças com as redes de Petri coloridas Hierárquicas [Jensen, 1997c,a,b]: uma rede de Petri colorida hierárquica é constituída por um conjunto de redes de Petri coloridas (páginas) não hierárquicas interligadas e que tomadas como um todo são equivalente a uma única rede de Petri colorida, a uma única página. No entanto, conforme já referido, enquanto as redes de Petri coloridas hierárquicas se baseiam no paradigma da programação estruturada, as redes de Petri coloridas componíveis baseiam-se no paradigma da programação orientada pelos objectos. Por outro lado, dado que uma rede de Petri colorida hierárquica é também uma rede de Petri colorida, podemos utilizar redes de Petri coloridas hierárquicas como páginas das redes de Petri coloridas componíveis. Tal oferece um complemento para a decomposição de operações mais complexas, de forma idêntica à utilizada na programação orientada pelos objectos quando se decompõe um método em métodos auxiliares.

As modificações mínimas, relativamente à sintaxe e semântica das redes de Petri coloridas, sugerem que os mecanismos de estruturação nas redes de Petri coloridas componíveis devem ser baseados nos conceitos de fusão de transições e fusão de lugares. O primeiro pode ser utilizado na modelação da invocação de métodos (ou troca de mensagens síncronas) e o segundo permite modelar comunicações assíncronas.

A fusão de nós é também utilizada para modelar a composição de classes e módulos. Já a classificação, ou seja, a abstracção para um conjunto de instâncias que partilham uma estrutura de dados e um conjunto de possíveis comportamentos, é suportada por marcas coloridas. Estas representam cada um dos objectos através de tuplos contendo um identificador da classe (tipicamente implícito), um identificador do objecto (instância) e uma estrutura de dados opcional correspondente aos atributos de cada objecto.

5.1.1 Definição das Redes de Petri Componíveis

Em primeiro lugar, e antes da definição das redes de Petri coloridas componíveis, define-se um sistema baseados em objectos. No que respeita à nomenclatura e conceitos utilizados, a definição que se apresenta é baseada na linguagem UML. Dessa forma pretendeu-se utilizar uma referência bem conhecida procurando minimizar possíveis ambiguidades nos nomes utilizados.

O sistema de objectos é definido como um conjunto de classes e um conjunto de pedidos². Os

²Em inglês: *requests*.

pedidos podem corresponder a "pedidos a classes" ou a "pedidos a objectos" que são satisfeitos, respectivamente, por **operações de classe** ou **operações de objecto**. Os pedidos a objectos são identificados pelo facto de especificarem um número de instância (ou de objecto), o qual não é necessário nos pedidos a classes. Assume-se a existência dos pedidos *create* e *destroy* para a criação e destruição dinâmica de objectos. O pedido *create* é uma operação de classe. O pedido *destroy* pode também ser visto como uma operação de classe.

Definição 5.1 (Sistema de Objectos): *Um sistema de objectos é um tuplo $SO = (CS, CP, V, CONST, class, attrib, Obj, \mathcal{E}, qp)$ que satisfaz os seguintes requisitos:*

1. $CS = \{CS_1, CS_2, \dots, CS_n\}$ é um conjunto finito de **classes de sistema**.
2. $CP = \{CP_1, CP_2, \dots, CP_m\}$ é um conjunto finito de **classes pedido**, tal que $CS \cap CP = \emptyset$.
3. V é um conjunto de variáveis.
4. $CONST$ é um conjunto de constantes.
5. $class$ é uma função que aplica variáveis e constantes em classes de sistema: $class : (V \cup CONST) \rightarrow CS$.
6. $attrib$ é uma função atributo que aplica classes em tuplos de variáveis e constantes: $attrib : (CS \cup CP) \rightarrow (V \cup CONST)^*$. Utiliza-se também $attrib_i(c)$ para o elemento na posição i do tuplo resultante.
7. $Obj \subseteq CS \times \mathbb{N}_0 \times (V \cup CONST)^*$ é um conjunto de objectos, tal que $\forall (c, i, data) \in Obj, attrib(c) = data$.
8. $\{create, destroy\} \subseteq CP \wedge attrib(create) = attrib(destroy) = \emptyset$.
9. $\mathcal{E} \subseteq \{SEND, RECV\} \times CS \times CP \times (V \cup CONST)^*$ é um conjunto finito de eventos para os quais também se utilizarão as seguintes notações: dado um $e = (sr, c, ped, lp) \in \mathcal{E}$, $attrib(e) = attrib(ped)$, $\forall p_i \in lp, class(p_i) = class(attrib_i(ped))$ e $class(e) = c$.
10. qp é uma função qualificadora de parâmetros que aplica atributos de eventos num de três qualificadores: $qp : AE \rightarrow \{IN, OUT, INOUT\}$, em que $AE = \{p \in lp \mid \exists (sr, c, ped, lp) \in \mathcal{E}\}$.

Note-se a distinção entre **emissores de pedidos**³ e **receptores de pedidos**⁴ através dos qualificadores *SEND* e *RECV* para eventos emissores e receptores, respectivamente. Nesse sentido, definem-se $\mathcal{E}^S = \{(SEND, c, ped, lp) \in \mathcal{E}\}$ e $\mathcal{E}^R = \{(RECV, c, ped, lp) \in \mathcal{E}\}$.

As especificações de eventos omitem o número da instância. Tal implica que, por omissão, os eventos especifiquem pedidos a classes (*class level requests*). Caso se pretenda especificar um pedido a uma instância específica, o número de instância deverá fazer parte da lista de atributos do pedido (os seus parâmetros). Como regra, utilizar-se-á o primeiro parâmetro. Em alternativa, poder-se-á utilizar uma notação específica em que a instância surge separada dos parâmetros, como por exemplo na linguagem Java.

³Em inglês: *request senders*.

⁴Em inglês: *request receivers*.

Também se define a função *attrib* sobre eventos. Para um qualquer evento $e = (sr, c, ped, lp)$, considera-se $attrib_i(e) = attrib_i(lp)$. Define-se também o conceito de **eventos compatíveis**:

Definição 5.2 (Eventos Compatíveis): *Dois eventos $e^S = (SEND, c^S, ped^S, lp^S) \in \mathcal{E}^S$ e $e^R = (RECV, c^R, ped^R, lp^R) \in \mathcal{E}^R$ dizem-se compatíveis, e denotam-se $e^S \underset{ped}{=} e^R$, se e só se verificarem as seguintes condições:*

- $ped^S = ped^R$
- $|lp^S| = |lp^R|$
- $\forall 1 \leq i \leq |lp^S|, class(attrib_i(e^S)) = class(attrib_i(e^R)) \wedge attrib_i(e^R) \notin CONST$

Para um pedido $ped \in \mathcal{CP}$ também se utilizará $attrib_{IN}(ped) = \{p \in attrib(ped) \mid qp(p) = IN\}$, bem como $attrib_{OUT}(ped)$ e $attrib_{INOUT}(ped)$ com as óbvias e análogas semânticas.

Diferentemente das redes de Petri lugar-transição, as redes de Petri coloridas suportam tipos de dados estruturados como marcas. Estes dados podem ser manipulados por expressões associadas aos arcos. O disparo das transições depende da existência de valores específicos de dados nos lugares. Conforme já apresentado no Capítulo 2, o suporte para tipos de dados estruturados captura simetrias estruturais no modelo. Por um lado, e comparativamente às redes de Petri lugar-transição, este facto permite uma redução, tipicamente muito significativa e por vezes drástica, da dimensão do modelo, nomeadamente da quantidade de nós e arcos. Por outro lado, oferece ao utilizador a possibilidade de distribuir a complexidade do modelo entre a parte gráfica e as anotações da rede, nomeadamente as cores, os valores das marcas, as expressões e funções.

Seguidamente apresenta-se a definição de uma rede de Petri colorida tal como proposta por Jensen em [Jensen, 1997c,a,b], onde se encontra uma explicação mais detalhada da definição que se segue. Para os objectivos desta exposição, basta ter presente que a definição assume a existência de uma linguagem de inscrições (no caso da ferramenta CPN Tools, trata-se da linguagem de programação Standard ML, embora ligeiramente adaptada). A linguagem de inscrições deve permitir a especificação de tipos de variável, expressões algébricas, variáveis e funções. A definição que se segue assume também a existência das funções *Type* e *Var*. A primeira devolve o tipo da variável, função ou expressão; a segunda retorna o conjunto de variáveis presentes numa expressão, numa função ou numa lista de atributos (ou parâmetros). O subscrito $_{MS}$ denota multiconjuntos (conjuntos que podem conter várias ocorrências de um mesmo elemento)⁵. Finalmente \mathbb{B} denota o conjunto de valores booleanos: $\mathbb{B} = \{verdade, falso\}$.

Definição 5.3 (Rede de Petri Colorida): *Uma rede de Petri colorida é um tuplo $R = (\Sigma, L, T, A, N, C, G, E, I)$ que satisfaz os seguintes requisitos:*

1. Σ é um conjunto finito de tipos não vazios denominados **conjuntos de cores**⁶.

⁵De *multiset* em inglês.

⁶Em inglês: *colour sets*.

2. L é um conjunto finito de lugares.
3. T é um conjunto finito de transições.
4. A é um conjunto de arcos tal que $(L \cap T) = (A \cap L) = (A \cap T) = \emptyset$.
5. N é uma função nó que aplica A em $((L \times T) \cup (T \times L))$.
6. C é uma função de cor que aplica cada lugar $l \in L$ num conjunto de cores de Σ .
7. G é uma função de guarda. Aplica T em expressões na linguagem de inscrições tal que

$$\forall t \in T, Type(G(t)) = \mathbb{B} \wedge Type(Var(G(t))) \subseteq \Sigma$$

8. E é uma função de expressões de arcos. Aplica A em expressões tal que: $\forall a \in A, Type(E(a)) = C(l(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma$ onde $l(a)$ é o lugar conectado a a ⁷.
9. I é uma função de inicialização que aplica cada lugar $l \in L$ num multiconjunto em $C(l)$.

A cada arco está associada uma expressão que pode conter variáveis livres. O tipo de cada uma destas variáveis é um conjunto de cores pertencente a Σ . No disparo de uma transição t , todas as variáveis nos respectivos arcos de entrada são vinculadas de acordo com os valores das marcas (cores) nos lugares de entrada da transição.

Seguidamente, classificam-se as variáveis de transição:

Definição 5.4 (Variáveis de Transição): *Seja $R = (\Sigma, L, T, A, N, C, G, E, I)$ uma rede de Petri Colorida. Para cada transição $t \in T$ utilizar-se-ão as seguintes definições:*

1. Cada variável que ocorra em pelo menos uma expressão de um arco de entrada é uma variável de entrada da transição: $VarIn(t) = \{v \mid \exists l \in L, \exists a \in A, N(a) = (l, t) \wedge v \in Var(E(a))\}$.
2. Cada variável que ocorra em pelo menos uma expressão de um arco de saída é uma variável de saída da transição: $VarOut(t) = \{v \mid \exists l \in L, \exists a \in A, N(a) = (t, l) \wedge v \in Var(E(a))\}$.
3. Cada variável que ocorra em pelo menos uma expressão de arco de entrada ou de saída é uma variável da respectiva transição t : $Var(t) = VarIn(t) \cup VarOut(t)$.

Definem-se também *vínculo de uma variável de transição* e *vínculo de uma transição*.

Definição 5.5 (Vínculo de uma variável de transição): *Dada uma transição t , um vínculo⁸ v de uma variável de transição $var \in Var(t)$, pertencente a um domínio D , é uma função que atribui um valor val do domínio D à variável var : $v : Var(t) \rightarrow D$. É especificado pela sintaxe: $v = \langle var, val \rangle$ em que $(var, val) \in Var(t) \times D$.*

Definição 5.6 (Vínculo de transição): *Um vínculo de transição vt é um conjunto constituído única e exclusivamente por um vínculo para cada variável da transição tal que $G(t) \langle vt \rangle = \text{verdade}$.*

⁷Também se utilizará $t(a)$ para a transição conectada ao arco a .

⁸Em inglês: *binding*.

Dado um sistema de objectos, cada classe é definida por uma rede de Petri colorida na qual cada transição pode ter eventos associados e cada lugar pode ser fundido com um ou mais lugares. Uma rede de Petri colorida componível pode ser vista como uma forma de interligar um conjunto de redes de Petri coloridas utilizando uma forma particular de fusão de transições — permitindo comunicação síncrona — e fusão de lugares — permitindo comunicação assíncrona. Assumir-se-á que cada rede de Petri colorida define um escopo próprio para cada variável. Desta forma, variáveis com identificadores iguais em redes distintas são sempre variáveis distintas. A única excepção surge quando ambas estiverem relacionadas pela passagem de parâmetros em eventos, tal como será apresentado adiante.

Seguidamente, define-se uma rede de Petri colorida componível (Def. 5.7). Cada rede de Petri colorida componível contém um conjunto de redes de Petri coloridas denominadas **páginas**. Este conjunto é denotado por \mathcal{S} . Genericamente, utilizar-se-á X_s para denotar " X de $s \in \mathcal{S}$ ", e $X_{\mathcal{S}}$ para denotar o conjunto $\bigcup_{s \in \mathcal{S}} X_s$. Por exemplo: L_s denota o conjunto de lugares na rede de Petri colorida $s \in \mathcal{S}$; $L_{\mathcal{S}}$ denota o conjunto de todos os lugares em todas as redes de Petri coloridas em \mathcal{S} . A Def. 5.7 utiliza uma notação semelhante à empregue por Christensen e Petrucci para a definição das redes de Petri coloridas modulares [Christensen e Petrucci, 1992]. Tal permite uma fácil comparação de ambas as classes de redes, ao mesmo tempo que evita a desnecessária introdução de novas notações e nomenclaturas. Os parágrafos que se seguem à definição explicam-na e como tal devem ser lidos juntamente com a mesma.

Definição 5.7 (Rede de Petri colorida componível): *Seja $SO = (\mathcal{CS}, \mathcal{CP}, V, CONST, class, attrib, Obj, \mathcal{E}, qp)$ um sistema de objectos. Uma rede de Petri colorida componível é um tuplo $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, A)$ que define o comportamento de um conjunto de objectos pertencentes a um conjunto de classes $\{CS_1, \dots, CS_n\} \subseteq \mathcal{CS}$ e que satisfaz os seguintes requisitos:*

1. \mathcal{S} é um conjunto de redes de Petri coloridas, tal que:

- (a) $\forall c \in \Sigma_{\mathcal{S}}, c \in \mathcal{CS}$.
- (b) cr é uma função "classe da rede" que aplica redes em $\mathcal{S} \setminus \mathbb{M}$ em classes de sistema em SO :
 $cr : (\mathcal{S} \setminus \mathbb{M}) \rightarrow \mathcal{CS}$.
- (c) \mathbb{M} é um subconjunto de redes em \mathcal{S} , denominadas módulos.
- (d) Os conjuntos de elementos de rede são disjuntos dois a dois:

$$\forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2 \Leftrightarrow (L_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (L_{s_2} \cup T_{s_2} \cup A_{s_2}) = \emptyset$$

- (e) $Var(T_{\mathcal{S}}) \subseteq V$.

2. LF é uma partição do conjunto de todos os lugares em \mathcal{S} (2a, 2b, 2c, 2d), em que todos os lugares num dado conjunto da partição têm o mesmo conjunto de cores associado e a mesma marcação inicial (2e):

- (a) $LF \subseteq \mathcal{P}(L_{\mathcal{S}})$
- (b) $LFE \in LF \Rightarrow LFE \neq \emptyset$
- (c) $\forall LFE_1, LFE_2 \in LF, (LFE_1 \neq LFE_2) \Rightarrow LFE_1 \cap LFE_2 = \emptyset$

$$(d) \forall l \in L_S, \exists LFE \in LF, l \in LFE$$

$$(e) \forall LFE \in LF, \forall l_1, l_2 \in LFE, C(l_1) = C(l_2) \wedge I(l_1) = I(l_2)$$

3. ET é uma função que aplica transições em eventos (3a), tal que:

$$(a) ET : T_S \rightarrow \mathcal{P}(\mathcal{E}), \text{ tal que } \forall t \in T_S, |ET(t) \cap \mathcal{E}^R| \leq 1$$

$$(b) \forall t \in T_S, \forall e^S \in ET(t)^S, class(e^S) \in VarIn(t^S) \Rightarrow \exists CC \in \Sigma_S, Type(class(e^S)) = CC$$

(c) Os atributos dos eventos têm necessariamente de satisfazer o seguinte:

$$\begin{aligned} & \forall t^S \in T_S, \forall e^S \in ET(t^S), \exists t^R \in T_S, \exists e^R \in ET(t^R), \\ & \left(t^S \neq t^R \wedge e^S \underset{ped}{=} e^R \wedge \right. \\ & \quad attrib_{IN}(e^S) \subseteq (VarIn(t^S) \cup CONST) \wedge attrib_{OUT}(e^S) \subseteq VarOut(t^S) \wedge \\ & \quad attrib_{IN}(e^R) \cap VarIn(t^R) = \emptyset \wedge \\ & \quad attrib_{OUT}(e^R) \subseteq VarIn(t^R) \wedge \\ & \quad attrib_{INOUT}(e^R) \subseteq VarIn(t^R) \wedge \\ & \quad \left. attrib_{INOUT}(e^S) \subseteq (VarIn(t^S) \cap VarOut(t^S)) \right) \end{aligned}$$

4. A é um conjunto de adições de redes em \mathcal{S} :

$$\begin{aligned} A = \{ (R_{op_1}, \dots, R_{op_n}, CI) \mid (R_{op_1}, \dots, R_{op_n}) \in \mathcal{S}^n \wedge CI \text{ é um colapso de interface} \wedge \\ \forall l \in L_{CI}, \{l\} \in LF \} \end{aligned}$$

1. Cada conjunto de cores é uma classe de sistema (1a). As redes de Petri coloridas componíveis não obrigam todas as classes a serem modeladas por uma rede pois tal seria totalmente contraproducente: a modelação excessiva de comportamentos tem sido apontada como uma das principais causas da "paralisia na análise"⁹[Desmond Francis D'Souza e Alan Cameron Wills, 1998]. Por exemplo, se todas as operações de um objecto podem ser invocadas independentemente do estado deste, a modelação de comportamento não é necessária. Algumas páginas correspondem a classes de sistema no sistema de objectos e são denominadas **classes da rede de Petri colorida componível** (1b). Outras correspondem a módulos que não modelam classes (1c). Finalmente, todos os nós e arcos são considerados disjuntos (1d) e todas as variáveis de transição são variáveis do sistema de objectos (1e).

2. Os conjuntos de lugares fundidos são conjuntos disjuntos de lugares, pertencentes a uma ou mais redes de Petri coloridas, que são considerados ocorrências múltiplas de um único lugar. Cada conjunto na partição LF em $\mathcal{P}(L_S)$ é denominado um conjunto de fusão de lugares. Todos os lugares, num dado conjunto de fusão de lugares, possuem o mesmo conjunto de cor associado e a mesma expressão de inicialização (2e). Denota-se por $[l]$ o conjunto de fusão contendo o lugar $l \in L_S$.

⁹Em inglês: *analysis paralysis*.

3. As transições podem ter zero ou mais eventos emissores (*SEND*) associados, e zero ou um evento receptor (*RECV*) (3a). Para uma dada transição t , utilizar-se-ão as notações $ET(t)^S = ET(t) \cap \mathcal{E}^S$ e $ET(t)^R = ET(t) \cap \mathcal{E}^R$.

Se a classe de um evento emissor (a classe onde se encontra o evento receptor) for especificada por uma variável de entrada da transição respectiva, então terá de existir um conjunto de cores em que uma das cores é classe desse evento (3b). Tal vinculação dinâmica¹⁰ permite uma invocação polimórfica associada, conforme será ilustrado no exemplo da Secção 6.2 na pág. 150. Para cada evento emissor numa transição, o evento receptor correspondente tem de estar associado a uma transição diferente da do emissor e os atributos dos eventos (os quais incluem parâmetros de pedido) têm de ser incluídos em subconjuntos próprios das respectivas variáveis das transições *SEND* e *RECV*, de acordo com os respectivos qualificadores de parâmetros (3c).

4. Quaisquer páginas podem ser adicionadas de forma a obter novas páginas (*vide* Def. 3.20 na pág. 75). Como forma de dificultar composições pouco legíveis, estas adições não podem utilizar conjuntos de fusão que incluam lugares que se encontrem fundidos com outros através da partição *LF*.

Uma **transição** diz-se **emissora** quando tem um, ou mais, eventos *SEND* associados. Uma **transição** diz-se **receptora** quando tem um evento *RECV* associado. Assim, uma transição pode ser simultaneamente emissora e receptora.

Utiliza-se também a função $local : \mathcal{E} \rightarrow \mathcal{P}(\mathcal{S})$ que indica as classes em que um dado evento e está declarado: $\forall s \in local(e), \exists t \in T_s, e \in ET(t)$.

Finalmente, definem-se **eventos comunicantes**:

Definição 5.8 (Eventos Comunicantes): Dada uma rede de Petri colorida componível $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, A)$, dois eventos compatíveis $e^S = (SEND, c^S, ped^S, lp^S) \in \mathcal{E}^S$ e $e^R = (RECV, c^R, ped^R, lp^R) \in \mathcal{E}^R$ dizem-se comunicantes, e denotam-se $e^S \rightleftharpoons e^R$, se e só se $\{class(e^S)\} \in local(e^R)$. Neste caso, é também verdadeiro que $\{class(e^R)\} \in local(e^S)$.

Seguidamente, define-se a adição em redes de Petri coloridas componíveis.

5.1.2 Adição nas Redes de Petri Coloridas Componíveis

A utilização da adição de redes (*vide* Def. 3.20 na pág. 75) no contexto das redes de Petri coloridas componíveis tem por objectivo oferecer um suporte adicional de composição de redes mas baseado em módulos e instâncias de módulos o que permite uma forma versátil de compor páginas da rede de Petri colorida componível. A adição de módulos pode ser vista como um

¹⁰Em inglês: *dynamic binding*.

suporte para os conceitos de módulo e de **mixins**. Este último existe para algumas linguagens de programação (e.g. Ruby [Ruby Home Page, 2004]) e permite uma forma simples de suportar herança de implementações.

A adição de redes é aplicada numa rede de Petri colorida componível, entre as várias páginas que as constituem. A operação é muito simples e pode ser definida informalmente pela seguinte sequência de passos, definíveis através de funções de transformação numa estrutura de transformação (vide Def. 3.28 na pág. 84):

1. As marcações nos lugares de fusão são adicionadas; tal implica a definição prévia da função de transformação para as marcações dos lugares adicionados.
2. Todos os arcos (e respectivas inscrições) são preservados.
3. Cada transição resultado tem por guarda a conjunção das guardas das transições interface respectivas.
4. Desde que não exista mais do que um evento receptor nas transições de um dado conjunto de fusão, a transição resultado respectiva fica com todos os eventos dessas transições. Se mais do que uma transição no conjunto de fusão tiver associado um evento receptor, a estrutura de transformação deve reflectir a atitude a tomar para evitar que surja mais do que um evento receptor na transição resultado. Uma hipótese será não permitir fusões com estas características.

A secção seguinte mostra de que modo uma rede de Petri colorida componível pode ser traduzida para uma rede de Petri colorida equivalente.

5.1.3 Das Redes de Petri Coloridas Componíveis às Redes de Petri Coloridas

Esta secção define a semântica das redes de Petri coloridas componíveis através da tradução destas para uma única rede de Petri colorida. Dado que as redes de Petri coloridas têm uma semântica bem documentada [Jensen, 1997c,a,b] e implementada [s.a., 2004c,b], esta tradução define implicitamente a semântica das redes de Petri coloridas componíveis.

Uma rede de Petri colorida componível contém um conjunto de redes de Petri coloridas. Por essa razão, a tradução de uma rede de Petri colorida componível para uma rede de Petri colorida centra-se na remoção da adição entre módulos (páginas na rede de Petri colorida componível) e na remoção das duas formas de interligar essas redes de Petri coloridas: o conjunto de conjuntos de lugares fundidos (LF) e a associação de eventos às transições através da função ET .

Começa-se pela remoção da adição entre páginas. Dada uma rede de Petri colorida componível $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, A)$, começa-se por definir o conjunto das redes operando (\mathcal{A}_{op}), o conjunto

das redes resultado (\mathcal{A}_{res}), o conjunto dos conjuntos de fusão (\mathcal{A}_{cf}) e a rede de Petri colorida componível sem adições:

$$\begin{aligned}\mathcal{A}_{op} &= \{R \in \mathcal{S} \mid \exists(R_{op_1}, \dots, R_{op_n}, CI) \in A, R \in \{R_{op_1}, \dots, R_{op_n}\}\} \\ \mathcal{A}_{res} &= \{R_{res} \mid \exists(R_{op_1}, \dots, R_{op_n}, CI) \in A, R_{res} = (R_{op_1} + \dots + R_{op_n})(CI)\} \\ \mathcal{A}_{cf} &= \{cf \mid \exists(R_{op_1}, \dots, R_{op_n}, CI) \in A, cf \in CI\}\end{aligned}$$

Considere-se também $T_{\mathcal{A}_{cf}} = \{t \in (T_S \cap CF) \mid CF \in \mathcal{A}_{cf}\}$. Podemos agora definir uma rede de Petri colorida componível sem adições, equivalente a outra com adições:

Definição 5.9: Rede de Petri colorida componível sem adições Dada uma rede de Petri colorida componível $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, A)$, a rede de Petri colorida componível sem adições equivalente é uma rede $R' = (\mathcal{S}', cr', \mathbb{M}', LF', ET', \emptyset)$ que satisfaz os seguintes requisitos:

1. $\mathcal{S}' = (\mathcal{S} \setminus \mathcal{A}_{op}) \cup \mathcal{A}_{res}$
2. $cr' = cr \cap cr(\mathcal{S}')$
3. $\mathbb{M}' = \mathbb{M} \setminus \mathcal{A}_{op}$
4. $LF' = LF$
5. $ET' = \{(t, E) \in ET \mid t \notin T_{\mathcal{A}_{cf}}\} \cup \{(Result(cf), E) \mid cf \in \mathcal{A}_{cf} \wedge E = \bigcup_{t \in cf} ET(t) \wedge |E^R| \leq 1\}$

Apesar da Def. 5.9 considerar todas as redes resultado como classes, tal deve ficar dependente da interpretação dada pelo modelador. Em particular, para cada rede resultado, o modelador deve poder classificá-la em classe (membro de \mathcal{S}) ou módulo (membro de \mathbb{M}).

De um ponto de vista centrado na rede de Petri, cada transição t , sem eventos associados ($ET(t) = \emptyset$), bem como as fusões resultantes dos eventos *SEND* e *RECV* definem **grupos de sincronismo**:

Definição 5.10 (Grupo de Sincronismo): Seja $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, \emptyset)$ uma rede de Petri colorida componível sem adições. Um grupo de sincronismo em R é constituído ou por uma transição única $t \in T_S$, sem eventos associados, ou por uma lista de conjuntos de transições não repetidas, com pelo menos dois elementos, em que o primeiro elemento têm uma única transição com um, ou mais, eventos emissores; as transições, no último elemento, não têm eventos emissores; e em cada um dos restantes elementos, se existirem, pelo menos uma transição tem um evento receptor e um evento emissor. O conjunto de todos os grupos de sincronismo é denotado por GS :

$$\begin{aligned}
GS = & \{t \in T_S \mid ET(t) = \emptyset\} \cup \\
& \left\{ (ct_1, \dots, ct_n) \in \bigcup_{n \geq 2} (\mathcal{P}(T_S))^n \mid \right. \\
& \left(\bigcap_{1 \leq i \leq n} ct_i \right) = \emptyset \wedge \\
& (\exists t \in T_S, ct_1 = \{t\} \wedge ET(t) = ET(t)^S) \wedge \\
& (\forall 2 \leq i \leq n, \exists t^S \in ct_{i-1}, \exists e^S \in t^S, \exists t^R \in ct_i, \exists e^R \in t^R, e^S \rightleftharpoons e^R) \wedge \\
& (\forall 2 \leq i < n, \exists t \in ct_i, ET(t)^R \neq \emptyset \wedge ET(t)^S \neq \emptyset) \wedge \\
& \left. (\forall t \in ct_n, ET(t)^S = \emptyset) \right\}
\end{aligned}$$

Cada grupo de sincronismo corresponde a uma única transição na rede de Petri colorida equivalente. Esta transição tem associados todos os arcos de entrada e de saída das transições que constituem o grupo de sincronismo.

Um grupo de sincronismo é *trivial* se contém uma única transição da rede de Petri colorida componível. Como tal, o conjunto $\{t \in T_S \mid ET(t) = \emptyset\}$ é denominado o conjunto dos grupos de sincronismo triviais. Os grupos de sincronismo não triviais podem ser vistos como caminhos, sem ciclos, entre as transições das várias páginas da rede de Petri colorida componível. Note-se que num caminho com mais do que dois conjuntos de transições, cada um dos conjuntos intermédios (depois do primeiro e antes do último) tem de conter, pelo menos, uma transição receptora e uma transição emissora, podendo estas corresponder a uma só transição. Só desta forma, cada um dos conjuntos intermédios pode funcionar como elo na cadeia de invocação. Cada uma destas cadeias comporta-se como uma única transição resultante da fusão de todas as transições que lhe pertencem.

Conforme já referido, a possibilidade de manipular o parâmetro classe (*c*) de cada evento ($SEND, c, ped, lp$) como uma variável de transição permite a especificação de invocações polimórficas de pedidos, através da tradução para fusões de transições: uma para cada possível invocação. Um exemplo é apresentado na Secção 6.2.

Os grupos de sincronismo são baseados nos canais síncronos propostos por Christensen e Hansen [Christensen e Hansen, 1992] e utilizados nas redes de Petri modulares [Christensen e Petrucci, 1992]. No entanto, os grupos de sincronismo têm por objectivo a modelação da invocação de métodos tal como normalmente suportado pelas linguagens de programação orientadas pelos objectos e definido na especificação da UML. Comparativamente com a proposta de Christensen e Hansen, tal implica duas diferenças fundamentais: (1) existe direcção de invocação; (2) a informação "transportada" pelos canais é qualificada segundo o sentido em que é transmitida.

A ferramenta RENEW [Kummer et al., 2004b,a] também implementa uma forma de canais síncronos com sentido de invocação. Ela difere da proposta aqui apresentada em três aspectos: (1) as redes na ferramenta RENEW são *redes objecto* nas quais as marcas podem ser referências

para objectos e as instâncias de redes são criadas dinamicamente (2) Os parâmetros dos canais não têm qualificadores; (3) não existe a noção de módulo, todas as composições são entre objectos.

As variáveis de evento são um subconjunto das variáveis de transição, respeitando as restrições especificadas no ponto 3c na Def. 5.7. Como tal, os grupos de sincronismo podem ser vistos como fusões de transições de alto-nível que restringem o conjunto de vínculos possíveis. Esta restrição é imposta pelos parâmetros *IN* porque podem surgir na guarda da transição receptora; pelos parâmetros *OUT* porque podem surgir na guarda da transição emissora; pelos parâmetros *INOUT* porque além de poderem surgir nas guardas, também estabelecem uma dependência entre variáveis das respectivas transições emissora e receptora. Relativamente às transições, refere-se a guarda de um grupo de sincronismo *gs* como $G(gs)$. Esta é a conjunção das guardas das transições em *gs* e um teste nos parâmetros classe dos eventos (*vide* ponto 5 na Def. 5.11 e exemplo na Secção 6.2).

Definição 5.11 (Vínculo do Grupo de Sincronismo): *Seja $R = (S, cr, \mathbb{M}, LF, ET, \emptyset)$ uma rede de Petri colorida componível sem adições e seja $gs = (ct_1, \dots, ct_n)$ um grupo de sincronismo não trivial em R . Um vínculo vin de gs (denotado $\langle gs \rangle$) é uma função definida nas variáveis de transição do grupo de sincronismo, $Var(gs) = \bigcup_{1 \leq i \leq n} Var(ct_i)$, tal que:*

1. $\forall v \in Var(gs), vin(v) \in Type(v)$
2. $\forall 1 \leq i < n, \forall t^S \in ct_i, \forall e^S \in ET(t^S), \exists t^R \in ct_{i+1}, \exists e^R \in ET(t^R),$

$$e^S_{ped} = e^R \wedge$$

$$(\forall 1 \leq j \leq |attrib(e^S)|, (attrib_j(e^S) \langle vin \rangle = attrib_j(e^R) \langle vin \rangle) \wedge$$

$$(Var(t^S) \setminus attrib(ET(t^S))) \cap (Var(t^R) \setminus attrib(ET(t^R))) = \emptyset$$
3. $G(gs) = \forall 1 \leq i < n, \forall t^S \in ct_i, \forall e^S \in ET(t^S), \exists t^R \in ct_{i+1}, \exists e^R \in ET(t^R),$

$$class(e^S) \langle vin \rangle = local(e^R) \wedge$$

$$G(t^S) \langle vin \rangle \wedge G(t^R) \langle vin \rangle$$

No ponto 1 especifica-se que todas as variáveis do grupo de sincronismo ficam vinculadas por um valor do seu tipo. O ponto 2 define uma igualdade posicional entre os parâmetros do evento emissor e do evento receptor de cada pedido e obriga a que todas as variáveis que não pertencem aos atributos dos eventos sejam diferentes. O ponto 3, correspondente à guarda do grupo de sincronismo, obriga a que a classe do evento emissor especifique a classe onde se encontra o evento receptor¹¹. Para além disso a conjunção das guardas de todas as transições tem de ser satisfeita pelo vínculo. Por outras palavras, a "guarda do grupo de sincronismo" inclui a conjunção das guardas de todas as suas transições.

¹¹Dado que a classe pode ser especificada por uma variável da transição, a especificação da classe onde se encontra o evento receptor só é necessariamente verdade considerando a aplicação do vínculo a essa variável. Tal implica que a Def. 5.10 origina um grupo de sincronismo para cada vínculo possível dessa variável.

A Fig. 5.1 apresenta um pedido síncrono e a respectiva semântica. Esta é ilustrada através de uma **transição pedido** (*GS AB* na Fig. 5.1b), ou seja uma transição cuja semântica é igual à do grupo de sincronismo. Assim, a guarda desta transição é a "guarda do grupo de sincronismo" atrás definida. Note-se que a igualdade entre parâmetros aparece expressa pela substituição textual (passagem por nome) dos atributos formais pelos atributos actuais. Tal será novamente referido mais adiante.

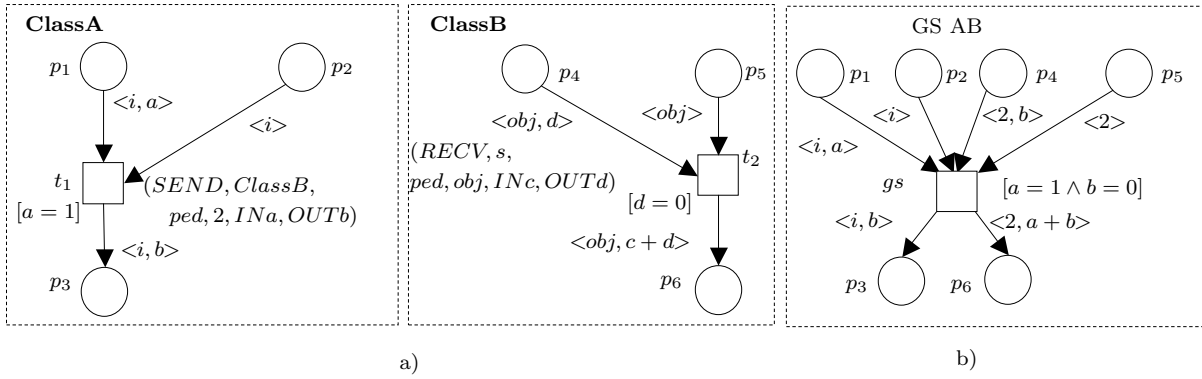


Figura 5.1: a) Um pedido síncrono com parâmetros, e b) a respectiva transição pedido com igual semântica.

As declarações dos eventos *SEND* e *RECV* têm o seguinte formato:

$(SEND, classeDoReceptor, pedido, [qualificador_1]parâmetro_1, \dots, [qualificador_n]parâmetro_n)$
 $(RECV, classeDoEmissor, pedido, [qualificador_1]parâmetro_1, \dots, [qualificador_n]parâmetro_n)$

Os seus elementos têm as seguintes funções:

SEND e RECV Os elementos *SEND* e *RECV* especificam o sentido de invocação. Numa linguagem de programação orientada pelos objectos tal corresponde, respectivamente, à invocação e à definição de um método.

classeDoReceptor O elemento *classeDoReceptor* pode ter um de dois sentidos distintos: (1) a classe a que pertence uma transição receptora, ou seja, a classe do objecto invocado; (2) uma variável cujo domínio é um conjunto de nomes de classes que contenham uma transição receptora para o pedido especificado (*vide* ponto 3 na Def. 5.11); conforme já referido, tal permite especificar uma invocação polimórfica, que está ilustrada no exemplo da Secção 6.2.

pedido O elemento *pedido* identifica o pedido e portanto também uma invocação. Desta forma, estabelece a ligação entre o evento emissor e o evento receptor, torna-os comunicantes.

parâmetro e qualificador Os elementos *parâmetro_i* são variáveis da transição; o qualificador *qualificador_i* pode tomar um dos seguintes valores: "IN", "OUT" ou "INOUT". Estes

nomes são atribuídos do ponto de vista da transição emissora. Qualquer qualificador pode ser omitido e nesse caso assume-se o qualificador *IN* ¹².

classeDoEmissor *classeDoEmissor* é o nome da classe da transição emissora; tal permite a sua utilização como uma variável de transição que especifica um canal de retorno até à classe emissora; mais especificamente, na transição receptora, pode ser visto como um parâmetro *IN* permanente. Tal permite a modelação de uma invocação de método utilizando dois grupos de sincronismo: um para invocar o método e outro para retornar para o invocador. Tal é discutido com mais detalhe na Secção 5.2.4 (*vide* em particular a Fig. 5.8 na pág. 136).

De acordo com os respectivos qualificadores, os parâmetros são vinculados da seguinte forma:

- Um parâmetro "*IN*" é vinculado pela transição emissora. Tipicamente será utilizado num ou mais arcos de saída da transição receptora.
- Um parâmetro "*OUT*" é vinculado pela transição receptora. Tipicamente, ele será utilizado num ou mais arcos de saída da transição emissora.
- Um parâmetro "*INOUT*" é vinculado quer pela transição emissora quer pela transição receptora. Tipicamente, ele será utilizado num ou mais arcos de saída da transição emissora e também num ou mais arcos de saída da transição receptora.

A passagem de parâmetros é feita "por nome": os atributos do evento *RECV* são textualmente substituídos pelos respectivos atributos do evento *SEND*. Tal é possível porque se considera que existem escopos distintos para as variáveis em transições distintas, ou variáveis com nomes distintos. Caso esta separação de escopos não seja conveniente, basta, por exemplo, impor a igualdade entre parâmetros na guarda da transição fundida (transição pedido) de forma a não ocorrerem sobreposições indesejadas de nomes. É essa a solução proposta em [Christensen e Hansen, 1992].

Na Fig. 5.1, as variáveis *a* e *b* substituem, respectivamente, as variáveis *c* e *d* "por nome": *a* e *b* constituem os parâmetros actuais do pedido, *c* e *d* são os parâmetros formais. Note-se que o identificador do objecto (2 no exemplo) é também um parâmetro do pedido: por omissão, é um parâmetro "*IN*". O identificador *ClassB* (*classeDoReceptor*) é utilizado para especificar a rede (classe) onde a correspondente declaração *RECV* reside. Na declaração *RECV*, o elemento *classeDoEmissor* identifica a classe onde a respectiva declaração *SEND* reside (parâmetro *s* no exemplo).

¹²Como conveniência sintáctica, é possível admitir a especificação de eventos com igual nome, mas diferindo nos parâmetros, podendo incluir parâmetros com valores por omissão como, por exemplo, na linguagem de programação C++ [Stroustrup, 1997].

Podemos agora definir a rede de Petri colorida equivalente a uma rede de Petri colorida componível sem adições:

Definição 5.12: Rede de Petri colorida equivalente *Dada uma rede de Petri colorida componível sem adições $R = (\mathcal{S}, cr, \mathbb{M}, LF, ET, \emptyset)$, seja GS o conjunto dos seus grupos de sincronismo. A rede de Petri colorida equivalente é uma rede CPN $= (\Sigma', L', T', A', N', C', G', E', I')$ que verifica os seguintes requisitos:*

1. $\Sigma' = \Sigma_{\mathcal{S}}$
2. $L' = LF$
3. $T' = GS$
4. $A' = \{(a, gs) \in A_{\mathcal{S}} \times GS \mid t(a) \in gs\}$
5. $\forall (a, gs) \in A', N'(a, gs) = N([l(a)], gs) \vee N'(gs, a) = N(gs, [l(a)])$
6. $\forall l' \in L', C'(l') = C(l')$
7. $\forall gs \in T', G'(gs) = G(gs)$
8. $\forall a' = (a, gs) \in A', E'(a') = E(a) <gs>$
9. $\forall l' \in L', I'(l') = I(l')$

A rede de Petri colorida equivalente inclui todos os conjuntos de cores das várias páginas (1), os seus lugares são os elementos dos conjuntos de fusão de lugares (2) e as suas transições são os grupos de sincronismo (3). Os arcos são definidos associando cada arco de cada página ao grupo de sincronismo da transição a que se encontra ligado (4). A função nó aplica os arcos em pares constituídos por um grupo de sincronismo e por um conjunto de lugares fundidos (5). As cores dos lugares são as dos conjuntos de lugares fundidos (6) e as guardas são as dos grupos de sincronismo (7). As expressões nos arcos mantém-se mas vinculadas pelo vínculo do grupo de sincronismo (8), e as marcações iniciais nos lugares são as dos conjuntos de lugares fundidos (9).

A secção seguinte apresenta um conjunto de idiomas para a modelação dos principais conceitos do desenvolvimento orientado pelos objectos utilizando redes de Petri coloridas componíveis.

5.2 Redes de Petri e Modelos Orientados pelos Objectos

Nesta secção discute-se de que forma os principais conceitos do desenvolvimento orientado pelos objectos podem ser suportados por idiomas aplicáveis a redes de Petri coloridas componíveis. Em particular, começa-se pela apresentação do conceito de rede de Petri colorida componível canónica que fornece o suporte para o ciclo de vida dos objectos. Seguidamente, apresenta-se o suporte para a classificação, atributos de objectos e de classes, pedidos síncronos e assíncronos,

composição e agregação. É dada especial ênfase à composição e à troca de mensagens, na qual se incluem as invocações polimórficas. Estes dois conceitos, enquadrados pelo suporte à classificação, constituem os elementos fundamentais para o desenvolvimento orientado pelos objectos. Neste sentido, a herança na sua forma mais geral (herança de implementação) é vista como inadequada para o desenvolvimento orientado pelos objectos utilizando redes de Petri coloridas componíveis.

5.2.1 Redes Canónicas e Adição de Módulos

Nesta secção define-se uma rede de Petri colorida componível canónica. Uma página de uma rede de Petri colorida componível canónica apresenta um conjunto de características que permitem uma especificação explícita e uniforme de uma classe, ou seja, dos atributos e do ciclo de vida dos respectivos objectos.

Uma página (classe) da rede de Petri colorida componível diz-se **canónica** se respeitar um dos idiomas ilustrados na Fig. 5.2 e descritos em seguida. Uma rede de Petri colorida componível diz-se canónica se todas as suas classes são canónicas. As classes canónicas incluem a modelação da criação ou da criação e destruição de objectos. É esta a sua característica fundamental. Note-se que outras redes podem ser canónicas ainda que incluam outros elementos que não os apresentados na Fig. 5.2.

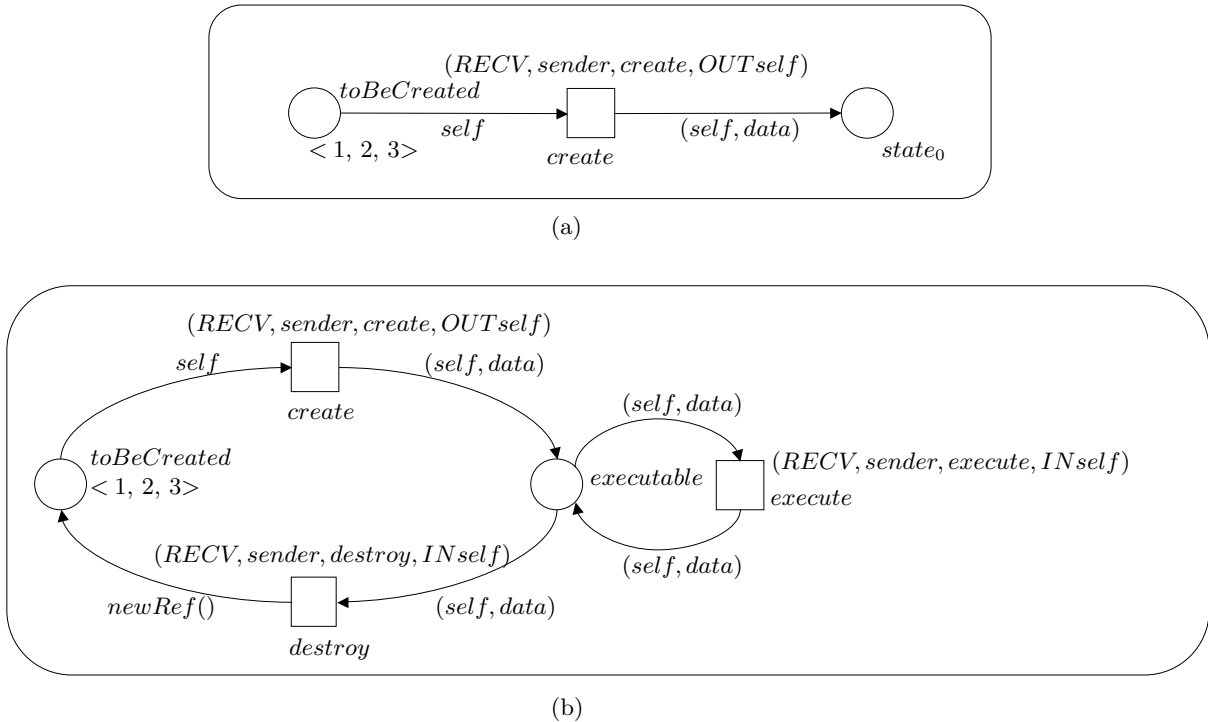


Figura 5.2: Dois idiomas para classes canónicas num rede de Petri colorida componível canónica: (a) sem destruição de objectos e (b) com destruição de objectos.

Para cada classe da rede de Petri colorida componível canónica pode ou não importar a modelação da destruição de objectos. Os dois idiomas na Fig. 5.2 correspondem a cada uma destas situações. Em ambas, um objecto é criado pelo evento *create* que devolve uma nova referência para o exterior (*self* nas Figs. 5.2a e 5.2b) e cria, ao mesmo tempo que inicializa, os atributos no novo objecto (*data* na Fig. 5.2a e 5.2b).

A modelação utilizando uma rede de Petri permite que os identificadores dos objectos que podem ser criados, estejam inicialmente disponíveis num lugar. Nas Figs. 5.2a e 5.2b esse lugar é denominado *toBeCreated*. Foram lá colocadas, a título meramente exemplificativo, três números inteiros que irão constituir os identificadores para um máximo de três objectos. A transição que aceita o pedido *create* remove uma marca desse lugar e deposita-a no lugar *state₀* ou *executable* juntamente com os dados do último objecto criado. Naturalmente, é possível adicionar parâmetros ao evento *create*. Estes parâmetros correspondem a parâmetros de um construtor de objectos.

No primeiro idioma (Fig. 5.2a), as marcas representativas dos objectos "movem-se" para outros estados, a partir do estado alcançado após a sua criação (estado inicial *state₀*). Este "movimento" corresponde ao comportamento de cada objecto.

No segundo idioma (Fig. 5.2b), a rede contém dois lugares (*toBeCreated* e *executable*) mais as respectivas transições (*create* e *destroy*). Para além deste ciclo de vida do objecto, a rede contém a transição *execute* que testa a existência de um dado objecto no lugar (estado) *executable*. Este objecto pode então ser executado. Tal significa que cada uma das operações do objecto que invoca o evento *execute* na rede ciclo de vida pode ficar habilitada. Cada objecto no estado *executable* pode, em qualquer momento, ser destruído. Para tal utiliza-se o evento receptor *destroy* associado à transição *destroy*, que recebe do exterior a referência para o objecto a destruir.

As marcas no lugar *executable* apenas podem ser removidas pela transição que aceita o pedido *destroy*. Esta transição trata de colocar novas marcas em *toBeCreated*, que serão as referências de possíveis novos e futuros objectos (a função *newRef()* garante referências originais). Este mecanismo corresponde a um idioma usualmente denominado lugar complementar na literatura da área das redes de Petri. Neste caso, um lugar contém o conjunto dos números de instâncias de cada um dos objectos que podem ser criados; e o outro lugar contém os objectos já criados. Este mecanismo também fornece a informação sobre a quantidade máxima de objectos que podem ser criados. Tal é útil caso se pretenda impor um limite na quantidade máxima de objectos que podem ser gerados.

A transição *execute* tem um evento *execute* associado o qual pode ser utilizado por todas as transições que definem o comportamento dos objectos no estado *executable* e que não necessitam de ler nem de modificar os dados dos objectos. Estas transições são (preferencialmente) modeladas noutra página e fundem-se com a transição *execute* na classe canónica. Esta outra

página modela o comportamento dos objectos da classe e será aqui referida como **módulo de comportamento**. A transição *execute* serve de "guarda" às transições neste módulo de comportamento testando se o objecto está num estado do seu ciclo de vida em que pode responder a cada um dos possíveis pedidos que lhe podem ser feitos.

Os módulos numa rede de Petri colorida componível são especialmente úteis para adicionar funcionalidades a classes existentes. A adição da leitura e modificação dos dados de cada objecto constitui um exemplo onde a adição de módulos (Def. 3.20 na pág. 75) permite uma modularidade acrescida. A Fig. 5.3 apresenta os módulos para especificação da leitura e escrita dos dados de cada objecto em qualquer classe. A Fig. 5.4 mostra o resultado (abreviado) da sua adição à classe canónica na Fig. 5.2b. Tal como a classe canónica, este módulo contém o lugar *executable* e as transições *create* e *destroy*. Para além deste ciclo de vida do objecto, a rede contém a transição *execute* que testa a existência de um dado objecto no lugar (estado) *executable*. Por fim, tem também as transições *readAndLock* e *writeAndUnlock* com os respectivos eventos associados. Estes permitem, respectivamente, ler e modificar os dados de um dado objecto.

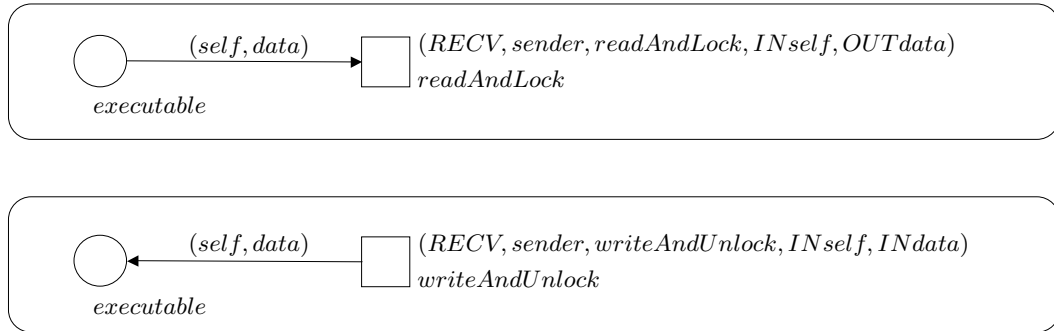


Figura 5.3: Módulos para leitura e escrita dos atributos dos objectos.

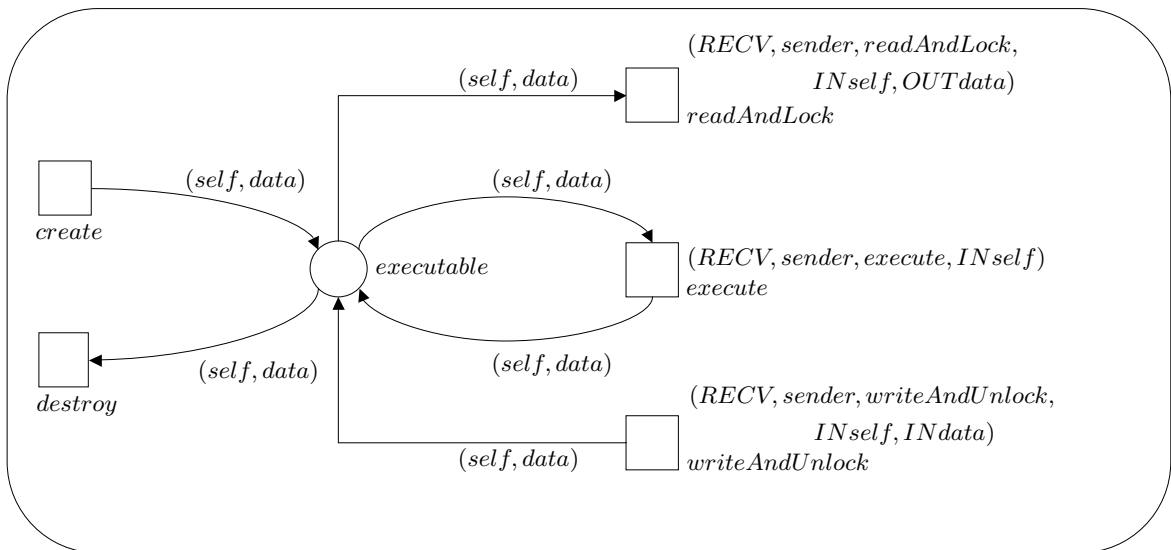


Figura 5.4: Classe canónica (abreviada) com destruição, leitura e escrita.

As transições *readAndLock* e *writeAndUnlock* permitem, respectivamente, ler e modificar os dados de cada objecto, em exclusão mútua. Por exemplo, considere-se uma operação num módulo de comportamento R_{comp} que envolva várias transições, que podem ser vistas como sub-operações, e que deva ser executada pelo objecto sem que este seja interrompido. Tal é modelado por uma transição inicial, uma transição final e zero ou mais passos intermédios. A transição inicial invoca o evento *readAndLock* para obter os dados. Esta transição e as seguintes podem então modificar esses dados. A transição final invoca o evento *writeAndUnlock*, actualizando assim os dados do objecto. Simultaneamente, este fica novamente disponível para executar operações. A Fig. 5.5b ilustra, de forma genérica este idioma.

A modelação do comportamento dos objectos em cada uma das classes canónicas (Fig. 5.2a ou 5.2b), assume a aplicação do respectivo idioma na Fig. 5.5a ou Fig. 5.5b, respectivamente.

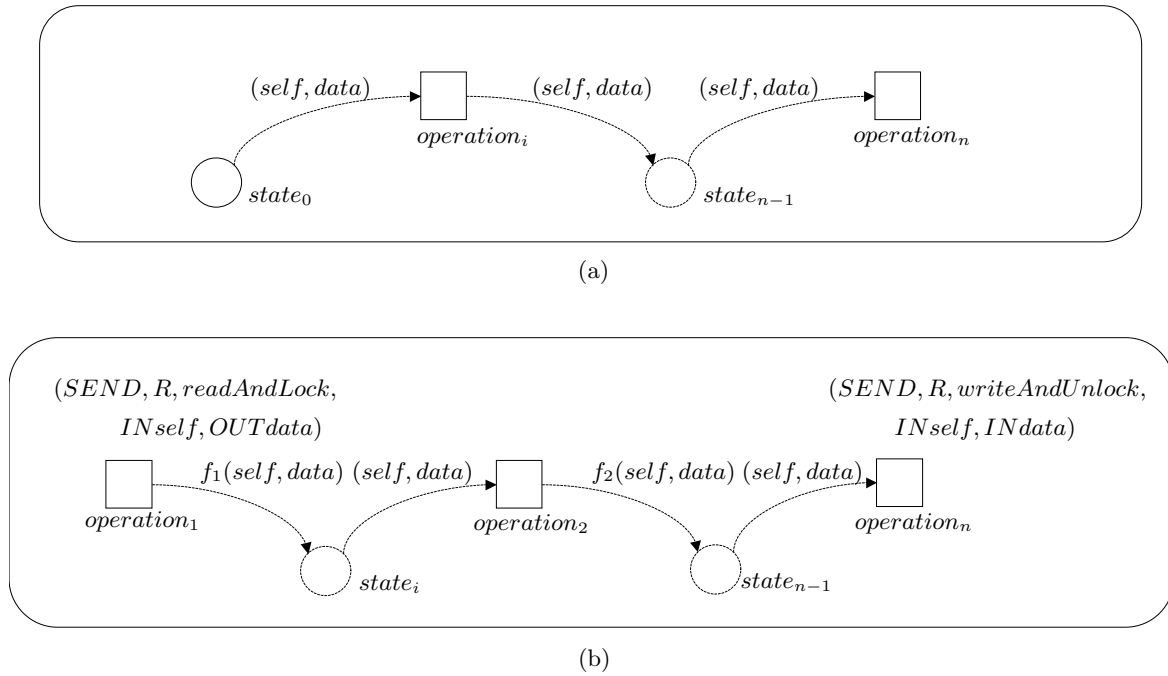


Figura 5.5: Dois idiomas para modelação do comportamento para cada uma classes canónicas na Fig. 5.2: (a) sem destruição de objectos e (b) com destruição de objectos.

Conforme já referido, no primeiro caso (Fig. 5.5a), as marcas representativas dos objectos "movem-se" para outros estados, a partir do estado alcançado após a sua criação (estado inicial). Este "movimento" corresponde ao comportamento de cada objecto e pode ser modelado por outra classe que se liga à especificação da classe canónica através de uma adição de páginas (neste caso, classe mais módulo) por meio da fusão dos lugares $state_0$. Na realidade, a adição deste módulo à classe pode ser vista como uma adição de funcionalidade à classe que assim recebe um conjunto de métodos.

No segundo caso, após o disparo da transição (operação) *destroy* para um dado objecto, esse objecto não poderá já efectuar operações dado estas não poderem ser executadas devido à não habilitação da transição *execute* e, por consequência, de todas as transições associadas a

operações.

Dado que as redes de Petri, inclusive as redes de Petri coloridas, não permitem a remoção de todas as marcas de um dado conjunto de lugares, num único disparo de uma transição (não permitem *preemption*), as marcas correspondentes aos objectos destruídos vão continuar presentes nos lugares onde se encontravam aquando da destruição do objecto¹³. Tais marcas irão implicar um processamento adicional que se poderá tornar muito significativo à medida que outros objectos vão sendo criados e destruídos. Caso tal situação se torne prejudicial, deverá ser considerada a hipótese de o simulador remover todas as marcas associadas aos objectos removidos. Tal não modificará o comportamento da rede, dado essas marcas não poderem nunca ser utilizadas devido à existência da fusão com a transição *execute*, *readAndLock* ou *writeAndUnlock*. No entanto, simplificarão o modelo permitindo uma execução mais rápida devida à diminuição da quantidade de vínculos a calcular em cada passo de execução.

A modularidade permitida pela adição de páginas é muito adequada à especificação de ciclos de vida mais complexos de forma modular. Dessa forma, torna-se possível dispor de um conjunto de ciclos de vida alternativos sem que tal interfira na especificação da respectiva classe canónica ou na especificação do comportamento. Estes ciclos de vida podem manter-se como páginas separadas: temos assim uma classe canónica e um módulo para a especificação de cada ciclo de vida. A Fig. 5.6 ilustra dois exemplos de ciclos de vida:

1. O primeiro (Fig. 5.6a) corresponde a uma versão do utilizado na classe canónica com destruição (Fig. 5.2b) mas agora com suporte para a criação e destruição assíncronas permitidas pela existência dos lugares *AssyncCreate* e *AssyncDestroy*. Neste caso a transição *create* já não é invocada (por um pedido *create*). Em vez disso depende do lugar *AssyncCreate* que é feito parte de uma fusão de lugares. Este lugar está exemplificado na Fig. 5.6a. Naturalmente, pode utilizar-se um idioma semelhante para a destruição de objectos: tal hipótese está representada na Fig. 5.6a pelo lugar *AssyncDestroy*.
2. O segundo (Fig. 5.6b) é um exemplo de um ciclo de vida mais complexo, com mais do que dois estados. Neste caso modelam-se objectos com quatro estados no seu ciclo de vida: (1) *toBeCreated* (não representado na Fig. 5.6 por constituir parte da classe canónica); (2) *ready*; (3) *executable* e (4) *suspended*. Neste caso, parte das transições entre estados no ciclo de vida serão, tipicamente, impostas por um objecto externo, nomeadamente um escalonador¹⁴ que imporá algum tipo de alternância nos estados de execução dos objectos.

Note-se que em ambos os exemplos e para simplificar a figura, omitiram-se as transições *readAndLock* e *writeAndUnlock*. Esta separação entre os módulos para leitura e escrita dos objectos e a página que especifica o ciclo de vida é facilmente suportada pelas redes de Petri coloridas componíveis. Com efeito, as transições *readAndLock* e *writeAndUnlock*, por um lado, e a

¹³Da destruição do objecto apenas resultou a remoção da respectiva marca no lugar *executable*.

¹⁴Em inglês: *scheduler*.

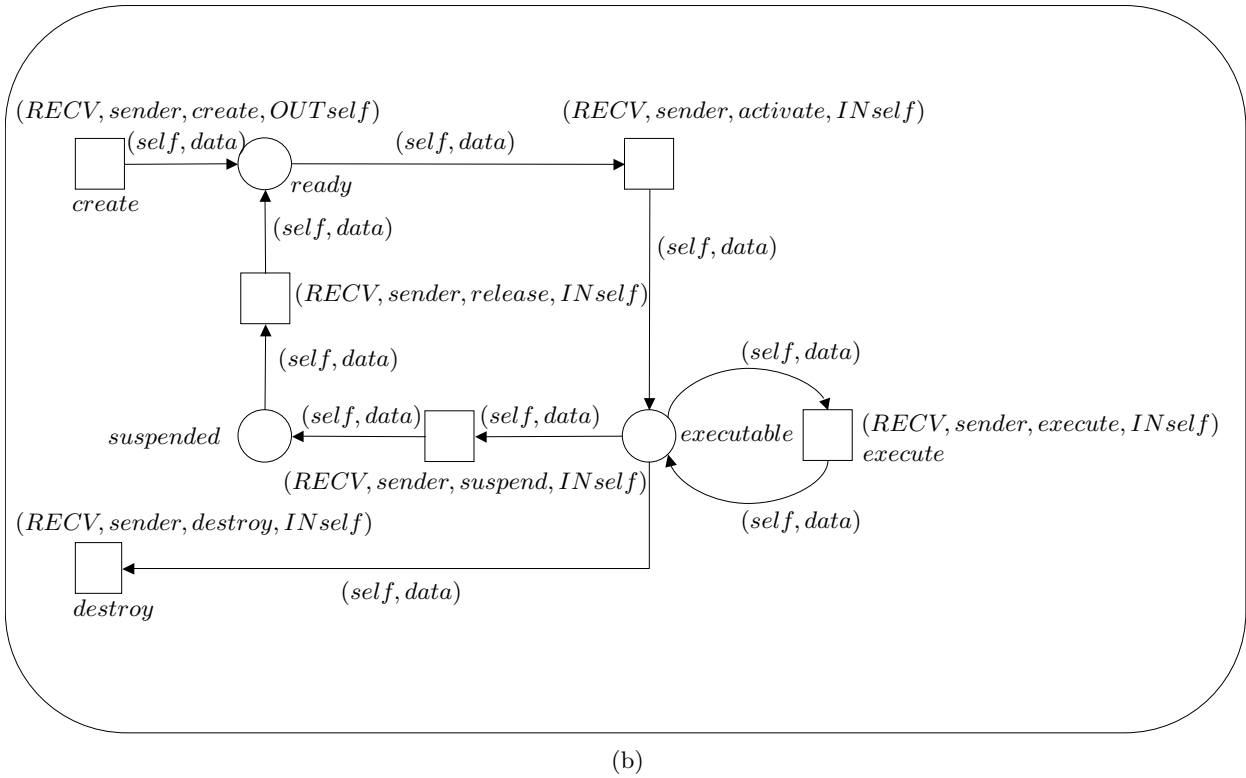
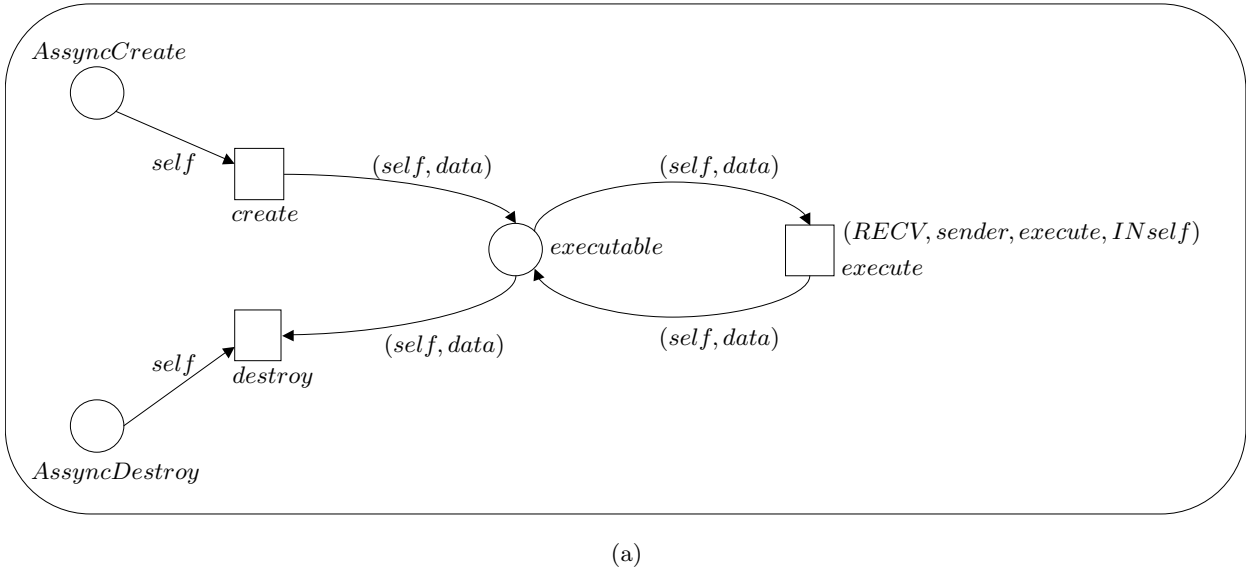


Figura 5.6: Dois ciclos de vida especificados por módulos adicionáveis a classes canónicas.

transição *execute* por outro, podem ser vistas como adições à rede que implementa o ciclo de vida propriamente dito. Por isso, as redes na Fig. 5.3 na pág. 129 são vistas como módulos: redes que não constituem uma classe mas que são adicionadas a outra rede (classe ou módulo).

Finalmente, apresenta-se na Fig. 5.7 uma rede canónica que utiliza o ciclo de vida de quatro estados da Fig. 5.6b.

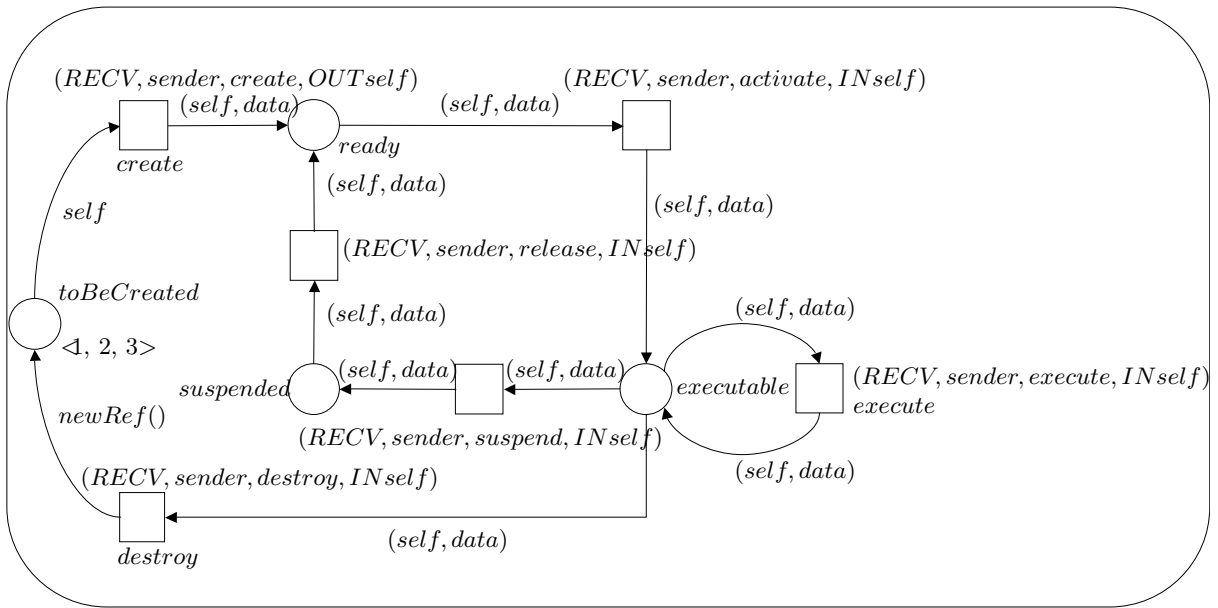


Figura 5.7: Rede canónica com um ciclo de vida com 4 estados.

5.2.2 Classificação

A classificação fornece a associação entre um conjunto de instâncias idênticas e uma classe. É um conceito extremamente intuitivo pois é frequente a necessidade de agrupar coisas semelhantes numa única classe. A semelhança inclui não apenas a estrutura, mas também o comportamento e as funcionalidades dele decorrentes. Por exemplo: um conjunto de seres humanos ou um conjunto de elevadores num prédio. As redes de Petri facilmente especificam classificações. Por um lado, a estrutura da rede força um mesmo conjunto de possíveis comportamentos (e funcionalidades) entre as várias marcas, que representam os objectos da classe, e que fluem ao longo da rede. Por outro lado, através da utilização de guardas, expressões nos arcos ou ambas, as redes de Petri coloridas podem também modelar comportamentos específicos para alguns objectos. Basicamente, cada rede é uma classe e cada objecto é associado a uma cor que representa os seus atributos e a referência para si próprio a qual o identifica.

A modelação de cada objecto por uma marca permite a especificação do comportamento de todos os objectos por uma única rede de Petri que representa a classe desses objectos. A rede de Petri torna-se a sobreposição ortogonal dos comportamentos de todos os objectos da classe. Por outras palavras, a rede de Petri é o sistema de objectos. Não há necessidade de pensar em instâncias distintas de um mesmo diagrama para modelar o comportamento de cada objecto, como sucede, por exemplo, com a utilização dos estadogramas no contexto da linguagem UML [OMG, 2003].

5.2.3 Atributos

Conforme já mencionado, os atributos de cada objecto são especificados nas marcas, fazendo por isso parte da definição dos tipos de cor. As marcas que representam objectos incluem o número de instância de cada objecto e os seus atributos. O número de instância corresponde ao identificador de cada objecto, ou seja, o *self* ou *this* de várias linguagens de programação orientadas pelos objectos. Também a execução concorrente interna¹⁵ de objectos pode ser modelada de forma intuitiva pelas redes de Petri, nomeadamente devido à visualização gráfica de garfos¹⁶ e junções¹⁷.

Os atributos de classe são especificados por uma marca única que é colocada num lugar específico, denominado LAC (Lugar para Atributos da Classe) cuja aplicação será exemplificada na Fig. 5.9. As modificações aos atributos da classe são efectuadas retirando, modificando e voltando a colocar a marca com os atributos modificados no mesmo lugar. Este idioma é semelhante ao descrito para modificação dos atributos de um dado objecto utilizando as transições *readAndLock* e *writeAndUnlock*. Tipicamente serão utilizados grupos de sincronismo associados tal como ilustrado na Fig. 5.9. Implicitamente, estas transições controlam também a visibilidade associada aos atributos da classe. Por exemplo, se um atributo não surge como parâmetro em nenhum evento receptor então esse atributo não é visível do exterior.

5.2.4 Pedidos

A troca de pedidos, ou *mensagens*, entre objectos constitui uma abstracção fundamental no desenvolvimento orientado pelos objectos. Após a estruturação em classes, o sistema diz-se orientado pelos objectos fundamentalmente porque todo o seu comportamento resulta de trocas de pedidos entre os vários objectos.

Um pedido é "uma mensagem inicial de um objecto para outro" [OMG, 2003, págs. 2-311]. A especificação da linguagem UML apresenta a seguinte definição para *sinais* e *operações*:

"Existem vários tipos de pedidos entre instâncias (e.g., envio de um sinal e invocação de uma operação). O primeiro é utilizado para desencadear uma reacção no receptor de uma forma assíncrona e sem que exista uma resposta, enquanto que o último aplica uma operação a uma instância, o que pode ser feito de forma síncrona ou assíncrona e pode obrigar a uma resposta do receptor para o emissor. Outros tipos de pedidos são utilizados, por exemplo para criar uma nova instância ou para apagar uma instância já existente." [OMG, 2003, pág. 2-110].

¹⁵Em inglês: *intra-object concurrency*.

¹⁶Em inglês: *forks*.

¹⁷Em inglês: *joins*.

A criação e destruição de instâncias são modeladas, respectivamente, pelas já referidas transições *create* e *destroy* (*vide* Secção 5.2.1). O envio de um sinal é modelado pela criação de grupos de fusão entre lugares e as invocações de operações são modeladas por grupos de sincronismo. As secções seguintes detalham estas modelações.

Pedidos Assíncronos

Tal como definidos na especificação da linguagem UML, os sinais parecem implicar alguma forma de *broadcast* de eventos. O efeito não local da comunicação *broadcast* acarreta dificuldades derivadas de uma das características intrínsecas das redes de Petri: o seu efeito local.

De forma a evitar extensões às redes de Petri, e portanto também às redes de Petri coloridas, propõe-se aqui a utilização de memória partilhada como suporte à comunicação por sinais, dado que a memória partilhada pode ser modelada por fusão de lugares. Mais especificamente, cada sinal é modelado por um conjunto de fusão de lugares. Cada lugar pertencente a um conjunto de fusão é anotado com o identificador desse conjunto de fusão.

A especificação da linguagem UML define sinal como "A especificação de um estímulo assíncrono que é comunicado entre duas instâncias." [OMG, 2003, pág. Glossary-13]. Uma interpretação estrita desta definição vê um sinal como um pedido assíncrono entre dois objectos, em que um dos objectos conhece o outro. Esta interpretação pode ser facilmente modelada por uma fusão de lugares entre o lugar onde o pedido é depositado e o lugar de onde o pedido é retirado para ser tratado. Conforme já referido, a fusão de lugares também suporta o *broadcast* de sinais em que um objecto envia um pedido assíncrono para um conjunto conhecido de objectos receptores, sob a forma de várias marcas idênticas. No caso mais simples, é permitido a qualquer dos receptores consumir todas estas marcas, o que impedirá os restantes de receberem o pedido. Caso tal não seja desejável, cada marca pode ser identificada com o objecto receptor, de forma que apenas este a possa consumir. Neste caso, será necessário o emissor conhecer todos os receptores. No caso mais geral em que o emissor não conhece os receptores terá ainda assim de conhecer a sua quantidade de forma a efectuar o *broadcast* dessa quantidade de marcas. Os receptores estarão obrigados a só consumir uma marca.

Pedidos Síncronos

Os pedidos síncronos são especificados por grupos de sincronismo. Estes suportam sincronização entre vários processos (*vide* Fig. 5.1), tal como o usual nas álgebras de processos (e.g. [Roscoe et al., 1997; Milner, 1995]). No entanto, mais interessante para a modelação orientada pelos objectos é o facto de os grupos de sincronismo satisfazerem o duplo objectivo de se manterem próximos das redes de Petri e de permitirem a modelação de invocações de métodos com passagem de parâmetros tal como frequentemente suportado pelas linguagens de desenho e pro-

gramação orientadas pelos objectos, em particular as linguagens UML, C++, Java e outras muito utilizadas.

De acordo com a especificação da linguagem UML, se o pedido for síncrono o receptor tem necessariamente de possuir um ponto de resposta¹⁸ bem definido. As redes de Petri coloridas componíveis podem especificar esta obrigatoriedade utilizando um par de pedidos: do emissor para o receptor (activação do método), e do receptor para o emissor (retorno do método). Esta segunda sincronização é também a única forma de o emissor "(...) suspender a execução até a actividade invocada pelo pedido alcançar um ponto bem definido e enviar uma resposta de volta para o emissor" [OMG, 2003, págs. 2-311].

A Fig. 5.8 ilustra esta dupla sincronização, semelhante à proposta nas OCP-nets de Maier e Moldt [Maier e Moldt, 2001], para modelar a invocação e o retorno de métodos.

Note-se que os parâmetros c e obj , na *ClasseB*, identificam o objecto emissor (instância) na *ClasseA*. Tal permite a resposta do receptor para o emissor através de um segundo grupo de sincronismo (r_2) entre t_4 e t_2 .

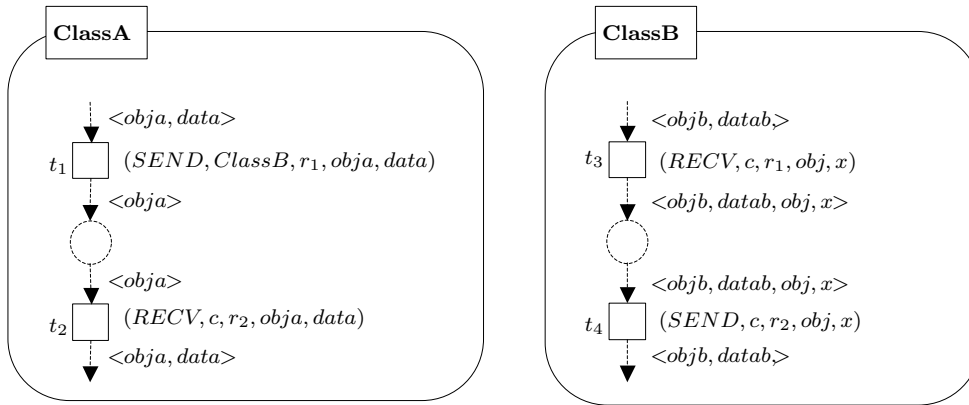


Figura 5.8: Sincronização dupla para uma invocação não-atômica.

Se o método não devolve dados e a sua execução deve ser atômica então é suficiente associar um *code segment* à transição receptora. Este *code segment* deve ser escrito na linguagem de inscrições de forma a melhor se integrar com as restantes inscrições. Estes *code segments* são suportados pelas ferramentas Design/CPN [s.a., 2004c] e CPN Tools [s.a., 2004b].

Os métodos de classe são modelados de forma implícita pelos pedidos que não especificam número de instância. Tipicamente, estes pedidos implicam aceder ao lugar LAC onde se especificam os atributos da classe. A Fig. 5.9 exemplifica este tipo de invocação e acesso ao lugar LAC.

Daqui em diante será utilizada a expressão "a transição t_1 invoca a transição t_2 " para significar que um evento emissor associado à transição t_1 tem um correspondente evento receptor associado à transição t_2 .

¹⁸Em inglês: *reply point*.

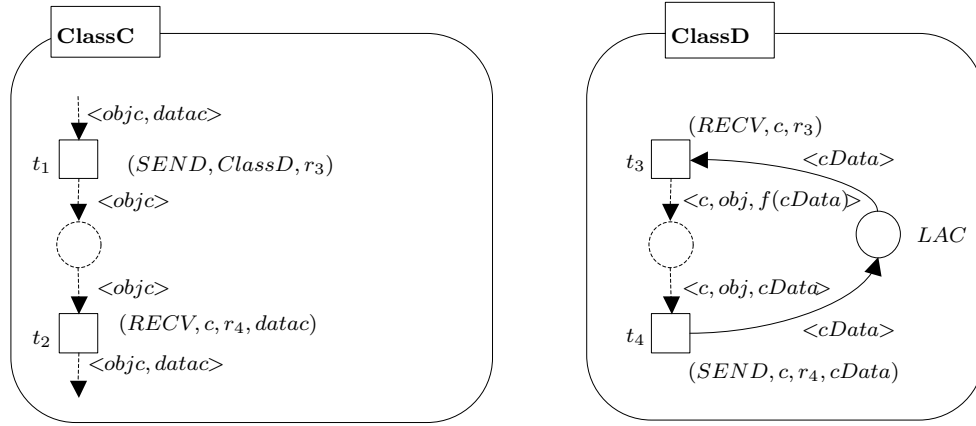


Figura 5.9: Sincronização dupla para invocação de uma operação de classe.

5.2.5 Composição

A composição é uma abstracção particularmente importante no desenvolvimento orientado pelo objectos e tal foi já fortemente defendido quer por ser considerada preferível à herança (e.g. [Gamma et al., 1995, págs. 18-20]), quer por ser vista como a abstracção fundamental no desenvolvimento orientado pelos objectos [Steimann et al., 2003]. De facto, comparativamente à herança, a composição não é claramente motivo de tantas polémicas quanto à sua utilização, nomeadamente não apresenta numerosas variações quer em termos teóricos, quer em termos do suporte oferecido pelas linguagens de programação (e.g. [Taivalsaari, 1996]). Outro ponto em favor da composição é a sua simplicidade claramente resultante da ubiquidade do conceito.

Consideramos dois tipos de composição: (1) um em que o **componente** é criado e destruído juntamente com o **composto**¹⁹; (2) outro em que o componente pode ser criado antes ou depois do composto e pode também ser destruído antes ou após o composto. O primeiro é frequentemente denominado **composição forte**²⁰ ou apenas **composição**²¹. O segundo é normalmente denominado **agregação**²², composição fraca, ou até, simplesmente, **associação**²³. No entanto, muitas vezes os últimos dois termos (agregação e associação) não se encontram bem definidos. Segundo Steimann, tal sucede também na especificação da linguagem UML [Steimann et al., 2003].

A Fig. 5.10 ilustra e permite comparar a especificação de composição forte (Fig. 5.10a) e agregação (Fig. 5.10b) entre dois objectos através de modelos abreviados. Mais especificamente, ilustra o caso em que o objecto componente é criado e destruído pelo objecto composto, algures durante a existência deste. Tal é conseguido utilizando grupos de sincronismo que modelam a invocação das operações *create* e *destroy*. Este idioma mimetiza a, por vezes implícita,

¹⁹Em inglês: *composite*.

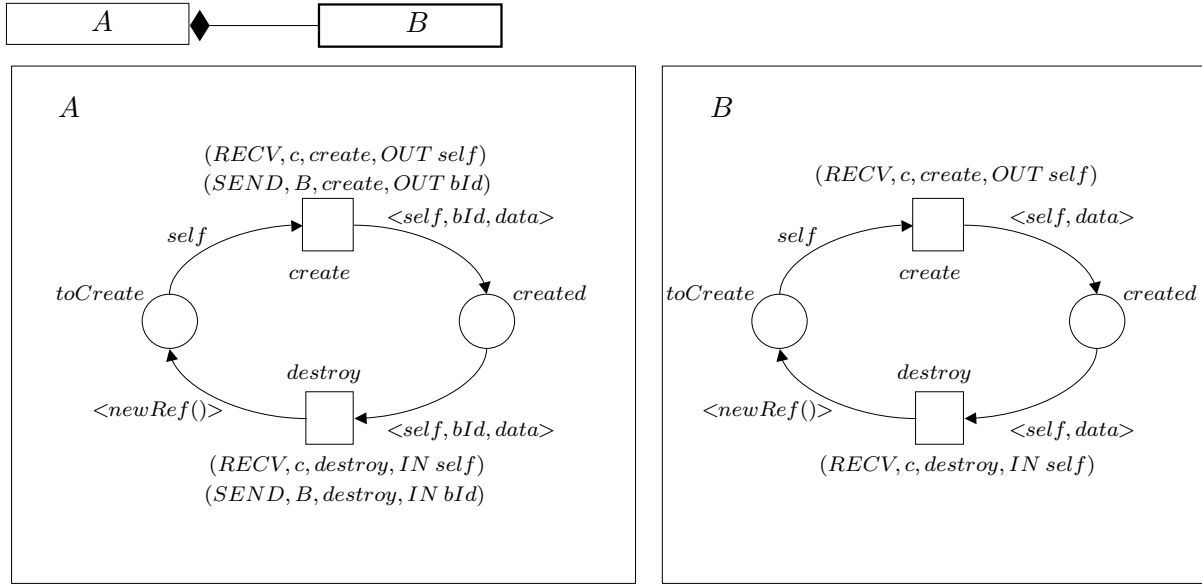
²⁰Em inglês: *strong composition*.

²¹Em inglês: *composition*.

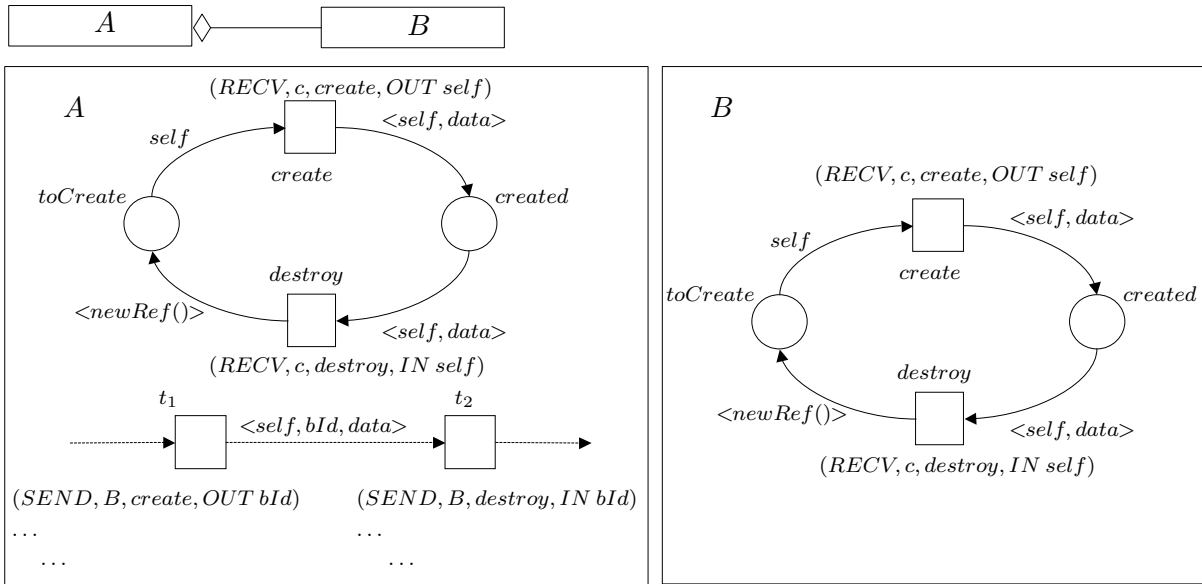
²²Em inglês: *aggregation*.

²³Em inglês: *association*.

invocação de construtores e destrutores nas linguagens de programação orientadas pelos objectos. A composição é caracterizada pela criação e destruição simultâneas do objecto composto e do objecto componente. A agregação é a forma mais geral de composição e não pressupõe a invocação das operações *create* ou *destroy* do componente pelas operações *create* ou *destroy* do composto.



(a)



(b)

Figura 5.10: a) Composição e b) agregação.

A distinção entre composição forte e agregação é claramente visível na rede de Petri: as transições *t1* e *t2* (Fig. 5.10b) são parte da classe A (ou de um seu módulo de comportamento, assumindo os idiomas da Fig. 5.5); os objectos da classe B são criados após a criação dos objectos da classe

A (na transição t_1) e destruídos (na transição t_2) antes da destruição dos objectos da classe A.

5.2.6 Generalização

A aplicação que provavelmente mais popularizou este tipo de abstracção foi o trabalho de Carl Linnaeus (e.g. [Natural History Museum, 2005]), o qual permitiu aos biólogos relacionar um grande número de espécies animais através de generalizações. De facto, a generalização é uma abstracção muito utilizada também no desenvolvimento orientado pelos objectos. É por vezes considerada a abstracção mais característica do desenvolvimento orientado pelos objectos provavelmente por se tratar daquela que é mais distintiva relativamente a metodologias não orientadas pelos objectos. Quando acompanhada de **vínculação dinâmica**²⁴ permite a distinção entre o desenvolvimento orientado pelos objectos e o **desenvolvimento baseado em objectos**²⁵.

Na linguagem UML, a definição do conceito de generalização é muito genérica:

”(A) Generalização é a relação taxonómica entre um elemento mais geral (o pai) e um elemento mais específico (o filho) que é totalmente consistente com o primeiro elemento e adiciona informação adicional.” [OMG, 2003, págs. 3-86]

Nas linguagens de programação orientada pelos objectos mais comuns (e.g. C++ e Java), ”totalmente consistente” corresponde à igualdade de assinaturas²⁶ entre métodos dos objectos da classe pai e métodos dos objectos da classe filho. Neste caso, em termos do suporte oferecido pelas rede de Petri colorida componível, basta aumentar a rede filho com os elementos da rede pai, ao mesmo tempo que se mantêm todos os eventos *RECV* da rede pai. Este aumento da rede filho pode ser especificado por sincronização entre transições na rede filho e transições na rede pai. Contudo, é conhecido que a concordância das assinaturas dos métodos pode não ser suficiente para os fins pretendidos, nomeadamente para garantir a possibilidade de substituição de tipos. Tal sucede porque a classe filho (a classe especializada) pode modificar o comportamento de um método específico da classe pai. Tal tem sido referido no contexto dos sistemas distribuídos (e.g. [Fischer e Wehrheim, 2000]) onde a **conformidade de comportamento**²⁷ é considerada especialmente importante [Harel e Kupferman, 2002]. Este tema tem sido estudado extensivamente também no contexto das redes de Petri [van der Aalst e Basten, 1997; Balzarotti et al., 1999] mas nenhuma solução efectivamente prática, ou seja, que permita uma forma simples de especificar herança em modelos em redes de Petri é do conhecimento do autor. Também não parece existir nenhuma ferramenta que suporte algum tipo de herança entre modelos em redes de Petri.

²⁴Em inglês: *dynamic binding*.

²⁵Ao leitor interessado neste tipo de classificações sugere-se a leitura do texto de Bergin [Bergin, 1997] que actualiza o de Wegner [Wegner, 1987].

²⁶Em inglês: *signatures*.

²⁷Em inglês: *behaviour conformity*.

Por outro lado, a **herança de tipos** (ou **herança de interfaces**) pode ser conseguida facilmente quando a rede pai se limite a definir transições receptoras puras (sem eventos emissores associados).

Quando qualquer rede pode ser rede pai, a especificação de uma herança que garanta a conformidade de comportamento entre tipos, ou seja, o princípio de Liskov [Barbara H. Liskov e Jeannette M. Wing, 1994], acaba por ficar exclusivamente na dependência do modelador.

Refira-se que também nas OCP-nets de Maier e Moldt [Maier e Moldt, 2001] é proposta a utilização de herança de subtipos sem exigência de conformidade de comportamentos.

Herança na Modelação com Redes de Petri Coloridas Componíveis

A seguinte lista resume a modelação de três tipos de herança comuns utilizando redes de Petri coloridas componíveis:

1. **Herança de Subtipos (implementação ou realização de *interfaces*)**. A interface não é especificada por uma rede de Petri colorida componível. Apenas as classes derivadas, que realizam a interface, têm necessariamente de especificar eventos *RECV* para cada uma das operações da interface.
2. **Herança de classes abstractas sem definição de operações ("interfaces com atributos")**. Semelhante à anterior. As classes abstractas também não são especificadas por uma rede de Petri colorida componível mas cada uma das suas classes derivadas tem de especificar um evento *RECV* para cada operação. Além disso, os seus objectos têm de conter os atributos da classe abstracta.
3. **Herança de classe concreta sem redefinição de operações e sem partilha dos atributos com a classe filha**. Pode ser especificada por adição entre uma instância da classe pai e a classe filho. Tal é realizado por fusão das respectivas transições *create* e das respectivas transições *destroy*. Na transição *create* resultante, o evento receptor da classe pai é removido e a guarda resultante é acrescida do teste de igualdade entre o identificador do objecto da classe pai e o objecto da classe filho. A Fig. 5.11 na pág. 144 ilustra, de forma abreviada, esta forma de herança. Note-se que esta forma de herança suporta a existência de operações com igual interface em ambas as classes. Através da adição descrita, estas operações mantêm-se na rede resultante. Caso se pretenda a semântica usual em que a operação da classe derivada substitui a da classe pai, então a operação deve ser removida da classe pai (removendo o evento *RECV* respectivo) e deve manter o evento da classe derivada. Ainda assim, é necessário que o modelador garanta que a parte do modelo associada à especificação da operação na classe pai não afecta de forma indesejável o funcionamento global do modelo.

Tal como nas linguagens de programação, a herança de subtipos é extremamente simples: a classe pai limita-se a especificar um conjunto de operações que têm de ser implementadas pela classe filha (derivada). Assim, a página na rede de Petri colorida componível correspondente à classe derivada define um evento *RECV* para cada operação a implementar, tal como definida na classe pai (ponto 1). Quando também se pretendam herdar dados (ponto 2), a classe filha define igualmente esses dados. Em qualquer destes dois casos, a classe pai (a interface ou classe abstracta) não é especificada por uma rede de Petri.

O ponto 3 respeita à situação em que pretende definir uma herança de uma classe concreta, mas em que os atributos na classe pai não podem ser modificados pela classe filho. Tal pode ser modelado por adição entre uma instância da classe pai e a classe filho. A classe filho passa a ser o resultado desta adição. Duas transformações simples ao nível das anotações permitem efectivar a relação de herança através da fusão das transições *create* e, se necessário, também das transições *destroy*: (1) a remoção do evento *RECV* na classe pai e (2) a criação de uma guarda que força a igualdade entre as referências dos objectos na classe pai e na classe filho, que se fundem devido à relação de herança. Conforme já referido, esta adição está ilustrada na Fig. 5.11 na pág. 144.

A generalização, sob a forma mais frequentemente suportada pelas linguagens orientadas pelos objectos, a **herança de implementação**, tem desvantagens já bem conhecidas e documentadas (e.g. Gamma et al. [1995]). Resumidamente, estas resultam do facto das hierarquias de classes resultantes de relações de herança quebrarem o encapsulamento das classes, ou seja a ocultação de informação. Com efeito, as classes que herdam passam a ter acesso a atributos e métodos das classes de onde herdam. Tal cria novas dependências entre classes. Tal como referido em [Gamma et al., 1995], "as classes pai definem frequentemente pelo menos parte da representação física das suas subclasses". Tal significa também que a herança fornece às subclasses, acesso aos detalhes da implementação da classe pai.

Para esta herança de implementação, comum nas linguagens de programação orientadas pelos objectos, não foi possível identificar uma forma "natural" e elegante de a especificar utilizando redes de Petri coloridas componíveis. Tal pode ser visto como um sinal da natureza fundamentalmente pouco encapsulada desse tipo de abstracção: o impedimento para uma especificação elegante resulta directamente da necessidade de herdar atributos, precisamente a característica que torna a herança de implementação pouco recomendável. A secção seguinte discute este tema.

Herança *versus* Composição

A escolha entre herança de implementação e composição no desenvolvimento orientado pelos objectos é subjectiva e como tal torna-se por vezes polémica. O próprio Bertrand Meyer, um dos principais e mais aguerridos defensores do desenvolvimento orientados pelos objectos admite

a dificuldade de escolha entre ambas as técnicas (*vide* em particular o Capítulo 24 do seu livro [Meyer, 1997]).

Por outro lado, a herança de implementação é frequentemente considerada algo a evitar. Em seu lugar é normalmente proposta a composição. A herança fica nesse caso reduzida à **herança de interfaces** (ou **herança de subtipos**). Tal permite a especificação de interfaces (conjuntos de assinaturas de métodos públicos) que devem obrigatoriamente ser suportadas pelas classes que implementam essas interfaces.

Pelos mesmos motivos, a recomendação de não utilização de herança de implementação aplica-se à modelação com redes de Petri coloridas componíveis. É preferível a utilização de relações de composição e agregação, em lugar de relações de herança. Esta deve ser preferida na sua forma mais simples: a herança de interface. Atente-se no seguinte texto:

”O favorecimento da composição de objectos em relação à herança de classes ajuda-nos a manter cada classe encapsulada e focada numa tarefa. As classes e hierarquias de classes manter-se-ão pequenas e será menos provável que cresçam até se tornarem monstros intratáveis. Por outro lado, um desenho baseado na composição de objectos terá mais objectos (ainda que menos classes), e o comportamento do sistema dependerá das suas inter-relações em lugar de estar definido numa classe.” [Gamma et al., 1995, pág. 19].

Esta ênfase nas relações entre objectos é a adequada à modelação com redes de Petri coloridas componíveis por duas razões:

1. Coloca a complexidade do sistema não na decomposição hierárquica de um conjunto de métodos/procedimentos (como é característico da programação estruturada e metodologias associadas), nem na definição de um relativamente reduzido número de classes, cada qual com muitos métodos, mas sim nas relações entre objectos, preferencialmente de classes distintas.
2. A modularidade baseada nas classes e nas relações entre objectos — preferencialmente de classes distintas — é directamente suportada pela noção de uma rede (página) para cada classe, pelas fusões de lugares e pelos grupos de sincronismo.

Por estas razões a herança de implementação em redes de Petri deve ser evitada e julgamos que este atitude não deve sequer constituir motivo de preocupação. Na linha do defendido por Brooks (*vide* citação no início da Secção 5.1 na pág. 112) considera-se que o encapsulamento em módulos, mais a sua reutilização, são os conceitos fundamentais. Com base neste encapsulamento, a troca de mensagens (outro dos conceitos fundamentais do desenvolvimento orientado pelos objectos), suportada pelos grupos de sincronismo, permite o suporte às restantes abstracções, nomeadamente a composição, a agregação, a delegação e, inclusive, o polimorfismo.

Em resumo, considera-se que a modularidade e o seu aproveitamento, através de composições, constitui a abstracção mais útil do ponto de vista prático. Por esta razão, as redes de Petri coloridas componíveis oferecem uma variedade alargada de mecanismos de composição: para além dos mecanismos presentes nas redes de Petri coloridas hierárquicas, dispõem também de grupos de sincronismo e da adição de módulos. Em particular, propõe-se a utilização da operação de adição de redes (Def. 3.20 na pág. 75). Se desejável, a adição de redes — numa forma adaptada às redes de Petri coloridas componíveis — pode ser utilizada para suporte à modelação de herança. Eventualmente, pode também ser utilizada para a modelação da composição. Como os grupos de sincronismo são utilizados para modelar as interacções entre objectos (a parte dinâmica), torna-se possível obter uma separação lógica entre os aspectos estáticos da composição de módulos e os aspectos dinâmicos resultantes da interacção dinâmica entre objectos.

5.2.7 Classes Genéricas, Navegabilidade, Visibilidade e Invocação Múltipla

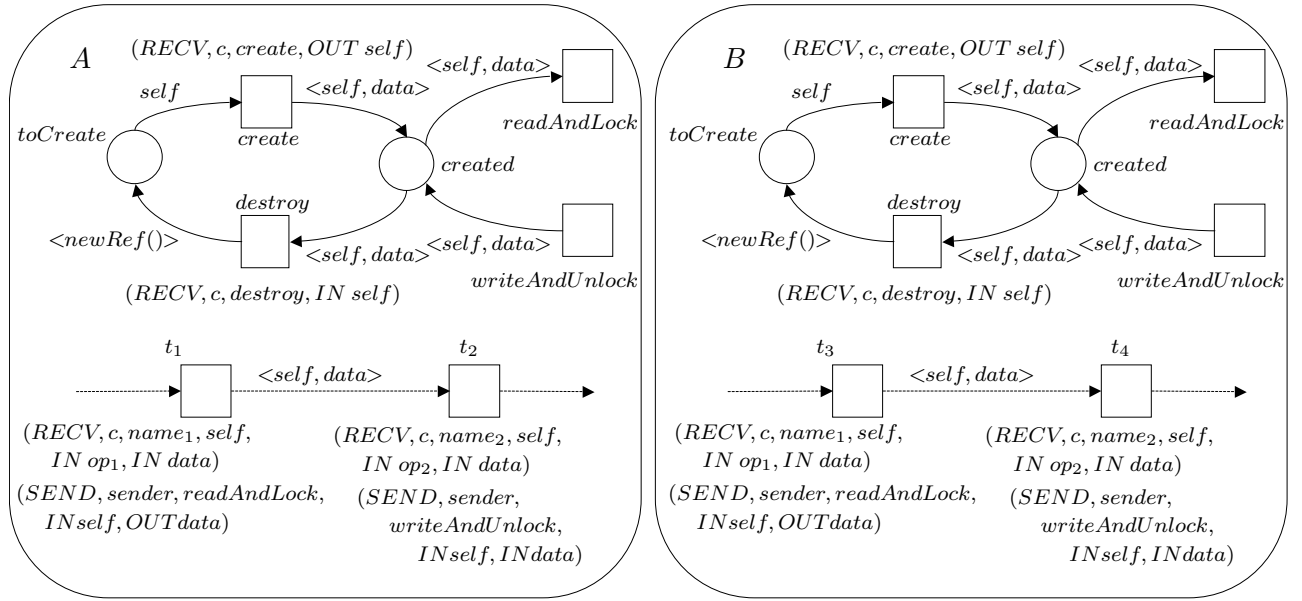
Outros conceitos que frequentemente se encontram no desenvolvimento orientado pelos objectos e, em particular, nas linguagens de especificação e programação orientadas pelos objectos também podem facilmente ser suportadas pelas redes de Petri coloridas componíveis. Nesta secção referem-se de forma muito resumida, vários destes conceitos, nomeadamente as classes genéricas, navegabilidade, visibilidade e invocação múltipla.

O suporte para classes genéricas pode facilmente ser adicionado. Para tal basta adicionar a necessária notação para que cada página suporte uma lista de parâmetros associados. Parâmetros estes que poderão ser utilizados como conjuntos de cores ou como classes associadas aos eventos. A este respeito veja-se a proposta de Mailund para a parametrização de redes de Petri coloridas [Mailund, 1999].

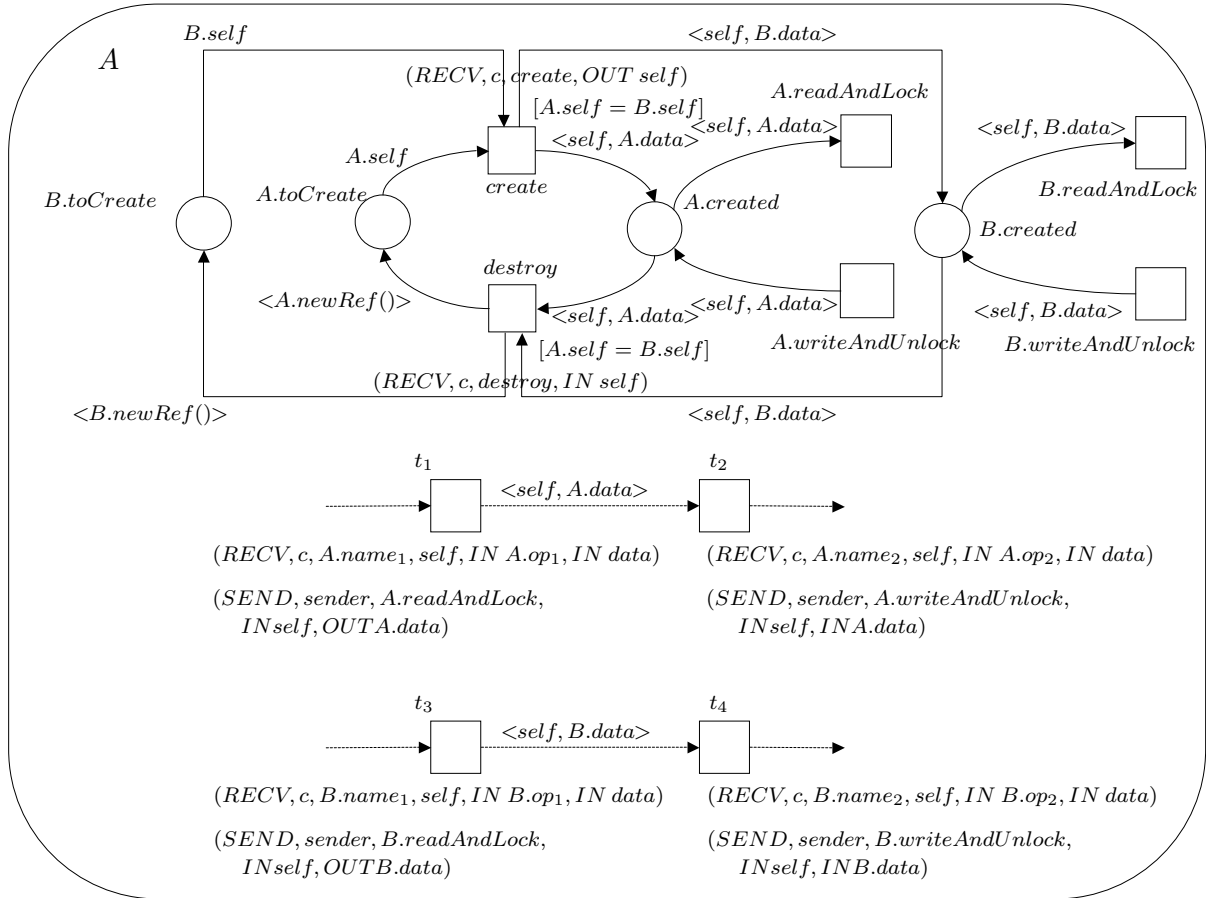
Tal como nas linguagens de programação orientadas pelos objectos, a navegabilidade entre objectos resulta da troca de mensagens através da utilização de eventos *SEND* e *RECV*.

A visibilidade é suportada ao nível da fornecida pelos qualificadores *public* e *private*, ou seja, a distinção entre elementos visíveis e invisíveis a partir do exterior da classe. As transições visíveis têm eventos *RECV* associados. À partida todos os lugares se encontram visíveis pois é possível incluí-los em conjuntos de fusão com lugares de outra classe (página). Mais uma vez, a simples adição de notações extra, pode permitir uma especificação mais detalhada para a visibilidade.

A invocação múltipla corresponde à fusão conjuntiva de transições. É suportada pela especificação de vários eventos emissores associados a uma mesma transição e também, de forma mais indirecta, pela possibilidade de associar um evento receptor a transições com eventos emissores, o que origina cadeias de invocação.



(a)



(b)

Figura 5.11: (a) Duas redes de Petri coloridas componíveis canónicas utilizando o idioma na Fig. 5.5b e (b) nova rede resultante de uma relação de herança entre A e B.

Capítulo 6

Redes de Petri e Objectos — Dois Exemplos

Este capítulo apresenta dois exemplos de modelação utilizando redes de Petri coloridas componíveis. Para tal recorre à ferramenta CPN Tools. O primeiro modelo é um exemplo simples de um sistema inspirado nos modelos clássicos produtor-consumidor e leitores-escritores. Para teste deste exemplo foi desenvolvido um protótipo que gera uma rede de Petri colorida a partir de uma rede de Petri colorida com grupos de sincronismo. O resultado da sua aplicação é apresentado neste capítulo. A segunda secção do capítulo compara um modelo da literatura, construído com redes de Petri coloridas hierárquicas, com um modelo equivalente construído com redes de Petri coloridas componíveis e aqui apresentado. Ambos modelam um mesmo sistema para controlo de um sistema de elevadores.

Actualmente, a CPN Tools é a única ferramenta que implementa as redes de Petri coloridas tal como definidas por Jensen [Jensen, 1997c,a,b]. Por outro lado, é provavelmente a ferramenta mais utilizada para modelação utilizando redes de Petri coloridas ou mesmo redes de alto-nível em geral. Cada uma das duas secções deste capítulo apresenta um exemplo que ilustra possíveis desenvolvimentos da CPN Tools e a sua aplicação à modelação de sistemas. Para tal utilizam-se conceitos e técnicas orientados pelos objectos tal como propostos no capítulo anterior sob a capa das redes de Petri coloridas componíveis. Ambos os exemplos se baseiam fortemente na utilização de grupos de sincronismo. O primeiro exemplo serviu também de teste ao desenvolvimento de um protótipo capaz de adicionar suporte para este mecanismo aos modelos gerados pela CPN Tools. O segundo exemplo permite comparar um modelo construído com base numa rede de Petri colorida hierárquica com um modelo equivalente para o mesmo sistema, utilizando redes de Petri coloridas componíveis, ou seja, recorrendo aos grupos de sincronismo e aos idiomas

propostos no Capítulo 5.

Na leitura dos exemplos que se apresentarão importa ter em atenção que uma rede de Petri colorida, tal como definida na CPN Tools, é constituída pelas seguintes quatro partes:

1. Declarações e definições de conjuntos de cores, variáveis e funções, utilizando a linguagem de inscrições.
2. Anotações no modelo gráfico, utilizando as declarações e definições do ponto anterior.
3. A representação gráfica de lugares, transições e arcos.
4. A representação gráfica de elementos gráficos informativos, ou seja que não influenciam o comportamento do modelo. Neste incluem-se caixas de texto, rectângulos, e outros elementos destinados unicamente a melhorar a legibilidade do modelo.

Por razões de simplificação e porque nada de significativo seria acrescentado aos exemplos a apresentar, omite-se a primeira parte em todos os modelos.

6.1 Grupos de sincronismo para as CPN Tools

Nesta secção discute-se a utilização de grupos de sincronismo na versão 1.2.0 da ferramenta CPN Tools [s.a., 2004b]. Em particular, discute-se de que forma podem ser implementados os grupos de sincronismo na CPN Tools, e apresenta-se uma proposta concreta que permitiu o teste prático relativamente à implementação dos grupos de sincronismo propostos no capítulo anterior.

Relativamente à CPN Tools, o protótipo implementado é uma ferramenta externa que transforma o modelo que utiliza grupos de sincronismo num outro modelo com igual comportamento mas sem grupos de sincronismo. A utilização de uma ferramenta externa justifica-se devido ao facto da CPN Tools ser uma ferramenta fechada, ou seja, que não disponibiliza o código fonte ou documentação para programadores que permita a sua modificação, ou sequer a adição de novas funcionalidades na aplicação. O protótipo utilizado foi objecto de uma apresentação numa *workshop* sobre a ferramenta CPN Tools que ocorreu em Aarhus em Setembro de 2004 [Barros e Gomes, 2004a].

6.1.1 Especificação de Eventos *SEND* e *RECV*

Para especificar os eventos *SEND* e *RECV* é utilizado o atributo para especificação de *code segments* associado às transições. O utilizador especifica as declarações *SEND* e *RECV* dentro

de comentários da linguagem de inscrições utilizada pela aplicação CPN Tools, a Standard ML. Para distinguir essas declarações de outros comentários, as declarações são escritas dentro de "(**" e "**)". A secção seguinte ilustra uma primeira proposta para a tradução de grupos de sincronismo para uma única **transição pedido**. Esta proposta segue uma abordagem que não é ainda possível na versão actual da CPN Tools (Setembro de 2005). No entanto, são duas as razões para a sua apresentação: (1) a hipótese de futuro suporte na CPN Tools e também (2) porque constitui uma introdução e justificação para a solução implementada e apresentada na Secção 6.2.

6.1.2 Tradução dos Grupos de Sincronismo

Os grupos de sincronismo aqui considerados são constituídos por um par de transições: uma transição emissora e uma transição receptora. Assim sendo, a sua tradução para uma única transição pedido implica a destruição de duas transições e a criação de uma nova transição resultante da fusão dessas transições. Os arcos que se ligam às transições iniciais passam a estar ligados à transição pedido correspondente. Os parâmetros na declaração *RECV* (parâmetros formais) são textualmente substituídos pelos parâmetros correspondentes na declaração *SEND* (parâmetros actuais). A guarda na transição pedido é a conjunção das guardas nas duas transições, após a substituição dos parâmetros formais.

Para minimizar as alterações gráficas na rede de Petri colorida original, a transição pedido é gerada sob a forma de uma nova página que fica associada à transição emissora: a **página pedido**. Os lugares que se ligam à transição emissora tornam-se lugares *socket*. Os lugares porto (*port places*) correspondentes são criados na transição pedido.

Por último, a transição receptora é removida da sua página inicial e todos os lugares que a ela se encontravam ligados tornam-se lugares de fusão¹. Para cada um destes lugares de fusão é criada uma cópia na página pedido. Estas cópias são então ligadas à transição pedido.

As Figs. 6.1, 6.2, 6.3 e 6.4 exemplificam esta tradução. As Figs. 6.1 e 6.2 mostram duas páginas, cada qual correspondente a uma classe de um modelo inicial. As Figs. 6.3 e 6.4 mostram as mesmas classes após tradução.

A transição **t1** na **ClassA** (*vide* Fig. 6.1) envia um pedido (**req1**) para a transição **t4** na **ClassB**. Também a transição **t6** na **ClassB** (*vide* Fig. 6.2) envia um pedido (**req2**) para a transição **t3** na **ClassA**.

Os lugares **check1**, **check2** e **check3** foram utilizados com o único propósito de evitar erros de sintaxe no modelo inicial, erros esses devidos à utilização de variáveis não vinculáveis nos arcos de saída. Tal sucede com as variáveis que são parâmetros *OUT* em declarações *SEND* (por

¹Em inglês: *fusion places*.

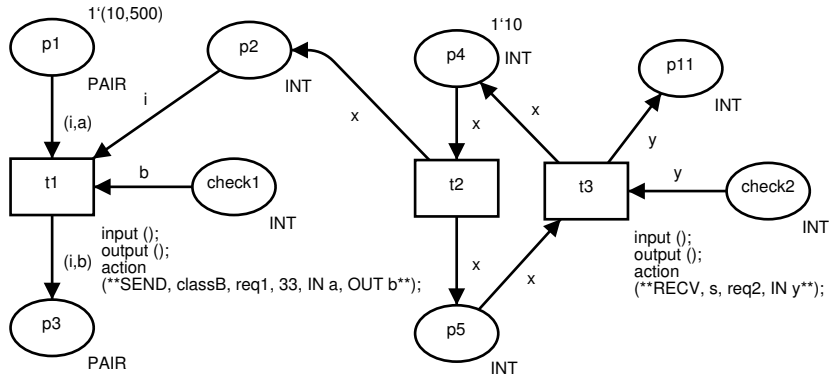


Figura 6.1: Página para a ClassA.

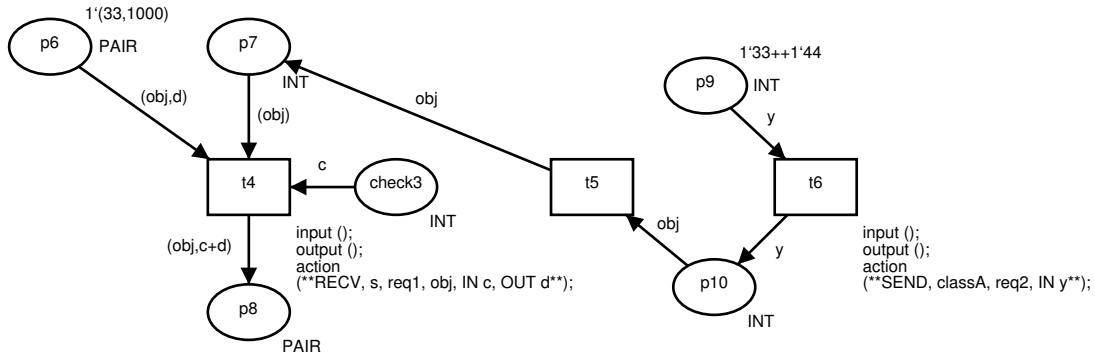


Figura 6.2: Página para a ClassB.

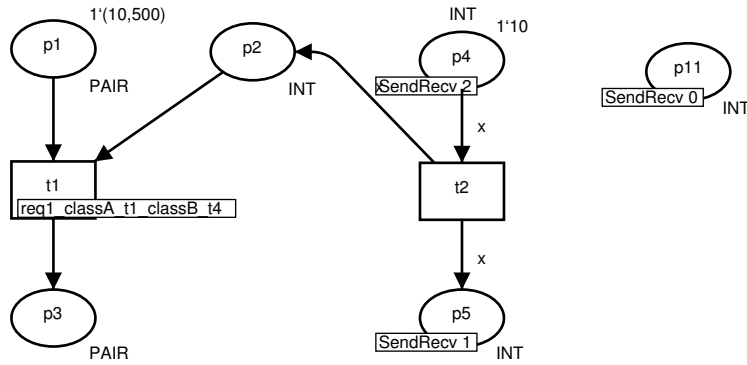


Figura 6.3: ClassA após transformação.

exemplo a variável *b* na *ClassA*); ou parâmetros *IN* em declarações *RECV* (por exemplo a variável *y* na *ClassA* e a variável *c* na *ClassB*).

Este tipo de erros não está presente no modelo final (com transições pedido). Tal deve-se à obrigatoriedade de presença dos parâmetros *OUT* nos arcos de entrada das transições receptoras (e.g. variável *d* na transição *t4* da Fig. 6.2), e dos parâmetros *IN* nos arcos de entrada das transições

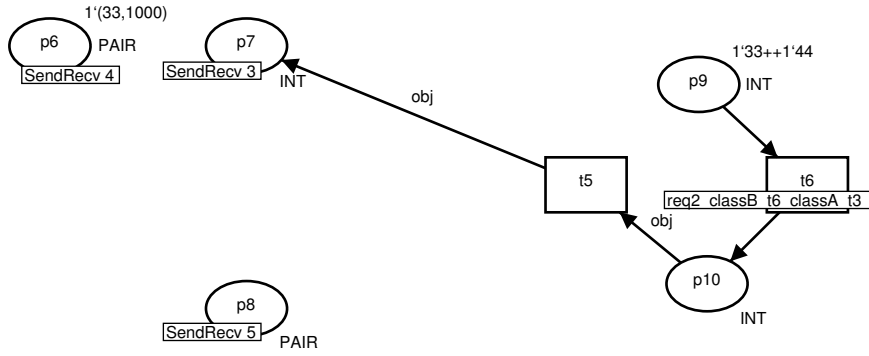


Figura 6.4: ClassB após transformação.

emissoras (e.g. variável y na transição $t6$ da Fig. 6.2)². Por esta razão, os lugares *check* são totalmente opcionais: se estiverem presentes serão removidos aquando da transformação³. Por fim, importa notar que estes lugares não necessitam de quaisquer marcações (cores) associadas: um tipo (conjunto de cores) associado é suficiente.

As Figs. 6.3 e 6.4 mostram as mesmas classes no modelo final. As transições *SEND* $t1$ e $t6$ mantêm-se, mas agora como transições de substituição associadas às páginas pedido respectivas (Figs. 6.5 e 6.6).

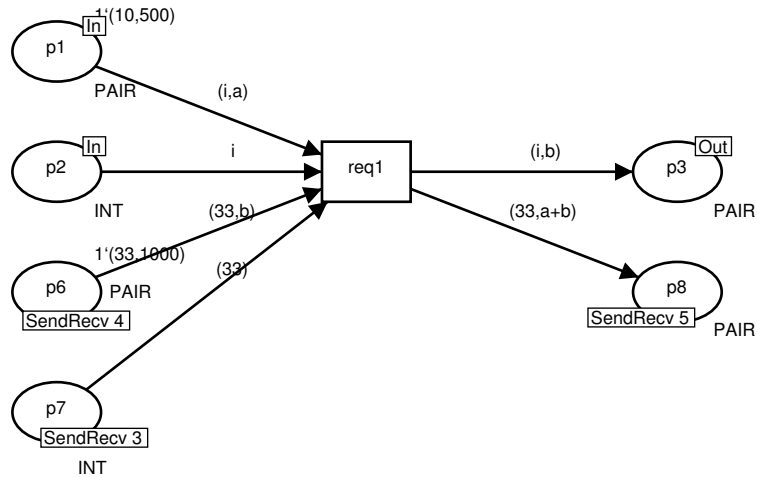


Figura 6.5: Página pedido para pedido $req1$ (página $req1_classA_t1_classB_t4$).

As transições receptoras $t3$ e $t4$ foram removidas e os seus lugares de entrada e saída foram transformados em lugares de fusão. Cada qual é fundido com a respectiva cópia na página pedido associada.

Infelizmente, a CPN Tools não permite que um lugar *socket/port* seja também um lugar de fusão. Tal inviabiliza a estratégia apresentada pois não seria possível tratar lugares que se encontrassem

²A referida obrigatoriedade resulta do ponto 3c da Def. 5.7 na pág. 117.

³O protótipo identifica estes lugares pelo prefixo *check*.

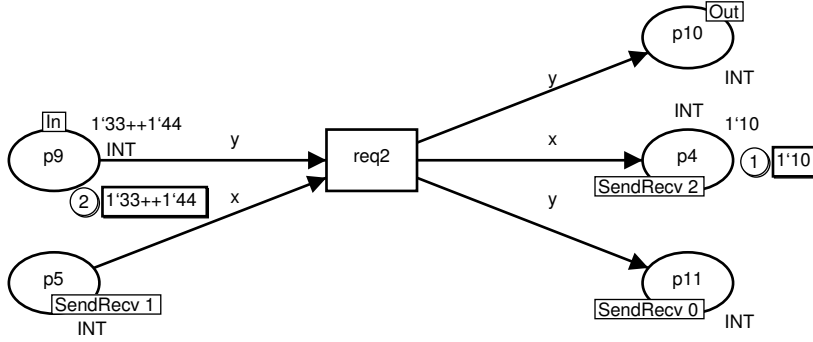


Figura 6.6: Página pedido para pedido **req2** (página *req2.classB.t6.classA.t3*).

ligados simultaneamente a uma transição emissora e a uma transição receptora. Por esta razão, a única solução consiste em traduzir também a transição emissora da forma sugerida para a transição receptora: removendo a transição inicial e utilizando lugares de fusão.

É esta a técnica utilizada no protótipo desenvolvido [Barros e Gomes, 2004b] e reflectida no exemplo apresentado na secção seguinte.

6.2 Um Exemplo de Tradução de Grupos de Sincronismo na *CPN Tools*

Nesta secção apresenta-se um exemplo de um modelo para um sistema de produção e consumo de dados entretanto guardados numa base de dados. O sistema será denominado *DBReadWrite* e foi inspirado nos problemas clássicos dos *leitores-escretores* e *produtor-consumidor* (e.g. [Reisig, 1998]).

No sistema existe um escritor que escreve numa base de dados os dados previamente pedidos a um produtor. Existem também dois leitores que lêem a base de dados e enviam esses dados para um de dois consumidores de tipos diferentes. O tipo de consumidor para onde os dados devem ser enviados, é especificado pelo produtor e fornecido juntamente com os dados ao escritor. Este coloca essa informação na base de dados.

Com base na especificação anterior é possível especificar um diagrama de classes na linguagem UML [OMG, 2003] contendo as operações envolvidas (*vide* Fig. 6.7). O diagrama apresenta uma interface (*ConsumerClass*) e as seguintes classes concretas⁴:

- **Producer.**
- **Writer.**

⁴Em inglês: *concrete classes*.

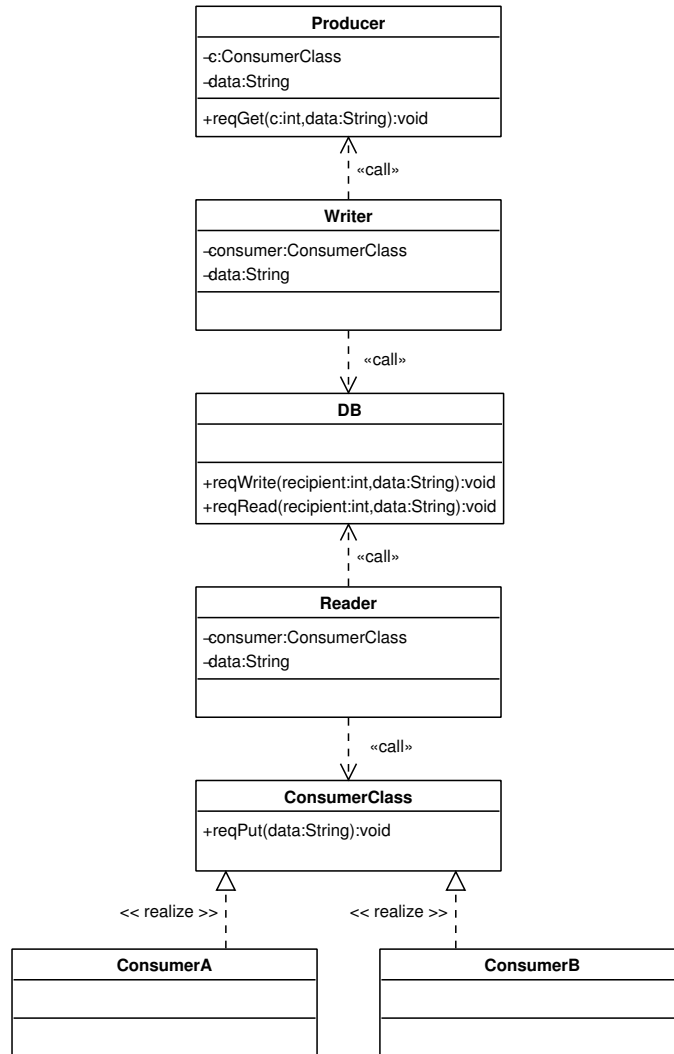


Figura 6.7: Diagrama de classes para o sistema *DBReadWrite*.

- DB.
- Reader.
- ConsumerA.
- ConsumerB.

O sistema contém seis objectos "activos": um produtor (da class **Producer**), um escritor (da classe **Writer**), dois leitores (da classe **Reader**), um consumidor do tipo A (da classe **ConsumerA**) e um consumidor do tipo B (da classe **ConsumerB**). O sistema também inclui um objecto passivo: uma base de dados (DB). Este objecto controla o acesso aos dados e disponibiliza métodos que permitem escrever e ler esses mesmos dados.

A partir deste diagrama de classes podemos construir um diagrama de adição (*vide* Fig. 6.8)⁵.

⁵Tal é possível, mas na prática será mais fácil construir o diagrama de adição em pequenas iterações, alternadas

Este fornece um passo intermédio para a rede de Petri. Tal como o diagrama de classes, é ainda um diagrama de estrutura mas reflecte de forma directa as composições entre páginas da rede de Petri. Note-se que os grupos de sincronismo são representados por transições resultado. O diagrama de classes é um diagrama no domínio dos objectos, mas o diagrama de adição é já um diagrama no domínio da rede de Petri. Em resumo, o diagrama de adição permite outro tipo de visualização estrutural, próxima não só do diagrama de classes, mas também do modelo em rede de Petri. Note-se que as transições resultado são anotadas com as operações. Na rede de Petri estas são especificadas por eventos *RECV*. Dado que os grupos de sincronismo têm uma direcção de invocação, os arcos ligados às transições resultado são dirigidos.

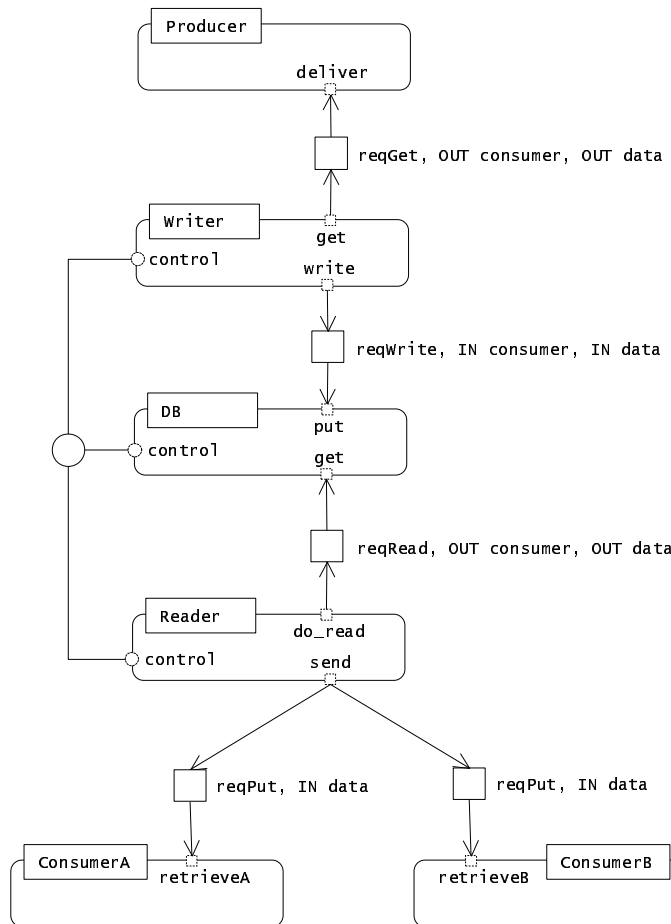


Figura 6.8: Diagrama de adição do modelo do sistema *DBReadWrite*.

Conforme proposto no capítulo anterior, cada classe está modelada por uma página própria. No entanto, neste modelo optou-se por não utilizar classes canónicas, ou seja, não são utilizadas transições, e respectivos eventos *RECV*, para criação de objectos⁶. A classe *Reader* tem dois objectos e todas as restantes definem apenas um objecto (um *singleton* [Gamma et al., 1995]). Por esta razão utilizar-se-á não só a expressão "um objecto C" para designar um objecto da com a construção da rede de Petri.

⁶Tal é suficiente devido ao facto da quantidade de objectos ser fixa ao longo de toda a execução do modelo e justifica-se a sua utilização por razões de simplificação do modelo. Na secção seguinte apresenta-se um modelo que utiliza várias classes canónicas.

classe *C*, mas também "o objecto *C*" para designar o único objecto da classe *C*.

A seguinte apresentação pode ser visualizada, de forma resumida, pelo diagrama de adição respectivo na Fig. 6.8. O objecto *Writer* (vide Fig. 6.9) pede dados ao objecto *Producer* (vide Fig. 6.10) utilizando o pedido *reqGet* na transição *get*. Esses dados são escritos no objecto da classe *DB* utilizando o pedido *reqWrite* na transição *write*. Tal é possível desde que o lugar *control* o permita, ou seja, desde que nenhum leitor esteja na posse de dados ainda não enviados para um dos dois tipos de consumidor. Tal traduz-se na presença de duas marcas no lugar *control* e na ausência de marcação no lugar *read* da classe *Reader* (vide Fig. 6.11).

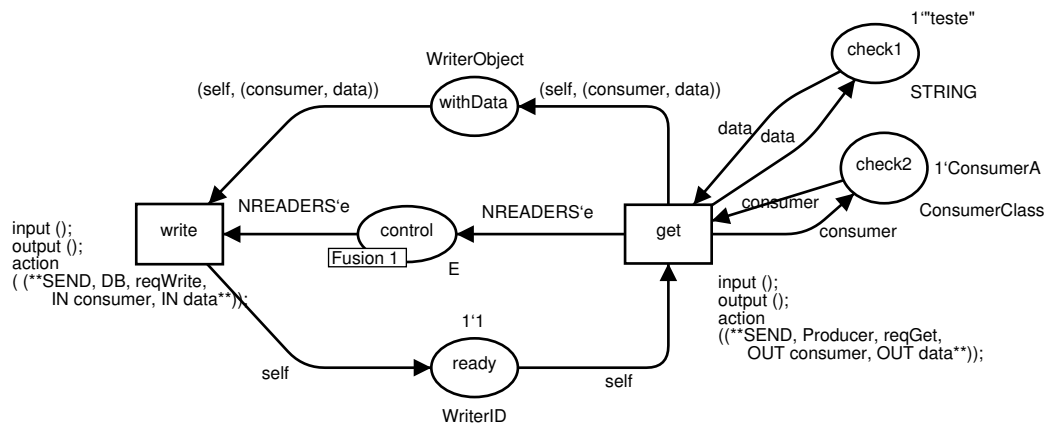


Figura 6.9: Class Writer.

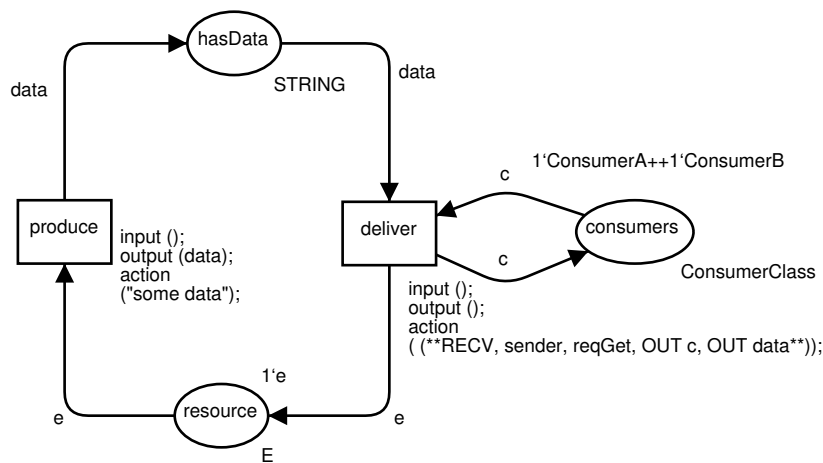


Figura 6.10: Classe Producer.

Cada objecto da classe *Reader* pode ler (pelo pedido *reqRead* na transição *do_read*) os dados presentes no objecto *DB*, escolher um de dois tipos de consumidor (*ConsumerA* ou *ConsumerB*) e passar-lhes os dados utilizando o evento *reqPut* na transição *send*. Esta invocação da operação *reqPut* é na verdade uma invocação polimórfica que resulta do valor da variável *consumer* aquando do disparo da transição *send* na classe *Reader*. Trata-se portanto da especificação de uma de duas possíveis invocações resultantes da vinculação dinâmica da variável *consumer* que

especifica a classe onde se encontra o pedido `reqPut` a invocar.

A transição `produce` na classe `Producer` (*vide* Fig. 6.10) gera os dados a serem consumidos. No exemplo, e exclusivamente por razões de simplificação, é sempre gerada a *string* "some data" mas, obviamente, poderiam ser gerados outros dados. É também o objecto da classe `Producer` que especifica qual a classe do consumidor que irá consumir os dados. Novamente por razões de simplificação, a classe do consumidor (`ConsumerA` ou `ConsumerB`) é escolhida de entre o conjunto de identificadores de classes presentes no lugar `consumers` na classe `Producer`.

A classe `Writer` utiliza dois lugares *check* (`check1` e `check2`). Conforme já referido, o seu único propósito consiste em evitar erros de sintaxe. Por exemplo, a variável `consumer` que na realidade é vinculada por um parâmetro OUT do pedido `reqGet`, não causa um erro de sintaxe devido à presença do lugar `check2` e respectivos arcos. Conforme já notado na Secção 6.1.2, todos os lugares *check* e respectivos arcos encontram-se removidos no modelo final. Finalmente, e conforme já referido, o objecto da classe `Writer` necessita de uma quantidade de marcas no lugar `control` igual à quantidade de leitores (`NREADERS`). Dado não existirem mais escritores, tal garante a escrita em exclusão mútua com a leitura.

Controlados pelo lugar `control`, os dois objectos `Reader` (Fig. 6.11), tentam ler informação do objecto da classe `DB` utilizando o pedido `reqRead` que é invocado pela transição `do_read` na classe `Reader`. Este pedido tem dois parâmetros OUT: `consumer` e `data`. O valor do último parâmetro é transmitido para o objecto consumidor da classe especificada pelo valor do primeiro parâmetro. Este pode tomar o valor `ConsumerA` ou `ConsumerB` e é especificado pelo pedido `reqPut`, associado à transição `send` na classe `Reader`. Aí, a variável `consumer` é vinculada dinamicamente. Diferentemente do objecto `Writer`, os objectos `Reader` necessitam apenas de uma marca no lugar de fusão `control`, que também surge na classe `Writer`, para poderem aceder à base de dados. Ou seja, de acordo com a semântica usual de um sistema de leitores e escritores, é permitida a leitura simultânea por dois leitores, mas a escrita é necessariamente exclusiva. Naturalmente, é também necessário que existam dados para ler ou escrever. Tal é garantido pela transição receptora `get` na classe `DB`. Apesar de não se encontrar ligado a quaisquer arcos na classe `DB`, o lugar `control` pertence logicamente à classe `DB` e é utilizado pelos objectos da classe `Reader` e `Writer`.

Os objectos das classes `ConsumerA` e `ConsumerB` (*vide* Fig. 6.13 e 6.14) foram também definidos como *singletons*. Ambos têm um pedido `reqPut` em comum que resulta da implementação da interface `ConsumerClass` (*vide* Fig. 6.7). Conforme já referido, estes pedidos são chamados com base no valor da variável `consumer` aquando do disparo da transição `send` na classe `Reader`. Mais especificamente, a transição `send` na classe `Reader` funde-se, de forma disjunta, quer com a transição `retrieveA`, na classe `ConsumerA`, quer com a transição `retrieveB`, na classe `ConsumerB`. Tal é facilmente visualizável no diagrama de adição na Fig. 6.8. A transição que é de facto disparada é escolhida conforme o vínculo da variável `consumer`: `ConsumerA` ou `ConsumerB`, respectivamente. Esta semântica surge de forma sintacticamente mais explícita na

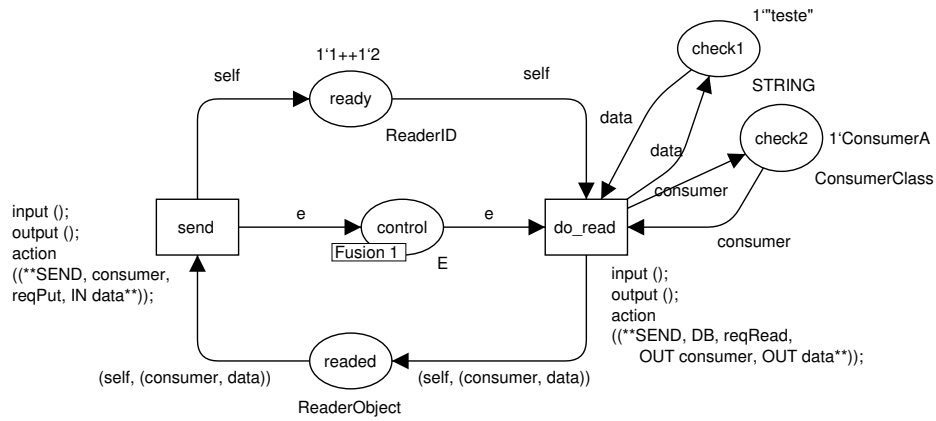


Figura 6.11: Classe Reader.

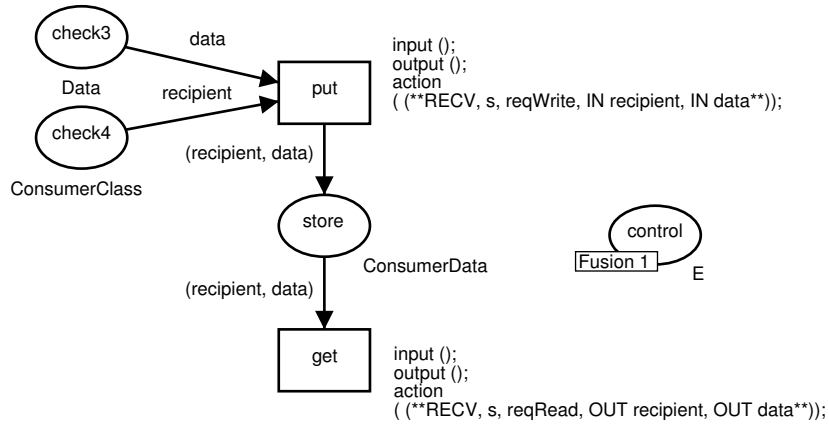


Figura 6.12: Classe DB.

secção seguinte, ou seja, no modelo final resultante da substituição dos pedidos nas transições, por transições pedido que implementam os grupos de sincronismo.

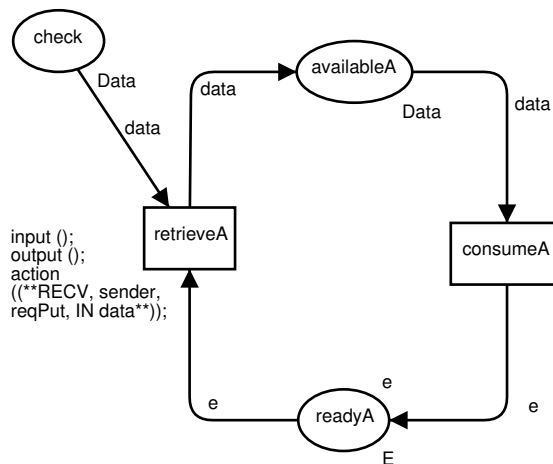


Figura 6.13: Classe ConsumerA.

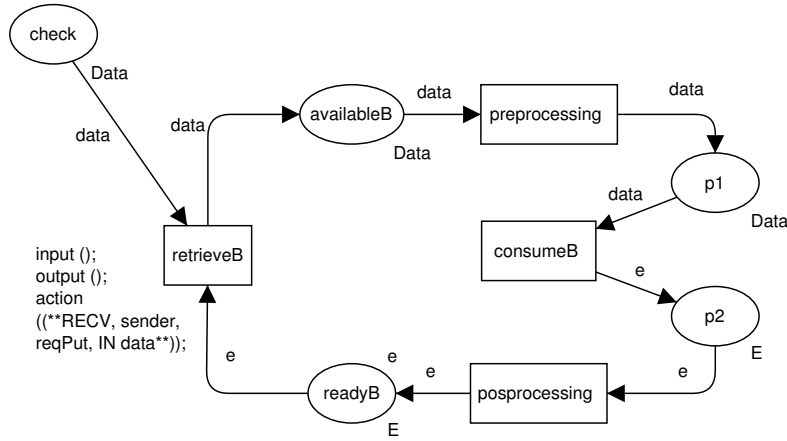


Figura 6.14: Classe ConsumerB.

6.2.1 Rede Colorida Hierárquica Equivalente

Esta secção apresenta o modelo equivalente sob a forma de uma rede de Petri colorida hierárquica. Este modelo foi gerado pelo protótipo já referido e exemplifica a tradução dos grupos de sincronismo com duas transições para uma única transição pedido.

O modelo traduzido é equivalente ao modelo inicial que utiliza grupos de sincronismo. A diferença fundamental reside no facto de se tratar de um modelo suportado pela ferramenta CPN Tools, logo, sem grupos de sincronismo. Na Fig. 6.15 na pág. 157 apresenta-se o diagrama de adição do modelo após transformação. A maior complexidade gráfica do modelo, relativamente ao modelo inicial na Fig. 6.8, reflecte a maior complexidade das composições entre páginas e a maior quantidade destas. Dado que o único tipo de transformação efectuado foi a substituição dos grupos de sincronismo (que implicam fusão de transições) pelo seu equivalente assíncrono (fusão de lugares), a comparação dos modelos inicial e final exemplifica também a abstracção e acrescida legibilidade permitidas pela utilização dos grupos de sincronismo.

A secção seguinte apresenta as classes resultantes da transformação das classes iniciais. Posteriormente, apresentam-se as páginas pedido. Todas as páginas devem ser enquadradas pelo diagrama de adição na Fig. 6.15.

Classes Iniciais Após Transformação

No que respeita às classes (páginas) iniciais, as transições emissoras e receptoras são removidas, juntamente com todos os seus arcos. Os lugares previamente ligados às transições removidas são transformados em lugares de fusão. Os restantes elementos, de cada classe, não são modificados. Em particular, a informação gráfica associada não é modificada. Tal permite um reconhecimento comparativo entre os modelos iniciais e os modelos transformados o que facilita a legibilidade dos

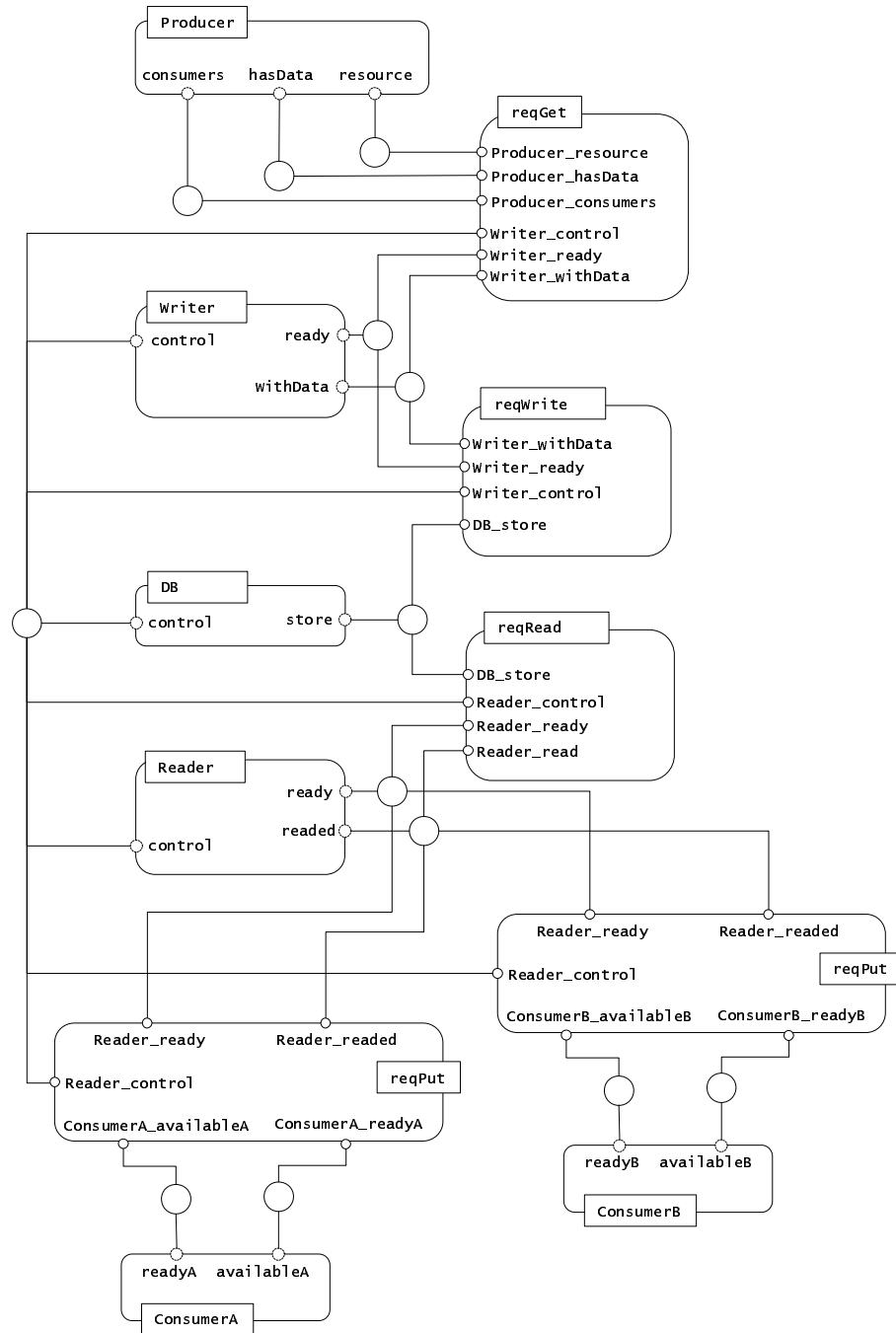


Figura 6.15: Diagrama de adição do modelo equivalente, sem grupos de sincronismo, do sistema *DBReadWrite*.

últimos. Também para melhorar esta comparação entre modelos, o protótipo gera informação textual que substitui as transições removidas nas classes transformadas pelo seu nome original. Estas duas estratégias — (1) remoção de transições sem modificação dos restantes elementos e (2) substituição das transições pelos seus nomes — permitem um modelo final graficamente semelhante ao modelo inicial⁷.

⁷Infelizmente a *CPN Tools* não permite a especificação de arcos falsos que substituam os arcos removidos. Tal permitiria uma semelhança gráfica total entre cada modelo inicial e final respectivo.

As Figs. 6.16, 6.17, 6.18, 6.19, 6.20 e 6.21 apresentam as classes iniciais após transformação. Note-se a substituição das transições emissoras e receptoras por anotações textuais e a passagem dos respectivos lugares a lugares de fusão.

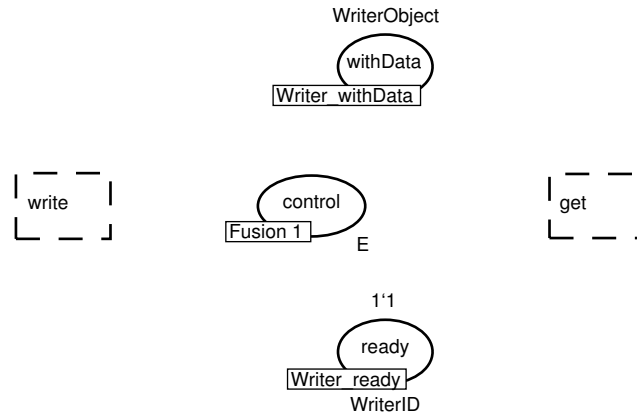


Figura 6.16: Classe **Writer** após transformação. Deve ser comparada com o modelo inicial na Fig. 6.9.

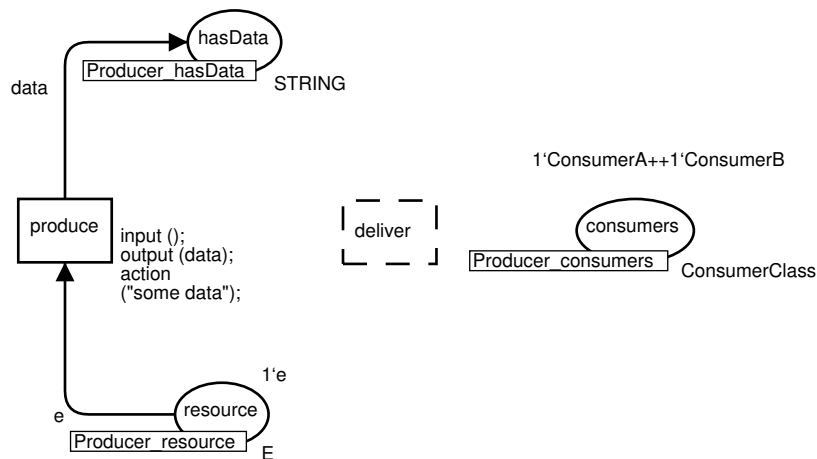


Figura 6.17: Classe **Producer** após transformação. Deve ser comparada com o modelo inicial na Fig. 6.10.

Páginas Pedido

Conforme já referido, para cada grupo de sincronismo é criada uma nova página pedido da rede de Petri colorida. Esta página contém a própria transição pedido e a sua vizinhança sob a forma de lugares de fusão. Estes fundem-se com os lugares nas páginas de origem das transições emissora e receptora.

As Figs. 6.22, 6.23, 6.24, 6.25 e 6.26 apresentam todas as páginas pedido geradas automaticamente pelo protótipo, juntamente com as classes transformadas já apresentadas na Secção 6.2.1.

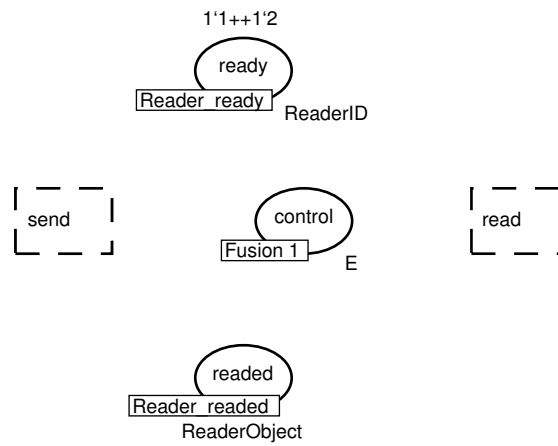


Figura 6.18: Classe Reader após transformação. Deve ser comparada com o modelo inicial na Fig. 6.11.

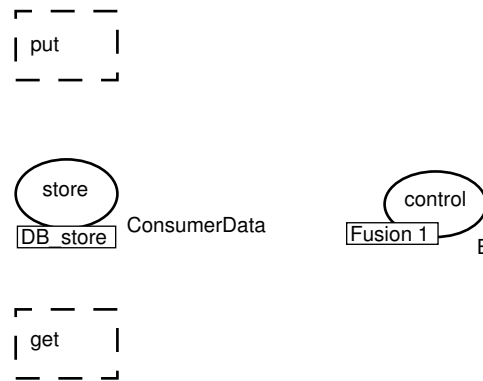


Figura 6.19: Classe DB após transformação. Deve ser comparada com o modelo inicial na Fig. 6.12.

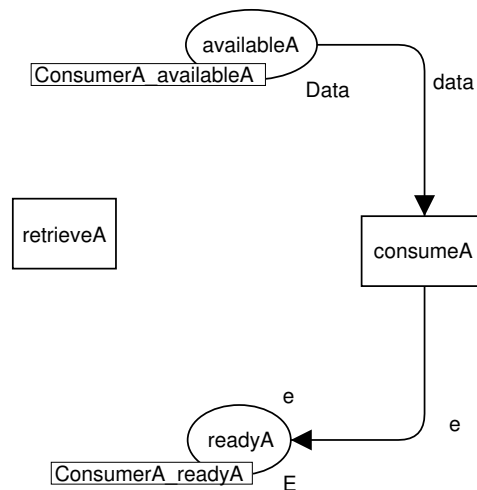


Figura 6.20: Classe ConsumerA após transformação. Deve ser comparada com o modelo inicial na Fig. 6.13.

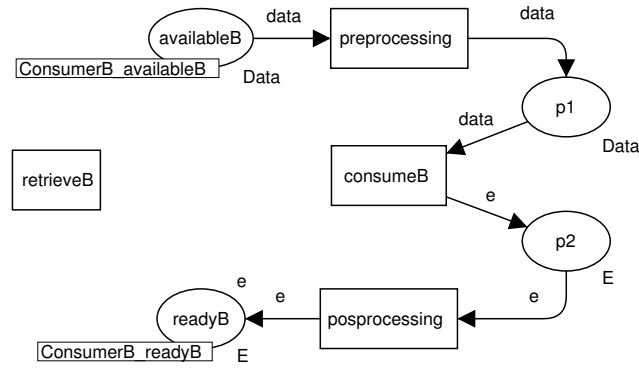


Figura 6.21: Classe `ConsumerB` após transformação. Deve ser comparada com o modelo inicial na Fig. 6.14.

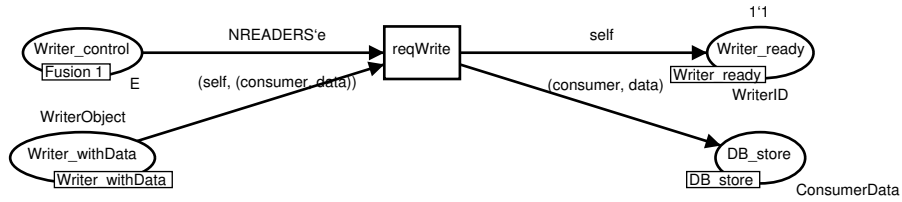


Figura 6.22: Página pedido `reqWrite`.

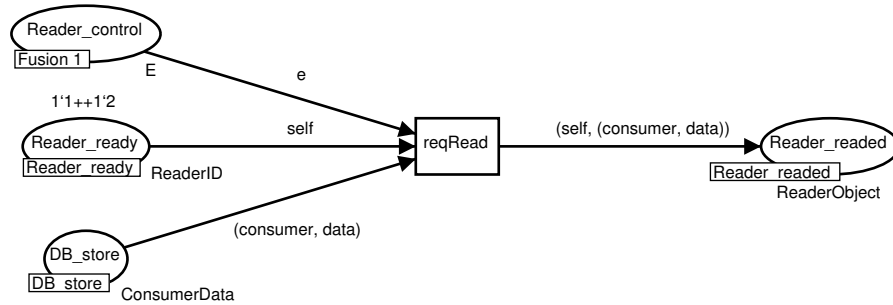


Figura 6.23: Página pedido `reqRead`.

Para evitar sobreposições gráficas, os lugares e arcos provenientes das transições emissora e receptora não podem manter a localização que apresentam no modelo inicial. Por essa razão optou-se por utilizar um algoritmo trivial, mas suficiente para a obtenção de bons resultados, para disposição dos lugares de fusão em cada página pedido: os lugares de entrada são dispostos verticalmente do lado esquerdo da transição pedido; os lugares de saída são dispostos verticalmente do lado direito da transição pedido. Um lugar que seja de entrada e de saída não é duplicado (por meio de uma nova fusão). Em vez disso é simplesmente colocado juntamente com os lugares de entrada ou de saída. Isto sucede com o lugar `Producer_consumers` no lado esquerdo na página `reqGet` (Fig. 6.24) em que os dois arcos estão sobrepostos e têm anotações com o mesmo conteúdo (`consumer`).

A invocação polimórfica especificada pela transição `send` na classe `Reader` (*vide* Fig. 6.11) dá

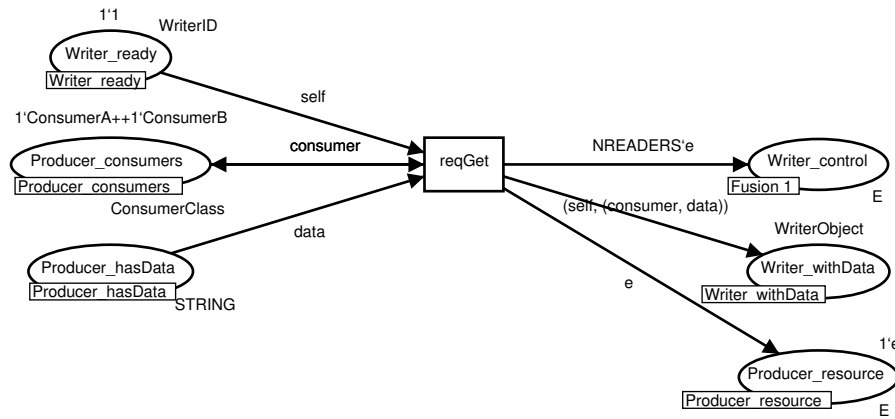


Figura 6.24: Página pedido reqGet.

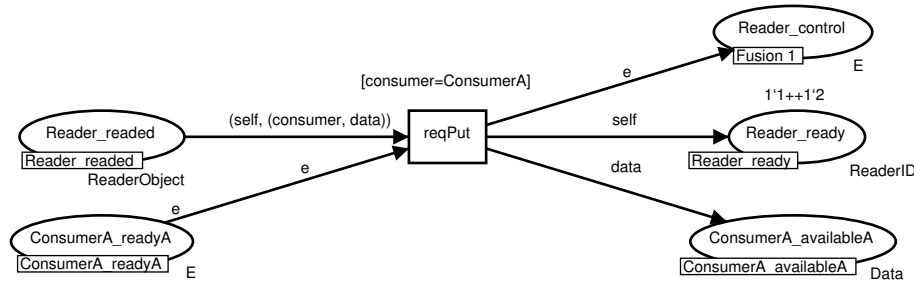


Figura 6.25: Página pedido reqPut para classe ConsumerA.

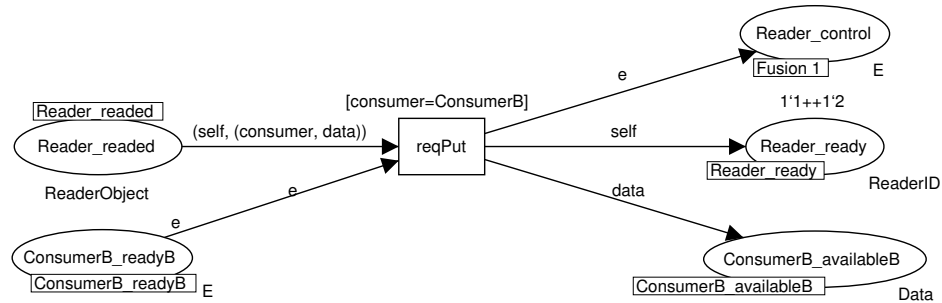


Figura 6.26: Página pedido reqPut para classe ConsumerB.

origem a duas páginas pedido —para cada um dos valores possíveis para a variável **consumer** na classe **Reader** — correspondentes às transições associadas ao pedido **reqPut** na parte inferior da Fig. 6.8. Estas páginas estão ilustradas na Fig. 6.25 e 6.26, respectivamente. A distinção é especificada pelas guardas da transição que testam o valor da variável (**consumer**) que especifica a classe.

Refira-se ainda que o Apêndice F, na pág. 265, apresenta informação complementar sobre o protótipo, incluindo o respectivo código fonte.

Na secção seguinte apresenta-se um exemplo de modelação que compara um modelo que utiliza

uma rede de Petri colorida hierárquica com outro modelo equivalente e para o mesmo sistema, que utiliza uma rede de Petri colorida componível. Nesse modelo utilizam-se redes canónicas, bem como, identificadores para os objectos em cada classe.

6.3 Controlo de Elevadores

Esta secção começa por apresentar um exemplo de um modelo que utiliza redes de Petri coloridas hierárquicas (e a CPN Tools) na construção de um modelo para um sistema de controlo de um conjunto de elevadores. Este será aqui designado por *modelo descendente*, dado tratar-se de um modelo hierárquico construído com base em refinamentos sucessivos de um modelo do sistema.

O sistema a modelar e o modelo descendente estão apresentados, de forma detalhada, num artigo de Jens Jørgensen apresentado numa *workshop* em Outubro de 2004 [Jens Bæk Jørgensen, 2004]. Por sua vez, o sistema e o modelo de Jørgensen foram inspirados por um exemplo de um livro de Roel Wieringa [Wieringa, 2003] onde é apresentado um modelo utilizando estadogramas. Na secção seguinte (§6.3.1) descreve-se de forma resumida esse mesmo sistema. Seguem-se o modelo descendente (§6.3.2) e o modelo orientado pelos objectos (§6.3.4).

O modelo orientado pelos objectos utiliza grupos de sincronismo e os idiomas propostos no capítulo anterior. Tal é feito no contexto da CPN Tools. Como a CPN Tools não suporta grupos de sincronismo, ou outra construção semelhante (e.g. canais síncronos) o modelo não é presentemente executável. No entanto, permite uma comparação com o modelo descendente no contexto restrito de uma única ferramenta, evidenciando dessa forma as diferenças entre ambas as atitudes de modelação que reflectem diferentes paradigmas: o estrutural e o orientado pelos objectos.

6.3.1 O Sistema a Modelar

O sistema a modelar corresponde a um controlador de dois elevadores num edifício de dez pisos. A principal responsabilidade do controlador é a de controlar o movimento dos dois elevadores (*cages* nos modelos). O movimento dos dois elevadores é desencadeado pelos utilizadores que primem *botões de pedido* (*request buttons*). Em cada piso existem *botões de piso* (*floor buttons*) que podem ser premidos para chamar um elevador. Quando estes botões são premidos o utilizador especifica se pretende ir para um piso inferior ou superior. Ou seja, cada botão admite três estados: *no*, *up* e *down*. Dentro de cada elevador existe outra variedade de botões de pedido: os *botões de elevador* (*cage buttons*), um para cada piso. Ao serem premidos indicam para que piso o utilizador se pretende deslocar.

Para além de controlar o movimento dos elevadores, o controlador é também responsável pela

atualização de um indicador de localização dentro de cada elevador. Este indicador informa sobre o piso onde o elevador se encontra em cada instante.

Podem ser identificados vários casos de uso⁸ que devem ser suportados pelo controlador. Os principais são os seguintes três:

Recolher passageiros Quando um passageiro prime um botão num piso p , um elevador deverá mover-se até esse piso e abrir a porta.

Entregar passageiros Quando um passageiro prime um botão, dentro do elevador e correspondente ao piso p pretendido, o elevador deverá mover-se até esse piso e abrir a porta.

Mostrar piso corrente Quando um elevador chega a um qualquer piso, os passageiros, dentro desse elevador, devem ser informados sobre qual o piso em que se encontram.

Como exemplo, vejamos de mais perto o caso de uso *Recolher passageiros* que é desencadeado pelo premir de um botão de piso. Tal gera um estímulo dirigido ao controlador dos elevadores que origina a necessidade de executar a seguinte sequência de tarefas:

1. A luz do botão premido deve ser ligada, caso não esteja, passando o botão ao estado **up** ou **down**.
2. O controlador atribui a execução do pedido a um dos elevadores. Tal implica que o controlador tem de determinar se o pedido pode ser satisfeito imediatamente. Isto só é possível quando o pedido tem origem num piso onde já se encontra um elevador parado. Neste caso, o elevador simplesmente abre a porta, não sendo necessário ligar o motor. Caso o elevador não se encontre no piso, o pedido ficará guardado numa lista de pedidos pendentes que o elevador irá satisfazendo ao longo do tempo.
3. Se for necessário ligar o motor, o controlador terá de gerar o sinal apropriado para esse efeito.
4. Se for suficiente abrir as portas, o controlador terá de gerar o sinal apropriado para esse efeito.

Este cenário e as suas possíveis continuações que incluem o movimento dos elevadores e a activação de sensores e actuadores, podem ser especificados utilizando, por exemplo, diagramas de sequência [Genest et al., 2004]. Em alternativa poderíamos descrever o comportamento geral do controlador dos elevadores e das entidades no seu ambiente, utilizando estadogramas [Harel, 1987, 1988] tal como é feito no livro de Wieringa [Wieringa, 2003]. Note-se que, quando comparados, os estadogramas e as redes de Petri coloridas têm as suas vantagens e desvantagens

⁸Em inglês: *use cases*.

relativas, já referidas na literatura (e.g. [Elkoutbi e Keller, 2000; Jørgensen e Christensen, 2002]). No entanto, o objectivo desta dissertação e deste exemplo em particular é o de demonstrar e aumentar a aplicabilidade das redes de Petri. Por essa razão são seguidamente apresentados dois modelos construídos com redes de Petri.

6.3.2 Modelo em Rede de Petri Colorida Hierárquica

Esta secção apresenta o modelo construído com base numa rede de Petri colorida hierárquica para o controlador dos elevadores. Para além da dobragem, característica das redes de Petri coloridas, o principal mecanismo de estruturação utilizado é a decomposição hierárquica baseada em macrotransições que no caso das redes de Petri coloridas são denominadas transições de substituição.

Dada a maior complexidade deste modelo, comparativamente com o modelo da Secção 6.2, será também explicado, de um modo informal, um pouco da sintaxe e semântica associadas à linguagem de programação Standard ML, a linguagem de inscrições utilizada na CPN Tools.

Anotações

As entidades no ambiente são representadas por conjuntos de cores. Tal é verdade quer para as entidades que constituem parte do ambiente, quer para outras entidades com as quais o controlador tem de interagir ainda que indirectamente. Por exemplo, o controlador controla cada elevador, porque controla o motor de cada elevador. O conjunto de cores (tipo de dados) CAGE utilizado para representar os elevadores é constituído por um tuplo com 4 elementos com o formato (`cageid`, `floor`, `requestlist`, `direction`). Os quatro elementos têm o seguinte significado:

`cageid` identifica o elevador.

`floor` é o número do piso onde o elevador se encontra, se o elevador estiver parado; ou o último piso que visitou, se o elevador se encontrar em movimento.

`requestlist` é a lista de pedidos pendentes que o elevador tem de satisfazer, sob a forma de números de pisos.

`direction` indica o sentido do movimento do elevador: `up` (a subir), `down` (a descer) ou `no` (parado).

Os pisos são representados por inteiros, e um botão de piso é representado por um par (`floor`, `direction`), onde `direction` tem o valor `no` se o botão não foi premido, caso contrário terá o valor `up` ou `down` indicando o sentido do pedido do utilizador. Os botões dentro de cada elevador

(*cage buttons*) são também representados por um par (*cageid,buttonlist*), onde *buttonlist* é uma lista de inteiros correspondentes aos andares para os quais foram premidos botões de elevador.

Comportamento dos Elevadores

O modelo que utiliza uma rede de Petri colorida hierárquica é constituído por três páginas (ou módulos):

1. Do Cage Cycle.
2. Requests.
3. UpDown.

O comportamento dos elevadores é considerado a tarefa de mais alto-nível e portanto modelado pela página de topo do modelo hierárquico. Esta denominou-se *Do Cage Cycle* e apresenta-se na Fig. 6.27.

Cada marca (cor) do conjunto de cores **CAGE** está em cada momento num, e só num, dos lugares **Idle**, **Moving**, **Opened** ou **Closed**. Por essa razão, todos esses lugares têm associado o conjunto de cores **CAGE**. Os lugares **Floor Buttons** e **Cage Buttons** são utilizados para modelar os botões de piso e os botões de elevador, respectivamente. As marcas nesses lugares correspondem a botões e modelam se os botões estão no estado *on* ou no estado *off*. A Fig. 6.27 mostra o estado inicial do modelo para a situação em que ambos os elevadores se encontram parados em espera no piso 1 e nenhum pedido foi ainda feito (lugar **Idle**). Note-se que a quantidade de marcas num qualquer lugar é indicada por um número inteiro dentro de um pequeno círculo perto ou dentro do lugar. Na CPN Tools é possível "clicar" no pequeno círculo abrindo dessa forma um rectângulo que mostra o estado actual do lugar. Isto foi feito para o lugar **Idle** mostrando os valores das duas marcas do conjunto de cores **CAGE**.

Cada expressão associada a um arco determina as marcas que são adicionadas ou removidas. Por exemplo, a expressão (*c,cf,f::r1,cd*) que aparece no arco entre o lugar **Idle** e a transição **Start Motor**, especifica uma marca **CAGE** na qual o terceiro elemento é compatível com o padrão⁹ *f::r1*, ou seja, é uma lista não vazia. *c*, *cf*, *f*, *r1*, e *cd* são variáveis dos tipos de dados adequados.

A habilitação da transição **Start Motor** requer duas condições:

1. Que o lugar **Idle** contenha alguma marca **CAGE** compatível com o padrão (*c,cf,f::r1,cd*), ou seja uma marca **CAGE** com uma lista de pedidos pendentes que não se encontre vazia.

⁹Em inglês: *pattern*.

Quando a transição **Start Motor** dispara, tal significa que um dos elevadores passou a mover-se: a marca correspondente passa do estado **Idle** para o estado **Moving** e o elemento **direction** desta marca é dado pela função **setdirection** que determina em que sentido deve o elevador deslocar-se. Para tal, baseia-se no piso actual do elevador e na sua lista de pedidos pendentes.

O disparo da transição **Arrive at Destination** modela a chegada de um elevador a um determinado piso, a sua paragem e a abertura da porta. A função **stophere** na guarda garante que o elevador só pára se o piso corrente está na lista de pedidos pendentes. Caso contrário o elevador continua em movimento (no estado e lugar **Moving**) e a sua localização é actualizada pelo módulo **UpDown**.

O disparo da transição **Arrive at Destination** origina a actualização de todas as marcas que representam botões nos lugares **Floor Buttons** e **Cage Buttons** para que os botões correspondentes à posição actual do elevador sejam desligados (*off*). Também o pedido para o piso actual é removido da lista de pedidos pendentes da marca **CAGE**. Tal é realizado pela função **removerequest** no arco entre **Arrive at Destination** e **Opened**.

O disparo da transição **Close Doors** modela o fecho da porta do elevador: a marca que representa o elevador é colocada no lugar **Closed**.

A transição **Stop** é habilitada se não existirem pedidos pendentes associados ao elevador (na marca **CAGE**). Tal é verificado pela função **turnidle** que constitui a guarda da transição. Nesse caso, o elevador fica parado em espera (estado e lugar **Idle**). Em alternativa, a transição **Serve Next Req** é habilitada se existirem pedidos pendentes. Nesse caso, o elevador continua a satisfazer pedidos, voltando a movimentar-se (lugar **Moving**).

Os módulos **Requests** (*vide* Fig. 6.28) e **UpDown** (*vide* Fig. 6.29), já referidos, são incluídos na página **Do Cage Cycle** sob a forma de transições de substituição **Handle Requests** e **Move UpDown**, respectivamente. Estes módulos são os elementos no nível seguinte da decomposição hierárquica do modelo.

A página **Requests** suporta a criação, tratamento e posterior atribuição de pedidos a cada um dos elevadores. Em particular, modela a activação dos botões de piso (transição **Push Floor Button**) e dos botões dentro dos elevadores (transição **Push Cage Button**). Tal origina a criação de pedidos (*requests*) que ficam disponíveis nos lugares **Floor Requests** e **Cage Requests**. Quando os elevadores se encontram no estado **Idle** podem então retirar pedidos desses lugares, passando esses pedidos a constituir parte da lista de pedidos pendentes de cada elevador. Por razões de simplificação, é dado o mesmo tratamento quer aos pedidos com origem nos botões de piso, quer aos pedidos com origem nos botões de elevador. No entanto, um modelo mais sofisticado pode especificar uma estratégia de escolha do elevador mais adequado para satisfazer o pedido, nomeadamente aproveitando a informação sobre o sentido de deslocamento actual de cada elevador.

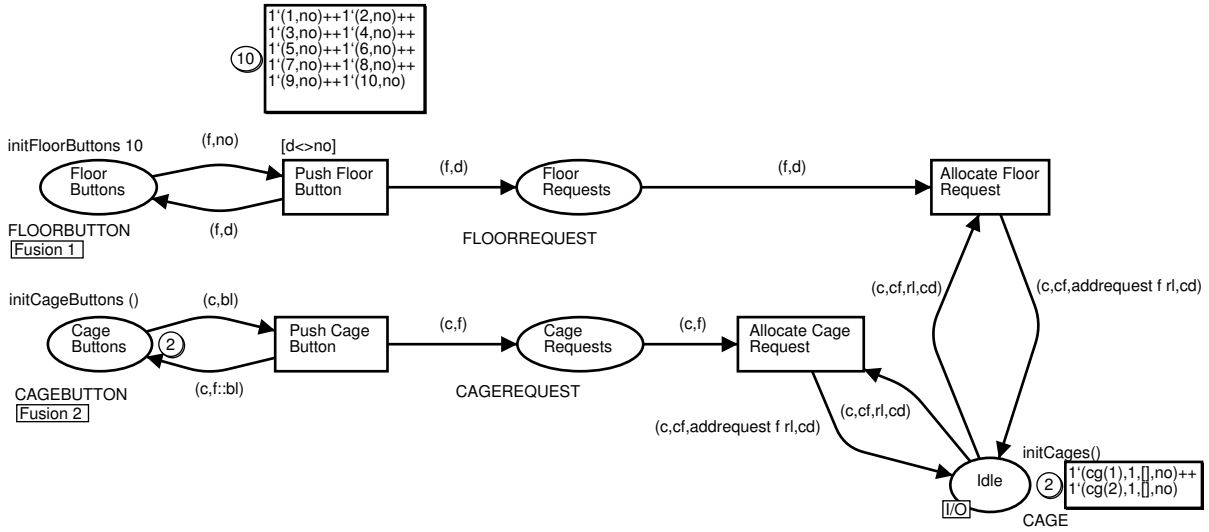


Figura 6.28: Página Requests.

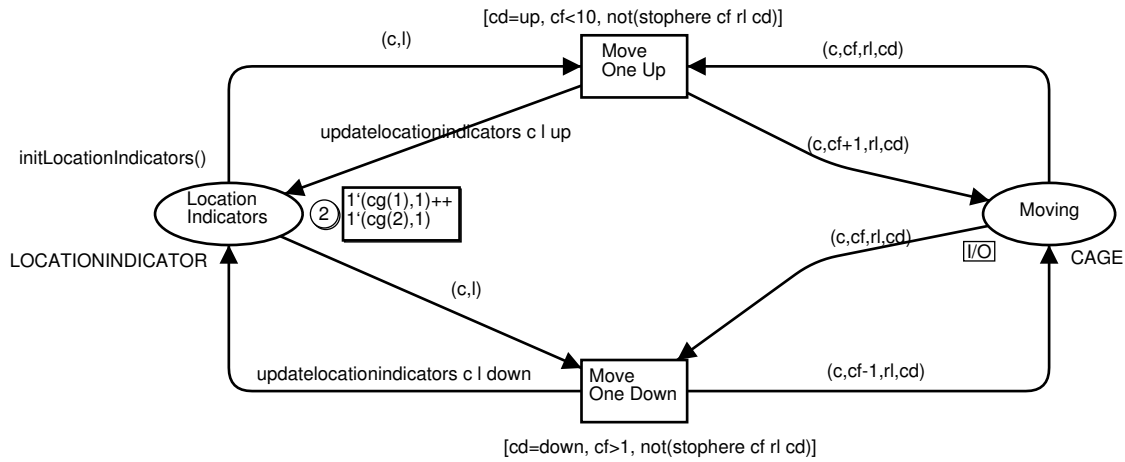


Figura 6.29: Página UpDown.

A página **UpDown** descreve o movimento dos elevadores entre pisos e a actualização do indicador dentro de cada elevador. Quando qualquer dos elevadores se encontra no lugar **Moving**, esta página altera a variável que indica o piso onde o elevador se encontra desde que não sejam ultrapassados os limites dos andares e não existam pedidos para esse piso. Esta condição é especificada pelas guardas associadas às transições **Move One Up** e **Move One Down**. Os indicadores de localização são modelados por marcas no lugar **Location Indicators** e são modificados pelas mesmas transições que actualizam a posição dos elevadores.

Finalmente, é interessante notar que a relação entre as três páginas que constituem este modelo é facilmente visualizada utilizando o diagrama de adição correspondente, que se apresenta na

Fig. 6.30.

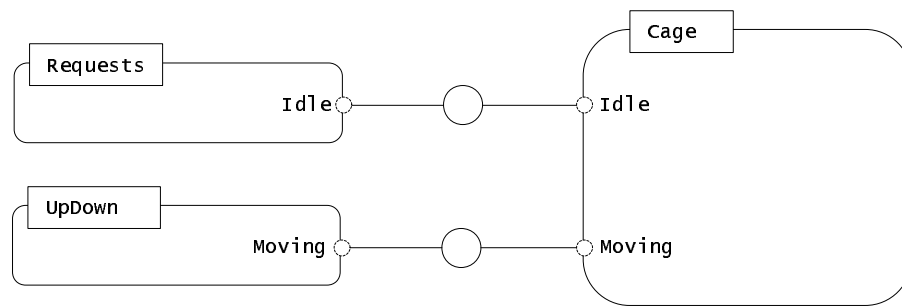


Figura 6.30: Diagrama de adição do controlador de elevadores utilizando uma rede de Petri colorida hierárquica.

6.3.3 Redes de Petri na Especificação de Requisitos e no Desenho

O modelo apresentado pode também ser visto como um modelo para a especificação de casos de uso executáveis [Jørgensen e Bossen, 2004; Jens Bæk Jørgensen, 2004; Barros e Jørgensen, 2005b,a]. A execução do modelo pode ser utilizada como forma de validar os três casos de uso referidos. Mais genericamente, o modelo que utiliza uma rede de Petri colorida hierárquica pode ser utilizado como uma ferramenta para a engenharia de requisitos. O modelo e a sua execução podem ser utilizados para especificar, validar e escolher requisitos para o controlador dos elevadores.

O modelo do controlador descreve o efeito desejado do controlador sobre o seu ambiente. Descreve quando o controlador deve interagir com as entidades externas, tais como motores, botões, sensores e portas. O modelo também descreve qual deve ser o efeito dessas interações no ambiente. Nem o modelo nem os casos de uso descrevem de forma explícita os detalhes do software que será necessário para implementar o controlador. O escopo quer do modelo que utiliza a rede de Petri colorida hierárquica, quer dos casos de uso, é o ambiente no qual o software terá de ser executado. Em ambos os casos, os elementos devem ser vistos como elementos do mundo-real: os motores, botões e portas são reais. Por esta razão, este é um modelo de muito alto-nível no qual se omitem detalhes que é usual encontrar nas especificações que utilizam linguagens de especificação ou de programação orientadas pelos objectos. No caso da CPN Tools, a utilização de uma linguagem funcional como linguagem de especificação convida também a uma maior abstracção. Tal facilita a especificação de protótipos de alto-nível úteis para investigar a utilidade e testar o comportamento dos vários casos de uso que vão sendo identificados na fase de análise. No entanto, estes modelos mantêm-se afastados da implementação e, em especial dos conceitos e técnicas usuais nas linguagens de especificação e programação orientadas pelos objectos.

A subsecção seguinte apresenta um diagrama de classes e o modelo alternativo mas equivalente ao modelo já apresentado, agora utilizando uma rede de Petri colorida componível. Esta é concretizada sob a forma de uma rede de Petri colorida hierárquica em que se assume o suporte para

os grupos de sincronismo e se utilizam os idiomas apresentados no capítulo anterior. Optou-se também pela construção de uma especificação muito mais detalhada a qual embora necessariamente muito mais extensa se mantém legível. Na verdade, dependendo da formação base do modelador, a rede de Petri colorida componível pode ser muito mais legível dado apresentar a expressividade dos conceitos do desenvolvimento orientado pelos objectos.

O modelo que utiliza uma rede de Petri colorida componível pode também ser visto como um modelo mais próximo de uma implementação numa linguagem de programação orientada pelos objectos [Barros e Jørgensen, 2005b,a]. Nesse sentido encurta a distância entre um modelo do nível de requisitos, típica e convenientemente mais abstracto, e uma sua implementação. Estes dois níveis de abstracção para os modelos podem também reflectir uma atitude base de separação entre o "mundo real" e o "mundo da máquina" tal como proposto por Jackson [Jackson, 2002]. No entanto, prefere-se aqui enfatizar a utilização de uma rede de Petri colorida componível como uma alternativa ao modelo tradicional da rede de Petri colorida hierárquica. As vantagens são três:

1. A rede de Petri colorida componível permite mais formas de estruturação do modelo, em especial através da utilização de grupos de sincronismo. Tal permite a construção de composições e dependências paralelas entre os vários módulos do sistema, o que favorece a especificação para a ocultação de informação. A rede de Petri colorida componível está para a rede de Petri colorida hierárquica tal como um programa orientado pelos objectos está para um programa estruturado.
2. É possível estabelecer uma relação clara entre os diagramas mais utilizados nas metodologias orientadas pelos objectos e a rede de Petri. Em particular, torna-se simples o estabelecimento de relações directas entre os diagramas de classes [OMG, 2003], ou outro tipo de diagramas de entidades e associações [Chen, 1976], e a rede de Petri.
3. Todos os principais conceitos do desenvolvimento orientado pelos objectos podem ser representados numa rede de Petri colorida componível utilizando os idiomas apresentados no capítulo anterior. Por essa razão, a construção do modelo da rede de Petri torna-se mais fácil para os modeladores que conheçam e prefiram o paradigma da orientação pelos objectos. Em particular, tal inclui grande parte dos engenheiros de software da actualidade.

6.3.4 Modelo em Rede de Petri Colorida Componível

Nesta secção apresenta-se um modelo alternativo para o controlador de elevadores da Secção 6.3.2. Na feitura deste modelo estiveram presentes duas preocupações antagónicas:

1. Maximizar as semelhanças com o modelo de Jørgensen de forma a facilitar a comparação entre ambos os modelos.

2. Modelar o mesmo sistema de forma a evidenciar as diferenças resultantes da utilização de redes de Petri coloridas componíveis e, consequentemente, de atitudes e técnicas de modelação distintas.

O compromisso entre estas duas atitudes resultou na modelação dos mesmos requisitos para o mesmo sistema mas considerando um diagrama de classes como ponto de partida para uma especificação mais detalhada. Em particular, o novo modelo especifica de forma explícita várias entidades e várias interações que surgem de forma apenas implícita, ou muito abreviada, no primeiro modelo. Tal sucede, por exemplo, com os motores, portas e respectivas interações com os elevadores (*cages*). Todos são agora entidades de "primeira classe", em pé de igualdade com os elevadores. No primeiro modelo, estes surgiam como entidades simultaneamente privilegiadas e aglutinadoras, correspondentes a um nível de topo do modelo.

Para a construção do modelo do controlador de elevadores utilizando uma rede de Petri colorida componível, apresenta-se em primeiro lugar um diagrama de classes (*vide* Fig. 6.31). Este foi construído tendo por base as entidades e os casos de uso identificados na fase de análise de requisitos.

Para além da classe **Cage**, que corresponde em grande parte à página **Do Cage Cycle** do primeiro modelo, definem-se mais oito classes concretas, uma classe abstracta (**Allocation**) e uma interface (**Button**). Seguidamente, apresentam-se cada uma destas classes concretas, as únicas que são especificadas por uma rede de Petri. No final, apresenta-se o diagrama de adição correspondente.

Class Initiator

A partir desta classe (*vide* Fig. 6.32¹⁰) cria-se um *singleton*. Este cria um objecto da classe **FloorButtonAllocation**, que inclui um conjunto de dez botões de piso, e dois objectos da classe **Cage**, correspondentes aos elevadores.

Classe Cage

A classe **Cage** (*vide* Fig. 6.33) segue de muito perto a página **Do Cage Cycle** do modelo anterior pois essa página especifica o comportamento dos controladores dos elevadores e é esse também o objectivo dos objectos desta classe. Esta classe apresenta, no entanto, cinco importantes diferenças relativamente à página **Do Cage Cycle**:

1. Os objectos **Cage** contêm as suas próprias referências (**self**) e referências para objectos das classes **Door**, **Motor**, **CageButtonAllocation**, **FloorButtonAllocation** e **LocationIndica**

¹⁰As transições em destaque são transições habilitadas no estado inicial.

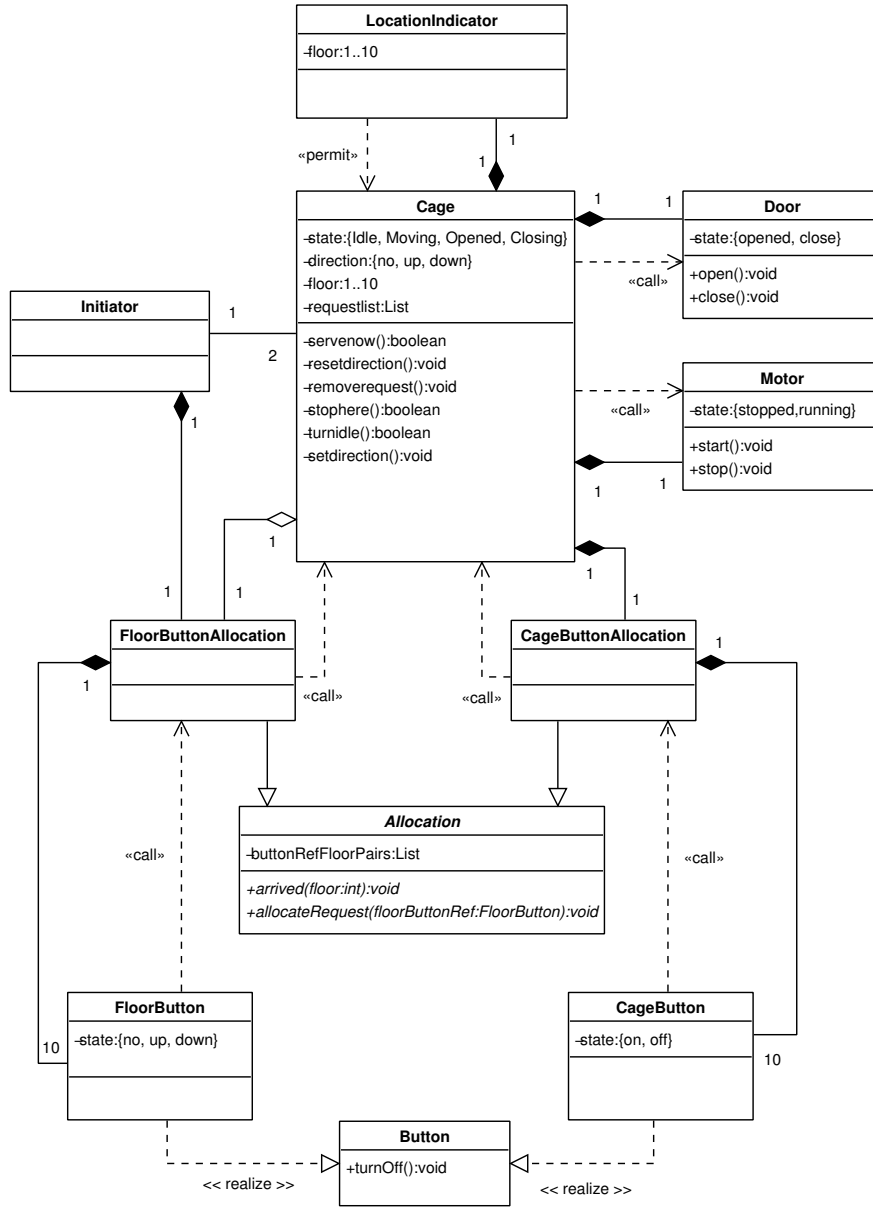


Figura 6.31: Diagrama de classes do controlador de elevadores.

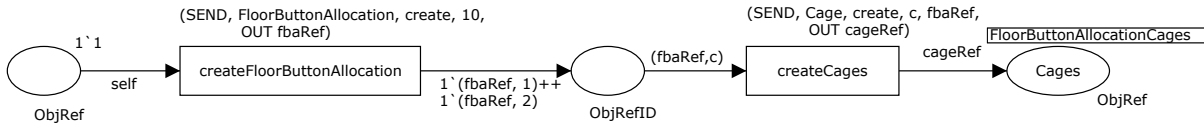


Figura 6.32: Classe Initiator.

tor. Por esta razão o tuplo (c, cf, rl, cd) do modelo descendente foi aumentado para $(self, c, cf, rl, cd, doorRef, motorRef, cbaRef, fbaRef, liRef)$.

- Foram adicionados dois lugares (*cageToCreate* e *beingCreated*) e duas transições (*cage CreateBegin* e *cageCreateEnd*) que se destinam apenas a criar os dois objectos da classe

Cage e a inicializá-los simultaneamente no estado **Idle**.

3. A transição **Handle Requests** juntamente com os lugares **Floor Buttons** e **Cage Buttons** foram substituídos por interacções síncronas com objectos das classes **FloorButtonAllocation** e **CageButtonAllocation**. Parte destas interacções são efectuadas através das transições **allocateCageRequest** e **allocateFloorRequest** pelas quais os objectos **Cage** recebem a notificação de um novo pedido (devido a ter sido premido um botão). Um elevador notifica os objectos **CageButtonAllocation** e **FloorButtonAllocation**, que se encarregam de notificar os botões correspondentes, em duas situações distintas: (1) quando chega ao destino (transição **Arrive at Destination**); (2) quando serve um pedido por já se encontrar no piso certo e no estado **Idle** (transição **Serve**). Essas mesmas transições pedem também às portas para abrirem.
4. A página **UpDown** foi substituída pela classe **Location Indicator** pelo que não surge já como transição de substituição ligada ao lugar **Moving**. Ainda assim, a comunicação continua a ser efectuada pelo lugar **Moving** agora visto como um lugar público dos objectos da classe **LocationIndicator**.
5. Os pedidos relativos ao funcionamento dos motores e das portas são feitos de forma explícita: as portas são abertas por pedidos associados às transições **Arrive at Destination** e **Serve**, e são fechadas por um pedido associado à transição **Close Doors**; os motores são postos a funcionar por um pedido associado à transição **Start Motor** e parados pelo pedido associado à transição **Stop**.

Classe **LocationIndicator**

A classe **LocationIndicator**, na Fig. 6.34, substitui de forma muito directa a página **UpDown** do primeiro modelo. Foi apenas adicionada uma transição responsável pela criação dos objectos. A comunicação com os objectos da classe **Cage** continua a ser feita pela fusão dos respectivos lugares **Moving**.

Classe **CageButtonAllocation**

No aspecto estrutural, os objectos da classe **CageButtonAllocation** na Fig. 6.35 agrupam os botões do elevador respectivo. No aspecto comportamental, servem de intermediário entre o elevador e os seus vários botões. A transição **AllocateRequest** recebe as notificações enviadas pelos objectos **CageButton** quando estes são premidos, e envia-as para o objecto **Cage** respectivo. A transição **arrived** recebe, do objecto **Cage** respectivo, a notificação de que um pedido foi satisfeito e notifica desse facto o botão respectivo.

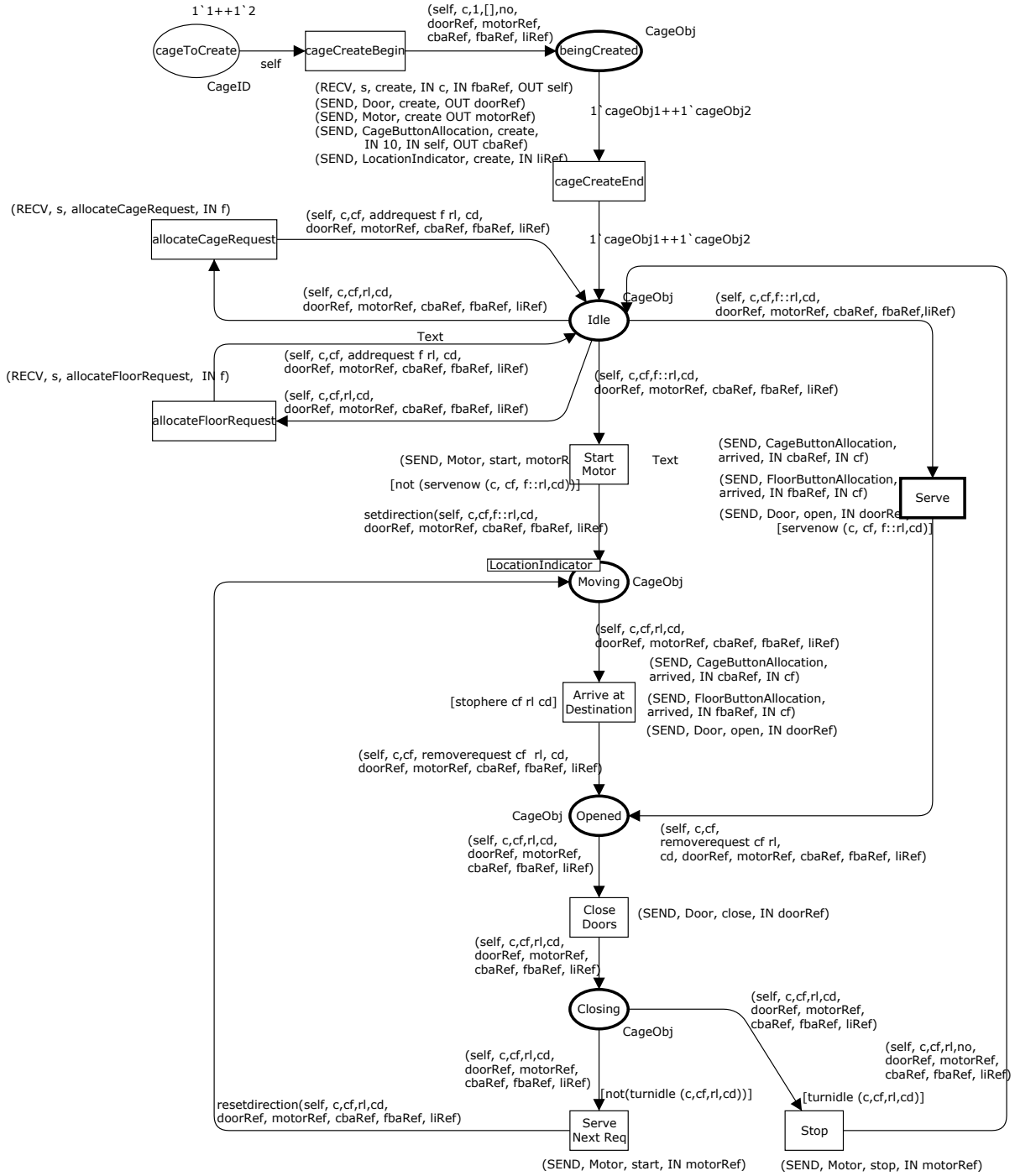


Figura 6.33: Classe de sistema **Cage**.

Para efectuar a tradução entre o número de piso e o objecto **CageButton** respectivo, o objecto **CageButtonAllocation** mantém uma lista de pares. Esta lista e os pedidos **arrived** e **allocateRequest** já referidos são herdados da classe abstracta **Allocation** (*vide* diagrama de classes na Fig. 6.31). Esta classe não é realizada sob a forma de uma página da rede de Petri colorida componível dado não se tratar de uma classe concreta. No entanto, a informação

envio da notificação respectiva (transição **Allocate Cage Request**) e a recepção da notificação de que o botão deve ser desactivado quando o pedido a ele associado já foi satisfeito (transição **turnOff**).

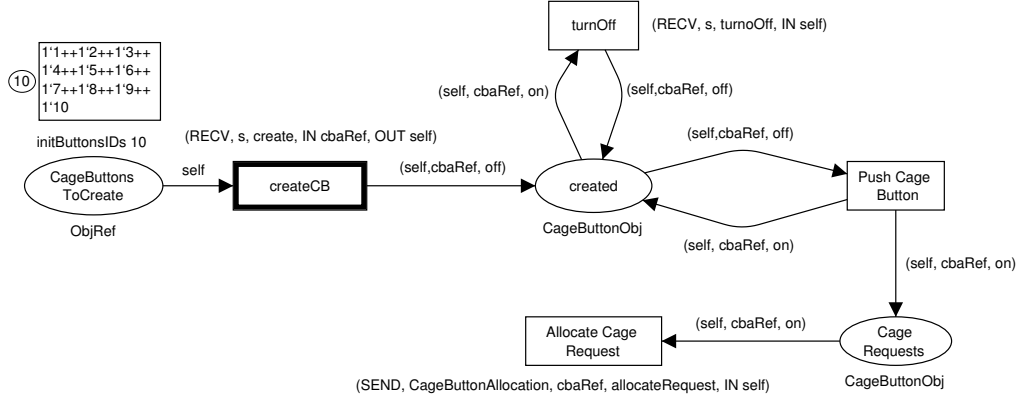


Figura 6.36: Classe de sistema **CageButton**.

Classe **FloorButtonAllocation**

A classe **FloorButtonAllocation** (*vide* Fig. 6.37) é muito semelhante à classe **CageButtonAllocation**. No entanto, como trata dos botões de piso existe apenas um objecto desta classe. Este objecto é criado pelo objecto **Initiator** (Fig. 6.32) que seguidamente cria os objectos da classe **Cage**. Nesse momento, o evento **create** na transição **createCages** passa-lhes a referência para o objecto **FloorButtonAllocation** (**fbaRef**). Por essa razão os objectos da classe **Cage** podem interagir com o objecto da classe **FloorButtonAllocation** de forma idêntica ao que fazem com os objectos das classe **CageButtonAllocation**. A única diferença prende-se com a escolha do elevador que vai satisfazer o pedido tratado pela transição **AllocateRequest** e proveniente de um botão de piso: esta escolha é feita de forma aleatória com base na escolha de uma das referências para objectos da classe **Cage** presentes no lugar **Cages** (*vide* canto inferior direito da Fig. 6.37). Este lugar é fundido com o lugar **FloorButtonAllocationCages** na classe **Initiator** (*vide* Fig. 6.32) como forma de ser inicializado com as referências dos objectos **Cage** aí criados.

Classe **FloorButton**

Tal como a classe **CageButton**, também a classe **FloorButton** (*vide* Fig. 6.38) implementa a interface **Button**. Os seus objectos representam os botões de piso e modelam o premir do botão (transição **Push Floor Button**), o envio da notificação respectiva (transição **Allocate Floor Request**) e a recepção da notificação de que o botão deve ser desactivado dado o pedido a ele associado já ter sido satisfeito (transição **turnOff**). É extremamente semelhante à classe **CageButton**. As únicas diferenças resultam de cada botão de piso admitir três estados (**up**,

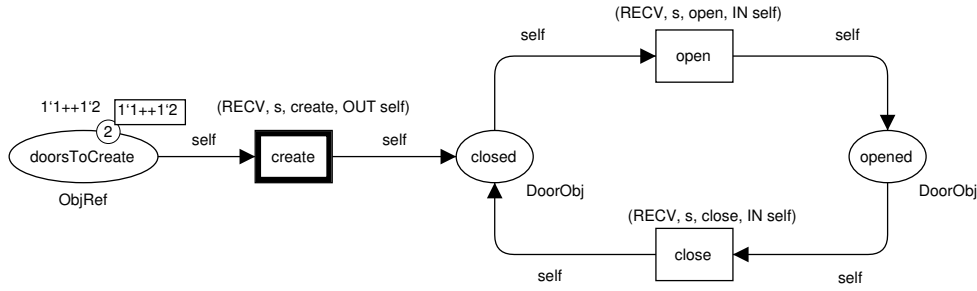


Figura 6.39: Classe de sistema Door.

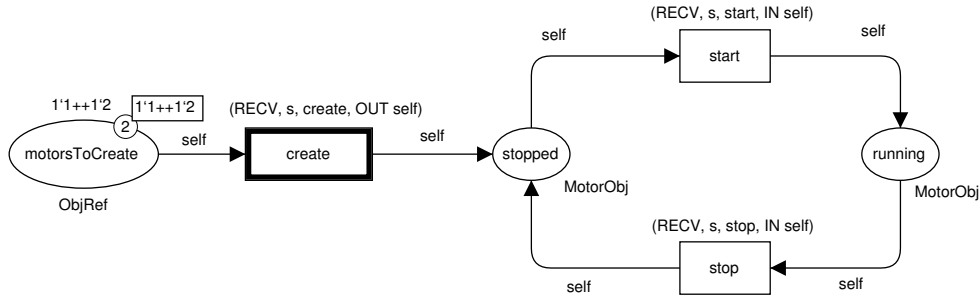


Figura 6.40: Classe de sistema Motor.

6.3.5 Diagrama de Adição

A Fig. 6.41 apresenta o diagrama de adição da rede de Petri colorida componível. Mais uma vez, este evidencia a semelhança estrutural entre a rede de Petri e o diagrama de classes respectivo (*vide* Fig. 6.31). Importa notar a ausência da classe abstracta **Allocation** e da interface **Button** dado serem consideradas na especificação das classes concretas que delas derivam (herdam).

Em conclusão, importa sublinhar a importância da semelhança estrutural entre a rede de Petri colorida componível e o diagrama de classes correspondente. Tal significa uma semelhança estrutural entre um diagrama de estrutura e um diagrama de comportamento. Só por si, tal constitui um aspecto inovador e útil para a efectiva utilização das redes de Petri no desenho orientado pelos objectos. Esta semelhança estrutural é ainda particularmente relevante porque não se encontra, na UML, entre o principal diagrama de estrutura — o diagrama de classes e objectos — e qualquer dos dois principais diagramas de comportamento, os estadosogramas e os diagramas de sequência. Na verdade, apenas entre os diagramas de classes e objectos e os diagramas de colaboração¹¹ existe semelhança estrutural significativa. No entanto, estes não apresentam as capacidades de modelação características das redes de Petri (paralelismo, sincronização, etc.), encontrando-se muito mais próximos dos diagramas de sequência.

¹¹Em inglês: *collaboration diagrams*.

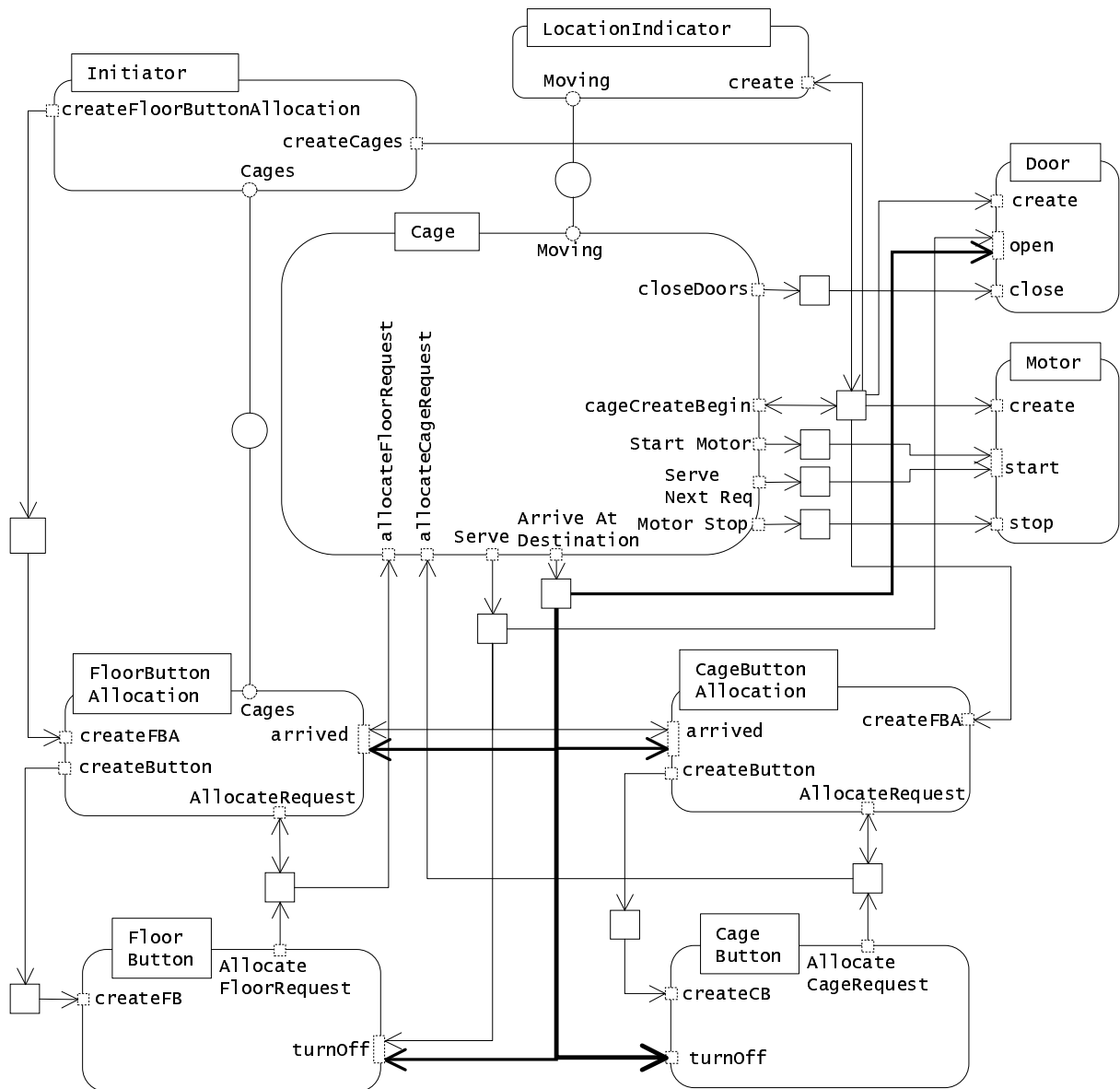


Figura 6.41: Diagrama de adição do controlador de elevadores utilizando uma rede de Petri colorida componível.

Também é interessante notar que, como resultado da dualidade intrínseca das redes de Petri, os diagramas de adição suportam duas formas de relação entre módulos: fusão de lugares e fusão de transições. A primeira pode corresponder à comunicação assíncrona, mas mais frequentemente irá servir para modelar partilha de dados. Já a fusão de transições é utilizada para modelar relações cliente-servidor equivalentes a chamadas de métodos. Tal permite distinguir algo que se encontra misturado nos diagramas de classes: relações de associação entre estruturas de dados e relações cliente servidor — relações com origem na visão estática misturadas com relações com origem na dinâmica do sistema — um problema para o qual vários autores têm chamado a atenção (e.g. [Simons e Graham, 1999] e [Génova et al., 2003]).

Em resumo, o diagrama de adição permite visualizar a rede de Petri colorida componível de uma

forma semelhante à oferecida pelo diagrama de classes, o que facilita a manutenção de coerência entre ambos. Tal permite também estabelecer uma ligação entre a rede de Petri e outras vistas do mesmo sistema.

Capítulo 7

Conclusão

*O atravessador de paredes foi a única coisa útil que Jácome inventou.
Tudo o resto nunca serviu para nada. Mas é muito importante.*

– José Eduardo Agualusa in *Estranhões, Bizarrocos e outros seres sem exemplo*

Em conclusão, discute-se o trabalho realizado e apontam-se dois conjuntos de trabalhos futuros: (1) os já planeados porque constituintes de projectos de investigação já aprovados; (2) os identificados como interessantes mas cuja realização não é ainda possível prever no tempo.

7.1 Sobre o que foi feito

Desde final da década de 1980, foram surgindo várias propostas para a estruturação de modelos em redes de Petri. Quase todas estas propostas se basearam na utilização de fusões entre lugares, entre transições ou ambas. Simultaneamente, foram surgindo cada vez mais classes de redes de Petri até que na segunda metade da década de 1990 começa a surgir a necessidade de encontrar um suporte genérico para todas essas classes de redes de Petri que permitisse alguma compatibilidade entre as várias ferramentas computacionais já existentes e futuras. Essa preocupação veio a ter a sua expressão máxima na linguagem Petri Net Markup Language (PNML). A acompanhar essa mesma linguagem surgem duas propostas para estruturação de modelos. No entanto, qualquer delas baseia-se num mecanismo minimalista de fusão de nós que não permite a efectiva fusão das anotações e consequente geração de novas anotações. A primeira parte desta dissertação apresentou uma proposta para uniformizar também as formas de estruturação, composição e modificação de modelos em qualquer classe de redes de Petri expressa em PNML. Essa proposta corresponde às operações de adição e subtracção. Para especificar os modelos resultantes da aplicação destas linguagens, foram também apresentadas

duas linguagens textuais, PN_{TEXT} e *Operational PNML* (OPNML), e uma linguagem gráfica denominada *diagramas de adição*.

Algumas das propostas, para estruturação de modelos em redes de Petri, aproveitaram a crescente popularidade do desenvolvimento orientado pelos objectos e propuseram extensões mais ou menos sofisticadas para as redes de Petri. Infelizmente estas extensões parecem perder algumas das vantagens das redes de Petri e algumas das vantagens de outras linguagens para descrição de comportamento mais utilizadas no desenvolvimento orientado pelos objectos. Mais especificamente, a complexidade adicional resultante da mistura de dois paradigmas à custa de sintaxes e semânticas adicionais parece afastar os potenciais utilizadores. É interessante comparar com o que sucedeu nas linguagens de programação. Quando o desenvolvimento orientado pelos objectos começou finalmente a tornar-se popular foi em grande medida porque Stroustrup resolveu partir da linguagem C [Kernighan e Ritchie, 1988] e acrescentar-lhe as classes [Stroustrup, 1994], um conceito que ele tinha encontrado na linguagem Simula. Tal permitiu ao C com classes e mais tarde, e muito em especial, ao C++ tornar-se a linguagem orientada pelos objectos mais popular contribuindo em grande medida para o sucesso do próprio paradigma. Algo semelhante foi tentado pelos autores das várias classes de redes de Petri orientadas pelos objectos: partir das redes de Petri coloridas e adicionar-lhe o que é preciso para as "objectificar". Esta estratégia não resultou, fundamentalmente porque as redes de Petri coloridas não eram e ainda não são tão populares no seio das linguagens gráficas para especificação de comportamento, como a linguagem C era e é no mundo das linguagens de programação imperativas estruturadas. Assim, o adicionar de características orientadas pelos objectos às redes de Petri coloridas não as tornou mais apelativas mas apenas mais complicadas.

Diferentemente, esta dissertação partiu da ideia de que a melhor forma de utilizar o paradigma do desenvolvimento orientado pelos objectos nas redes de Petri seria a de conseguir criar modelos orientados pelos objectos utilizando uma classe de redes de Petri já existente e o mais popular possível. A melhor candidata era claramente uma classe de redes de Petri de alto-nível, muito em especial as redes de Petri coloridas. Isto por duas razões: (1) por ser a classe de redes de Petri de alto-nível mais conhecida e, muito provavelmente, também a mais utilizada; (2) por ser a mais bem suportada em termos de ferramentas computacionais.

Tal como é possível criar programas orientados pelos objectos utilizando a linguagem C, também seria possível criar modelos orientados pelos objectos utilizando redes de Petri coloridas, desde que fosse possível identificar os idiomas adequados. Tal foi demonstrado na segunda parte desta dissertação em que um conjunto de idiomas e técnicas permitem a utilização de redes de Petri Coloridas no desenvolvimento orientado pelos objectos, nomeadamente, na fase de desenho. Para a efectivação dessas técnicas, definiu-se uma classe de redes de Petri que define de forma precisa a sintaxe e a semântica a utilizar. No entanto, esta classe limita-se a precisar a utilização dada às redes de Petri coloridas adicionando-lhes apenas uma forma particular dos canais síncronos. Esta constitui uma adição muito ligeira à sintaxe e semântica das redes de Petri coloridas mas que permite uma muito significativa simplificação dos modelos.

É interessante notar que a construção mais polémica da programação orientada pelos objectos, a herança de implementação, é de facto aquela que pior se adapta à modelação com redes de Petri. Parece por isso, que as redes de Petri obrigam a herança de implementação a mostrar a sua pouca elegância. Já a composição, a agregação e mesmo o polimorfismo são facilmente modelados pela rede de Petri colorida com canais síncronos.

Verificou-se também que os diagramas de adição quando aplicados no contexto de modelos de redes de Petri orientados pelos objectos, permitem visualizar uma semelhança estrutural entre redes de Petri e diagramas de classes, desta forma estabelecendo uma ligação entre as redes de Petri e outras vistas do sistema modelado. Naturalmente, tal facilita também a manutenção de coerência entre as diversas vistas. Esta semelhança estrutural merece ser mais estudada mas é já defensável que, neste aspecto, as redes de Petri coloridas são superiores a algumas das linguagens comportamentais presentes na UML, nomeadamente os estadogramas e os diagramas de sequência.

7.2 Trabalho Futuro

Divide-se o trabalho futuro entre aquele que já se encontra calendarizado, e o que se julga merecedor de futura investigação ainda não planeada.

7.2.1 Trabalho Futuro já Planeado

Ao longo desta dissertação foram desenvolvidos exemplos académicos e dois protótipos que permitiram validar as ideias fundamentais apresentadas. Embora o autor esteja certo da aplicabilidade das propostas efectuadas, fica ainda por demonstrar a sua aplicação num exemplo de dimensão realista. Está o autor ciente que tal só é possível a partir do momento em que existam ferramentas que suportem a especificação de sistemas de grande dimensão. Exemplos da utilização de métodos formais construídos sem a utilização de ferramentas, são ainda e apenas uma extensão à teoria. Embora úteis, estes exemplos não podem representar a verdadeira validação da utilidade do método em causa: a sua aplicabilidade. Esta só pode ser aferida através da criação de exemplos de dimensão realista, só realizáveis através de ferramentas adequadas.

Em particular, prevê-se a necessidade de ferramentas de suporte às propostas constantes da primeira e segunda parte desta dissertação: (1) uma ferramenta capaz de por em prática a adição de redes especificada a partir da linguagem *PN_{TEXT}* ou da linguagem *OPNML*; (2) outra ferramenta (ou a mesma) que suporte a construção de modelos utilizando redes de Petri coloridas componíveis.

Estando o autor certo da impossibilidade temporal de obter estas ferramentas como parte in-

tegrante dos trabalhos que conduziram a esta dissertação, é com especial agrado que vê essa possibilidade assumir contornos realistas devido à aprovação de um projecto de investigação proposto em 2004 e aprovado em 2005 pela Fundação para a Ciência e a Tecnologia. Entre outras tarefas, esse projecto prevê a construção de componentes do ambiente descrito no Apêndice C, o que inclui o suporte para as propostas na Parte I. Prevê-se que com um pouco mais de esforço será possível desenvolver também o suporte para a utilização das redes de Petri coloridas componíveis. O projecto também irá permitir a efectiva realização de um gerador de código para redes de Petri *input-output* que suporte as tarefas de análise e verificação de código. Prevê-se também a construção do já referido editor que suportará descrições nas linguagens PNML, OPNML e PN_{TEXT} .

É pois com fundamentada esperança que se verifica a existência de uma oportunidade real e efectiva de ver realizado o sistema descrito nesta dissertação e, muito em particular, as linguagens de composição de modelos que estiveram no cerne deste trabalho.

7.2.2 Outros Trabalhos Futuros

No final é possível olhar para trás e identificar não apenas vários pontos que merecem ser mais explorados, mas também linhas de investigação que continuam, em grande medida, por explorar. A lista seguinte, enquadra todos estes na estrutura desta dissertação:

1. Infelizmente, nem as macros nem as invocações são exemplos claros das abstracções no sentido usual do termo definido, por exemplo, por Gries: "Por abstracção entendemos o acto de evidenciar algumas propriedades de um objecto com o objectivo de o utilizar ou estudar, ao mesmo tempo que não consideramos outras propriedades que, por enquanto, não nos interessam" [Gries, 1981]. Esta definição vê as abstracções como um filtro de propriedades e suporta a utilização da abstracção como uma representação simplificada do estado de um determinado elemento. Nas superpáginas, as macros e as invocações são normalmente representadas como um tipo especial de lugar ou transição associado a outra rede. No entanto, os macrolugares não têm marcas e as macrotransições não disparam instantaneamente. Claramente, as semânticas e visualizações fornecidas por estas abstracções distam mais do que seria desejável das dos objectos que lhes servem de inspiração: os lugares e transições. A literatura sobre este problema é escassa e os trabalhos que abordam este tema ([Buchholz, 1994] e Lakos [Lakos, 1997, 2000]) não tiveram ainda concretização prática, provavelmente por obrigarem a demasiados cuidados por parte do modelador. Assim, este é um tema que merece ser estudado, em especial no que respeita ao seu suporte em ferramentas computacionais de modelação.
2. Na linguagem PN_{TEXT} pode ser interessante permitir a especificação de transformações de anotações *caso a caso*: para cada conjunto de fusão e para cada anotação, seria possível especificar uma dada função de transformação diferente da utilizada para a mesma anotação

noutros conjuntos de fusão. Tal tem o risco de uma menor estruturação no que respeita às transformações de anotações, mas importa verificar na prática a sua real utilidade. Mais uma vez, as ferramentas são fundamentais.

3. Com o devido suporte computacional, a linguagem PN_{TEXT} poderá ser útil no ensino. Em particular, pode ser utilizada para definir modelos constituídos por máquinas de estado concorrentes que comuniquem de forma síncrona ou assíncrona.
4. Os diagramas de adição utilizados no Capítulo 3 são suficientemente simples e legíveis para poderem ser utilizados como auxiliares na criação de modelos em várias ferramentas já existentes.
5. A adição e subtracção de redes, parecem suficientemente genéricas para poderem ser facilmente adaptadas a outras linguagens gráficas. Nesse sentido e no âmbito desta dissertação, foi já desenvolvido um trabalho preliminar o qual pode contribuir para uma possível linha de investigação [Barros e Gomes, 2003c].
6. Também o suporte ao desenho orientado pelos aspectos possibilitado pela utilização de adição e subtracção pode ser adaptado a outras linguagens gráficas. Também aqui foi já publicado um trabalho preliminar [Barros e Gomes, 2002].
7. Quanto às propostas apresentadas na Parte II é intenção do autor e do grupo de investigação em que se integra, continuar a explorar as capacidades das redes de Petri coloridas componíveis na criação de modelos orientados pelos objectos o que incluirá a exploração das semelhanças estruturais entre diagramas de classe e diagramas de adição.

Apêndice A

Gramática da Linguagem PN_{TEXT}

A sintaxe da linguagem PN_{TEXT} é aqui ilustrada por um conjunto de exemplos que se pretendem ilustrativos da sintaxe da linguagem. Seguidamente, define-se a respectiva gramática EBNF. Esta gramática foi definida de acordo com a sintaxe do gerador de parsers JavaCC [s.a., 2005b].

Listagem A.1: Exemplos de instruções permitidas pela linguagem PN_{TEXT} .

```
1  N := Net("OneNetID")
   N := (N[0...3] + M[2...5])((/N.t1[i]/M.t1[i+1]/-> tt[i+j])[i:0...3])[j:10...12]
   //-----
   R2cf := (Na + Nb[0...1])(/t1/Nb[i].t2/ -> t[i])[i:0...1]
   R1cf := (Na + Nb[0...1])(/t1/(Nb[i].t2)[i:0...1]/ -> t)
6  //-----
   E3Park2Exit2Counter := (E3Park2Exit2 + ParkingArea + Alternate)
                           ( (/E3Park2Exit2.E2Park.in[i]/ParkingArea.in/
                              -> in[i])[i:1...2] ,
                              /E3Park2Exit2.ParkAPassage2.ParkA.Enter.in/ParkingArea.in/
11                              -> in[3] ,
                              /E3Park2Exit2.E2Park.Leave.out/ParkingArea.out/turn[1]/
                              -> out[1] ,
                              /E3Park2Exit2.ParkAPassage2.ParkA.Leave.out/ParkingArea.out/turn[2]/
                              -> out[2])
16 //-----
   ParkExit := (ParkA - Enter)
               (/ParkA.Enter.in /Enter.gotTicket/ -> in)
               (/ParkA.Enter.entranceFree/Enter.entranceFree/ ,
               /ParkA.Enter.arriveUp/Enter.arriveUp/ ,
21 /ParkA.Enter.arriveDown/Enter.arriveDown/ ,
               /ParkA.Enter.waitingTicket/Enter.waitingTicket/ ,
               /ParkA.Enter.gateOpenE/Enter.gateOpenE/
               )
26 Dinner5 := (Phil[0] + Phil[1] + Phil[2] + Phil[3] + Phil[4] +
               Fork[0] + Fork[1] + Fork[2] + Fork[3] + Fork[4])
               (/Phil[0].right/Phil[1].left/Fork[0].f/ -> f[0] ,
               /Phil[1].right/Phil[2].left/Fork[1].f/ -> f[1] ,
               /Phil[2].right/Phil[3].left/Fork[2].f/ -> f[2] ,
31 /Phil[3].right/Phil[4].left/Fork[3].f/ -> f[3] ,
               /Phil[4].right/Phil[0].left/Fork[4].f/ -> f[4])
```

Dinner := (Phil[0...n-1] + Fork[0...n-1])
 (/Phil[i].right/Phil[(i+1) MOD n].left/Fork[i].f/ -> f[i])[i:0...n-1]

Listagem A.2: Gramática da linguagem PN_{TEXT} em JavaCC.

```

/** PnText grammar
    author: Joao Paulo Barros
    version: 1.0
4  */

PARSER_BEGIN(PnText)

9  public class PnText {
    public static void main(String args[]) throws ParseException {
        PnText parser = new PnText(System.in);
        parser.start();
14 }

PARSER_END(PnText)

19 SKIP :
{
    " "
    | "\t"
    | "\n"
24 | "\r"
    | "\r\n"
}

SPECIAL_TOKEN : /* COMMENTS */
29 {
    <SINGLE_LINE_COMMENT: "//" (~["\n", "\r"])* (" \n" | " \r" | " \r\n")>
    | <FORMAL_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*", "/"] (~["*"])* "*" ))* "/">
    | <MULTILINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*", "/"] (~["*"])* "*" ))* "/">
34 }

/** Reserved words */
TOKEN : {
    <NET: "Net" >
    | <PLACE: "Place" >
39 | <PLACES: "places" >
    | <TRANSITION: "Transition" >
    | <TRANSITIONS: "transitions" >
    | <ARC: "Arc" >
    | <ARCS: "arcs" >
44 | <MOD: "MOD" >
    | <DIV: "DIV" >
}

TOKEN : {
49 <EOL : "\n" | "\r" | "\r\n" >
    | <LEFT_SQUARE: "[" >
    | <RIGHT_SQUARE: "]" >
    | <LEFT_PAR: "(" >

```

```

| <RIGHT_PAR: ")" >
54 | <INT: ([ "0" - "9" ])+ >
| <ID: ([ "a" - "z", "A" - "Z", "_" ])+ ([ "a" - "z", "A" - "Z", "_", "0" - "9" ])* >
| <ID_STRING: "\"" ([ "a" - "z", "A" - "Z", "_" ])+ ([ "a" - "z", "A" - "Z", "_", "0" - "9" ])* "\"" >
| <REAL: ([ "0" - "9" ])+ "." ([ "0" - "9" ])+ >
| <ASSIGN: ":@" >
59 }

void start() : { } {
    (statement())+ <EOF>
    {System.out.println("ok!");} }

64 void statement() : { } {
    assignment() }

void assignment() : { } {
69 <ID> ( newNetDefinition() | netPartDefinition() )}

void newNetDefinition() : { } {
    <ASSIGN>
    (
74 net()
    |
    operation() (collapse() (remove())?)?
    )}

79 //*****
/** operation between nets */
void operation() : { } {
    <LEFT_PAR> (" -")? idv() ((" + " | " - ") idv())* <RIGHT_PAR> }

84 void idv() : { } { // ex.: aNet[2...3]
    <ID> (subVectorIndexes())? }

void subVectorIndexes() : { } { // ex.: [2] ou [i + 3] ou [2...3] ou [i+1...3]
    <LEFT_SQUARE> intExpression() ("..." intExpression())? <RIGHT_SQUARE> }

89 void collapse() : { } { // ex.: (net1.t1 / net2.t3[i], p1/p2 -> a)[i:0...3]
    <LEFT_PAR> collapseContents() <RIGHT_PAR> (iterator())?
    }

94 void collapseContents() : { } {
    (collapse() | namedFusionSet()) ("," (collapse() | namedFusionSet()))* }

void namedFusionSet() : { } { // ex.: /Nb.a/b.c/a/x.y[2...3]/ -> a.b[2]
    "/" (completeIDVector() "/" )+ "->" completeIDUnique() }

99 void completeIDVector() : { } { // ex.: Nb.Na[i+4].t2 ou (Nb.Na[i+4].t2)[i:0...3]
    completeID()
    |
    <LEFT_PAR> completeID() <RIGHT_PAR> iterator() }

104 void completeID() : { } { // ex.: Nb.Na[i+4].t2
    idPart() ("." idPart())* }

void idPart() : { } { // ex.: a[2] ou p[i+3] ou p[2...3] ou p[i+1...3]
109 <ID> (subVectorIndexes())? }

```

```

void completeIDUnique() : { } { // ex.: Nb.Na[i+4].t2
    idPartUnique() (". " idPartUnique())* }

114 void idPartUnique() : { } { // ex.: a[2] ou p[i+3]
    <ID> (<LEFT_SQUARE> intExpression() <RIGHT_SQUARE>)? }

void iterator() : { } { // ex.: [i:0...3]
    <LEFT_SQUARE> <ID> ":" intExpression() "... " intExpression() <RIGHT_SQUARE> }

119 /** integer expressions should be the ones supported by the used
    inscription language. Here we only specify some tokens
    that should be allowed*/
void intExpression() : { } {
124    (<LEFT_PAR> | <RIGHT_PAR> | <INT> | <ID> | <MOD> | <DIV> | "+" | "-" | "*" | "/" )+ }

    /** *****
    /** for node removal in subtraction */

129 void remove() : { } { // ex.: (/net1.t1 / net2.t3/, /p1/p2/)[i:0...3]
    <LEFT_PAR> removeContents() <RIGHT_PAR> (iterator())?
    }

    void removeContents() : { } {
134    (remove() | fusionSet()) ("," (remove() | fusionSet()))* }

    void fusionSet() : { } { // ex.: /Nb.a/b.c/a/x.y[2...3]/
    "/" (completeIDVector() "/" )+ }

139 /** *****
    /** net "constructor" */
void net() : { } {
    <NET> <LEFT_PAR> netContents() <RIGHT_PAR> }

144 /** net contents */
void netContents() : { } {
    <ID_STRING> ("," parameter())* }

void netPartDefinition() : { } {
149    "." part() }

void part() : { } {
    <PLACES> <ASSIGN> places()
    |
154    <TRANSITIONS> <ASSIGN> transitions()
    |
    <ARCS> <ASSIGN> arcs() }

    /** places ' list */
159 void places() : { } {
    (place()) ("+" place())* }

    /** place "constructor" */
void place() : { } {
164    <PLACE> <LEFT_PAR> ptContents() <RIGHT_PAR> }

    /** transitions ' list */
void transitions() : { } {
    (transition()) ("+" transition())* }

```

```

169  /** transition "constructor" */
    void transition() : { } {
        <TRANSITION> <LEFT_PAR> ptContents() <RIGHT_PAR> }

174  /** place or transition contents */
    void ptContents() : { } {
        <ID_STRING> ("," parameter())* }

    /** arcs list */
179  void arcs() : { } {
        (arc()) ("+" arc())* }

    /** arc "constructor" */
    void arc() : { } {
184        <ARC> <LEFT_PAR> arcContents() <RIGHT_PAR> }

    /** arc contents */
    void arcContents() : { } {
        <LEFT_PAR> <ID_STRING> "," <ID_STRING> <RIGHT_PAR>
189        ("," parameter())* }

    /** constructor parameter */
    void parameter() : { } {
        <INT> | <REAL> | <ID> | <ID_STRING> }

```


Apêndice B

Gramática da Linguagem *Operational PNML*

A sintaxe da linguagem *Operational PNML* é aqui definida por uma gramática em Relax NG [s.a., 2003c], apresentada na Listagem B.1.

Listagem B.1: Gramática da linguagem *Operational PNML*.

```
<?xml version="1.0" encoding="UTF-8"?>
2 <grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <a:documentation>
    Operational PNML v0.2 by Joao Paulo Barros and Luis Gomes
7    Last revision: 2004/06/17
    Strongly based on the
    Petri Net Markup Language schema
    RELAX NG implementation of basic PNML
    version: 1.3.2b
12    according to the paper by Billington et al
    (c) 2001–2004
    Michael Weber (mweber@informatik.hu-berlin.de),
    Ekkart Kindler,
    Christian Stehno (for the graphical elements)
17  </a:documentation>
  <include href="http://www.informatik.hu-berlin.de/top/pnml/basicPNML.rng">
    <define name="net.element">
      <element name="net">
        <a:documentation>
22          A net has a unique identifier (id) and refers to
            its Petri Net Type Definition (PNTD) (type).
        </a:documentation>
        <attribute name="id">
          <data type="ID"/>
27        </attribute>
        <attribute name="type">
          <ref name="nettype.uri"/>
        </attribute>
```

```

32      <a:documentation>
      The sub-elements of a net may occur in any order.
      A net consists of several net labels (net.labels), several
      objects (net.content), tools specific information, and a set of
      graphical information in any order.
      Alternatively, in Operational PNML, the net can be defined by
37      a set of operations.
      </a:documentation>
      <interleave>
      <ref name="net.labels"/>
      <choice>
42      <ref name="net.operations"/>
      <group>
      <zeroOrMore>
      <ref name="net.content"/>
      </zeroOrMore>
47      <zeroOrMore>
      <ref name="toolspecific.element"/>
      </zeroOrMore>
      <optional>
      <element name="graphics">
52      <ref name="netgraphics.content"/>
      </element>
      </optional>
      </group>
      </choice>
57      </interleave>
      </element>
      </define>
</include>
<a:documentation>#####</a:documentation>
62 <a:documentation>##### Operations #####</a:documentation>
<a:documentation>#####</a:documentation>
<define name="net.operations">
  <a:documentation>A net can be just a set of operations.</a:documentation>
  <element name="operations">
67    <interleave>
    <zeroOrMore>
    <element name="addition">
    <ref name="addition.content"/>
    </element>
72    </zeroOrMore>
    <zeroOrMore>
    <element name="subtraction">
    <ref name="subtraction.content"/>
    </element>
77    </zeroOrMore>
    </interleave>
    </element>
  </define>
  <a:documentation>#####</a:documentation>
82 <a:documentation>##### Addition #####</a:documentation>
<a:documentation>#####</a:documentation>
<define name="addition.content">
  <a:documentation>Multiplication performs multiple additions on a set of nets.</a:documentation>
  <ref name="result"/>
87  <oneOrMore>
  <choice>

```

```

        <ref name="netRef"/>
        <ref name="netRefVector"/>
    </choice>
92 </oneOrMore>
    <optional>
        <element name="merge">
            <oneOrMore>
                <choice>
97         <ref name="placeFusion.element"/>
            <ref name="transitionFusion.element"/>
            <ref name="placeFusionVector.element"/>
            <ref name="transitionFusionVector.element"/>
                </choice>
102        </oneOrMore>
    </element>
</optional>
</define>
<a:documentation>#####</a:documentation>
107 <a:documentation>##### Subtraction #####</a:documentation>
<a:documentation>#####</a:documentation>
<define name="subtraction.content">
    <a:documentation>
112        Given a list of nets and two sets of fusion sets , subtraction uses
        the first set of fusion sets (interface) to perform a net addition .
        The resulting net is this net , after removing the fused nodes in
        the second set .
    </a:documentation>
    <ref name="result"/>
117 <oneOrMore>
    <choice>
        <ref name="netRef"/>
        <ref name="netRefVector"/>
    </choice>
122 </oneOrMore>
    <element name="merge">
        <oneOrMore>
            <choice>
127         <ref name="placeFusion.element"/>
            <ref name="transitionFusion.element"/>
            <ref name="placeFusionVector.element"/>
            <ref name="transitionFusionVector.element"/>
                </choice>
        </oneOrMore>
132 </element>
    <element name="removal">
        <oneOrMore>
            <choice>
137         <ref name="placeFusion.element"/>
            <ref name="transitionFusion.element"/>
            <ref name="placeFusionVector.element"/>
            <ref name="transitionFusionVector.element"/>
                </choice>
        </oneOrMore>
142 </element>
</define>
<a:documentation>#####</a:documentation>
<a:documentation>##### placeRef and transitionRef #####</a:documentation>
<a:documentation>#####</a:documentation>

```

```

147  <define name="placeRef">
      <element name="place">
        <ref name="nodeRef.content"/>
      </element>
    </define>
152  <define name="transitionRef">
      <element name="transition">
        <ref name="nodeRef.content"/>
      </element>
    </define>
157  <define name="nodeRef.content">
      <attribute name="netID">
        <data type="token"/>
      </attribute>
      <attribute name="nodeID">
162    <data type="token"/>
      </attribute>
    </define>
    <a:documentation>#####</a:documentation>
    <a:documentation>##### result #####</a:documentation>
167  <a:documentation>#####</a:documentation>
    <define name="result">
      <a:documentation>Result attribute.</a:documentation>
      <attribute name="result">
        <data type="ID"/>
172    </attribute>
    </define>
    <a:documentation>#####</a:documentation>
    <a:documentation>##### netRef and netRefVector #####</a:documentation>
    <a:documentation>#####</a:documentation>
177  <define name="netRef">
      <a:documentation>Reference to one net.</a:documentation>
      <element name="net">
        <ref name="netRef.content" />
      </element>
182  </define>
    <define name="netRefVector">
      <a:documentation>Reference to a vector of nets.</a:documentation>
      <element name="netVector">
        <ref name="netRef.content" />
187      <attribute name="first">
        <data type="nonNegativeInteger"/>
      </attribute>
      <attribute name="last">
        <data type="nonNegativeInteger"/>
192      </attribute>
    </element>
    </define>
    <define name="netRef.content">
      <attribute name="netID">
197      <data type="token"/>
      </attribute>
      <optional>
        <attribute name="sign">
          <data type="token"/>
202      </attribute>
      </optional>
    </define>

```

```

207 <a:documentation>#####</a:documentation>
    <a:documentation>##### Fusion Sets #####</a:documentation>
    <a:documentation>#####</a:documentation>
    <define name="transitionFusion.element">
        <element name="transitionFusion">
            <ref name="transitionFusion.content"/>
        </element>
212 </define>
    <define name="placeFusion.element">
        <element name="placeFusion">
            <ref name="placeFusion.content"/>
        </element>
217 </define>
    <define name="transitionFusion.content">
        <attribute name="nodeID">
            <data type="token"/>
        </attribute>
222 <oneOrMore>
            <ref name="transitionRef"/>
        </oneOrMore>
    </define>
    <define name="placeFusion.content">
227 <attribute name="nodeID">
            <data type="token"/>
        </attribute>
        <oneOrMore>
            <ref name="placeRef"/>
232 </oneOrMore>
    </define>
    <define name="placeFusionVector.element">
        <element name="placeFusionVector">
            <attribute name="iterator">
237 <data type="NMTOKEN"/>
            </attribute>
            <attribute name="first">
                <data type="nonNegativeInteger"/>
            </attribute>
242 <attribute name="last">
                <data type="nonNegativeInteger"/>
            </attribute>
            <ref name="placeFusion.content"/>
        </element>
247 </define>
    <define name="transitionFusionVector.element">
        <element name="transitionFusionVector">
            <attribute name="iterator">
                <data type="NMTOKEN"/>
252 </attribute>
            <attribute name="first">
                <data type="nonNegativeInteger"/>
            </attribute>
            <attribute name="last">
257 <data type="nonNegativeInteger"/>
            </attribute>
            <ref name="transitionFusion.content"/>
        </element>
    </define>
262 </grammar>

```


Apêndice C

Redes de Petri para Controladores Digitais

Este apêndice contém duas secções que correspondem a trabalho preliminar no sentido de criar um ambiente de desenvolvimento baseado em redes de Petri.

Esse ambiente irá suportar a linguagem PNML e qualquer classe de redes de Petri que possa ser especificada no contexto dessa linguagem (tipicamente, utilizando um *Petri Net Type Definition*). A proposta inclui também um editor gráfico que suportará a linguagem PN_{TEXT} e OPNML, bem como um gerador de código capaz de produzir código executável em vários tipos de plataformas. Através de interfaces adequadas, este mesmo código pode ser utilizado para a simulação e análise dos modelos a partir dos quais foi produzido.

A primeira secção apresenta uma nova classe de redes de Petri que constituirá a primeira classe de redes a ser suportada pelo ambiente. Denomina-se **rede de Petri *input-output*** e destina-se à modelação do comportamento de controladores de sistemas conduzidos por eventos discretos. Dado que se procurou simplicidade e um conjunto mínimo de modificações a classes de redes de Petri bem conhecidas, as redes de Petri *input-output* constituem uma extensão simples e que se pretende intuitiva às bem conhecidas e estudadas redes de Petri lugar-transição [Reisig, 1985; Desel e Reisig, 1998]. As extensões têm como objectivo a aplicação desta classe de redes de Petri na modelação e geração automática de código executável em controladores, também aplicável em plataformas com poucos recursos computacionais.

A segunda secção apresenta a proposta para o ambiente de desenvolvimento baseado em redes de Petri. Finalmente, apresenta-se um exemplo de um modelo expresso numa rede de Petri *input-output*.

C.1 Uma Classe de Redes para Controladores Digitais

Uma área típica de aplicação das redes de Petri é a da modelação de sistemas conduzidos por eventos discretos. Numa sistematização das várias utilizações de redes de Petri na análise e síntese de controladores para sistemas conduzidos por eventos discretos, Holloway, Krogh e Giua [Holloway et al., 1997] identificam três abordagens principais:

1. A abordagem do controlo do comportamento.
2. A abordagem do controlador lógico.
3. A abordagem baseada na teoria para o controlo de sistema contínuos.

A primeira é a mais utilizada na modelação de sistemas de manufactura. Nela, o controlador e a fábrica¹ são modelados em conjunto. Quando o comportamento desejado é obtido, é então necessário extrair o controlador do modelo.

Na abordagem do controlador lógico modela-se o comportamento do próprio controlador. Nesse sentido, a rede passa a modelar entradas e saídas. Esta abordagem facilita a implementação do controlador mas obriga à simulação em malha fechada (ou em anel fechado²). Corresponde também à aproximação seguida por David e Alla [David e Alla, 1992; Silva, 1985] e outros (e.g. [Gomes e Steiger-Garção, 1995; Gomes, 1997]) aquando da utilização de redes de Petri sincronizadas ou redes de Petri interpretadas.

Na terceira abordagem, o objectivo é sintetizar o controlador a partir de um modelo do comportamento da fábrica e uma especificação do comportamento desejado para a malha fechada.

Neste capítulo propõe-se uma nova classe de redes de Petri, as redes de Petri *input-output*. Estas redes são aplicáveis no âmbito do segundo tipo de abordagem acima referida, ou seja, na modelação do controlador lógico. O objectivo é o da implementação física do controlador. Assim, a rede modela o comportamento do controlador: as entradas do controlador ficam associadas à evolução da rede, nomeadamente às transições, e as saídas do controlador ficam associadas, quer à evolução da rede (disparo da transições) quer ao estado da rede (marcações dos lugares).

O desenho desta nova classe de redes de Petri teve por base os seguintes três objectivos:

1. Suporte para a modelação de sinais e eventos externos.
2. Proximidade sintáctica e semântica relativamente a uma classe de redes de Petri bem conhecida e bem estudada (redes lugar-transição).

¹Em inglês: *plant*.

²Em inglês: *closed-loop*.

3. Simplicidade de execução.

O modelo de execução implícito no desenho das redes de Petri *input-output* corresponde à utilização da rede de Petri como modelo do comportamento do próprio controlador lógico. Como tal, a rede modela explicitamente eventos de entrada, sinais de entrada, eventos de saída e sinais de saída. A especificação de eventos e sinais de entrada externos é associada às transições. Cada transição pode ter associados zero ou mais eventos de entrada e também zero ou mais eventos de saída. É também possível associar, a cada transição, uma função booleana, cujas variáveis são sinais de entrada. Esta função booleana corresponde a uma guarda da transição: à semelhança das redes de Petri coloridas estas guardas são condição necessária, mas não suficiente, para a habilitação das transições respectivas. Por último, os sinais de saída podem depender das marcações dos lugares.

Conforme já referido, as redes de Petri *input-output* baseiam-se nas redes de Petri lugar-transição. Estas constituem a mais conhecida das classes de redes de Petri de baixo-nível e aquela a que é frequente a designação redes de Petri corresponder. As extensões necessárias para a especificação de entradas e saídas adicionam uma quantidade mínima de novas anotações. Estas possibilitam uma ligação semântica explícita entre o modelo da rede de Petri e o ambiente em que o controlador irá operar. Por outras palavras, as redes de Petri *input-output* são redes não-autónomas pois a sua evolução pode depender de eventos e sinais externos ao modelo do controlador (*vide* [David e Alla, 1992; Gomes e Steiger-Garção, 1995; Gomes, 1997]).

Seguidamente apresenta-se o tipo de sistema para o qual as redes de Petri *input-output* foram desenhadas. Posteriormente, definem-se as redes de Petri *input-output*.

C.1.1 Um Modelo para o Controlo de Sistemas Conduzidos por Eventos Discretos

Os sistemas aqui considerados podem disponibilizar sinais de entrada que permitem que o controlador percepcione o estado do sistema, e sinais de saída que permitem que o controlador actue sobre o sistema. Os primeiros têm origem em **sensores**; os segundos afectam **actuadores**. Adicionalmente, definem-se eventos de entrada que modelam alterações entre dois estados consecutivos de sinais de entrada específicos. Definem-se também eventos de saída associados a sinais de saída e que são activados pelo disparo de transições. Assim, a interface do controlador é definida da seguinte forma:

Definição C.1 (Interface do Controlador): A interface de uma rede de Petri *input-output* com o sistema a controlar é um tuplo $IC = (SE, SEE, SS, SSL, EE, ES)$ que satisfaz os seguintes requisitos:

1. SE é um conjunto finito de sinais de entrada, incluindo os sinais de entrada externos (SEE) e os sinais de saída que podem ser lidos (SSL): $SE = SEE \cup SSL$.

2. SEE é um conjunto finito de sinais de entrada externos.
3. SS é um conjunto finito de sinais de saída.
4. SSL é um conjunto finito de sinais de saída que podem ser lidos. Como tal, semanticamente, são sinais de entrada e sinais de saída, surgindo por isso em ambos os conjuntos: $SSL = SE \cap SS$.
5. EE é um conjunto finito de eventos de entrada.
6. ES é um conjunto finito de eventos de saída.
7. $(SE \cap (EE \cup ES) = \emptyset) \wedge (SS \cap (EE \cup ES) = \emptyset) \wedge (EE \cap ES = \emptyset)$.

Define-se também o estado das entradas e saídas do controlador, dado ser este estado que irá determinar a evolução da rede de Petri que modela o controlador.

Definição C.2 (Estado das Entradas e Saídas do Controlador): *Dada a interface de um controlador $IC = (SE, SEE, SS, SSL, EE, ES)$, o estado das entradas e saída do sistema é definido por um tuplo $EESC = (VEE, VES, VSE, VSS)$ tal que:*

1. VEE é um conjunto finito de vínculos de eventos de entrada: $VEE \subseteq (EE \times \mathbb{B})$.
2. VES é um conjunto finito de vínculos de eventos de saída: $VES \subseteq (ES \times \mathbb{B})$.
3. VSE é um conjunto finito de vínculos de sinais de entrada: $VSE \subseteq (SE \times \mathbb{N}_0)$.
4. VSS é um conjunto finito de vínculos de sinais de saída: $VSS \subseteq (SS \times \mathbb{N}_0)$.

Os valores dos eventos de entrada são calculados com base em alterações de valores nos sinais de entrada associados: resultam da comparação dos valores actuais dos sinais de entrada com os valores, dos mesmos sinais, no passo de execução anterior. No entanto, o controlador, ou seja a rede de Petri *input-output*, vê os eventos de entrada como outro tipo de sinais. Para que tal seja possível, o cálculo para determinação dos eventos é efectuado fora do controlador por um módulo também responsável pela aquisição dos sinais físicos. Os valores dos sinais de entrada surgem na Def. C.2 como números inteiros positivos, no entanto, nada obsta a que possam também ser incluídos os números inteiros negativos (\mathbb{Z} em lugar de \mathbb{N}_0) ou até os números reais (\mathbb{R} em lugar de \mathbb{N}_0). Seguidamente, definem-se as redes de Petri *input-output*.

C.1.2 Redes de Petri Input/Output

Esta secção define as redes de Petri *input-output*. Conforme já referido, esta classe pode ser vista como uma rede de Petri lugar-transição não-autónoma estendida com eventos e sinais externos de entrada e saída.

Na Def. C.3, que se apresenta seguidamente, assume-se a existência de uma linguagem de inscrições que permita a especificação de expressões algébricas, variáveis e funções. Utilizam-se também as seguintes notações: o conjunto de expressões booleanas é designado por EB ; a

função $Var(e)$ devolve o conjunto de variáveis presentes numa expressão e ; e MI é o conjunto de identificadores das marcações de cada lugar na rede, no final de um dado passo de execução; e $MI(l)$ é o identificador da marcação do lugar l .

Dado que o objectivo último de uma rede de Petri *input-output* consiste em dar suporte à sua implementação através de sistemas de controlo autónomos, a linguagem de inscrições deve, preferencialmente, ser a utilizada no código gerado. Tal permite simplificar a geração de código, nomeadamente por permitir evitar, ou reduzir significativamente, o *parsing*, ao mesmo tempo que evita a utilização de mais uma linguagem no processo de desenvolvimento. Para controladores embutidos, e quando a implementação for realizada em software, tal significa, muito provavelmente, a utilização da linguagem ANSI C dada a quase ubiquidade da mesma nesse domínio.

Definição C.3 (Rede de Petri *input-output*): Dada uma interface do controlador $IC = (SE, SEE, SS, SSL, EE, ES)$, uma rede de Petri *input-output* é um tuplo $R = (L, T, A, AT, M, peso, pesoT, GE, ge, ee, es, RA, ra)$ que satisfaz os seguintes requisitos:

1. L é um conjunto finito de lugares.
2. T é um conjunto finito de transições (disjunto de L).
3. A é um conjunto de arcos tal que $A \subseteq ((L \times T) \cup (T \times L))$.
4. AT é um conjunto de arcos de teste tal que $AT \subseteq (T \times L)$.
5. M é uma função marcação que aplica lugares em números inteiros não negativos: $M : L \rightarrow \mathbb{N}_0$.
6. $peso$ é uma função que aplica A em \mathbb{N} : $peso : A \rightarrow \mathbb{N}$.
7. $pesoT$ é uma função que aplica AT em \mathbb{N} : $pesoT : AT \rightarrow \mathbb{N}$.
8. GE é um conjunto de expressões booleanas, denominadas **guardas de entrada**, cujas variáveis são sinais de entrada e marcações de lugares: $GE = \{eb \in EB \mid Var(eb) \subseteq (SE \cup MI)\}$.
9. ge é uma função parcial que aplica transições em guardas de entrada: $T \rightarrow GE$.
10. ee é uma função parcial, denominada **eventos de entrada**, que associa eventos externos de entrada a transições: $ee : T \rightarrow \mathcal{P}(EE)$.
11. es é uma função parcial, denominada **eventos de saída**, que associa eventos externos de saída a transições: $es : T \rightarrow \mathcal{P}(ES)$.
12. RA é um conjunto de regras de actuação $RA \subseteq REG$, em que REG é o conjunto das regras constituídas por ternos que contêm uma expressão booleana, um sinal e um valor que irá afectar esse sinal: $\forall (e, s, v) \in REG, Var(e) \subseteq MI \wedge s \in SS \wedge v \in \mathbb{N}_0$.
13. ra é uma função que aplica lugares em conjuntos de regras de actuação em que o único identificador de marcação é relativo ao próprio lugar: $ra : L \rightarrow \mathcal{P}(RA)$, tal que $\forall l \in L, l \mapsto (e, s, v), Var(e) = MI(l)$.

Os elementos L , T , A , M e $peso$ têm o significado usual no contexto das redes de Petri lugar/transição. Os arcos de teste AT têm a semântica usual para arcos de teste: para um dado

arco de teste at , com peso $pesoT(at) = n$, devem existir n ou mais marcas no lugar respectivo para que a transição esteja apta. No entanto, este arco não consome nem cria marcas.

Os sinais de entrada externos, correspondentes a entradas físicas, são associados a transições de duas formas distintas:

1. De uma forma directa, sob a forma de variáveis constituintes de uma expressão booleana que constitui uma guarda da transição (*vide* GE e ge nos pontos 8 e 9 da Def. C.3).
2. De uma forma indirecta, sob a forma de eventos. Cada um destes eventos é calculado com base na modificação, entre dois passos de execução, do sinal de entrada associado (ee no ponto 10 da Def. C.3) por código dependente da plataforma.

Os sinais de saída externos são modelados e modificados de duas formas:

1. Com base em eventos de saída. Estes correspondem a sinais de saída que são afectados aquando do disparo das respectivas transições durante o passo de execução (es no ponto 11 da Def. C.3), à semelhança das denominadas "máquinas de Mealy" [Mealy, 1955].
2. Com base no estado da rede no início de cada passo de execução, mais especificamente com base nas marcações de cada lugar (ra no ponto 13 na Def. C.3) à semelhança das denominadas "máquinas de Moore" [Moore, 1956].

Um controlador é então constituído pela (1) respectiva interface, (2) o estado das respectivas entradas e saídas e por (3) uma rede de Petri *input-output*:

Definição C.4 (Controlador): Um controlador é um tuplo $C = (IC, EESC, R)$ constituído por uma interface, um estado das entradas e saídas do controlador e por uma rede de Petri *input-output* tal que:

1. IC é a interface do controlador: $IC = (SE, SEE, SS, SSL, EE, ES)$.
2. $EESC$ é um estado das entradas e saídas do controlador: $EESC = (VEE, VES, VSE, VSS)$.
3. R é uma rede de Petri *input-output*: $R = (L, T, A, AT, M, peso, pesoT, GE, ge, ee, es, RA, ra)$.

O disparo das transições segue o modelo de David e Alla [David e Alla, 1992]: uma transição t está apta a disparar sempre que se encontra habilitada, do ponto de vista das marcações dos lugares de entrada, e está **pronta**³. A transição diz-se pronta quando a condição externa associada é verdadeira. Para que esta condição externa seja verdadeira é necessário que no estado actual das entradas e saídas do controlador, e da marcação da rede, os eventos de entrada e a guarda sejam verdadeiros (condições 3 e 4 da Def. C.5). Assim sendo, do ponto de vista da rede de Petri, é sempre escolhido o passo máximo de execução, ou seja um passo ao qual não se

³Em inglês: *ready*.

podem adicionar mais transições sem que surjam conflitos efectivos entre essas inscrições e as presentes no passo. A evolução do modelo só é possível em instantes específicos (denominados *tics*) impostos por um relógio global externo, e um passo de execução é definido como o intervalo entre dois *tics* consecutivos.

Nas definições seguintes utilizar-se-á $M(l)$ para denotar a marcação de um lugar l numa rede com a marcação M . O conjunto dos lugares de entrada de uma transição t ou de um conjunto de transições S serão denotados por $\bullet t = \{l \mid (l, t) \in A\}$ e $\bullet S = \bigcup_{t \in S} \bullet t$. De forma idêntica, considerando os arcos de teste, utilizar-se-á $ot = \{l \mid (t, l) \in AT\}$ e $oS = \bigcup_{t \in S} ot$.

Definição C.5 (Aptidão de uma Transição numa Rede de Petri *input-output*): *Dado um controlador $C = (IC, EESC, R)$ com $IC = (SE, SEE, SS, SSL, EE, ES)$, num estado $EESC = (VEE, VES, VSE, VSS)$ e $R = (L, T, A, AT, M, peso, pesoT, GE, ge, ee, es, RA, ra)$, uma transição $t \in T$ diz-se apta em $EESC$ se, e só se, satisfizer as seguintes condições:*

1. $\forall l \in \bullet t, M(l) \geq peso(l, t)$.
2. $\forall l \in ot, M(l) \geq pesoT(t, l)$.
3. $ge(t) < VSE, M \geq \text{verdade}$.
4. $\forall e \in ee(t), (e, \text{verdade}) \in VEE$.

Definição C.6 (Passo de uma Rede de Petri *input-output*): *Seja $C = (IC, EESC, R)$ um controlador com $IC = (SE, SEE, SS, SSL, EE, ES)$, num estado $EESC = (VEE, VES, VSE, VSS)$ e $R = (L, T, A, AT, M, peso, pesoT, GE, ge, ee, es, RA, ra)$ uma rede de Petri *input-output*. Seja também $CTA \subseteq T$ o subconjunto de todas as transições aptas em R (vide Def. C.5). Então, Y é um passo em R se e só se a seguinte condição for satisfeita:*

$$Y \subseteq CTA \wedge \forall t_{out} \in (CTA \setminus Y), \exists SY \subseteq Y, \exists l \in (\bullet t_{out} \cap \bullet SY), \left(peso(l, t_{out}) + \sum_{t \in SY} peso(l, t) \right) > M(l)$$

Conforme já referido, o passo de uma rede de Petri *input-output* é máximo: não é possível adicionar mais transições (aptas) a um passo sem que surjam um ou mais conflitos efectivos entre qualquer transição que se pretenda adicionar e uma ou mais das que estão incluídas no passo.

Definição C.7 (Ocorrência de um Passo e Marcação Sucessora numa RPIO): *Seja $R = (L, T, A, AT, M, peso, pesoT, GE, ge, ee, es, RA, ra)$ uma rede de Petri *input-output* com um sistema a controlar no estado $EESC = (VEE, VES, VSE, VSS)$ e com uma interface do controlador $IC = (SE, SEE, SS, SSL, EE, ES)$. A ocorrência de um passo Y na rede R devolve a rede $R' = (L, T, A, AT, M', peso, pesoT, GE, ge, ee, es, RA, ra)$ que difere de R apenas na sua marcação sucessora M' :*

$$M' = \left\{ \left(l, m - \sum_{t \in Y \wedge (l, t) \in A} peso(l, t) + \sum_{t \in Y \wedge (t, l) \in A} peso(t, l) \right) \in (L \times \mathbb{N}_0) \mid (l, m) \in M \right\}$$

A secção seguinte resume as principais características da gramática para definição de redes de Petri *input-output* utilizando PNML.

C.1.3 Especificação em PNML

Conforme já apresentado na Secção 4.2.1, para especificar novos tipos de redes de Petri em PNML é necessário especificar o **Petri Net Type Definition** (PNTD) correspondente. Este explicita os vários elementos que podem estar associados à rede, a cada lugar, a cada transição e a cada arco.

As redes de Petri *input-output* são uma extensão às redes de Petri lugar-transição e este facto reflecte-se na gramática proposta que contem todos os elementos do PNTD para redes de Petri lugar/transição mais os elementos específicos das redes de Petri *input-output*.

Comparativamente com as redes de Petri lugar-transição, a gramática adiciona suporte para as seguintes especificações:

1. Declaração de sinais e eventos de entrada, e também de sinais e eventos de saída, todos associados à rede (elemento XML `net`).
2. Eventos de entrada e de saída associados às transições (elemento XML `transition`).
3. Uma guarda baseada em eventos de entrada e associada às transições (elemento XML `transition`).
4. Regras do tipo "sinal = valor *if* condição" associadas aos lugares e em que a condição é dependente da marcação do lugar, permitindo especificar as activações dos sinais de saída.

A gramática consta da Listagem E.1 no Apêndice E (pág. 259). Estas gramáticas constituem peças fundamentais dos ambientes de desenvolvimento baseado em modelos em redes de Petri especificados em PNML. A Secção C.3 apresenta uma proposta para um destes ambientes. Na mesma secção, apresenta-se um exemplo de uma rede de Petri *input-output*. A secção seguinte discute a execução de modelos de redes de Petri.

C.2 Sobre a Execução de Modelos

Quando se pensa na execução de redes de Petri, a forma mais intuitiva é certamente a execução pelo denominado "jogador de marcas"⁴. Tal implica a utilização de um simulador, provavelmente gráfico, dos quais os mais conhecidos, também por suportarem redes de Petri coloridas, são a

⁴Em inglês: *token player*.

CPN Tools [s.a., 2004b] e o RENEW[Kummer et al., 2004a]. Contudo, tal nem sempre é possível devido a uma ou mais das seguintes três razões:

1. Pode ser necessário executar o modelo sobre uma plataforma, hardware ou software, para a qual não existem jogadores de marcas, dado estes serem frequentemente construídos para funcionarem sobre sistemas operativos populares para computadores pessoais.
2. Os jogadores de marcas podem não funcionar, ou funcionar de forma demasiado lenta, sobre a plataforma em causa.
3. Pode não ser desejável a presença de um jogador de marcas gráfico dado apenas se pretender a execução da rede para controlo ou obtenção de resultados de teste ou verificação.

A alternativa é a geração de código adequado ao que se pretende obter do modelo: teste de vários cenários desejáveis ou indesejáveis, vários tipos de verificação, obtenção de código que possa ser executado numa ou mais plataformas específicas, etc..

Uma forma de implementar o modelo consiste em traduzi-lo para uma máquina de estados [Gomes e Steiger-Garção, 1995; Barros, 1996; Barros et al., 1997]. No entanto, tipicamente, esta máquina de estados é demasiado grande para poder ser utilizada. Na verdade, tal corresponde a uma implementação de todo o espaço de estados o que, em muitos casos, não é realista. Ainda assim, para espaços de estado suficientemente pequenos, esta pode ser uma solução eficiente, em especial para plataformas hardware com relativamente pouca memória e capacidade de processamento, mas que podem facilmente executar uma máquina de estados. Estas máquinas de estado podem ser codificadas na linguagem de programação ANSI C a qual está normalmente disponível para sistemas operativos embutidos. Em resumo: por esta via podemos construir um modelo de alto-nível que é implementado como uma máquina de estados. Neste sentido, o modelo da rede de Petri deve ser visto como uma alternativa, de mais alto-nível, aos modelos baseados em máquinas de estado ou estadogramas.

Conforme já referido, a abordagem alternativa é a de gerar um interpretador para o modelo. Este pode ser visto como sendo constituído por dois pacotes⁵ distintos: (1) a especificação da estrutura da rede; (2) o executor da rede.

Para cada passo de execução, o pacote executor realiza três subpassos:

1. Calcula o conjunto de transições aptas.
2. Selecciona o subconjunto de transições aptas que irá disparar.
3. Dispara as transições no subconjunto referido em 2.

⁵Em inglês: *packages*.

Em redes de grande dimensão, o cálculo no primeiro subpasso pode demorar demasiado tempo. Após o primeiro passo de execução, este cálculo pode ser minimizado aplicado-o às transições que no passo anterior tenham sido consideradas inaptas, devido à marcação nos respectivos lugares de entrada, mas apenas nos casos em que a marcação de pelo menos um lugar de entrada foi alterada.

A clássica dicotomia memória versus eficiência surge de forma muito clara na implementação de modelos de redes de Petri, em especial nas redes de Petri coloridas: ou se abdica da compacidade da rede de Petri através da desdobragem da rede de Petri colorida numa máquina de estados; ou a implementação directa, e portanto mais compacta, da rede de Petri colorida tem de pagar o preço de uma mais lenta execução devido ao cálculo dos vínculos de cada transição.

Finalmente, importa mencionar que o problema geral de execução de uma rede de Petri é muito mais complexo pois deve incluir implementações distribuídas. Desta forma importa encontrar as melhores estratégias para decidir qual a arquitectura adequada e qual a melhor forma de executar o modelo nessa arquitectura. Tal parece constituir um bom tema de investigação futura pois de acordo com o conhecimento do autor não existem ainda soluções efectivamente práticas para execução de redes de Petri neste âmbito generalizado.

A secção seguinte apresenta uma proposta para um ambiente de desenvolvimento baseado em redes de Petri que se destina a obter como produto final, código executável em vários tipos de plataforma, em especial em sistemas embutidos com relativamente pouco recursos computacionais. Pretende-se que este mesmo ambiente permita futuras extensões para suporte à geração de modelos a serem implementados em arquitecturas distribuídas.

C.3 Um Ambiente para Desenvolvimento Utilizando Redes de Petri

Existe actualmente uma quantidade muito significativa de ferramentas computacionais que permitem a utilização de redes de Petri das mais variadas classes. Em particular, a *Petri net Tools Database* [s.a., 2005d] lista aproximadamente sessenta ferramentas. Assim sendo, começa-se por explicitar as razões pelas quais se propõe aqui um ambiente de desenvolvimento constituído por um conjunto de ferramentas para a utilização de redes de Petri. São seis as motivações para a realização deste ambiente:

1. A quase inexistência de geradores de código para modelos em redes de Petri. Em Maio de 2005, na *Petri net Tools Database* encontravam-se três ferramentas como possuindo nas suas características alguma forma de geração de código: CoopnBuilder, SIPN-Editor, e Syroco. Na informação relativa à CPN Tools também se menciona geração de código, mas este código é gerado internamente à própria ferramenta.

2. A quase inexistência de ferramentas computacionais para redes de Petri compatíveis com a linguagem PNML, a qual a breve prazo constituirá o formato normalizado para intercâmbio de classes de redes de Petri e de modelos que utilizem essas classes. Mesmo as poucas ferramentas que suportam o formato PNML, fazem-no para as suas classes específicas de redes de Petri.
3. A intenção de criar um sistema facilmente adaptável a diferentes classes de redes de Petri. Actualmente, a única ferramenta com tal objectivo é a Petri net Kernel (PNK) [Abdourahaman et al., 2002], cujo desenvolvimento se encontra parado.
4. A disponibilização de um editor gráfico capaz de suportar as várias possibilidades de estruturação de modelos em redes de Petri permitidas pelas linguagens PN_{TEXT} e OPNML, definidas na primeira parte desta dissertação.
5. A intenção de criar um sistema em que o código gerado é optimizado em função da classe de redes de Petri em utilização e também em função das características concretas de cada modelo.
6. A intenção de criar um sistema em que o código gerado é utilizado quer para implementação final sobre uma plataforma hardware ou software específica, quer para a sua prévia simulação e análise.

A Fig. C.1 esquematiza a arquitectura do ambiente proposto, que pode ser decomposta em cinco áreas:

1. A **interface com o utilizador** que se centra num editor gráfico e textual de redes e que permite a interacção entre o modelador e o modelo. Denomina-se *PnEditor* e está representado no canto superior esquerdo da Fig. C.1.
2. O **gerador de código** (*PnGenerator*) que pode ser visto como o núcleo do ambiente de desenvolvimento. Está representado no centro da Fig. C.1 e é responsável pela geração de código executável a partir da especificação da redes de Petri fornecida pelo editor, ou por outra aplicação capaz de gerar ficheiros PNML para uma classe de redes de Petri por si suportada.
3. O **simulador** que irá utilizar código gerado pelo PnGenerator.
4. O **analizador** que também utilizará código gerado pelo PnGenerator.
5. A **plataforma específica** sobre a qual será executado o código gerado.

O PnEditor deve suportar qualquer classe de redes de Petri desde que a respectiva especificação esteja descrita em PNML e exista uma gramática (um PNTD) que defina a sintaxe de cada uma dessas classes. Conforme já referido (*vide* Capítulo 4), a PNML [s.a., 2004d; Jünger et al.,

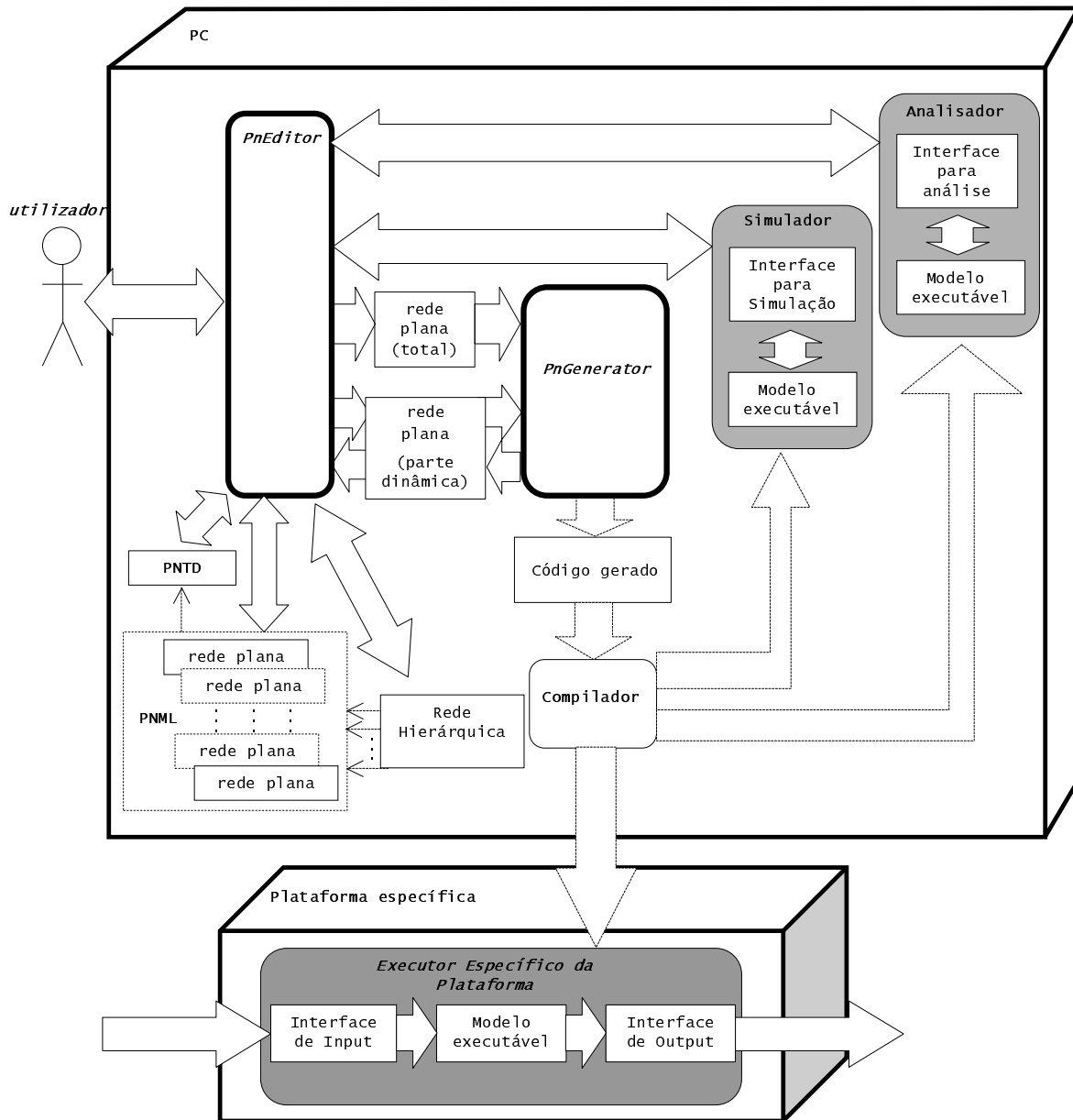


Figura C.1: A arquitectura do ambiente de desenvolvimento.

2000; Billington et al., 2003] é um formato para intercâmbio de modelos em redes de Petri que irá constituir a segunda parte da norma internacional para as redes de Petri de alto-nível: a ISO/IEC 15909 [s.a., 2005c, 2004e]. O editor suportará ainda a definição de novos PNTDs ou a alteração dos existentes. Para tal será possível adicionar ou remover elementos a cada um dos elementos "primitivos" da linguagem PNML: **net**, **place**, **transition** e **arc**. Actualmente, apenas o editor associado à ferramenta Petri Net Kernel [Abdourahaman et al., 2002] suporta esta funcionalidade.

Por outro lado, o editor deverá também suportar as operações de adição e subtracção de redes descritas na primeira parte desta dissertação. Conforme também aí apresentado, a operação de

adição permitirá a especificação de estruturas hierárquicas e não-hierárquicas. Apesar deste suporte para vários tipos de estruturação, o modelo que é gerado pelo editor e lido pelo gerador de código (PnGenerator) será sempre um modelo "plano" equivalente, ou seja, correspondente a uma especificação equivalente mas numa única página. Tal é possível porque a adição e subtração fornecem mecanismos de estruturação gráfica que não implicam quaisquer modificações da semântica da classe de redes em utilização. O editor suportará ainda a simulação interactiva, com base no simulador, e a análise do código com base no analisador.

No caso do gerador de código, foi já desenvolvido um protótipo gerador de código na linguagem ANSI C, executável num sistema embutido baseado num microprocessador PIC 16F877 do fabricante Microchip com as seguintes características [Pais, 2004]:

- Arquitectura RISC em que a execução de todas as instruções é realizada num único ciclo de relógio.
- Velocidade de operação do relógio DC - 20MHz.
- Memória Flash para programação de 8 KBytes.
- 368 Bytes de memória RAM para dados.
- 256 Bytes de memória EEPROM para dados.
- Circuito para programação série embutido.
- Entrada/Saída através de 33 portos de Entrada/Saída, com controlo individual de direcção.
- Circuito integrado de 40 pinos.

O ambiente é totalmente extensível a outras classes de redes de Petri e o código gerado para execução do modelo pode ser utilizado de três formas distintas cada uma das quais correspondente a um diferente objectivo:

1. Para execução na plataforma específica.
2. Para suporte à verificação de propriedades, nomeadamente através da construção e verificação do espaço de espaços.
3. Para suporte à simulação interactiva do modelo através de um "jogador de marcas".

Outra característica fundamental é a da geração de código optimizado para a plataforma de execução e para a implementação do modelo. A optimização da implementação do modelo corresponde a uma adaptação quer às características da classe de redes de Petri em utilização, quer às características exibidas por cada modelo específico. Por exemplo, ainda que a classe de

redes permita a especificação de guardas, se o modelo em causa não as utilizar então não deve ser gerado código que as considere, quer a nível estático (na descrição da estrutura da rede), quer a nível dinâmico (no código que testa a aptidão das transições).

A secção seguinte apresenta um exemplo de um modelo utilizando uma rede de Petri *input-output*.

C.4 Um Controlador para um Parque de Estacionamento

Nesta secção apresenta-se uma versão do modelo **ParkA**, definido na pág. 53, mas agora utilizando uma rede de Petri *input-output*. Este novo modelo, denominado **ParkAIO**, resulta da adição das redes **EnterIO**, **ParkingAreaIO** e **LeaveIO** apresentadas na Fig. C.2. Estas três redes têm a mesma estrutura que as suas congéneres não marcadas (*vide* Fig. 3.1 na pág. 53). Para além das inscrições da rede autónoma já presentes nos modelos não marcados, são especificados os eventos de entrada e saída através de triângulos com um vértice na transição respectiva. Representam-se também guardas correspondentes a condições que utilizam sinais de entrada como variáveis, e acções associadas a lugares e dependentes da marcação destes. Todos os elementos não-autónomos do modelo são representados com um fundo cinzento claro.

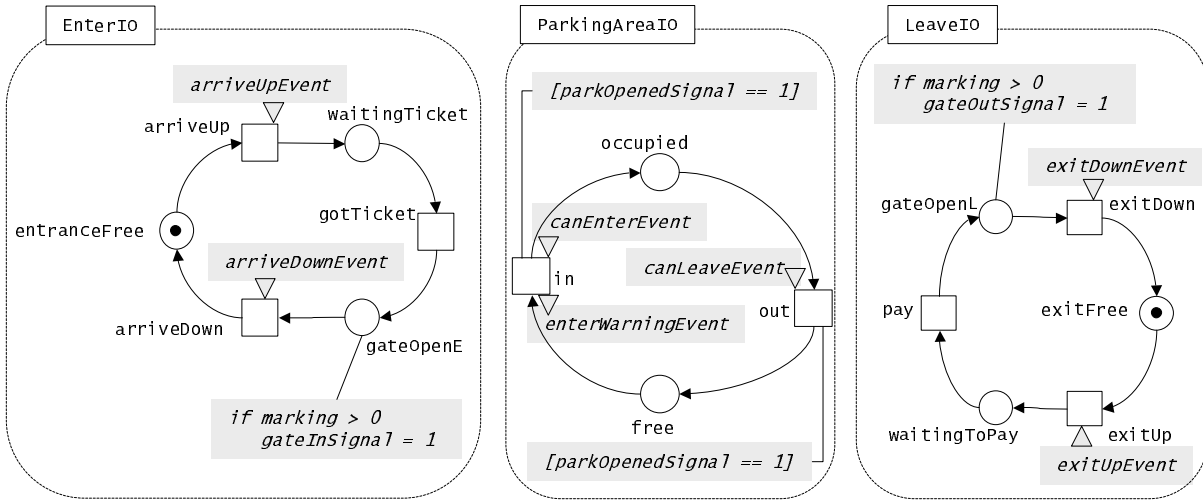


Figura C.2: Três redes de Petri *input-output*: EnterIO, ParkingAreaIO e LeaveIO.

A rede **ParkAIO** é definida pela adição de uma instância de cada uma das três redes na Fig. C.2 segundo a estrutura de transformação ET_{ptio} a qual se apresenta na Listagem D.5 na pág. 232.

$$\text{ParkAIO} := (\text{EnterIO} + \text{ParkingAreaIO} + \text{LeaveIO})(\text{/gotTicket/in/} \rightarrow \text{in}, \text{/out/pay/} \rightarrow \text{out})$$

A Fig. C.3 apresenta uma representação gráfica para o modelo ParkAI0.

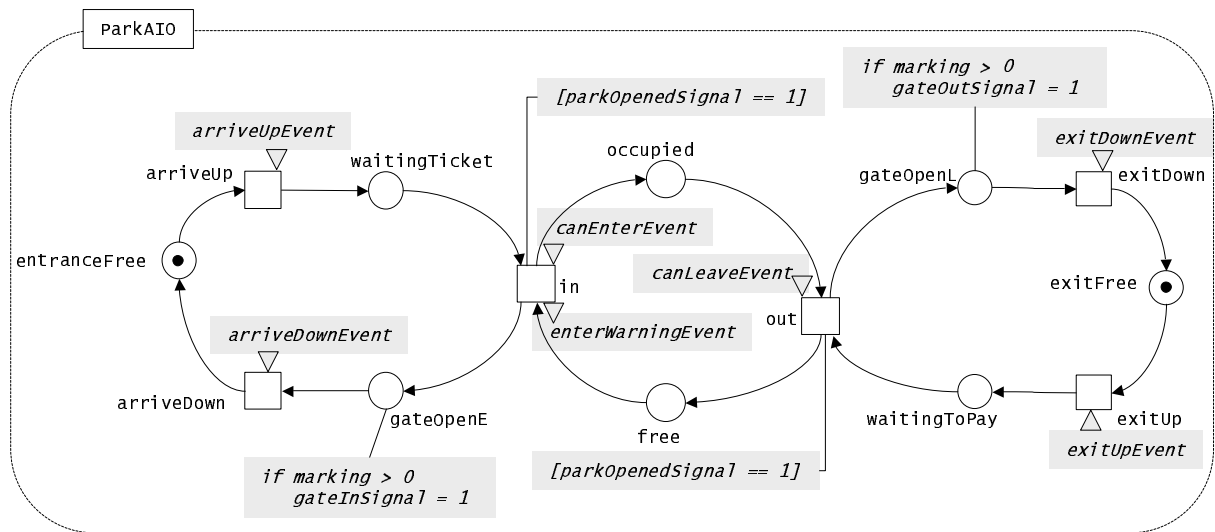


Figura C.3: Modelo ParkAI0.

Na Secção D.2.3 apresenta-se a especificação em PNML das três redes: **EnterIO** na Listagem D.14 da pág. 247, **ParkingAreaIO** na Listagem D.15 da pág. 249 e **LeaveIO** na Listagem D.16 da pág. 250. Na Listagem D.17 da mesma secção (pág. 252) apresenta-se a especificação em Operational PNML para o modelo ParkAI0. O código PNML resultante consta da Listagem D.18. O protótipo desenvolvido por Pais [Pais, 2004] gera código executável a partir deste tipo de especificação.

Finalmente, na Fig. D.3, apresenta-se um grafo gerado automaticamente a partir da especificação PNML do modelo ParkAI0.

Apêndice D

Protótipo para Tradução de Especificações OPNML

Este apêndice apresenta o código fonte da aplicação descrita na Secção 4.3. A Secção D.1 apresenta as listagens do componente principal e de quatro componentes correspondentes a estruturas de transformação. Note-se que estes podem conter dependências entre si, expressas em Ruby por meio da instrução `require`.

A Secção D.2 apresenta exemplos de listagens de vários ficheiros OPNML. Estes estão também disponíveis na Internet [Barros e Gomes, 2004e]. Alguns foram já apresentados na Secção 4.2.2.

D.1 Código Fonte do Protótipo `opnml2pnml`

Listagem D.1: Código fonte do componente principal (`opnml2pnml.rb`) do protótipo `opnml2pnml`.

```
# OperationalPNML to PNML
3 # Author: Joao Paulo Barros
  # Beja, Portugal
  # opnml2pnml.rb v. 0.5
  # Beja, 2005/09/25

8 # Compared to previous versions adds the dynamic loading of net structures
  # based on the PNID in the opnml file.
  # See method loadTransformationStructure in class OPNML.

  # When generating a specific label for a result node, allows
13 # the use of all other labels in the fusion element for that node.

  require "rexml/document"
  include REXML
```

```

18 # Prints an error message and exits the program
    def Error(errorString)
        puts "\nError! ".concat(errorString)
        exit
    end
23
    # "Main" class. Reads OPNML file

    class OPNML
        SupportedOperations = ["addition", "subtraction"]
28        attr_reader :xmlns, :type, :netID

        def initialize(fileNameInput)
            @fileNameInput = fileNameInput
            f = File.new(fileNameInput)
33            @doc = Document.new(f)
            f.close
            Error("Document has no root element") if (!@doc.root)
            @xmlns = @doc.root.attributes["xmlns"]
            @type = ""
38            @@writtenFiles = "\n"
        end

        # Tells user which PNML files (one net in each) were generated
        def OPNML.addToWrittenFiles(filename)
43            @@writtenFiles.concat("\nCreated file #{filename}")
        end

        # Reads OPNML file and delegates operation processing to the corresponding class
        def readFile
48            net = @doc.root.elements["net"]
            Error("No net element found.") if (!net)
            @type = net.attributes["type"]
            Error("Attribute 'type' not found in element pnml/net.") if (!@type)

53            loadTransformationStructure(@type)

            @netID = net.attributes["id"]
            Error("Attribute 'id' not found in element pnml/net.") if (!@netID)
            operations = net.elements["operations"]
58            Error("'pnml/net' element has no operations element.") if (!operations)
            operations.elements.each { |operation|
                if (SupportedOperations.include?(operation.name)) then
                    operationName = operation.name.capitalize()
                    op = eval(operationName).new(operation, self)
63                    op.compute()
                else
                    Error("Error! Operation #{operation.name} not supported." +
                        "Supported operations are: #{SupportedOperations}")
                end
68            }
            puts @@writtenFiles
        end

        #loads the transformation structure for the net type in the opnml file
73        # if the net type has '/' is loads the file with the string after it as name
        # if there are not '/' it loads the file with the type as name
        # For example for type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb"

```



```

# it tries to loads the file ptNetb.rb
# For type="colorSimple" it tries to load the file colorSimple.rb
78   def loadTransformationStructure(type)
      pos = type.rindex('/')
      toLoad = (pos == nil) ? type : type[pos + 1, type.length - pos - 1]
      puts 'Loading transformation structure "' + toLoad + '"...'
      require toLoad + '.ts'
83   end
end # end class OPNML

# Operations list
class Operations
88   def initialize()
      @operations = Array.new # operations list
      end
      def addOperation(operation)
        @operations.push(operation)
93   end
end # end class Operations

# Used by NodeData
class ArcData
98   attr_reader :xml, :sign, :id
      def initialize(xml, sign)
        @xml = xml
        @sign = sign
        @id = xml.attributes['id']
103  end
end

# Node data for each node to be fused (see FusionElement)
class NodeData
108  attr_reader :xml, :arcs, :sign
      def initialize(xml, arcs, sign)
        @xml = xml
        # two arrays containing ArcData objects
        @arcs = arcs #{"place" => Element Array,      arcs with place source
113          # "transition" => Element Array} arcs with transition source
        # sign from the respective net
        @sign = sign
      end
end # end class NodeData
118

# Groups nodes to be fused.
class FusionElement
  # fused node name
  attr_reader :final
123  # list of nodes to be fused (each one is a NodeData)
  attr_reader :toFuse
  # is this a "place" or a "transition"?
  attr_reader :placeTransition
  # labels, with the given name, in all nodes
128  attr_reader :labels
  def initialize(final)
    @final = final
    @toFuse = Hash.new
    @placeTransition = ""
133    @labels = Hash.new

```

```

end
def isInterface?()
  return @toFuse.length() > 1
end
138 def addNodeToFuse(completeNodeID, nodeXML, arcs, sign)
  @toFuse[completeNodeID] = NodeData.new(nodeXML, arcs, sign)

  nodeXML.each_element{ |elem|
    @labels[elem.name()] = Array.new if !@labels[elem.name()]
    143   #@labels[elem.name()].push(elem.elements['text'].text)
    @labels[elem.name()].push(elem)
  }
end
def setPlaceTransition(placeTransition)
148   @placeTransition = placeTransition
end
end # end class FusionElement

# Stores nets for performing the operations. Each Operation#operands is a net
153 class Net
  attr_reader :netID, :docXML, :sign
  def initialize(netID, sign, vectorIndex = nil)
    @netID = netID
    @sign = sign
    158   @docXML = nil
    file = netFileOK(@netID)
    Error("Error opening file for #{netID}") if !file
    @docXML = Document.new(file)
    file.close()

    163   net = @docXML.elements["pnml"].elements["net"]

    if vectorIndex then
      netIDF = net.attributes['id']
      168   Error("netID attribute in file #{netID} is #{netIDF}.") if netID != netIDF
      net.attributes['id'] = netID.concat("#{vectorIndex}")
    end
    if $prefix or vectorIndex then
      net.each_element do |element|
        173   #replace id='x' by id='netID.x'
        if element.name() =~ /place|transition|arc/
          element.attributes['id'] = "#{netID}.".concat(element.attributes['id'])
        end
        #replace target='x' by target='netID.x'
        178   #replace source='x' by source='netID.x'
        if (element.name() == "arc")
          element.attributes['source'] = "#{netID}.".concat(element.attributes['source'])
          element.attributes['target'] = "#{netID}.".concat(element.attributes['target'])
        end
      end
    183   end
    else # add prefix only to arc ids
      net.each_element("arc") do |arc|
        #replace id='x' by id='netID.x'
        arc.attributes['id'] = "#{netID}.".concat(arc.attributes['id'])
      end
    188   end
  end
end
end

```

```

def netFileOK(netID)
193   fileName = netID + ".pnml.xml"
      Error(" File #{fileName} does not exist.") if !File.exists?(fileName)
      Error(" File #{fileName} is not readable.") if !File.readable?(fileName)
      Error(" File #{fileName} has zero size.") if File.zero?(fileName)
      return File.new(fileName)
198   end

# Called from Operation#collapse
def Net.removeNodes(net, nodesToFuse, final)
  nodesToFuse.each_key do |completeNodeID|
203    net.each_element do |element|
      if (element.attributes['id'] == completeNodeID)
        net.delete_element(element)
      end
      if (element.attributes['source'] == completeNodeID)
208        element.attributes['source'] = final
      end
      if (element.attributes['target'] == completeNodeID)
        element.attributes['target'] = final
      end
    end
213  end
  end
end

# Called from Subtraction#subtract
218 def Net.removeNodesAndArcs(net, nodes)
  nodes.each_key { |completeNodeID|
    net.each_element { |element|
      if (element.attributes['id'] == completeNodeID ||
          element.attributes['source'] == completeNodeID ||
223         element.attributes['target'] == completeNodeID)
        net.delete_element(element)
      end
    }
  }
228   end
end # end class Net

# This class is used to handle the arcs between a pair of interface nodes.
class IPairsList
233   # list of IPair objects
  attr_reader :list
  def initialize()
    @list = Array.new
  end
238   def addIPair(sourcePlaceTransition, sourceFE, targetFE, arcData)
    sourceID = sourceFE.final
    targetID = targetFE.final
    # if the IPair exists just add a new arc and return
    @list.each do |ip|
243      if sourcePlaceTransition == "place" &&
        ip.placeFE.final == sourceID && ip.transitionFE.final == targetID
        ip.ptArcs.push(arcData)
        return
      end
248      if sourcePlaceTransition == "transition" &&
        ip.transitionFE.final == sourceID && ip.placeFE.final == targetID

```

```

        ip.tpArcs.push(arcData)
        return
    end
253 end
    ip = nil
    if sourcePlaceTransition == "place" then
        ip = IPair.new(sourceFE, targetFE)
        ip.addPTArc(arcData)
258 elsif sourcePlaceTransition == "transition"
        ip = IPair.new(targetFE, sourceFE)
        ip.addTPArc(arcData)
    else
        Error(" Unexpected error in IPairList#addIPair")
263 end
    @list.push(ip) # add the new IPair to the list
end
end # end class IPairList

268 class Operation
    def initialize(xml, pnml, op)
        @xml = xml
        @operands = Hash.new
        # merge and removal are going to be Arrays of FusionElements
273 @fusionSets = { "merge" => Array.new, "removal" => Array.new }
        @iPairsList = IPairsList.new
        @pnml = pnml
        @op = op
        @result = @xml.attributes["result"]
278 @result = pnml.netID if @result == "return"
    end
    def createNets()
        @xml.elements.each("net"){ |net|
            netID = net.attributes["netID"]
283 netSign = net.attributes["sign"] == "minus" ? -1: 1 # can be nil
            createNet(netID, netSign)
        }
        @xml.elements.each("netVector"){ |netVector|
            netID = netVector.attributes["netID"]
288 netSign = netVector.attributes["sign"] == "minus" ? -1: 1 # can be nil
            createNetVector(netVector, netID, netSign)
        }
    end
    def createNet(netID, netSign, i=nil)
293 net = Net.new(netID, netSign, i)
        #add new net to operands
        @operands[net.netID] = net
    end

298 def createNetVector(netVector, netID, netSign)
        first = netVector.attributes["first"].to_i
        last = netVector.attributes["last"].to_i
        first.upto(last) do |i| createNet(netID.dup, netSign, i) end
    end
303 def createFusionSets(mergeOrRemoval)
        ["placeFusion", "transitionFusion"].each{|pt|
            @xml.elements.each("#{"mergeOrRemoval}/#{pt}"){|fusionSet|
                nodeID = fusionSet.attributes["nodeID"]

```

```

308         createFusionSet(pt, nodeID, fusionSet, mergeOrRemoval)
        }
    }
    ["placeFusionVector", "transitionFusionVector"].each{|pt|
        @xml.elements.each("merge/#{pt}"){|fusionSetVector|
313             createFusionSetVector(pt, fusionSetVector, mergeOrRemoval)
        }
    }
end

318 def createFusionSet(ptType, finalNodeID, fusionSetXML, mergeOrRemoval,
        iterator=nil, iteratorValue=nil)
    fe = FusionElement.new(finalNodeID)
    @fusionSets[mergeOrRemoval].push(fe)
    fusionSetXML.elements.each do |pt|
323         if (pt.name() != "place" && pt.name() != "transition")
            Error("Illegal element '#{pt.name()}' inside '#{ptType}'")
        end
        if pt.name() == "place" && ptType[0,9] == "transition"
            Error("'place' in transition fusion")
328         end
        if pt.name() == "transition" && ptType[0,4] == "place"
            Error("'transition' in place fusion")
        end
        fe.setPlaceTransition(pt.name())
333         netID = pt.attributes["netID"].dup
        nodeToFuseID = pt.attributes["nodeID"].dup
        if (iterator) then
            netID.gsub!(/(.*)\[#{iterator}\](.*)/) do |s|
                "#{s1}[#{iteratorValue}]#{s2}"
338             end
            nodeToFuseID.gsub!(/(.*)\[#{iterator}\](.*)/) do |s|
                "#{s1}[#{iteratorValue}]#{s2}"
            end
        end
        end
        if $prefix or (netID =~ /.*\[.*\].*/) #has index
            completeNodeID = "#{netID}.#{nodeToFuseID}"
        else
            completeNodeID = "#{nodeToFuseID}"
        end
        netSign = @operands[netID].sign
        nodeXML, arcs = getNodeXMLData(netID, nodeToFuseID, completeNodeID, netSign)
        fe.addNodeToFuse(completeNodeID, nodeXML, arcs, netSign)
    end #do fusionSetXML.elements.each
end

353 def getNodeXMLData(netID, nodeToFuseID, completeNodeID, netSign)
    net = @operands[netID].docXML.elements['pnml'].elements['net']
    c = 0
    pt = "" # 'place' or 'transition'
358     nodeXML = Element.new
    # define two arrays of ArcData objects
    arcs = {'place' => Array.new, 'transition' => Array.new}
    #net.each_element_with_attribute('id', '#{completeNodeID}', 2){ |e|
    net.each_element() do |e|
363         if e.attributes['id'] == completeNodeID then
            nodeXML = Document.new(e.to_s).root
            c = c + 1
        end
    end
end

```

```

    pt = e.name()
    if pt != 'place' && pt != 'transition' then
368       Error("#{completeNodeID} is a #{pt}")
    end
  end
  Error("More than one element with 'id' '#{completeNodeID}'") if c > 1
end
373 Error("No elements with 'id' '#{completeNodeID}'") if c == 0

net.each_element() do |e|
  if (e.name() == "arc") &&
    (e.attributes['source'] == completeNodeID) then
378     arcs[pt].push(ArcData.new(Document.new(e.to_s).root, netSign))
  end
end
return nodeXML, arcs
end

383
def createFusionSetVector(ptType, fusionSetVector, mergeOrRemoval)
  nodeID = fusionSetVector.attributes["nodeID"]
  iterator = fusionSetVector.attributes["iterator"]
  if !(iterator.to_s =~ /[a-zA-Z][a-zA-Z0-9]*/)
388     Error("Attribute 'iterator', in fusionSetVector" +
      "with nodeID '#{nodeID}', must be a single variable." +
      "It is '#{iterator.to_s}'")
  end
  first = fusionSetVector.attributes["first"].to_i
393 if !(first.to_s =~ /[0-9]+)/
    Error("Attribute 'first', in fusion vector with " +
      "nodeID '#{nodeID}' #{nodeID}, must be a positive " +
      "integer. It is '#{first.to_s}'")
  end
398 last = fusionSetVector.attributes["last"].to_i
  if !(last.to_s =~ /[0-9]+)/
    Error("attribute 'last', in fusion vector with " +
      "nodeID #{nodeID}, must be a positive integer." +
      "It is '#{last.to_s}'")
  end
403 end
  first.upto(last) do |i|
    n = nodeID.gsub(/(.*)\[#{iterator}\](.*)/) { |s|
      "#{s}[#{i}]"
    }
408     createFusionSet(ptType, n, fusionSetVector, mergeOrRemoval,
      iterator, i)
  end
end # createFusionSetVector

413 # Creates a new net with all the nodes and arcs in the operands
def unifyNets()
  newNetDoc = Document.new
  p = newNetDoc.add_element("pnml", { 'xmlns' => @pnml.xmlns })
  net = p.add_element("net", { 'id' => @result, 'type' => @pnml.type })
418 netLabels = Array.new
  netLevelLabels = Hash.new
  @operands.each_pair do |netID, netX|
    #the following copy is necessary to avoid destructing netXML
    netXMLCopy = Document.new(netX.docXML.to_s)
423    n = netXMLCopy.elements["pnml"].elements["net"]
  end
end

```

```

        n.each_element do |e|
            if (e.name() != 'place' &&
                e.name() != 'transition' &&
                e.name() != 'arc')
428                labelList = netLevelLabels[e.name]
                netLevelLabels[e.name] = Array.new if (labelList == nil)
                netLevelLabels[e.name].push(Label.new(e, netX.sign))
            else
                netLabels.push(e)
433            end
        end
    end

    # we like net labels to appear first
438    # here they are fused and the result is added to the result net
    netLevelLabels.each_pair do |label, netLevelElementList|
        net.add_element(
            Document.new(
                NetFusion.new(label, netLevelElementList).addsub(@op).root)
443        end

    # sort all net elements so that each set of "inherited" labels is ordered
    netLabels.sort!() { |x,y| x.to_s <=> y.to_s }
    # then places
448    netLabels.each { |e| net.add_element(e) if (e.name() == 'place') }
    # then transitions
    netLabels.each { |e| net.add_element(e) if (e.name() == 'transition') }
    # and finally the arcs
    netLabels.each { |e| net.add_element(e) if (e.name() == 'arc') }
453    # to be created nodes and arcs will appear after
    # all these "inherited" elements.
    return newNetDoc
end

458    def collectLabels(nodesToFuse)
    #collects all the label elements inside a node
        labels = Array.new
        nodesToFuse.each_value do |nodeData|
            nodeData.xml.each_element { |elem| labels.push(elem.name()) }
463        end
        if (not labels.empty?()) then
            labels.uniq!()
            labels.sort!() { |x,y| x.to_s <=> y.to_s }
        end
468        return labels
    end

    # If the node was fused in the merge part then
    # it is an interface node
473    def isInterface?(nodeID)
        @fusionSets["merge"].each do |fe|
            if fe.isInterface?()
                fe.toFuse.each_key do |completeNodeID|
                    return fe if completeNodeID == nodeID
478                end
            end
        end
        return nil
    end

```

```

end
483
# Creates all the interface pairs.
# These can then be manipulated through the definition of
# IPairsFusion#add and IPairsFusion#sub methods
def createIPairs()
488   @fusionSets["merge"].each do |fe|
       if fe.isInterface?()
           fe.toFuse.each_value do |nodeData|
               nodeData.arcs.each_pair do |pt, arcs|
                   arcs.each do |arcData|
493                       targetID = arcData.xml.attributes['target']
                       targetFE = isInterface?(targetID)
                       if targetFE != nil then
                           @iPairsList.addIPair(fe.placeTransition,
                                                   fe, targetFE, arcData)
498                       end
                   end
               end
           end
       end
end
503
end
end
end # class Operation

class Addition < Operation
508   def initialize(xml, netID, op="add-") super(xml, netID, op) end

   def compute()
       totalNet = add("merge")
       writeFile(totalNet)
513   end

   def add(mergeRemoval)
       createNets()
       createFusionSets(mergeRemoval)
518       totalNet = unifyNets()
       createIPairs()
       processIPairs(totalNet)
       collapse(totalNet)
       return totalNet
523   end

   def processIPairs(totalNet)
       ip = IPairsFusion.new(@iPairsList, totalNet)
       ip.add()
528   end

   def writeFile(totalNet)
       fileName = totalNet.elements["pnml"].elements["net"].attributes["id"]
       fname = fileName + ".pnml.xml"
533       f = File.new(fname, "w")
       #printf(f, totalNet.to_s)
       totalNet.write(f,3)
       f.close
       OPNML.addToWrittenFiles(fname)
538   end
end

```



```

# Applies the fusion elements in the fusion set merge
# to the unified total net
def collapse(totalNet)
543   net = totalNet.elements["pnml"].elements["net"]
      @fusionSets["merge"].each do |fe|
        #first remove the fused tokens from net
        Net.removeNodes(net, fe.toFuse, fe.final)
        #add element resulting from fusion
548   pt = net.add_element(fe.placeTransition, {'id' => fe.final})
        #add labels for the element resulting from fusion
        labels = collectLabels(fe.toFuse)
        if (not labels.empty?()) then
          labels.each{ |labelID|
553             p = pt.add_element(labelID)
                p.add_element(Document.new(
                    NodeFusion.new(labelID, fe).addsub(@op)).root)
          }
        end
558   end
      end
end # class Addition

563 class Subtraction < Addition
  def initialize(xml, netID, op="sub_") super(xml, netID, op) end
  def compute()
    totalNet = add("merge")
    createFusionSets("removal")
568   subtract(totalNet)
    writeFile(totalNet)
  end

  def processIPairs(totalNet)
573   ip = IPairsFusion.new(@iPairsList, totalNet)
      ip.sub()
  end

  def subtract(totalNet)
578   net = totalNet.elements["pnml"].elements["net"]
      @fusionSets["removal"].each do |fe|
        Net.removeNodesAndArcs(net, fe.toFuse)
      end
  end
583 end # end class Subtraction

# Stores the xml representation for a node label.
# Also stores the sign from the respective net.
# Used in the NodeFusion class
588 class Label
  attr_reader :xml, :sign
  def initialize(xml, sign)
    @xml = xml
    @sign = sign
593   end
end # end class Label

# Defines the transformations to be applied to

```

```

598 # the several fused net labels.
# Each label "lab" can have a method named "add_lab" and
# another named "sub_lab".
class NetFusion
  def initialize(label, netLevelLabelList)
603     @label = label
        @labels = netLevelLabelList
  end
  #calls method with @label name
  def addsub(as)
608     return eval(as + @label)
  end
end

613

# Defines the transformations to be applied to
# the several fused labels.
# Each label "lab" can have a method named "add_lab" and
618 # another named "sub_lab". The former is applied to interface places
# (merge part) in net addition. The latter is applied to interface places
# (merge part) in net subtraction.
# NetFusion objects are created in methods Addition#collapse and
# Subtraction#collapse .
623 class NodeFusion
  def initialize(label, fe)
    @label = label
    @labels = Array.new
    @fe = fe
628    @nodesToFuse = fe.toFuse
    @nodesToFuse.each_value do |nodeData|
      labelXML = nodeData.xml.elements[label]
      @labels.push(Label.new(labelXML, nodeData.sign)) if labelXML
    end
633  end

  def addsub(as)
    return eval(as + @label)
  end
end
638 end # end class NodeFusion

# Interface pair. This is a pair where both elements are a set of places
# or a set of transitions. The arcs between the place and the transition
# are also stored. All the elements in the IPairList class are IPair objects.
643 # Used in the IPairsFusion class
class IPair
  attr_reader :placeFE, :transitionFE, :ptArcs, :tpArcs
  def initialize(placeFE, transitionFE)
    @placeFE = placeFE
648    @transitionFE = transitionFE
    @ptArcs = Array.new
    @tpArcs = Array.new
  end
  def addPTArc(arcData)
653    @ptArcs.push(arcData)
  end
  def addTPArc(arcData)

```

```

        @tpArcs.push(arcData)
    end
658 end

# Defines the transformations to be applied to
# the arcs between interface nodes (the IPair objects).
# A method named "add" and another named "sub_lab" must be defined.
663 # The former is applied to interface pairs
# (merge part) in net addition. The latter is applied to interface pairs
# (merge part) in net subtraction.
# IPairsFusion objects are created in methods Addition#processIPairs and
# Subtraction#processIPairs.
668 class IPairsFusion
    def initialize(iPairsList, totalNet)
        @iPairsList = iPairsList
        @iPairsList.list.each do |ip|
            puts "Interface Pair (result place, result transition) = (' +
673             ip.placeFE.final.to_s + ', ' + ip.transitionFE.final.to_s + ')"
        end
        @net = totalNet.elements["pnml"].elements["net"]
    end

    def deleteArc(arc)
        #auxiliary function used in add() and sub() implementations
        @net.elements.each do |e|
            @net.delete_element(e) if (e.name() == "arc") && (e.attributes['id'] == arc.id)
        end
683 end
end # class IPairsFusion

if ARGV.size == 1 || ARGV.size == 2 then
    $prefix = (ARGV[0] == "-p")
688 cpnReader = OPNML.new(ARGV[$prefix ? 1: 0]).readFile()
else
    puts "-----"
    puts "OperationalPNML to PNML"
    puts "opnml2pnml.rb v. 0.5"
693 puts "Beja, 2005/09/19"
    puts "Joao Paulo Barros"
    puts "-----"
    puts "Usage: ruby [-p] opnml2pnml.rb net.opnml.xml\n"
    puts "Option [-p] adds net identifiers as prefix to all nodes in operand nets\n"
698 exit
end
end

```

Listagem D.2: Código fonte para a estrutura de transformação para redes de Petri lugar-transição, em particular para o PNTD `ptNetb`.

```

# For Operational PNML to PNML
# Author: Joao Paulo Barros
# Beja, Portugal
# 2005/09/25
5
# Transformation structure for nets with type "ptNetb" as PNTD

class NetFusion
    def add_name()
10     result = ""

```

```

    @labels.each do |label|
      labelContent = label.xml.elements["text"].text
      result.concat(labelContent)
    end
15   return "<name><text>#{result}</text></name>"
end

def sub_name()
  return add_name()
20   end
end

class NodeFusion
  def add_initialMarking()
25     result = 0
    @labels.each do |label|
      result += label.xml.elements["text"].text.to_i
    end
    return "<text>#{result}</text>"
30   end

  def sub_initialMarking()
    result = 0
    @labels.each do |label|
35      result += label.sign * label.xml.elements["text"].text.to_i
    end
    return "<text>#{result}</text>"
end

40   def add_name()
    result = ""
    @labels.each do |label|
      labelContent = label.xml.elements["text"].text
      result.concat(labelContent)
45   end
    return "<text>#{result}</text>"
end

  def sub_name()
50     result = ""
    @labels.each do |label|
      labelContent = label.xml.elements["text"].text
      result.concat(labelContent) if label.sign > 0
    end
55     @labels.each do |label|
      labelContent = label.xml.elements["text"].text
      result.gsub!(/#{labelContent}/, "") if label.sign < 0
    end
    return "<text>#{result}</text>"
60   end
end # end class NodeFusion

class IPairsFusion
65   def add() # this one merges the several arcs between each interface pair
    @iPairsList.list.each do |ip| # for each interface pair
      weightPT = 0
      weightTP = 0

```

```

70      ip.ptArcs.each do |ptArc| #for each place->transition arc
          weightPT = weightPT +
              ptArc.xml.elements['inscription'].elements['text'].text.to_i
          deleteArc(ptArc)
        end
      ip.tpArcs.each do |tpArc| #for each transition->place arc
75          weightTP = weightTP +
              tpArc.xml.elements['inscription'].elements['text'].text.to_i
          deleteArc(tpArc)
        end
      p = ip.placeFE.final
80      t = ip.transitionFE.final
      if weightPT > 0
          # add p->t arc element
          a = @net.add_element('arc', {'id' => p + t, 'source'=> p, 'target' => t})
          a.add_element('inscription').add_element('text').add_text(weightPT.to_s)
85      end
      if weightTP > 0
          # add t->p arc element
          a = @net.add_element('arc', {'id' => t + p, 'source'=> t, 'target' => p})
          a.add_element('inscription').add_element('text').add_text(weightTP.to_s)
90      end
    end
  end

  def sub() # this one merges the several arcs between each interface pair
95      @iPairsList.list.each do |ip| # for each interface pair
          weightPT = 0
          weightTP = 0
          ip.ptArcs.each do |ptArc| #for each place->transition arc
              weightPT = weightPT +
100                  ptArc.sign *
                  ptArc.xml.elements['inscription'].elements['text'].text.to_i
              deleteArc(ptArc)
            end
          ip.tpArcs.each do |tpArc| #for each transition->place arc
105              weightTP = weightTP +
                  tpArc.sign *
                  tpArc.xml.elements['inscription'].elements['text'].text.to_i
              deleteArc(tpArc)
            end
          p = ip.placeFE.final
110          t = ip.transitionFE.final
          if weightPT > 0
              # add p->t arc element
              a = @net.add_element('arc', {'id'=> p + t, 'source'=> p, 'target' => t})
              a.add_element('inscription').add_element('text').add_text(weightPT.to_s)
115          elsif weightPT < 0
              Error("Arc weight between place #{p} and transition #{t} is #{weightPT}.")
            end
          if weightTP > 0
              # add t->p arc element
120              a = @net.add_element('arc', {'id'=> t + p, 'source'=> t, 'target' => p})
              a.add_element('inscription').add_element('text').add_text(weightTP.to_s)
            elsif weightTP < 0
              Error("Arc weight between transition #{t} and place #{p} is #{weightTP}.")
125            end
          end
        end
  end
end

```

```

end
end # class IPairsFusion

```

Listagem D.3: Código fonte para uma estrutura de transformação `ColorSimple` referida na Secção 3.5.3 (pág. 86).

```

1  # For opnml2pnml
  # Author: Joao Paulo Barros
  # Beja, Portugal
  # 2005/09/19

6  # Defines a transformation structure for a PNID named colorSimple
  # The net structure only overrides ptNetb.ts for addition.
  # initialMarking labels are "added" by simply choosing one of them for
  #   the result node.
  # Channel annotations are removed.
11 # Guard addition

  require 'ptNetb.ts' #defaults come from place/transition nets

  class NodeFusion
16    def add_initialMarking()
      result = @labels[0].xml.elements["text"].text
      return "<text>#{result}</text>"
    end

21    def add_channel()
      return "<text></text>"
    end

    def add_guard()
26    result = '(' + @labels[0].xml.elements["text"].text + ')'
      for i in 1..@labels.size - 1
        result = result + ' and (' + @labels[i].xml.elements["text"].text + ')'
      end

31    channels = @fe.labels['channel']

    if (!channels.empty?())
      # add condition to guard due to the channels.
      # Only two channels are supported.
36    if (channels.size() != 2)
        Error("The number of channels must be 2.")
      end
      paramLists = Array.new
      channels.each do |ch|
41        m = Regexp.new('(.*)\!\\?.*').match(ch.elements['text'].text)
        Error("Syntax error in channel expression \"#{ch}\".") if m == nil
        paramLists.push(Regexp.last_match(1).to_s.split(','))
      end
      if (paramLists[0].size() != paramLists[1].size())
46        Error('Mismatch in the number of channel parameters');
      end
      # Add comparisons, between parameters, to the guard
      for i in 0..(paramLists[0].size() - 1) do
        result = '(' + paramLists[0][i].strip() +
51        ' = ' + paramLists[1][i].strip() + ')' and ' + result
      end
    end
  end

```

```

        end
        return "<text>#{result}</text>"
    end
56 end # end class NodeFusion

```

Listagem D.4: Código fonte para a estrutura de transformação `ptTest` referida na Secção 3.5.3 (pág. 89).

```

# For Operational PNML to PNML
# Author: Joao Paulo Barros
3 # Beja, Portugal
# 2005/09/15

# Transformation structure for nets with type "ptTest" as PNID
# Extends the ptNetb transformation structure
8 # Test arcs MUST have a transition as source

require "ptNetb.ts"

class NodeFusion
13 end # end class NodeFusion

class IPairsFusion
    def add() # this one merges the several arcs between each interface pair
        @iPairsList.list.each do |ip| # for each interface pair
18            weightPT = 0
            weightTP = 0
            weightTPTest = 0
            ip.ptArcs.each do |ptArc| #for each place->transition arc
                weight = ptArc.xml.elements['inscription'].elements['text'].text.to_i
23                weightPT += weight
                deleteArc(ptArc)
            end
            ip.tpArcs.each do |tpArc| #for each transition->place arc
                weight = tpArc.xml.elements['inscription'].elements['text'].text.to_i
28                if (ptArc.xml.elements['type'] &&
                    ptArc.xml.elements['type'].elements['text'] &&
                    ptArc.xml.elements['type'].elements['text'].text.eql?('test'))
                    weightTPTest += weight
                else
33                    weightTP += weight
                end
                deleteArc(tpArc)
            end
            p = ip.placeFE.final
            t = ip.transitionFE.final
38            if weightPT > 0
                # add p->t arc element
                a = @net.add_element('arc', {'id' => p + t, 'source'=> p, 'target' => t})
                a.add_element('inscription').add_element('text').add_text(weightPT.to_s)
43            end
            if weightTP > 0
                # add t->p arc element
                a = @net.add_element('arc', {'id' => t + p, 'source'=> t, 'target' => p})
                a.add_element('inscription').add_element('text').add_text(weightTP.to_s)
48            end

            if weightTPTest > 0 # add test arc

```

```

# add t->p arc element
a = @net.add_element('arc', {'id' => t + p, 'source'=> t, 'target' => p})
53 a.add_element('inscription').add_element('text').add_text(weightTPTest.to_s)
a.add_element('type').add_element('text').add_text('test')
end
end
end
58
def sub() # this one merges the several arcs between each interface pair
  @iPairsList.list.each do |ip| # for each interface pair
    weightPT = 0
    weightTP = 0
63 weightTPTest = 0
    ip.ptArcs.each do |ptArc| #for each place->transition arc
      weightPT = weightPT +
        ptArc.sign *
        ptArc.xml.elements['inscription'].elements['text'].text.to_i
68 deleteArc(ptArc)
    end
    ip.tpArcs.each do |tpArc| #for each transition->place arc
      weight = tpArc.sign *
        tpArc.xml.elements['inscription'].elements['text'].text.to_i
73 if (ptArc.xml.elements['test'] == nil)
        weightTPTest += weight
      else
        weightTP += weight
      end
78 deleteArc(tpArc)
    end
    p = ip.placeFE.final
    t = ip.transitionFE.final
    if weightPT > 0
83 # add p->t arc element
      a = @net.add_element('arc', {'id'=> p + t, 'source'=> p, 'target' => t})
      a.add_element('inscription').add_element('text').add_text(weightPT.to_s)
    elsif weightPT < 0
      Error("Arc weight between place #{p} and transition #{t} is #{weightPT}.")
88 end
    if weightTP > 0
      # add t->p arc element
      a = @net.add_element('arc', {'id'=> t + p, 'source'=> t, 'target' => p})
      a.add_element('inscription').add_element('text').add_text(weightTP.to_s)
93 elsif weightTP < 0
      Error("Arc weight between transition #{t} and place #{p} is #{weightTP}.")
    end

    if weightTPTest > 0 # add test arc
98 # add t->p arc element
      a = @net.add_element('arc', {'id' => t + p, 'source'=> t, 'target' => p})
      a.add_element('inscription').add_element('text').add_text(weightTPTest.to_s)
      a.add_element('test').add_element('text').add_text(weightTPTest.to_s)
    end
103
  end
end
end # class IPairsFusion

```


Listagem D.5: Código fonte para a estrutura de transformação `ptio` referida na Secção C.4 (pág. 212).

```
# For Operational PNML to PNML
# Author: Joao Paulo Barros
3 # Beja, Portugal
# 2005/09/25

# Transformation structure for nets with type "ptio" as PNID
# Extends the ptTest transformation structure
8
require "ptTest.ts"

class NetFusion
  def add_input()
13     result = ''
        @labels.each do |label|
            label.xml.each_element do |elem|
                result += elem.to_s
            end
18     end
        return "<output>#{result}</output>"
    end

    def add_output()
23     result = ''
        @labels.each do |label|
            label.xml.each_element do |elem|
                result += elem.to_s
            end
28     end
        return "<output>#{result}</output>"
    end
end

33
class NodeFusion

  def add_inputEvents()
      result = ''
38     @labels.each do |label|
        label.xml.each_element do |elem|
            result += elem.to_s
        end
      end
43     return result
  end

  def add_outputEvents()
      result = ''
48     @labels.each do |label|
        label.xml.each_element do |elem|
            result += elem.to_s
        end
      end
53     return result
  end

  def add_signalInputGuard()
```

```

    result = @labels[0].xml.elements["text"].text
58   for i in 1..(@labels.size() - 1)
        result = '(' + result + ')'
        result += ' && (' + @labels[i].xml.elements["text"].text + ')'
    end
    return "<text>#{result}</text>"
63   end

    def add_signalOutputActions()
        result = ''
        @labels.each do |label|
68           label.xml.each_element do |elem|
                result += elem.to_s
            end
        end
        return result
73   end

end # end class NodeFusion

```

D.2 Exemplos de Modelos em Operational PNML

Esta secção apresenta exemplos de modelos em Operational PNML e de modelos em PNML que servem como operandos aos primeiros. Alguns dos quais foram já apresentados na Secção 4.2.2.

D.2.1 Canal Síncrono (*vide* Secção 3.5.3 na pág. 86)

Listagem D.6: Especificação PNML para o modelo *Producer* na Fig. 2.6 da pág. 34.

```

1  <pnml xmlns="colorSimpleNS">
    <net id="ProducerChannel" type="colorSimple">
        <name>
            <text>Producer</text>
        </name>
6   <place id="produced">
        <name>
            <text>produced</text>
        </name>
        <initialMarking>
11         <text>(P1, 5)</text>
        </initialMarking>
    </place>
    <place id="sent">
        <name>
16         <text>sent</text>
        </name>
        <initialMarking>
            <text></text>
        </initialMarking>
21    </place>
    <transition id="produce">
        <name>
            <text>produce</text>

```

```

26      </name>
      </transition>
      <transition id="send">
        <name>
          <text>send</text>
        </name>
31      <guard>
        <text>p='P1'</text>
      </guard>
      <channel>
        <text>x!?!ch</text>
36      </channel>
      </transition>
      <arc id="a1" source="produce" target="produced">
        <inscription>
          <text>(p,x)</text>
41      </inscription>
      </arc>
      <arc id="a2" source="produced" target="send">
        <inscription>
          <text>(p,x)</text>
46      </inscription>
      </arc>
      <arc id="a3" source="send" target="sent">
        <inscription>
          <text>p</text>
51      </inscription>
      </arc>
      <arc id="a4" source="sent" target="produce">
        <inscription>
          <text>p</text>
56      </inscription>
      </arc>
    </net>
  </pnml>

```

Listagem D.7: Especificação PNML para o modelo *Consumer* na Fig. 2.6 da pág. 34.

```

1  <pnml xmlns="colorSimpleNS">
    <net id="ConsumerChannel" type="colorSimple">
      <name>
        <text>Consumer</text>
      </name>
6    <place id="received">
      <name>
        <text>produced</text>
      </name>
      <initialMarking>
11      <text></text>
      </initialMarking>
    </place>
    <place id="consumed">
      <name>
16      <text>sent</text>
      </name>
      <initialMarking>
        <text>5</text>
      </initialMarking>

```

```

21  </place>
    <transition id="receive">
      <name>
        <text>receive</text>
      </name>
26  <guard>
      <text>c='C1' or c='C2'</text>
    </guard>
    <channel>
      <text>y!?ch</text>
31  </channel>
    </transition>
    <transition id="consume">
      <name>
        <text>consume</text>
36  </name>
    </transition>
    <arc id="a1" source="receive" target="received">
      <inscription>
        <text>(c,y)</text>
41  </inscription>
    </arc>
    <arc id="a2" source="received" target="consume">
      <inscription>
        <text>(c,y)</text>
46  </inscription>
    </arc>
    <arc id="a3" source="consume" target="consumed">
      <inscription>
        <text>c</text>
51  </inscription>
    </arc>
    <arc id="a4" source="consumed" target="receive">
      <inscription>
        <text>c</text>
56  </inscription>
    </arc>
  </net>
</pnml>

```

Listagem D.8: Especificação OPNML para o modelo na Fig. 2.7 da pág. 34.

```

1  <pnml xmlns="colorSimpleNS">
    <net id="ProducerConsumerChannel"
      type="colorSimple">
      <operations>
        <addition result="return">
6      <net netID="ProducerChannel"/>
        <net netID="ConsumerChannel"/>
        <merge>
          <transitionFusion nodeID="sendReceive">
            <transition netID="ProducerChannel" nodeID="send"/>
11          <transition netID="ConsumerChannel" nodeID="receive"/>
          </transitionFusion>
        </merge>
      </addition>
    </operations>
16  </net>

```

</pnml>

Listagem D.9: Especificação PNML produzida pelo protótipo `opnml2pnml` a partir da especificação na Listagem D.8 e da estrutura de transformação na Listagem D.3.

```

3  <pnml xmlns='colorSimpleNS'>
    <net type='colorSimple' id='ProducerConsumerChannel'>
        <name>
            <text>ConsumerProducer</text>
        </name>
        <place id='consumed'>
8         <name>
            <text>sent</text>
        </name>
        <initialMarking>
            <text>5</text>
13        </initialMarking>
        </place>
        <place id='produced'>
            <name>
18            <text>produced</text>
        </name>
        <initialMarking>
            <text>(P1, 5)</text>
        </initialMarking>
        </place>
23        <place id='received'>
            <name>
                <text>produced</text>
            </name>
            <initialMarking>
28            <text/>
        </initialMarking>
        </place>
        <place id='sent'>
            <name>
33            <text>sent</text>
        </name>
        <initialMarking>
            <text/>
        </initialMarking>
38        </place>
        <transition id='consume'>
            <name>
                <text>consume</text>
            </name>
43        </transition>
        <transition id='produce'>
            <name>
                <text>produce</text>
            </name>
48        </transition>
        <arc id='ConsumerChannel.a1' source='sendReceive' target='received'>
            <inscription>
                <text>(c,y)</text>
            </inscription>
53        </arc>

```

```

    <arc id='ConsumerChannel.a2' source='received' target='consume'>
      <inscription>
        <text>(c,y)</text>
      </inscription>
58 </arc>
    <arc id='ConsumerChannel.a3' source='consume' target='consumed'>
      <inscription>
        <text>c</text>
      </inscription>
63 </arc>
    <arc id='ConsumerChannel.a4' source='consumed' target='sendReceive'>
      <inscription>
        <text>c</text>
      </inscription>
68 </arc>
    <arc id='ProducerChannel.a1' source='produce' target='produced'>
      <inscription>
        <text>(p,x)</text>
      </inscription>
73 </arc>
    <arc id='ProducerChannel.a2' source='produced' target='sendReceive'>
      <inscription>
        <text>(p,x)</text>
      </inscription>
78 </arc>
    <arc id='ProducerChannel.a3' source='sendReceive' target='sent'>
      <inscription>
        <text>p</text>
      </inscription>
83 </arc>
    <arc id='ProducerChannel.a4' source='sent' target='produce'>
      <inscription>
        <text>p</text>
      </inscription>
88 </arc>
    <transition id='sendReceive'>
      <channel>
        <text/>
      </channel>
93 <guard>
      <text>(x = y) and (c='C1' or c='C2') and (p='P1')</text>
    </guard>
      <name>
        <text>receivesend</text>
98 </name>
    </transition>
  </net>
</pnml>

```

D.2.2 Adição de Árbitro (*vide* Secção 3.5.3 na pág. 89)

Listagem D.10: Especificação PNML para o modelo Arbiter na Fig. 3.27a da pág. 90.

```

<pnml>
  <net id="Arbiter" type="ptTest">
    <place id="p1">
4      <name>

```

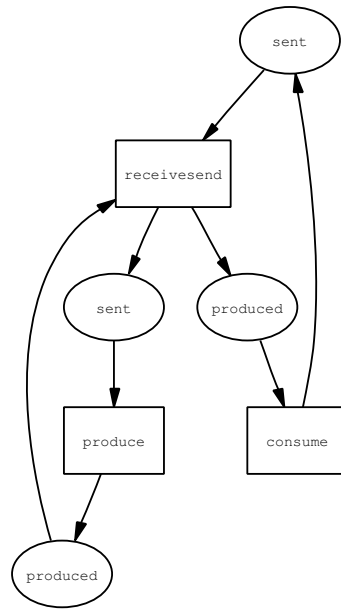


Figura D.1: Representação gráfica obtida automaticamente a partir do modelo na Listagem D.9 e correspondente ao modelo na Fig. 2.7 da pág. 34. Como anotações, foram utilizados os atributos **name** dos nós, em lugar dos respectivos identificadores.

```

    <text>p1</text>
  </name>
  <initialMarking>
    <text>1</text>
  </initialMarking>
9  </place>
  <place id="p2">
    <name>
      <text>p2</text>
14    </name>
    <initialMarking>
      <text>0</text>
    </initialMarking>
  </place>
19  <transition id="t1">
    <name>
      <text>t1</text>
    </name>
  </transition>
24  <transition id="t2">
    <name>
      <text>t2</text>
    </name>
  </transition>
29  <transition id="turn[1]">
    <name>
      <text>turn[1]</text>
    </name>
  </transition>
34  <transition id="turn[2]">
    <name>
      <text>turn[2]</text>

```

```

    </name>
  </transition>
39 <arc id="a1" source="p1" target="t1">
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
44 <arc id="a2" source="t1" target="p2">
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
49 <arc id="a3" source="p2" target="t2">
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
54 <arc id="a4" source="t2" target="p1">
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
59 <arc id="a5" source="turn[1]" target="p1">
    <inscription>
      <text>1</text>
    </inscription>
    <type>
64 <text>test</text>
    </type>
  </arc>
    <arc id="a6" source="turn[2]" target="p2">
    <inscription>
69 <text>1</text>
    </inscription>
    <type>
      <text>test</text>
    </type>
74 </arc>
  </net>
</pnml>

```

Listagem D.11: Especificação PNML para o modelo *Par* na Fig. 3.27b da pág. 90.

```

<pnml>
  <net id="Par" type="ptTest">
    <place id="p3">
4    <name>
      <text>p3</text>
    </name>
    <initialMarking>
      <text>1</text>
9    </initialMarking>
    </place>
    <place id="p4">
      <name>
14 <text>p4</text>
      </name>
      <initialMarking>

```



```

        <text>1</text>
    </initialMarking>
</place>
19 <place id="p5">
    <name>
        <text>p5</text>
    </name>
    <initialMarking>
24     <text>0</text>
    </initialMarking>
</place>
<place id="p6">
    <name>
29     <text>p6</text>
    </name>
    <initialMarking>
        <text>1</text>
    </initialMarking>
34 </place>
<place id="p7">
    <name>
        <text>p7</text>
    </name>
39 <initialMarking>
        <text>0</text>
    </initialMarking>
</place>
<transition id="start [1]" >
44     <name>
        <text>start [1] </text>
    </name>
</transition>
<transition id="start [2]" >
49     <name>
        <text>start [2] </text>
    </name>
</transition>
<transition id="entry [1]" >
54     <name>
        <text>entry [1] </text>
    </name>
</transition>
<transition id="entry [2]" >
59     <name>
        <text>entry [2] </text>
    </name>
</transition>
<transition id="exit [1]" >
64     <name>
        <text>exit [1] </text>
    </name>
</transition>
<transition id="exit [2]" >
69     <name>
        <text>exit [2] </text>
    </name>
</transition>
<arc id="a1" source="start [1]" target="p3">

```

```

74      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a2" source="start [2]" target="p4">
79      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a3" source="p3" target="entry [1]" >
84      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a4" source="p4" target="entry [2]" >
89      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a5" source="entry [1]" target="p5">
94      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a6" source="entry [2]" target="p7">
99      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a7" source="p5" target="exit [1]" >
104      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a8" source="p7" target="exit [2]" >
109      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a9" source="exit [1]" target="p3">
114      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a10" source="exit [2]" target="p4">
119      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a11" source="p6" target="entry [1]" >
124      <inscription>
          <text>1</text>
        </inscription>
    </arc>
    <arc id="a12" source="p6" target="entry [2]" >
129      <inscription>
          <text>1</text>
        </inscription>

```

```

134     </arc>
    <arc id="a13" source="exit [1]" target="p6">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a14" source="exit [2]" target="p6">
139     <inscription>
      <text>1</text>
    </inscription>
    </arc>
  </net>
144 </pnml>

```

Listagem D.12: Especificação OPNML para o modelo na Fig. 3.27c da pág. 90.

```

1 <pnml>
  <net id="ParArbiter"
    type="ptTest">
    <operations>
      <addition result="return">
6      <net netID="Arbiter"/>
      <net netID="Par"/>
      <merge>
        <transitionFusionVector nodeID="t[i]" iterator="i" first="1" last="2">
          <transition netID="Arbiter" nodeID="turn[i]"/>
11      <transition netID="Par" nodeID="entry[i]"/>
        </transitionFusionVector>
      </merge>
    </addition>
    </operations>
16  </net>
</pnml>

```

Listagem D.13: Especificação PNML produzida pelo protótipo `opnml2pnml` a partir da especificação na Listagem D.12 e da estrutura de transformação na Listagem D.4 na pág. 231.

```

3 <pnml xmlns='iopt'>
  <net type='ptTest' id='ParArbiter'>
    <place id='p1'>
      <name>
        <text>p1</text>
      </name>
8      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <place id='p2'>
13      <name>
        <text>p2</text>
      </name>
      <initialMarking>
        <text>0</text>
18      </initialMarking>
    </place>
    <place id='p3'>

```

```

    <name>
      <text>p3</text>
23    </name>
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
28    <place id='p4'>
      <name>
        <text>p4</text>
      </name>
      <initialMarking>
33        <text>1</text>
      </initialMarking>
    </place>
    <place id='p5'>
      <name>
38        <text>p5</text>
      </name>
      <initialMarking>
        <text>0</text>
      </initialMarking>
43    </place>
    <place id='p6'>
      <name>
        <text>p6</text>
      </name>
48    <initialMarking>
      <text>1</text>
    </initialMarking>
    </place>
    <place id='p7'>
53    <name>
      <text>p7</text>
    </name>
    <initialMarking>
      <text>0</text>
58    </initialMarking>
    </place>
    <transition id='exit[1] '>
      <name>
        <text>exit[1]</text>
63    </name>
    </transition>
    <transition id='exit[2] '>
      <name>
        <text>exit[2]</text>
68    </name>
    </transition>
    <transition id='start[1] '>
      <name>
        <text>start[1]</text>
73    </name>
    </transition>
    <transition id='start[2] '>
      <name>
        <text>start[2]</text>
78    </name>

```

```

      </transition>
      <transition id='t1'>
        <name>
          <text>t1</text>
83      </name>
      </transition>
      <transition id='t2'>
        <name>
          <text>t2</text>
88      </name>
      </transition>
      <arc id='Arbiter.a1' source='p1' target='t1'>
        <inscription>
          <text>l</text>
93      </inscription>
      </arc>
      <arc id='Arbiter.a2' source='t1' target='p2'>
        <inscription>
          <text>l</text>
98      </inscription>
      </arc>
      <arc id='Arbiter.a3' source='p2' target='t2'>
        <inscription>
          <text>l</text>
103      </inscription>
      </arc>
      <arc id='Arbiter.a4' source='t2' target='p1'>
        <inscription>
          <text>l</text>
108      </inscription>
      </arc>
      <arc id='Arbiter.a5' source='t[1]' target='p1'>
        <inscription>
          <text>l</text>
113      </inscription>
        <type>
          <text>test</text>
        </type>
      </arc>
118      <arc id='Arbiter.a6' source='t[2]' target='p2'>
        <inscription>
          <text>l</text>
        </inscription>
        <type>
          <text>test</text>
123      </type>
      </arc>
      <arc id='Par.a1' source='start[1]' target='p3'>
        <inscription>
          <text>l</text>
128      </inscription>
      </arc>
      <arc id='Par.a10' source='exit[2]' target='p4'>
        <inscription>
          <text>l</text>
133      </inscription>
      </arc>
      <arc id='Par.a11' source='p6' target='t[1]'>

```

```

138      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a12' source='p6' target='t[2] '>
      <inscription>
143      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a13' source='exit[1]' target='p6'>
      <inscription>
148      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a14' source='exit[2]' target='p6'>
      <inscription>
153      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a2' source='start[2]' target='p4'>
      <inscription>
158      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a3' source='p3' target='t[1] '>
      <inscription>
163      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a4' source='p4' target='t[2] '>
      <inscription>
168      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a5' source='t[1]' target='p5'>
      <inscription>
173      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a6' source='t[2]' target='p7'>
      <inscription>
178      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a7' source='p5' target='exit[1] '>
      <inscription>
183      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a8' source='p7' target='exit[2] '>
      <inscription>
188      <text>1</text>
      </inscription>
    </arc>
    <arc id='Par.a9' source='exit[1]' target='p3'>
      <inscription>
193      <text>1</text>
      </inscription>

```

```

198     </arc>
        <transition id='t[1] '>
            <name>
                <text>entry [1] turn [1]</text>
            </name>
        </transition>
        <transition id='t[2] '>
            <name>
203         <text>entry [2] turn [2]</text>
            </name>
        </transition>
    </net>
</pnml>

```

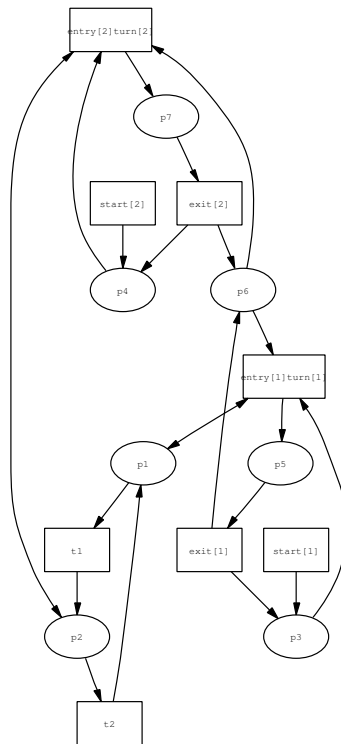


Figura D.2: Representação gráfica obtida automaticamente a partir do modelo na Listagem D.13 e correspondente ao modelo na Fig. 3.27c da pág. 90. Como anotações, foram utilizados os atributos **name** dos nós, em lugar dos respectivos identificadores.

D.2.3 Modelo ParkAI0 (*vide* Secção C.4 na pág. 212)

Listagem D.14: Especificação PNML para o modelo **EnterIO** na Fig. C.2 da pág. 212.

```

<pnml>
  <net id="EnterIO" type="ptio">
3    <input>
        <event id="arriveUpEvent"/>
        <event id="arriveDownEvent"/>
    </input>
    <output>
8    <signal id="gateInSignal" value="0"/>

```

```

</output>
<place id="entranceFree">
  <name>
    <text>entranceFree</text>
13  </name>
    <initialMarking>
      <text>1</text>
    </initialMarking>
  </place>
18  <place id="waitingTicket">
    <name>
      <text>waitingTicket</text>
    </name>
    <initialMarking>
23  <text>0</text>
    </initialMarking>
  </place>
  <place id="gateOpenE">
    <name>
28  <text>gateOpenE</text>
    </name>
    <initialMarking>
      <text>0</text>
    </initialMarking>
33  <signalOutputActions>
      <signalOutputAction idref="gateInSignal" value="1">marking > 0</signalOutputAction>
    </signalOutputActions>
  </place>
  <transition id="arriveUp">
38  <name>
      <text>arriveUp</text>
    </name>
    <inputEvents>
      <event idref="arriveUpEvent"/>
43  </inputEvents>
    </transition>
    <transition id="arriveDown">
      <name>
        <text>arriveDown</text>
48  </name>
      <inputEvents>
        <event idref="arriveDownEvent"/>
      </inputEvents>
    </transition>
53  <transition id="gotTicket">
      <name>
        <text>gotTicket</text>
      </name>
    </transition>
58  <arc id="a1" source="entranceFree" target="arriveUp">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
63  <arc id="a2" source="arriveUp" target="waitingTicket">
      <inscription>
        <text>1</text>
      </inscription>

```



```

68     </arc>
    <arc id="a3" source="waitingTicket" target="gotTicket">
        <inscription>
            <text>1</text>
        </inscription>
    </arc>
73     <arc id="a4" source="gotTicket" target="gateOpenE">
        <inscription>
            <text>1</text>
        </inscription>
    </arc>
78     <arc id="a5" source="gateOpenE" target="arriveDown">
        <inscription>
            <text>1</text>
        </inscription>
    </arc>
83     <arc id="a6" source="arriveDown" target="entranceFree">
        <inscription>
            <text>1</text>
        </inscription>
    </arc>
88 </net>
</pnml>

```

Listagem D.15: Especificação PNML para o modelo `ParkingAreaIO` na Fig. C.2 da pág. 212.

```

1 <pnml xmlns="ptio">
    <net id="ParkingAreaIO" type="ptio">
        <input>
            <signal id="parkOpenedSignal"/>
            <event id="canEnterEvent"/>
6            <event id="canLeaveEvent"/>
        </input>
        <output>
            <event id="enterWarningEvent" value="0"/>
        </output>
11    <place id="occupied">
        <name>
            <text>occupied</text>
        </name>
        <initialMarking>
16            <text>0</text>
        </initialMarking>
    </place>
    <place id="free">
        <name>
21            <text>free</text>
        </name>
        <initialMarking>
            <text>0</text>
        </initialMarking>
26    </place>
    <transition id="in">
        <name>
            <text>in</text>
        </name>
31    <inputEvents>

```

```

    <event idref="canEnterEvent"/>
  </inputEvents>
  <outputEvents>
    <event idref="enterWarningEvent"/>
36  </outputEvents>
    <signalInputGuard>
      <text>(parkOpenedSignal == 1)</text>
    </signalInputGuard>
  </transition>
41  <transition id="out">
    <name>
      <text>out</text>
    </name>
    <inputEvents>
46    <event idref="canLeaveEvent"/>
    </inputEvents>
    <signalInputGuard>
      <text>(parkOpenedSignal == 1)</text>
    </signalInputGuard>
51  </transition>
  <arc id="a1" source="in" target="occupied">
    <inscription>
      <text>1</text>
    </inscription>
56  </arc>
  <arc id="a2" source="occupied" target="out">
    <inscription>
      <text>1</text>
    </inscription>
61  </arc>
  <arc id="a3" source="out" target="free">
    <inscription>
      <text>1</text>
    </inscription>
66  </arc>
  <arc id="a4" source="free" target="in">
    <inscription>
      <text>1</text>
    </inscription>
71  </arc>
</net>
</pnml>

```

Listagem D.16: Especificação PNML para o modelo LeaveIO na Fig. C.2 da pág. 212.

```

<pnml xmlns="ptio">
2  <net id="LeaveIO" type="ptio">
    <input>
      <event id="exitUpEvent"/>
      <event id="exitDownEvent"/>
    </input>
7  <output>
    <signal id="gateOutSignal" value="0"/>
  </output>
  <place id="gateOpenL">
    <name>
12    <text>gateOpenL</text>
    </name>

```

```

    <initialMarking>
      <text>0</text>
    </initialMarking>
17    <signalOutputActions>
      <signalOutputAction idref="gateOutSignal" value="1">marking > 0</signalOutputAction>
    </signalOutputActions>
  </place>
  <place id="exitFree">
22    <name>
      <text>exitFree</text>
    </name>
    <initialMarking>
      <text>1</text>
27    </initialMarking>
  </place>
  <place id="waitingToPay">
    <name>
      <text>waitingToPay</text>
32    </name>
    <initialMarking>
      <text>0</text>
    </initialMarking>
  </place>
37  <transition id="pay">
    <name>
      <text>pay</text>
    </name>
  </transition>
42  <transition id="exitDown">
    <name>
      <text>exitDown</text>
    </name>
    <inputEvents>
47    <event idref="exitDownEvent"/>
    </inputEvents>
  </transition>
  <transition id="exitUp">
    <name>
52    <text>exitUp</text>
    </name>
    <inputEvents>
      <event idref="exitUpEvent"/>
    </inputEvents>
57  </transition>
  <arc id="a1" source="pay" target="gateOpenL">
    <inscription>
      <text>1</text>
    </inscription>
62  </arc>
  <arc id="a2" source="gateOpenL" target="exitDown">
    <inscription>
      <text>1</text>
    </inscription>
67  </arc>
  <arc id="a3" source="exitDown" target="exitFree">
    <inscription>
      <text>1</text>
    </inscription>

```

```

72     </arc>
    <arc id="a4" source="exitFree" target="exitUp">
        <inscription>
            <text>1</text>
        </inscription>
77     </arc>
    <arc id="a5" source="exitUp" target="waitingToPay">
        <inscription>
            <text>1</text>
        </inscription>
82     </arc>
    <arc id="a6" source="waitingToPay" target="pay">
        <inscription>
            <text>1</text>
        </inscription>
87     </arc>
</net>
</pnml>

```

Listagem D.17: Especificação OPNML para o modelo ParkAIO na Fig. C.3 da pág. 213.

```

1  <pnml xmlns="ptio">
    <net id="ParkAIO"
        type="ptio">
        <operations>
            <addition result="return">
6              <net netID="EnterIO"/>
              <net netID="ParkingAreaIO"/>
              <net netID="LeaveIO"/>
              <merge>
11                 <transitionFusion nodeID="in">
                    <transition netID="EnterIO" nodeID="gotTicket"/>
                    <transition netID="ParkingAreaIO" nodeID="in"/>
                </transitionFusion>
                <transitionFusion nodeID="out">
                    <transition netID="ParkingAreaIO" nodeID="out"/>
16                 <transition netID="LeaveIO" nodeID="pay"/>
                </transitionFusion>
            </merge>
        </addition>
    </operations>
21 </net>
</pnml>

```

Listagem D.18: Especificação PNML produzida pelo protótipo `opnml2pnml` a partir da especificação na Listagem D.17 e da estrutura de transformação na Listagem D.5 na pág. 232.

```

    <net type='ptio' id='ParkAIO'>
        <output>
4            <signal id='gateOutSignal' value='0'/>
            <signal id='gateInSignal' value='0'/>
            <event id='enterWarningEvent' value='0'/>
        </output>
        <output>
9            <event id='exitUpEvent'/>
            <event id='exitDownEvent'/>
            <event id='arriveUpEvent'/>

```

```

        <event id='arriveDownEvent'/>
        <signal id='parkOpenedSignal'/>
14      <event id='canEnterEvent'/>
        <event id='canLeaveEvent'/>
        </output>
        <place id='entranceFree'>
        <name>
19      <text>entranceFree</text>
        </name>
        <initialMarking>
        <text>1</text>
        </initialMarking>
24    </place>
        <place id='exitFree'>
        <name>
        <text>exitFree</text>
        </name>
29    <initialMarking>
        <text>1</text>
        </initialMarking>
        </place>
        <place id='free'>
34    <name>
        <text>free</text>
        </name>
        <initialMarking>
        <text>0</text>
39    </initialMarking>
        </place>
        <place id='gateOpenE'>
        <name>
        <text>gateOpenE</text>
44    </name>
        <initialMarking>
        <text>0</text>
        </initialMarking>
        <signalOutputActions>
49      <signalOutputAction idref='gateInSignal' value='1'>marking > 0</signalOutputAction>
        </signalOutputActions>
        </place>
        <place id='gateOpenL'>
        <name>
54    <text>gateOpenL</text>
        </name>
        <initialMarking>
        <text>0</text>
        </initialMarking>
59    <signalOutputActions>
        <signalOutputAction idref='gateOutSignal' value='1'>marking > 0</signalOutputAction>
        </signalOutputActions>
        </place>
        <place id='occupied'>
64    <name>
        <text>occupied</text>
        </name>
        <initialMarking>
        <text>0</text>
69    </initialMarking>

```

```

</place>
  <place id='waitingTicket'>
    <name>
      <text>waitingTicket</text>
74    </name>
    <initialMarking>
      <text>0</text>
    </initialMarking>
  </place>
79  <place id='waitingToPay'>
    <name>
      <text>waitingToPay</text>
    </name>
    <initialMarking>
84      <text>0</text>
    </initialMarking>
  </place>
    <transition id='arriveDown'>
      <name>
89        <text>arriveDown</text>
      </name>
      <inputEvents>
        <event idref='arriveDownEvent' />
      </inputEvents>
94    </transition>
      <transition id='arriveUp'>
        <name>
          <text>arriveUp</text>
        </name>
99        <inputEvents>
          <event idref='arriveUpEvent' />
        </inputEvents>
    </transition>
      <transition id='exitDown'>
104      <name>
        <text>exitDown</text>
      </name>
      <inputEvents>
        <event idref='exitDownEvent' />
109      </inputEvents>
    </transition>
      <transition id='exitUp'>
        <name>
114        <text>exitUp</text>
      </name>
      <inputEvents>
        <event idref='exitUpEvent' />
      </inputEvents>
    </transition>
119    <arc id='EnterIO.a1' source='entranceFree' target='arriveUp'>
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
124    <arc id='EnterIO.a2' source='arriveUp' target='waitingTicket'>
      <inscription>
        <text>1</text>
      </inscription>

```

```

129      </arc>
      <arc id='EnterIO.a3' source='waitingTicket' target='in'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
134      <arc id='EnterIO.a4' source='in' target='gateOpenE'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
139      <arc id='EnterIO.a5' source='gateOpenE' target='arriveDown'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
144      <arc id='EnterIO.a6' source='arriveDown' target='entranceFree'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
149      <arc id='LeaveIO.a1' source='out' target='gateOpenL'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
154      <arc id='LeaveIO.a2' source='gateOpenL' target='exitDown'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
159      <arc id='LeaveIO.a3' source='exitDown' target='exitFree'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
164      <arc id='LeaveIO.a4' source='exitFree' target='exitUp'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
169      <arc id='LeaveIO.a5' source='exitUp' target='waitingToPay'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
174      <arc id='LeaveIO.a6' source='waitingToPay' target='out'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
179      <arc id='ParkingAreaIO.a1' source='in' target='occupied'>
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
184      <arc id='ParkingAreaIO.a2' source='occupied' target='out'>
        <inscription>

```

```

    <text>1</text>
  </inscription>
</arc>
189   <arc id='ParkingAreaIO.a3' source='out' target='free'>
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
194   <arc id='ParkingAreaIO.a4' source='free' target='in'>
    <inscription>
      <text>1</text>
    </inscription>
  </arc>
199   <transition id='in'>
    <inputEvents>
      <event idref='canEnterEvent' />
    </inputEvents>
    <name>
204     <text>gotTicketin</text>
    </name>
    <outputEvents>
      <event idref='enterWarningEvent' />
    </outputEvents>
209   <signalInputGuard>
    <text>(parkOpenedSignal == 1)</text>
  </signalInputGuard>
</transition>
  <transition id='out'>
214   <inputEvents>
    <event idref='canLeaveEvent' />
  </inputEvents>
  <name>
    <text>outpay</text>
219   </name>
  <signalInputGuard>
    <text>(parkOpenedSignal == 1)</text>
  </signalInputGuard>
</transition>
224 </net>
</pnml>

```

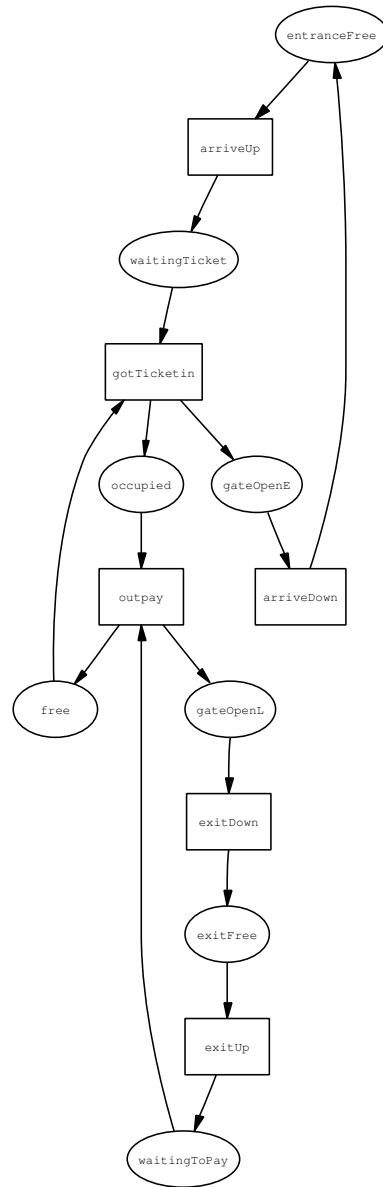



Figura D.3: Representação gráfica obtida automaticamente a partir do modelo na Listagem D.18 e correspondente ao modelo na Fig. C.3 da pág. 213. Como anotações, foram utilizados os atributos **name** dos nós, em lugar dos respectivos identificadores.

Apêndice E

Gramática Relax NG para Redes de Petri Input-Output

Apresenta-se seguidamente a gramática para redes de Petri *input-output* utilizando a linguagem Relax NG [s.a., 2003c].

Listagem E.1: Gramática para redes de Petri *input-output* em Relax NG.

```
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
12
  <a:documentation>
    Petri Net Type Definition for IO Place/Transition nets (based on basic PNML)
    RELAX NG implementation of IOptNetb.pntd
    version: 2.0
17    (c) 2005-09-19 Joao Paulo Barros
    Based on version 1.0 by Rui Pais and Joao Paulo Barros

    Based on:
    Petri Net Type Definition for Place/Transition nets (based on basic PNML)
22    RELAX NG implementation of ptNetb.pntd
    version: 1.0
    (c) 2001-2003
    Michael Weber, mweber@informatik.hu-berlin.de
  </a:documentation>
27
  <a:documentation>
    We include PNML with the correct URI for our Petri Net Type Definition.
  </a:documentation>
  <include href="http://www.informatik.hu-berlin.de/top/pnml/basicPNML.rng">
32  <define name="nettype.uri" combine="choice">
    <a:documentation>
      We define the net type URI declaring the namespace of this
      Petri net type definition.
    </a:documentation>
37  <value>http://www.informatik.hu-berlin.de/top/pntd/ptNetb</value>
```

```

    </define>
</include>

<a:documentation>
42    All labels of this Petri net type come from the Conventions Document.
</a:documentation>
<include href="http://www.informatik.hu-berlin.de/top/pnml/conv.rng" />

<define name="net.labels" combine="interleave">
47    <a:documentation>
        A P/T net may have a name.
    </a:documentation>
    <interleave>
        <optional<xref name="Name" /></optional>
52    <zeroOrMore<xref name="NetIOInput" /></zeroOrMore>
        <zeroOrMore<xref name="NetIOOutput" /></zeroOrMore>
    </interleave>
</define>

57    <define name="place.labels" combine="interleave">
    <a:documentation>
        A place of a P/T net may have a name and an initial marking.
    </a:documentation>
    <interleave>
62    <optional<xref name="PTMarking" /></optional>
        <optional<xref name="Name" /></optional>
        <optional<xref name="Capacity" /></optional>
        <optional<xref name="SignalOutputActions" /></optional>
    </interleave>
67    </define>

    <define name="transition.labels" combine="interleave">
    <a:documentation>
        A transition of a P/T net may have a name.
72    </a:documentation>
    <interleave>
        <optional<xref name="Name" /></optional>
        <optional<xref name="NetIOTransGuard" /></optional>
        <optional<xref name="NetIOTransInput" /></optional>
77    <optional<xref name="NetIOTransOutput" /></optional>
    </interleave>
    </define>

    <define name="arc.labels" combine="interleave">
82    <a:documentation>
        An arc of a P/T net may have an inscription.
    </a:documentation>
    <optional<xref name="PTArcInscription" /></optional>
    <optional<xref name="ArcType" /></optional>
87    </define>

<a:documentation>
    *****
    *** IO net labels ***
92    *****
</a:documentation>

<define name="NetIOInput">

```

```

    <a:documentation>
97      Input section in IO P/T-nets.
    </a:documentation>
    <element name="input">
      <interleave>
        <zeroOrMore>
102          <ref name="signalIn" />
        </zeroOrMore>
        <zeroOrMore>
          <ref name="eventIn" />
        </zeroOrMore>
107      </interleave>
    </element>
  </define>

  <define name="NetIOOutput">
112    <a:documentation>
      Output section in IO P/T-nets.
    </a:documentation>
    <element name="output">
      <interleave>
117        <zeroOrMore>
          <ref name="signalOut" />
        </zeroOrMore>
        <zeroOrMore>
          <ref name="eventOut" />
122        </zeroOrMore>
      </interleave>
    </element>
  </define>

127 <define name="event">
  <a:documentation>
    An event is an event id
  </a:documentation>
  <element name="event">
132    <ref name="signalEvent.content" />
  </element>
</define>

  <define name="signalEvent.content">
137    <attribute name="id"> <data type="ID" /> </attribute>
  </define>

  <define name="eventIn">
    <a:documentation>
142      Input event. Only specifies an event id.
    </a:documentation>
    <ref name="event" />
  </define>

147 <define name="eventOut">
  <a:documentation>
    Output event. Includes a default value.
  </a:documentation>
  <element name="event">
152    <ref name="signalEvent.content" />
    <attribute name="value"> <data type="nonNegativeInteger" /> </attribute>

```

```

    </element>
</define>

157 <define name="signal">
    <a:documentation>
        A signal is a sugnal id
    </a:documentation>
    <element name="signal">
162     <ref name="signalEvent.content" />
    </element>
</define>

<define name="signalIn">
167 <a:documentation>
    Input signal. Only specifies a signal id.
</a:documentation>
    <ref name="signal" />
</define>

172 <define name="signalOut">
    <a:documentation>
        Output signal
    </a:documentation>
177 <element name="signal">
    <ref name="signalEvent.content" />
    <attribute name="value"> <data type="nonNegativeInteger" /> </attribute>
    </element>
</define>

182 <a:documentation>
    *****
    *** IO places labels ***
    *****
187 </a:documentation>

<define name="Capacity" combine="interleave">
    <a:documentation>
        Place capacity given by a positive integer.
192 </a:documentation>
    <element name="capacity">
        <data type="positiveInteger"
            datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
        </data>
197 </element>
    </define>

<define name="SignalOutputActions" combine="interleave">
    <a:documentation>
202     If condition is true the output signal is assigned the value
        specified in the net output element
    </a:documentation>
    <element name="signalOutputActions">
        <oneOrMore>
207     <element name="signalOutputAction">
        <attribute name="idref"> <data type="IDREF" /> </attribute>
        <attribute name="value"> <data type="nonNegativeInteger" /> </attribute>
        <data type="string" />
    </element>

```

```

212         </oneOrMore>
           </element>
        </define>

        <a:documentation>
217  *****
        *** IO transitions labels ***
        *****
        </a:documentation>

222 <define name="NetIOTransGuard" combine="interleave">
        <a:documentation>
            Transition guard based on input signals
        </a:documentation>
        <element name="signalInputGuard">
227     <element name="text">
            <data type="string"
                datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
            </data>
        </element>
232 </element>
    </define>

    <define name="NetIOTransInput" combine="interleave">
        <a:documentation>
237     Transition input event
        </a:documentation>
        <element name="inputEvents">
            <ref name="transInputOutput.content" />
        </element>
242 </define>

    <define name="NetIOTransOutput" combine="interleave">
        <a:documentation>
            Transition output event
247 </a:documentation>
        <element name="outputEvents">
            <ref name="transInputOutput.content" />
        </element>
    </define>
252 <define name="transInputOutput.content" combine="interleave">
        <interleave>
            <zeroOrMore>
                <element name="event">
257     <attribute name="idref">
                <data type="IDREF" />
            </attribute>
        </element>
            </zeroOrMore>
262 </interleave>
    </define>

    <a:documentation>
        *****
267  *** IO arc labels ***
        *****
    </a:documentation>

```

```
<define name="ArcType" combine="interleave">
  <element name="type">
272    <ref name="simpletextlabel.content"/>
  </element>
</define>

</grammar>
```


Apêndice F

Protótipo para Tradução de Grupos de Sincronismo

Com vista a demonstrar e testar o funcionamento da tradução de grupos de sincronismo para transições simples, foi implementado um protótipo que opera tendo por base a ferramenta CPN Tools [s.a., 2004b]. Conforme descrito na Secção 6.1, o protótipo lê um ficheiro gerado pela CPN Tools e gera outro ficheiro, no mesmo formato mas contendo um modelo transformado .

Resumidamente, o protótipo implementa o tipo de transformação exemplificado na Fig. 5.1 da pág. 124, bem como na Secção 6.1: lê um modelo gerado com a ferramenta CPN Tools (um ficheiro `.cpn`) e gera um outro ficheiro `.cpn` que especifica o modelo transformado. O protótipo denomina-se `ccpn2hcpn` e está escrito na linguagem Ruby [Ruby Home Page, 2004], uma linguagem de *script* orientada pelos objectos.

A versão actual da ferramenta está disponível na Internet [Barros e Gomes, 2004b]. No final deste apêndice apresenta-se o código integral da mesma.

É importante notar que, do ponto de vista de um utilizador da CPN Tools, a abordagem seguida pelo protótipo não é a mais interessante. Com efeito, a CPN Tools baseia toda a sua interacção com o utilizador numa sofisticada interface gráfica pelo que a utilização concomitante de um protótipo com uma interface de linha do comando não é muito apelativa. No entanto, tal poderá ser útil enquanto não existir suporte para grupos de sincronismo na CPN Tools. Quando tal suceder, o presente protótipo será útil apenas para testar novos conceitos e técnicas com ele relacionados. Entretanto serve como prova de conceito e a sua apresentação no seio do grupo que desenvolveu e desenvolve a CPN Tools [Barros e Gomes, 2004a] reforçou a intenção desse mesmo grupo de incluir este tipo de mecanismo em futuras versões da CPN Tools.

Caso se pretendam futuros desenvolvimentos deste protótipo no sentido de o tornar mais útil, as seguintes duas funcionalidades devem ser prioritárias:

- O protótipo desenvolvido considera apenas grupos de sincronismo com uma transição emissora e uma transição receptora. Tal como proposto no Capítulo 5, será particularmente interessante adicionar suporte para grupos de sincronismo com várias transições pois tal irá permitir uma maior compacidade dos modelos, bem como, uma modelação mais próxima da utilizada nas linguagens de programação orientadas pelos objectos.
- Verificação sintáctica das declarações dos eventos.

Conforme já referido, a solução preferível, do ponto de vista dos utilizadores da CPN Tools, é a integração do suporte para grupos de sincronismo na CPN Tools. Tal evitaria a necessidade de gerar modelos transformados, necessariamente distintos do modelo construído pelo utilizador.

Termina-se esta secção com uma lista de funcionalidades que o autor julga deverem ser implementadas na CPN Tools com vista à obtenção de um bom suporte para a utilização de grupos de sincronismo:

- Cada transição deve providenciar um novo campo para especificação dos eventos relacionados com os pedidos.
- Eventos *SEND* sem os correspondentes pedidos *RECV* devem ser assinalados como erros de sintaxe.
- A interface gráfica da CPN Tools deve permitir a navegação entre as transições emissoras e as respectivas transições receptoras.
- Deve ser feita a verificação de tipos entre parâmetros de eventos.
- A ocorrência de parâmetros de entrada, nos arcos de entrada da transição associada, deve ser verificada. O mesmo deve suceder para as outras regras definidas no ponto 3c da Def. 5.7 na pág. 117.
- Os parâmetros *IN* nos eventos *SEND* e os parâmetros *OUT* nos eventos *RECV* devem poder vincular variáveis, tornando os lugares *check* desnecessários (*vide* Secção 6.2).
- Deve existir suporte sintáctico para a criação de variáveis do tipo *metaclass*, ou seja, conjuntos de cores cujas cores sejam nomes de páginas que sejam reconhecidas como tal. Esta funcionalidade permitirá uma melhor visibilidade para a invocação polimórfica descrita na Secção 6.1

A listagens seguintes apresentam todo o código fonte do protótipo desenvolvido.

Listagem F.1: Código iniciador do protótipo `ccpn2hcpn`.

```

require "cpnet"

4  if (ARGV.size != 2) then
    puts "-----"
    puts "Composable Coloured Petri Nets to Hierarchical Coloured Petri Nets"
    puts "ccpn2hcpn v. 0.1"
    puts "Implements the send/recv translation."
9   puts "Beja, 2004/07/19"
    puts "Joao Paulo Barros"
    puts "-----"
    puts "Usage: ruby ccpn2hcpn.rb input.cpn output.cpn\n"
    exit
14  else
    puts "\nReading file ...\n"
    $log = File.new("ccpn2hcpn.log", "w")
    $cpnet = CPNet.new(ARGV[0], ARGV[1])
    $cpnet.toCPN()
19  end

```

Listagem F.2: Classe `CPNet` (protótipo `ccpn2hcpn`).

```

require "rexml/document"
include REXML

require "expression"
5  require "arc"
require "place"
require "transition"
require "generators"
require "request"
10 require "fusionElement"

def log(s) $log.puts s end

class CPNet
15   attr_reader :sendTransitionsHash, :recvTransitionsHash
    attr_reader :placeIdPlaceHash, :fusionElements
    def initialize(cpnFileNameInput, cpnFileNameOutput)
      @doc = Document.new(File.new(cpnFileNameInput))
      @xml = @doc.root.elements['cpnet']
20   @cpnFileNameOutput = cpnFileNameOutput
      # pointers to transitions in XML file
      @sendTransitionsHash = Hash.new
      @recvTransitionsHash = Hash.new
      @placeIdPlaceHash = Hash.new
25   @pageNamePageIdHash = Hash.new
      @requests = Hash.new
      @fusionElements = Hash.new
    end

30   def toCPN()
      readCCPNFile
      generateRequests
      writeCPNFile
    end
35

```

```

def readCCPNFile
  #read pages
  @doc.root.elements.each("/workspaceElements/cpnet/page") do |pageXML|
    pageId = pageXML.attributes["id"]
    40    pageName = pageXML.elements["pageattr"].attributes["name"]
    @pageNamePageIdHash[pageName] = pageId
    log "Page #{pageName} has id #{@pageNamePageIdHash[pageName]}"
  end

  45  #read page elements
  @doc.root.elements.each("/workspaceElements/cpnet/page") do |pageXML|
    pageId = pageXML.attributes["id"]
    pageName = pageXML.elements["pageattr"].attributes["name"]
    readTransitions(pageXML, pageId, pageName)
    50    readPlaces(pageXML, pageName)
    readArcs(pageXML)
  end

  #read fusion elements
  readFusionElements()
  55  end

def readTransitions(pageXML, pageId, pageName)
  pageXML.elements.each("trans") do |transitionXML|
    transition = Transition.new(pageXML, pageId, pageName, transitionXML)
    60    if transition.exp
      if transition.exp.isSend? then
        log "Transition #{transition.name} in page " +
          "#{transition.pageId} is SEND, has id " +
          "#{transition.id}, and has exp: #{transition.exp.exp}"
        65    transition.removeTransitionExpression()
        @sendTransitionsHash[transition.id] = transition
      elsif transition.exp.isRecv? then
        log "Transition #{transition.name} in page " +
          "#{transition.pageId} is RECV, has id " +
          "#{transition.id}, and has exp: #{transition.exp.exp}"
        70    transition.removeTransitionExpression()
        @recvTransitionsHash[transition.id] = transition
      end
    end
  end
  75  end
end

def readPlaces(pageXML, pageName)
  pageXML.elements.each("place") do |placeXML|
    80    p = Place.new(placeXML, pageName)
    @placeIdPlaceHash[p.id] = p
  end
end

85  def readArcs(pageXML)
  pageXML.elements.each("arc") do |arcXML|
    transitionId = arcXML.elements["transend"].attributes["idref"]
    t = @sendTransitionsHash[transitionId]
    t = @recvTransitionsHash[transitionId] if (!t)
    90    placeId = arcXML.elements["placeend"].attributes["idref"]
    p = placeIdPlaceHash[placeId]
    # if connected to a send or recv transition
    # create an arc and add it to the transition

```

```

    if (t)
95         arc = Arc.new(arcXML, t, p)
            t.addArc(arc)
            arc.addPlaceToTransition()
        end
    end
100 end

def readFusionElements()
    @doc.root.elements.each("/workspaceElements/cpnet/fusion") do |fusionXML|
        @fusionElements[fusionXML.attributes['name']] =
105         FusionElement.createFromXML(fusionXML)
    end
end

def extractTargetClasses(classVariableName)
110     if @pageNamePageIdHash[classVariableName]
        # class exists
        return nil
    else
        notFound = true
115         @doc.root.elements.each("/workspaceElements/cpnet/globbox/var") do |var|
            id = var.elements['id']
            if id && (id.text == classVariableName)
                notFound = false
                return idIsDefined(var.elements['type'].elements['id'].text,
120                                     classVariableName)
            end
        end
        Error("#{classVariableName} is not defined as a variable.") if notFound
    end
125 end

def idIsDefined(type, variable)
    @doc.root.elements.each("/workspaceElements/cpnet/globbox/color") do |color|
        if color.elements['id'].text == type
130             enum = color.elements['enum']
            Error("Variable #{variable} is not an enum type.") if !enum
            targetClasses = Array.new
            enum.each_element do |id| targetClasses.push(id.text) end
            return targetClasses
135         end
    end
    Error("Variable #{variable} is not defined.")
    return nil
end
140

def generateRequests
    @recvTransitionsHash.each_value do |recvTransition|
        log "BEGIN in generateRequests @recvTransitionsHash.each_value"
        request = recvTransition.exp.request
145         reqKey = RequestKey.new(recvTransition.pageId, request)
        if @requests[reqKey.value]
            Error("Two requests '#{request}' in page #{recvTransition.pageName}")
        end
        @requests[reqKey.value] = Request.new(recvTransition)
150         log "END in generateRequests @recvTransitionsHash.each_value"
    end
end

```

```

@sendTransitionsHash.each_value do |sendTransition|
  targetClasses = sendTransition.exp.targetClasses
  st = false
155   requestName = ""
  pageName = ""
  targetClasses.each do |tc|
    pageName = tc
    requestName = sendTransition.exp.request
160    reqKey = RequestKey.new(@pageNamePageIdHash[pageName], requestName)
    sendRequest = @requests[reqKey.value]
    if sendRequest != nil then
      #checkRequest
      sendRequest.addSendTransition(sendTransition)
165      st = true
    end
  end
  if st == false
    Error("No RECV part was found for SEND request #{requestName} in "+
170      "transition #{sendTransition.name}, page #{pageName}.")
  end
end
end

175 def writeCPNFile
  # make modification to loaded .cpn file
  @requests.each_pair do |requestKey, request|
    #ignore RECV with no send
    next if request.sendTransitions.empty?()
180    recvTransition = request.recvTransition
    recvTransition.removeCheckPlacesAndArcs()
    recvTransition.makeEachPlaceAFusionPlace(@xml, @fusionElements)
    #for each send transition
    toDelete = Array.new
185    request.sendTransitions.each do |sendTransition|
      puts "\nProcessing request \"#{sendTransition.exp.request}\":\n" +
        "Page \"#{sendTransition.pageName}\", send transition" +
        "\"#{sendTransition.name}\" -> Page " +
        "\"#{recvTransition.pageName}\", receive transition " +
190        "\"#{recvTransition.name}\"...\n"

      #remove check places and respective arcs
      sendTransition.removeCheckPlacesAndArcs()
      sendTransition.makeEachPlaceAFusionPlace(@xml, @fusionElements)
195

      # Create new page for this request
      Page.new(@xml, sendTransition, recvTransition)
      toDelete.push(sendTransition)
    end
200    toDelete.each do |t| t.removeTransition() end
    recvTransition.removeTransition
  end

  # remove superfluous arcs
205  @recvTransitionsHash.each_value {|t| t.arcs.each {|a| a.removeFromXMLFile }}
  @sendTransitionsHash.each_value {|t| t.arcs.each {|a| a.removeFromXMLFile }}

  # write new .cpn file
  File.open(@cpnFileNameOutput, "w+").puts "#{@doc}"

```

```

210     end #def writeCPNFile
    end #class CPNet

```

Listagem F.3: Classe Page (protótipo ccpn2hcpn).

```

require "error"
require "generators"

4  module XML
    def createXMLDataCopy()
        return Document.new(@xml.to_s).root
    end
    def assignNewId()
9      @id = IdGenerator.new.value()
        @xml.attributes['id'] = @id
    end
    def addElement(element)
        @xml.add_element(element.xml)
14    end
    def deleteElement(elementName)
        @xml.delete_element(elementName)
    end
end
19
# Page with a transition for eac send/rcv pair
class Page
    include XML
    attr_reader :id, :pageName, :xml
24    def initialize(cpnetXML, sendTransition, rcvTransition)
        @parentXML = cpnetXML
        @pageName = sendTransition.exp.request() + "_" +
                    sendTransition.longName + "_" +
                    rcvTransition.longName
29        @sendTransition = sendTransition
        @rcvTransition = rcvTransition
        @id = IdGenerator.new.value()
        #add page to parent
        @xml = @parentXML.add_element('page', {'id' => @id})
34        @xml.add_element('pageattr', {'name' => @pageName})
        @transition = nil
        @newFusedPlaces = Hash.new
        createTransition()
        addFusedPlacesAndArcs()
39    end

    def createTransition()
        @transition = Transition.new(@xml, @id, @pageName,
                                     @sendTransition.createXMLDataCopy())
44        @transition.assignNewId()
        @transition.name = @sendTransition.exp.request()
        @transition.setInAndOutPlaces(@sendTransition, @rcvTransition)
        if @transition.exp && @transition.exp.isPolymorphic
            @transition.addGuard(@rcvTransition.pageName)
49        end
        #remove code segment. Must be after guard addition!
        @transition.deleteElement('code')
        #add it to the page
        addElement(@transition)

```

```

54      return @transition
      end

      def addFusedPlacesAndArcs()
        #substitute formal parameters by actual ones
59      @sendTransition.replaceParameters(@recvTransition)
        # add places fused with recv transition places
        @recvTransition.arcs.each { |arc|
          if (!arc.place.isCheckPlace) then
            addTransitionPlaceAndArc(arc, @recvTransition.pageName)
64          end
        }
        @sendTransition.arcs.each { |arc|
          if (!arc.place.isCheckPlace) then
            addTransitionPlaceAndArc(arc, @sendTransition.pageName)
69          end
        }
      end

      def addTransitionPlaceAndArc(arc, pageName)
74      newPlace = @newFusedPlaces[arc.place]
        if !newPlace then
          newPlace = arc.place.clone()
          newPlace.xml = arc.place.createXMLDataCopy
          newPlace.updatePosRelativeTo(@transition)
79          newPlace.assignNewId()
          newPlace.changeNameToCompleteName(pageName)
          @newFusedPlaces[arc.place] = newPlace
          #add it to the page
          addElement(newPlace)
84          #add the new place to the fusion element of the copied place
          newPlace.fusionElement.addPlaceId(newPlace.id)
        end
        #copy arc to subpage
        newArc = copyArcToNewFusedPlace(arc, newPlace)
89      end

      def copyArcToNewFusedPlace(oldArc, newFusedPlace)
        newArc = Arc.new(oldArc.createXMLDataCopy, @transition, newFusedPlace)
        newArc.assignNewId()
94      newArc.xml.elements['transend'].attributes['idref'] = @transition.id
        newArc.xml.elements['placeend'].attributes['idref'] = newFusedPlace.id
        newArc.setPositionForArcAnnotation()
        newArc.xml = addElement(newArc)
        return newArc
99      end
    end
  end
end

```

Listagem F.4: Classe Transition (protótipo ccpn2hcpn).

```

require "page"
require "rexml/document"
include REXML
4
class Transition
  include XML
  attr_reader :pageXML, :pageId, :pageName, :xml, :id, :name, :exp, :arcs
  attr_reader :inPlaces, :outPlaces

```



```

9      def initialize(pageXML, pageId, pageName, xml)
        @pageXML      = pageXML
        @pageId        = pageId
        @pageName      = pageName
        @xml           = xml
14      @exp           = getTransitionExpression(xml)
        @id            = xml.attributes["id"]
        @name          = xml.elements["text"].text
        @arcs          = Array.new
        @inPlaces      = Array.new
19      @outPlaces     = Array.new
        @posInput      = -1
        @posOutput     = -1
        @x             = xml.elements['posattr'].attributes['x'].to_f
        @y             = xml.elements['posattr'].attributes['y'].to_f
24      @h             = xml.elements['box'].attributes['h'].to_f
        @w             = xml.elements['box'].attributes['w'].to_f
        @parametersReplaced = false
      end

29      def longName() @pageName + "-" + @name end

      def name=(n) @name = @xml.elements["text"].text = n end

      def addArc(arc)
34      @arcs.push(arc)
        log "Arc #{arc.id} added to transition #{self}"
      end

      def addOutPlace(p) @outPlaces.push(p) if !p.isCheckPlace end
39      def addInPlace(p) @inPlaces.push(p) if !p.isCheckPlace end

      def setInAndOutPlaces(sendTransition, recvTransition)
        @inPlaces = sendTransition.inPlaces + recvTransition.inPlaces
        @outPlaces = sendTransition.outPlaces + recvTransition.outPlaces
44      end

      def hasInput?(p)
        @inPlaces.each{|ip| return true if p.id == ip.id}
        return false
49      end
      def hasOutput?(p)
        @outPlaces.each{|ip| return true if p.id == ip.id}
        return false
      end
54

      def getPosInputOffset()
        @posInput = @posInput + 1
        return @posInput - (nInputPlaces() / 2)
      end

59      def getPosOutputOffset()
        @posOutput = @posOutput + 1
        return @posOutput - (nOutputPlaces() / 2)
      end

      end

64      def getInputX() @x - 4.5 * @w end
      def getInputY() @y + getPosInputOffset() * 2 * @h end

```

```

def getOutputX() @x + 4.5 * @w end
def getOutputY() @y + getPosOutputOffset() * 2 * @h end

69
def nInputPlaces() return @inPlaces.length end
def nOutputPlaces() return @outPlaces.length end

def addGuard(className)
74 # used for polymorphic behaviour
    variable = exp.classVariableName()
    #className = sendTransition.exp.getTargetClass()
    #uses the code element as a basis for the generated xml
    codeXML = @xml.elements['code']
79 Error("No code element found in transition #{@name}.") if !codeXML
    condXML = @xml.add_element('cond', {'id' => IdGenerator.new.value()})
    codeXML.each_element { |e| condXML.add_element(e) }
    # uses an offset of 10 relative to the transition's position (could be better!)
    condXML.elements['posattr'].attributes['x'] =
84      (@xml.elements['posattr'].attributes['x'].to_i + 10).to_s
    condXML.elements['posattr'].attributes['y'] =
      (@xml.elements['posattr'].attributes['y'].to_i + 10).to_s
    condXML.elements['text'].text = "[#{variable}##{className}]"
end

89
#depois testar sem parametro e utilizando o @xml
def getTransitionExpression(xml)
    codeText = xml.elements["code/text"]
    return nil if codeText == nil
94 #get rid of newlines
    ct = codeText.to_s.gsub("\n", " ")
    m = Regexp.new('.*\(\*\*(.*)\*\*\).*').match(ct)
    return nil if m == nil
    return Expression.new(Regexp.last_match(1).to_s)
99 end

def removeCheckPlacesAndArcs()
    log "BEGIN Transition#removeCheckPlacesAndArcs for transition #{@name}"
    @arcs.each { |arc|
104     log "removeCheckPlacesAndArcs found arc #{arc.id}"
        if (arc.place.isCheckPlace) then
            log "removeCheckPlacesAndArcs " +
                "Remove check place #{arc.place.name} and arc #{arc.id} in page #{@pageName}"
            @pageXML.delete_element(arc.place.xml)
109             @pageXML.delete_element(arc.xml)
        end
    }
    log "END Transition#removeCheckPlacesAndArcs in page #{@pageName}"
end

114
def makeEachPlaceAFusionPlace(cpnetXML, fusionElements)
    @arcs.each { |arc|
        if (!arc.place.isCheckPlace) then
            arc.place.makeItAFusionPlace(cpnetXML, @pageName, fusionElements)
119         end
    }
end

def removeTransitionExpression()
124     log "Transition#removeTransitionExpression begin"

```



```

def makeSendRecvHash(sendExpression, recvExpression)
184   h = Hash.new
      for i in 1..sendExpression.nParameters
        h[sendExpression.parameterName(i)] = recvExpression.parameterName(i)
      end
      return h
189   end

end # class Transition

```

Listagem F.5: Classe Place (protótipo ccpn2hcpn).

```

require "page"
require "rexml/document"
4  include REXML

class Place
  include XML
  attr_reader :xml, :id, :name, :isCheckPlace, :fusionElement
9   attr_writer :xml

  def initialize(xml, pageName)
    @xml = xml
    @id = xml.attributes['id']
14   @name = xml.elements['text'].text
    Error("Place #{@id} in page #{pageName} has no name!") if not @name
    @isCheckPlace = @name[0..4] == 'check'
    @fusionElement = nil
  end
19

  def isFusionPlace?(cpnetXML, fusionElements)
    fusionInfo = @xml.elements['fusioninfo']
    if fusionInfo
      if @fusionElement == nil
24       #this place is a fusion place in the original net
        name = fusionInfo.attributes['name']
        @fusionElement = fusionElements[name]
        return (@fusionElement != nil)
      end
29     return true
    else
      return false
    end
  end
34

  def changeNameToCompleteName(pageName)
    @name = xml.elements['text'].text = pageName + "_" + @name
  end

39  def updatePosRelativeTo(transition)
    if transition.hasInput?(self)
      setPos(transition.getInputX(), transition.getInputY())
    elsif transition.hasOutput?(self)
      setPos(transition.getOutputX(), transition.getOutputY())
44   else
      Error 'Inconsistent state in Page#addRecvTransitionPlaceAndArc.'
    end
  end

```

```

end

49 def setPos(x, y)
    diffX = x - @xml.elements['posattr'].attributes['x'].to_f
    diffY = y - @xml.elements['posattr'].attributes['y'].to_f

    @xml.elements['posattr'].attributes['x'] = x.to_s
54 @xml.elements['posattr'].attributes['y'] = y.to_s
    updateXY('type', diffX, diffY)
    updateXY('fusioninfo', diffX, diffY)
    updateXY('initmark', diffX, diffY)
end

59 def updateXY(elem, diffX, diffY)
    p = @xml.elements[elem]
    if p then
        tmp = p.elements['posattr'].attributes['x'].to_f
64 tmp += diffX
        p.elements['posattr'].attributes['x'] = tmp.to_s
        tmp = p.elements['posattr'].attributes['y'].to_f
        tmp += diffY
        p.elements['posattr'].attributes['y'] = tmp.to_s
69 end
end

def makeItAFusionPlace(cpnetXML, pageName, fusionElements)
    #if already a fusion place then exit
74 return if isFusionPlace?(cpnetXML, fusionElements)
    log "PLACE #{@id} is going to be made into a fusion place"
    #create a fusion element
    @fusionElement = FusionElement.new(pageName + "_" + @name)
    @fusionElement.addToXML(cpnetXML)
79 log "Fusion element #{@fusionElement.fusionName} " +
    "created due to place #{pageName + "_" + @name}"
    #add fusionInfo element to the place
    # this makes it a fusion place
    addFusionInfo(IdGenerator.new.value())
84 #add the place to the respective fusion element
    @fusionElement.addPlaceId(@id)
end

def addFusionInfo(fusionInfoId)
89 x = @xml.elements['posattr'].attributes['x'].to_f - 7.0
    y = @xml.elements['posattr'].attributes['y'].to_f - 7.0
    fusionInfo = @xml.add_element('fusioninfo',
        {'id'=>fusionInfoId,
         'name'=>@fusionElement.fusionName})
94 fusionInfo.add_element('posattr', {'x'=>x.to_s(), 'y'=>y.to_s()})
    fusionInfo.add_element('fillattr',
        {'colour'=>'White', 'pattern'=>'Solid',
         'filled'=>'false'})
    fusionInfo.add_element('lineattr',
99 {'colour'=>'Black', 'thick'=>'0',
        'type'=>'Solid'})
    fusionInfo.add_element('textattr',
        {'colour'=>'Black', 'bold'=>'false'})

end
104 end # class Place

```

Listagem F.6: Classe FusionElement (protótipo ccpn2hcpn).

```

1  require "rexml/document"
   include REXML

   class FusionElement
     attr_reader :fusionName
6   def initialize(fusionName, id = nil, xml = nil)
       @id = id ? id : IdGenerator.new.value()
       @fusionName = fusionName
       @xml = xml
   end
11
   def addToXML(cpnetXML)
       @xml = cpnetXML.add_element('fusion', {'id'=>@id, 'name'=>@fusionName})
   end
16
   def FusionElement.createFromXML(fusionXML)
       return FusionElement.new(fusionXML.attributes['name'],
                                fusionXML.attributes['id'],
                                fusionXML)
   end
21
   def addPlaceId(placeId)
       @xml.add_element("fusion_elm", {'idref' => placeId})
   end
   end
end

```

Listagem F.7: Classe Expression (protótipo ccpn2hcpn).

```

# expressions have the form of the following example:
# req, SEND, classA, par1, IN par2, OUT par2
5
class Expression
  attr_reader :exp, :nParameters, :targetClasses, :isPolymorphic
  def initialize(exp)
    @exp = exp;
10   @parts = exp.split(',')
    @parts.each { |p| p.strip! } #remove spaces
    @nParameters = @parts.length - 3
    @targetClasses = nil
    @targetClasses = $cpnet.extractTargetClasses(@parts[1]) if isSend?()
15   @isPolymorphic = (@targetClasses != nil)
    @targetClasses = Array.new.push(@parts[1]) if not @isPolymorphic
  end
  def type() return @parts[0] end
  def isSend?() return type == "SEND" end
20  def isRecv?() return type == "RECV" end
  def classVariableName() return @parts[1] end
  def request() return @parts[2] end
  # first parameter has index 1
  def parameterName(i)
25   par = @parts[2+i]
    parParts = par.split
    if parParts.length == 1 then
      # no qualifier in parameter (constant parameter)
      return par
    end
  end
end

```

```

30     elsif parParts.length == 2 then # IN, OUT, or INOUT qualifier
        return parParts[1] # return parameter name
    else
        puts("Error in parameter syntax\n")
    end
35 end
# first parameter has index 1
def parameterType(i)
    par = @parts[2+i]
    parParts = par.split
40     if parParts.length == 1 then
        # no qualifier in parameter (constant parameter)
        return "CONST"
    elsif parParts.length == 2 then # IN, OUT, or INOUT qualifier
        return parParts[0] # return qualifier
45     else
        puts("Error in parameter syntax\n")
    end
end
end
end

```

Listagem F.8: Classe Request e classe RequestKey (protótipo ccpn2hcupn).

```

1  class Request
    #sendTransitions 1..N Transition
    #recvTransition 1 Transition
    attr_reader :sendTransitions, :recvTransition

6      def initialize(recvTransition)
        @recvTransition = recvTransition
        @sendTransitions = Array.new
    end
    def addSendTransition(sendTransition)
11        @sendTransitions.push(sendTransition)
    end
end

class RequestKey
16    attr_reader :value, :pageId, :requestName
    def initialize(pageId, requestName)
        @pageId = pageId
        @requestName = requestName
        # the following is used to use RequestKeys as Hash keys
21        @value = @pageId + @requestName
    end
end
end

```

Listagem F.9: Classe IdGenerator (protótipo ccpn2hcupn).

```

2  class IdGenerator
    @@idGeneratorInit = 99999
    def value()
        @@idGeneratorInit += 1
        return "ID" + @@idGeneratorInit.to_s()
7      end
end
end

```

Listagem F.10: Método `Error` (protótipo `ccpn2hcpn`).

```
def Error(errorString)
    puts "\nError! " + errorString
    exit
4 end
```


Bibliografia

- Abdourahaman, Fischer, Erik, Gruenewald, Alexander, Hauptmann, Jens, Hohberg, Bodo, Jünger, Matthias, Kindler, Ekkart, Klein, Raimund, Oschmann, Frank, Schulz, Reiner, Schwenzer, Ines, e Weber, Michael. 2002. Petri Net Kernel. <http://www.informatik.hu-berlin.de/top/pnk/>, 2002. A lista de autores está por ordem alfabética tal como no sítio da ferramenta. (Citado nas págs. 52, 209 e 210)
- Agha, Gul, de Cindio, Fiorella, e Rozenberg, Grzegorz, editores. 2001. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer. (Citado nas págs. 23, 42, 111 e 281)
- Aksit, Mehmet, Barry, Brian, Elrad, Tzilla, Filman, Bob, Kiczales, Gregor, Lieberherr, Karl, Mezini, Mira, Murphy, Gail, e Ossher, Harold. 2005. aspect-oriented software development. <http://www.acpn.de/>, 2005. O *site* é editado pelos membros do *Steering Committee* da *Aspect-Oriented Software Association* que por essa razão constam como autores desta referência. O *site* foi consultado em 20 de Janeiro de 2005. (Citado na pág. 66)
- Arbib, Michael A., Kfoury, A. J., e Moll, Robert N. 1981. *A Basis for Theoretical Computer Science*. Springer-Verlag. (Citado na pág. xxxi)
- Balzarotti, C., de Cindio, F., e Pomello, L. 1999. Observation Equivalences for the Semantics of Inheritance. In *Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, págs. 67–82. Kluwer Academic Publishers, 1999. (Citado na pág. 139)
- Barbara H. Liskov e Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841. (Citado na pág. 140)
- Baresi, Luciano e Pezzè, Mauro. 2001. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, capítulo On Formalizing UML with High-Level Petri Nets, págs. 276–304. Volume 2001 of , Agha et al. [2001]. (Citado nas págs. 8 e 22)
- Barros, João Paulo. 1996. CpPNeTS: uma Classe de Redes de Petri de Alto-nível – Implementação de um Sistema de Suporte à sua Aplicação e Análise. Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática, 1996. (Citado na pág. 207)
- Barros, João Paulo e Gomes, Luis. 2003a. Actions as Activities and Activities as Petri Nets. In Jürjens, Jan, Rumpe, Bernhard, France, Robert, e Fernandez, Eduardo B., editores, *Workshop on Critical Systems Development with UML - Proceedings of the UML'03 workshop*, págs. 129–135. Institut für Informatik der Technische Universität München, 2003a. TUM-INFO-09-I0323-89/1.-FI. (Citado na pág. 6)
- Barros, João Paulo e Gomes, Luís. 2003b. Modifying Petri Net Models by Means of Crosscutting Ope-

- rations. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD 2003)*. IEEE Computer Society Press, 2003b. (Citado na pág. 13)
- Barros, João Paulo e Gomes, Luís. 2003c. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In Akkawi, Faisal, Aldawud, Omar, Booch, Grady, Clarke, Siobhán, Gray, Jeff, Harrison, Bill, Kandé, Mohamed, Stein, Dominik, Tarr, Peri, e Zakaria, Aida, editores, *The 4th AOSD Modeling With UML Workshop*, pág. 8, 2003c. (Citado na pág. 185)
- Barros, João Paulo e Gomes, Luís. 2004a. A Unidirectional Transition Fusion for Coloured Petri Nets and its Implementation for the CPNTools. In Jensen, Kurt, editor, *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools*, DAIMI PB - 570, págs. 133–150, Aarhus, Dinamarca. Department of Computer Science, University of Aarhus. ISSN 0105-8517, disponível em <http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/CPN04proceedings.pdf>. (Citado nas págs. 14, 146 e 265)
- Barros, João Paulo e Gomes, Luís. 2004b. Composable CPNs Homepage. <http://www.uninova.pt/gres/ccpn>, 2004b. (Citado nas págs. 150 e 265)
- Barros, João Paulo e Gomes, Luís. 2004c. Net Model Composition and Modification by Net Operations: a Pragmatic Approach. In *Proceedings of the 2th IEEE International Conference on Industrial Informatics (INDIN 2004)*, 2004c. (Citado nas págs. 13, 31, 32 e 76)
- Barros, João Paulo e Gomes, Luís. 2004d. On the Use of Coloured Petri Nets for Object-Oriented Design. In Cortadella, Jordi e Reisig, Wolfgang, editores, *Applications and Theory of Petri Nets 2004 25th International Conference (ICATPN 2004)*, Bolonha, Itália, June 21-25, 2004, volume 3099 of *Proceedings Series: Lecture Notes in Computer Science*, págs. 117–136. Springer. ISBN: 3-540-22236-7. (Citado nas págs. 14 e 35)
- Barros, João Paulo e Gomes, Luís. 2004e. Operational PNML. <http://www.uninova.pt/gres/opnml>, 2004e. (Citado nas págs. 13, 93, 106 e 215)
- Barros, João Paulo e Gomes, Luís. 2004f. Operational PNML: Towards a PNML Support for Model Construction and Modification. In *Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets; Workshop satélite da International Conference on Application and Theory of Petri Nets 2004*, 2004f. (Citado nas págs. 13 e 93)
- Barros, João Paulo, Gomes, Luís, e Adolfo Steiger-Garção. 1997. Implementation of a Non-Autonomous High-Level Petri Net Model for Reactive Real-time Systems. In *Proceedings of the 4th IFAC Workshop on Algorithms and Architectures for Real-Time Control*, 1997. (Citado na pág. 207)
- Barros, João Paulo, Gomes, Luís, Pais, Rui, e Dias, Rui. 2004. From Petri Nets to Executable Systems: an Environment for Code Generation and Analysis. In *1st International Conference on Informatics in Control, Automation and Robotics (ICINCO'2004)*, Setúbal, Portugal. (Citado na pág. 16)
- Barros, João Paulo e Jørgensen, Jens Bæk. 2005a. A Case Study on Coloured Petri Nets in Object-Oriented Analysis and Design. *Nordic Journal of Computing*, 12(3):229–250. (Citado nas págs. 14, 169 e 170)
- Barros, João Paulo e Jørgensen, Jens Bæk. 2005b. Model Transformations for an Elevator Controller: Coloured Petri Nets in Object-Oriented Analysis and Design. In *Second International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005)*, Renes, França. (Citado nas págs. 14, 169 e 170)
- Barros, João Paulo. 2002. Specific Proposals for the Use of Petri Nets in a Concurrent Programming Course. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology*

- in computer science education*, págs. 165–167, New York, NY, USA. ACM Press. (Citado na pág. 8)
- Barros, João Paulo e Gomes, Luís. 2002. Activities as Behaviour Aspects. In Kandé, Mohamed, Aldawud, Omar, Booch, Grady, e Harrison, Bill, editores, *Workshop on Aspect-Oriented Modeling with UML*, 2002. Workshop satélite da «UML» 2002 - The Unified Modeling Language 5th International Conference(UML'2002). (Citado na pág. 185)
- Basten, Twan. 1998. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, Holanda. (Citado na pág. 27)
- Battiston, E., F. de Cindio, e Mauri, G. 1988. OBJSA Nets: a Class of High-level Nets Having Objects as Domains. *Lecture Notes in Computer Science: Advances in Petri Nets 1988*, 340:20–43. (Citado na pág. 27)
- Bergin, Joseph. 1997. What IS Object-Oriented Programming—Really? <http://csis.pace.edu/~bergin/papers/oop.html>, 1997. Consultado em 13 de Setembro de 2005. (Citado na pág. 139)
- Bernardi, Simona, Donatelli, Susanna, e Merseguer, José. 2002. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In *Proceedings of the 3rd international workshop on Software and performance*, págs. 35–45. ACM Press, 2002. (Citado nas págs. 8 e 22)
- Bernardinello, Luca e Fiorella De Cindio. 1992. A Survey of Basic Net Models and Modular Net Classes. *Lecture Notes in Computer Science; Advances in Petri Nets 1992*, 609:304–351. (Citado na pág. 23)
- Berthelot, Gérard e Petrucci, Laure. 2001. Specification and Validation of a Concurrent System: an Educational Project. *Journal of Software Tools for Technology Transfer*, 3(4):372–381. (Citado na pág. 8)
- Best, Eike, Devillers, Raymond, e Koutny, Maciej. 2001. *Petri Net Algebra*. Monographs in Theoretical Computer Science. An EATCS Series. Springer. (Citado na pág. 26)
- Biberstein, O., Buchs, D., e Guelfi, N. 2001. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 2001:73–130. (Citado na pág. 111)
- Billington, Jonathan, Christensen, Søren, van Hee, Kees, Kindler, Ekkart, Kummer, Olaf, Petrucci, Laure, Post, Reinier, Stehno, Christian, e Weber, Michael. 2003. The Petri Net Markup Language: Concepts, Technology, and Tools. In van der Aalst, W. e Best, E., editores, *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, volume 2679 of *LNCS*, págs. 483–505, Eindhoven, Holland. Springer-Verlag. (Citado nas págs. 51, 93 e 210)
- Bowen, Jonathan. 2003. Formal Methods Publications. <http://www.afm.sbu.ac.uk/pubs/>, 2003. (Citado na pág. 4)
- Brauer, Wilfried, Gold, Robert, e Vogler, Walter. 1991. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. *Lecture Notes in Computer Science; Advances in Petri Nets 1990*, 483:1–46. (Citado na pág. 23)
- Brooks, Frederick P. Jr. 1995. *The Mythical Man-Month Essays on Software Engineering Anniversary Edition*. Addison-Wesley. (Citado nas págs. 300 e 301)
- Buchholz, Peter. 1994. Hierarchical High Level Petri Nets for Complex System Analysis. In Valette, R., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, págs. 119–138. Springer-Verlag, 1994. (Citado na pág. 184)

- Buchs, Didier e Guelfi, Nicolas. 1991. CO-OPN: a Concurrent Object Oriented Petri Net Approach. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, 1991, Gjern, Denmark*, págs. 432–454, 1991. NewsletterInfo: 39. (Citado na pág. 111)
- Chen, P. P. 1976. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(?):9–36. (Citado na pág. 170)
- Chen, Y., W. T. Tsai, e Chao, D. 1993. Dependency Analysis-A Petri-Net-Based Technique for Synthesizing Large Concurrent Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4 (4):414–426. (Citado na pág. 27)
- Christensen, Søren e Hansen, N. D. 1992. Coloured Petri Nets Extended with Channels for Synchronous Communication. *Daimi PB-390*. Also in: Valette, R.: *Lecture Notes in Computer Science*, Vol. 815; *Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, pages 159–178. Springer-Verlag, 1994. Abridged version; available at <http://www.daimi.au.dk/CPnets/publ/full-papers/ChrHan1994.pdf>. (Citado nas págs. 23, 28, 33, 43, 111, 122 e 125)
- Christensen, Søren e Jørgensen, Jens Bæk. 2004. Teaching Coloured Petri Nets: Examples of Courses and Lessons Learned. In Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science: Advances in Petri Nets*, págs. 402–412. Springer-Verlag, Berlin Heidelberg. (Citado na pág. 8)
- Christensen, Søren e Petrucci, Laure. 1992. Towards a Modular Analysis of Coloured Petri Nets. In Jensen, K., editor, *Lecture Notes in Computer Science; 13th International Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, volume 616, págs. 113–133. Springer-Verlag, 1992. (Citado nas págs. 23, 117 e 122)
- Christensen, Søren e Petrucci, Laure. 1995. Modular State Space Analysis of Coloured Petri Nets. In *Proceeding of the 16th International Conference on Application and Theory of Petri Nets, Turin.*, págs. 201–217, 1995. available at <http://www.daimi.aau.dk/CPnets/publ/full-papers/ChrPet1995.pdf>. (Citado na pág. 28)
- Christensen, Søren e Petrucci, Laure. 2000. Modular analysis of Petri nets. *Computer Journal*, 43(3): 224–242. (Citado na pág. 28)
- Clocksin, W. F. e Mellish, C. S. 1994. *Programming in Prolog*. Springer-Verlag, 4 edition. (Citado na pág. 35)
- Czarnecki, Krzysztof e Eisenecker, Ulrich. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. (Citado nas págs. xxii e 66)
- David, René. 1991. Modeling of Dynamic Systems by Petri Nets. In *European Control Conference*, págs. 136–147, Grenoble, France. (Citado nas págs. 17 e 24)
- David, René e Alla, Hassane. 1992. *Petri Nets & Grafcet; Tools for Modelling Discrete Event Systems*. Prentice Hall International (UK) Ltd. (Citado nas págs. 17, 24, 200, 201 e 204)
- DeMarco, Tom. 2002. *Structured Analysis: Beginnings of a New Discipline*, págs. 520–27. (Citado na pág. 299)
- Desel, Jörg e Reisig, Wolfgang. 1998. Place/transition Petri nets. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:122–173. (Citado na pág. 199)
- Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores. 2004. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg. (Citado na pág. 22)

- Desmond Francis D'Souza e Alan Cameron Wills. 1998. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison Wesley Longman. (Citado na pág. 118)
- Dijkstra, Edsger W. 1971. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138. Também publicado em [Dijkstra, 2002]. (Citado na pág. 77)
- Dijkstra, Edsger W. 1974. On the Role of Scientific Thought, 1974. publicado como [Dijkstra, 1982] e em <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>. (Citado na pág. 67)
- Dijkstra, Edsger W. 1982. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, págs. 60–66. Springer-Verlag. (Citado na pág. 285)
- Dijkstra, Edsger W. 1993. On the Economy of doing Mathematics. In *Proceedings of the Second International Conference on Mathematics of Program Construction*, volume 669, págs. 2–10, London, UK. Springer-Verlag. (Citado na pág. 1)
- Dijkstra, Edsger W. 2002. *The Origin of Concurrent Programming: from Semaphores to Remote Procedure Calls*, capítulo Hierarchical Ordering of Sequential Processes, págs. 198–227. Springer-Verlag New York, Inc. (Citado na pág. 285)
- Doldi, Laurent. 2003. *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*. Wiley. (Citado na pág. 1)
- Eccles, Peter J. 1997. *An Introduction to Mathematical Reasoning numbers, sets and functions*. Cambridge University Press. (Citado na pág. xxxi)
- Ehrig, Hartmut, Juhás, Gabriel, Padberg, Julia, e Rozenberg, Grzegorz, editores. 2001. *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*. Springer. (Citado nas págs. 23, 110 e 288)
- Ehrig, Hartmut e Padberg, Julia. 2004. Graph Grammars and Petri Net Transformations. In Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science: Advances in Petri Nets*, págs. 496–536. Springer-Verlag, Berlin Heidelberg. Baseado na apresentação feita no 4th Advanced Course on Petri Nets realizado em Setembro de 2004 em Eichstätt na Alemanha. (Citado na pág. 11)
- Ehrig, Hartmut, Reisig, Wolfgang, Rozenberg, Grzegorz, e Weber, H., editores. 2003. *Petri Net Technology for Communication-Based Systems : Advances in Petri Nets*, volume 2472 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg. Disponível em <http://www.springerlink.com/link.asp?id=p7a1heq6a3u2>. (Citado na pág. 22)
- Elkoutbi, M. e Keller, R. K. 2000. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *Proceedings of 21st Petri Nets Conference*, volume 1825 of *LNCS*, págs. 166–186, Aarhus, Denmark. Springer. (Citado na pág. 164)
- Ellson, John, Gansner, Emden, Koren, Yehuda, Koutsofios, Eleftherios, Mocenigo, John, North, Stephen, Woodhull, Gordon, Dobkin, David, Alexiev, Vladimir, Lilly, Bruce, Scheerder, Jeroen, G., Daniel Richard, e Low), Glen. 2004. Graphviz - Graph Visualization Software. <http://www.graphviz.org>, 2004. (Citado na pág. 52)
- Elrad, Tzilla, Filman, Robert E., e Bader, Atef. 2001. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, 44(10):29–32. Este número inclui uma secção especial sobre programação orientada pelos aspectos. (Citado na pág. 66)
- Eric Y. T. Juan, Jeffrey J. P. Tsai, e Murata, Tadao. 1996. A New Compositional Method for Condensed State-Space Verification. In *Procs. of the 1st IEEE High-Assurance Systems Engineering Workshop, Ontario, Canada, Oct. 22, 1996*, págs. 104–111. IEEE Computer Society Press, 1996. (Citado na pág. 27)

- Eric Y. T. Juan, Jeffrey J. P. Tsai, e Murata, Tadao. 1998. Compositional Verification of Concurrent Systems Using Petri Net Based Condensation Rules. *ACM Trans. on Programming Languages and Systems*, 20(5):917–979. (Citado na pág. 27)
- Filman, Robert E., Elrad, Tzila, Clarke, Siobhán, e Akşit, Mehmet, editores. 2005. *Aspect-Oriented Software Development*. Addison-Wesley. (Citado na pág. 66)
- Fischer, Clemens e Wehrheim, Heike. 2000. Behavioural Subtyping Relations for Object-Oriented Formalisms. In *AMAST 2000: Algebraic Methodology and Software Technology, LNCS 1816*. Springer, 2000. (Citado na pág. 139)
- Fowler, Martin. 2004. UmlSketchingTools. <http://martinfowler.com/bliki/UmlSketchingTools.html>, 2004. 16 de Junho de 2004. (Citado na pág. 2)
- Gamma, E., Helm, R., Johnson, R., e Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (Citado nas págs. 137, 141, 142, 152 e 301)
- Genest, Blaise, Muscholl, Anca, e Peled, Doron. 2004. Message Sequence Charts. In Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science: Advances in Petri Nets*, págs. 537–558. Springer-Verlag, Berlin Heidelberg. Based on the lectures given at the 4th Advanced Course on Petri Nets held in Eichstätt, Germany in September 2004. (Citado nas págs. 2 e 163)
- Génova, Gonzalo, Llorens, Juan, e Palacios, Vicente. 2003. Sending Messages in UML. *Journal of Object Technology*, 2(1):99–115. (Citado na pág. 179)
- Genrich, Hartmann J. 1987. Predicate/transition nets. In Brauer, W., Reisig, W., e Rozenberg, G., editores, *Lecture Notes in Computer Science: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 254, págs. 207–247. Springer-Verlag, 1987. (Citado na pág. 30)
- Genrich, Hartmann J. e Lautenbach, Kurt. 1981. System Modelling with High-Level Petri Nets. *Theoretical Computer Science*, 13:109–136. (Citado na pág. 23)
- Gerhart, Susan L. 1990. Applications of Formal Methods: Developing Virtuoso Software. *IEEE Softw.*, 7(5):6–10. Introdução pelo editor convidado. (Citado na pág. 6)
- Girault, Claude e Valk, Rüdiger. 2003. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag. (Citado nas págs. 8, 17, 22, 24 e 293)
- Gomes, Luís. 1997. *Redes de Petri Reactivas e Hierárquicas - Integração de formalismos no projecto de sistemas reactivos de tempo-real*. PhD thesis, Universidade Nova de Lisboa. (Citado nas págs. 29, 31, 32, 38, 39, 200 e 201)
- Gomes, Luís. 2005. On Conflict Resolution in Petri Nets Models Through Model Structuring and Composition. In *Proceedings of the 3rd IEEE International Conference on Industrial Informatics (INDIN 2005)*, 2005. (Citado na pág. 86)
- Gomes, Luís e Barros, João Paulo. 2003. On Structuring Mechanisms for Petri Nets Based System Design. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, págs. 431–438. IEEE Catalog Number: 03TH8696, 2003. (Citado nas págs. 13, 31, 32, 38, 39 e 72)
- Gomes, Luís e Barros, João Paulo. 2005. Structuring and Composability Issues in Petri Nets Modeling. *IEEE Transactions on Industrial Informatics*, 1(2):112–123. (Citado na pág. 12)
- Gomes, Luís, Barros, João Paulo, e Costa, Anikó. 2002. Petri Net Model Node Structuring Techniques for Embedded System Design. In *Proceedings of the 5th Portuguese Conference on Automatic*

- Control (CONTROLO'2002)*, Aveiro, Portugal. Associação Portuguesa de Controlo Automatico (APCA). (Citado nas págs. 13, 29, 31, 32, 38 e 39)
- Gomes, Luís, Barros, João Paulo, e Costa, Anikó. 2005. Modeling Formalisms for Embedded Systems Design. In Zurawski, Richard, editor, *Embedded Systems Handbook*. CRC. (Citado na pág. 2)
- Gomes, Luís, Barros, João Paulo, e Pais, Rui. 2004. From Non-Autonomous Petri net Models to Code in Embedded Systems Design. In *Second International Workshop on Discrete-Event System Design (DESDes'04)*, Zielona Gora, Polónia. (Citado na pág. 16)
- Gomes, Luís e Costa, Anikó. 2004. Concurrent Systems' Hardware Design using Petri nets. In *Proceedings of the 5th European Workshop on Microelectronics Education (EWME 2004)*, Lausanne, Suíça). (Citado na pág. 8)
- Gomes, Luis e Steiger-Garção, Adolfo. 1995. Programmable Controller Design Based on a Synchronized Colored Petri Net Model and Integrating Fuzzy reasoning. In Michelis, G., De e Diaz, M., editores, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1995, 16th International Conference*, volume 935, págs. 218–237. Springer, 1995. (Citado nas págs. 200, 201 e 207)
- Gomes, Luís e Steiger-Garção, Adolfo. 1996. Towards the Implementation of Conflict Resolution on a Non Autonomous High-Level Petri Net Model. In *Workshop Manufacturing and Petri Nets, Osaka, Japan, 1996*. Satellite workshop of the 17th International Conference on Application and Theory of Petri Nets (ICATPN'96). (Citado na pág. 86)
- Gries, David. 1981. *The Science of Programming*. Springer-Verlag. (Citado nas págs. 184 e 301)
- Hall, Anthony, Dill, David L., Rushby, John, Holloway, C. Michael, Butler, Ricky W., e Zave, Pamela. 1996. Industrial Practice. *Computer*, 29(4):22–27. (Citado nas págs. 8, 9 e 300)
- Harel, David. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274. (Citado nas págs. 2, 109 e 163)
- Harel, David. 1988. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530. (Citado nas págs. 2 e 163)
- Harel, David e Kupferman, Orna. 2002. On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 28(9):889–903. (Citado na pág. 139)
- Harel, David e Naamad, Amnon. 1996. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333. (Citado na pág. 2)
- Harel, David e Rumpe, Bernhard. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72. (Citado na pág. 2)
- He, Xudong. 1996. A Formal Definition of Hierarchical Predicate Transition Nets. In *Lecture Notes in Computer Science; Proc. 17th International Conference in Application and Theory of Petri Nets (ICATPN'96)*, Osaka, Japan, volume 1091, págs. 212–229. Springer-Verlag, 1996. (Citado nas págs. 30 e 31)
- He, Xudong. 2004. Página pessoal na internet. <http://www.cs.fiu.edu/faculty/hex/>, 2004. (Citado na pág. 31)
- He, Xudong e Lee, John A. N. 1991. A Methodology for Constructing Predicate Transition Net Specifications. *Software-Practice and Experience*, 21(8):845–875. (Citado nas págs. 36 e 38)
- Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Inc. (Citado nas págs. 26 e 55)
- Holloway, L. E., Krogh, B. H., e Giua, A. 1997. A Survey of Petri Net Methods for Controlled Discrete Event Systems. *Discrete Event Dynamic Systems*, 7:151–190. (Citado na pág. 200)

- Huber, P., Jensen, K., e Shapiro, R. M. 1989. Hierarchies in Coloured Petri Nets. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989, Bonn, Germany*, págs. 192–209, 1989. (Citado nas págs. 23, 27, 28, 38, 39 e 41)
- Jackson, Michael. 1995. *Software Requirements & Specifications*. Addison-Wesley. (Citado nas págs. 64 e 300)
- Jackson, Michael. 2002. Some Basic Tenets of Description. *Software and Systems Modeling*, 1(1):5–9. (Citado nas págs. 3, 14 e 170)
- Jeffries, Ronald. 1976. Carta,. ACM SIGPLAN Notices, 1976. (Citado na pág. 3)
- Jens Bæk Jørgensen. 2004. CPN Models as Enhancement to a Traditional Software Specification for an Elevator Controller. In Moldt, Daniel, editor, *Proceedings of the Third Workshop on Modelling of Objects, Components, and Agents*, DAIMI PB - 571, págs. 99–116, Aarhus, Dinamarca. Department of Computer Science, University of Aarhus. (Citado nas págs. 14, 162 e 169)
- Jensen, Kurt. 1981. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14: 317–336. (Citado na pág. 23)
- Jensen, Kurt. 1996. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9:7–40. InternalNote: Submitted by: kjensen@daimi.aau.dk. (Citado na pág. 30)
- Jensen, Kurt. 1997a. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 1 Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 2nd corrected printing edition. (Citado nas págs. 17, 21, 27, 30, 37, 72, 81, 110, 113, 115, 120 e 145)
- Jensen, Kurt. 1997b. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 2 Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 2nd corrected printing edition. (Citado nas págs. 17, 21, 27, 30, 37, 72, 110, 113, 115, 120 e 145)
- Jensen, Kurt. 1997c. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 3 Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany. (Citado nas págs. 8, 17, 21, 27, 30, 37, 72, 110, 113, 115, 120 e 145)
- Jensen, Kurt. s.d. Overview of Design/CPN, s.d. (Citado na pág. 9)
- Jensen, Kurt e Rozenberg, Grzegorz. 1991. *High-level Petri Nets: Theory and Application*. Monographs in Theoretical Computer Science. Springer-Verlag. (Citado nas págs. 29 e 30)
- Jörg Desel, Gabriel Juhás. 2001. What is a Petri Net. In Ehrig et al. [2001], págs. 1–25. (Citado nas págs. 23 e 110)
- Jørgensen, Jens Bæk e Bossen, Claus. 2004. Executable Use Cases: Requirements for a Pervasive Health Care System. *IEEE Software*, 21(2):34–41. (Citado na pág. 169)
- Jørgensen, Jens Bæk e Christensen, Søren. 2002. Executable Design Models for a Pervasive Healthcare Middleware System. In *UML 2002: Proceedings of the 5th International Conference on The Unified Modeling Language*, volume 2460 of LNCS, págs. 140–149, London, UK. Springer-Verlag. (Citado nas págs. 9 e 164)
- Jünger, M., Kindler, E., e Weber, M. 2000. The Petri Net Markup Language. In Phillipi, S., editor, *Workshop Algorithmen und Werkzeuge für Petrinetze*, 2000. (Citado nas págs. 51, 93 e 209)
- Kernighan, Brian W. e Ritchie, Dennis. 1988. *The C Programming Language*. Prentice Hall. (Citado na pág. 182)

- Kiczales, Gregor, Lamping, John, Mendhekar, Anurag, Maeda, Chris, Lopes, Cristina, Loingtier, Jean-Marc, e Irwin, John. 1997. Aspect-oriented programming. In Akşit, Mehmet e Matsuoka, Satoshi, editores, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, págs. 220–242, Berlin, Heidelberg, and New York. Springer Verlag. (Citado na pág. 66)
- Kindler, Ekkart e Weber, Michael. 2001. A Universal Module Concept for Petri nets. In *Proceedings des 8. Workshops Algorithmen und Werkzeuge für Petrinetze / Gabriel Juhas und Robert Lorenz (Hrsg.) – Katholische Universität Eichstätt, 2001*, págs. 7–12, 2001. (Citado nas págs. 51 e 95)
- Köhler, Michael, Moldt, Daniel, e HeikoRölke. 2001. Modelling the Structure and Behaviour of Petri Net Agents. In Colom, J.M. e Koutny, M., editores, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, págs. 224–241. Springer-Verlag, 2001. (Citado na pág. 43)
- Köster, Frank, Schöf, Stefan, Sonnenschein, Michael, e Wieting, Ralf. 2001. Modeling of a Library with THORNs. In Agha, G.A., Cindio, F., De, e Rozenberg, G., editores, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science: Advances in Petri nets*, págs. 406–427. Springer-Verlag, Berlin, Heidelberg, Germany. (Citado na pág. 111)
- Kristensen, Lars Michael, Jørgensen, Jens Bæk, e Jensen, Kurt. 2004. Application of Coloured Petri Nets in System Development. In Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science: Advances in Petri Nets*, págs. 626–685. Springer-Verlag, Berlin Heidelberg. Based on the lectures given at the 4th Advanced Course on Petri Nets held in Eichstätt, Germany in September 2004. (Citado na pág. 17)
- Kummer, Olaf. 2002. Referenznetze. Logos Verlag, 2002. (Citado nas págs. 31, 35, 42 e 111)
- Kummer, Olaf, Wienberg, Frank, e Duvigneau, Michael. 2004a. RENEW – the reference net workshop. <http://www.renew.de/>, 2004a. (Citado nas págs. 24, 31, 35, 43, 52, 111, 112, 122 e 207)
- Kummer, Olaf, Wienberg, Frank, Duvigneau, Michael, Schumacher, Jörn, Köhler, Michael, Moldt, Daniel, Rölke, Heiko, e Valk, Rüdiger. 2004b. An Extensible Editor and Simulation Engine for Petri Nets: RENEW. In Cortadella, Jordi e Reisig, Wolfgang, editores, *Applications and Theory of Petri Nets 2004 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004*, volume 3099 of *Proceedings Series: Lecture Notes in Computer Science*, págs. 484–493. Springer. ISBN: 3-540-22236-7. (Citado nas págs. 31, 35, 43, 52, 111, 112 e 122)
- Lakin, Fred. 1998. comp.lang.visual Frequently-Asked Questions (FAQ). <http://www.faqs.org/faqs/visual-lang/faq/index.html>, 1998. Referido na resposta à pergunta 12: "Q12: What is the Deutsch Limit?". (Citado nas págs. 23 e 300)
- Lakos, Charles A. 1995. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets, Turin*, págs. 278–297, 1995. (Citado nas págs. 111 e 112)
- Lakos, Charles A. 1997. On the Abstraction of Coloured Petri Nets. In Azéma, P. e Balbo, G., editores, *Lecture Notes in Computer Science: 18th International Conference on Application and Theory of Petri Nets, Toulouse, France.*, volume 1248, págs. 42–61, Berlin, Germany. Springer-Verlag. (Citado na pág. 184)
- Lakos, Charles A. 2000. Composing Abstractions of Coloured Petri Nets. In Nielsen, M. e Simpson, D., editores, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark.*, volume 1825, págs. 323–345. Springer-Verlag, 2000. (Citado na pág. 184)

- Lakos, Charles A. 2001. Object Oriented Modelling with Object Petri Nets. In Agha, G.A., Cindio, F., De, e Rozenberg, G., editores, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science: Advances in Petri nets*, págs. 1–37. Springer-Verlag, Berlin, Heidelberg, Germany. (Citado na pág. 42)
- Leveson, Nancy G. 1990. Guest Editor's Introduction Formal Methods in Software Engineering. *IEEE Transactions on Software Engineering*, 16(9):929–931. (Citado na pág. 6)
- Machado, Ricardo J. e Fernandes, João M. 2001. A Petri Net Meta-Model to Develop Software Components for Embedded Systems. In *Proceedings of the 2nd International Conference on Application of Concurrency to System Design (ACSD 2003)*, págs. 113–122. IEEE Computer Society Press, 2001. (Citado na pág. 111)
- Magee, Jeff e Kramer, Jeff. 1999. *Concurrency: State Models & Java Programs*. Wiley. (Citado na pág. 54)
- Maier, Christoph e Moldt, Daniel. 2001. Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling. In Agha, G.A., Cindio, F., De, e Rozenberg, G., editores, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science: Advances in Petri nets*, págs. 406–427. Springer-Verlag, Berlin, Heidelberg, Germany. (Citado nas págs. 111, 136 e 140)
- Mailund, Thomas. 1999. Parameterised Coloured Petri Nets. In Jensen, Kurt, editor, *Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, DAIMI PB - 541, págs. 133–150, Aarhus, Dinamarca. Department of Computer Science, University of Aarhus. ISSN 0105-8517, disponível em <http://www.daimi.au.dk/CPnets/workshop99/papers/Proceedings.pdf>. (Citado na pág. 143)
- Mäkelä, Marko. 2004. MARIA The Modular Reachability Analyzer. <http://www.tcs.hut.fi/Software/maria/>, 2004. (Citado na pág. 24)
- Mealy, George H. 1955. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079. (Citado na pág. 204)
- Meyer, Bertrand. 1997. *Object-oriented Software Construction*. Prentice-Hall, 2nd edition. (Citado nas págs. 17 e 142)
- Milner, Robin. 1995. *Communication and Concurrency*. Lecture Notes in Computer Science. Prentice Hall PTR. (Citado nas págs. 26 e 135)
- Moldt, Daniel e Wienberg, Frank. 1997. Multi-Agent-Systems Based on Coloured Petri Nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, págs. 82–101, London, UK. Springer-Verlag. (Citado na pág. 43)
- Moore, Edward F. 1956. Gedanken-Experiments on Sequential Machines. In Shannon, Claude e McCarthy, John, editores, *Automata Studies*, págs. 129–153. Princeton University Press, Princeton, NJ. (Citado na pág. 204)
- Murata, Tadao. 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580. (Citado nas págs. 17 e 24)
- Natural History Museum. 2005. Linnaeus Link. <http://www.nhm.ac.uk/research-curation/projects/linnaeus-link/index.html>, 2005. Consultado em 13 de Setembro de 2005. (Citado na pág. 139)
- Notomi, Masato e Murata, Tadao. 1994. Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent-Software Analysis. *IEEE Transactions on Software Engineering (TSE)*, 20(4):325–336. (Citado na pág. 27)

- OMG. 2003. Unified Modeling Language Specification, version 1.5. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>, 2003. Unified Modeling Language, v1.5, Object Management Group. (Citado nas págs. 1, 4, 22, 102, 112, 133, 134, 135, 136, 139, 150, 170 e 301)
- Pais, Rui. 2004. Geração de Executores e Analisadores de Redes de Petri. Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática, 2004. (Citado nas págs. 211 e 213)
- Pais, Rui, Barros, João Paulo, e Gomes, Luís. 2005. A Tool for Tailored Code Generation from Petri Net Models. In *Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005)*. IEEE, 2005. (Citado na pág. 16)
- Parnas, David L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058. (Citado na pág. 26)
- Parnas, David L. 1974. On a "Buzzword": Hierarchical Structure. *Information Processing*, 74:336–339. (Citado nas págs. 36 e 64)
- Parnas, David L. 1978. Another view of the Dijkstra-dMLP controversy. *ACM SIGSOFT Software Engineering Notes*, 3(4):20–21. Letter. (Citado na pág. 6)
- Peterson, James L. 1977. Petri nets. *ACM Computing Surveys*, 9(3):223 – 252. (Citado nas págs. 17 e 24)
- Petri, Carl Adam. 1962. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, West Germany. (Citado nas págs. 2, 6, 18 e 38)
- Reinhold Plösch. 2004. *Contracts, Scenarios and Prototypes*. Springer. (Citado nas págs. 5 e 7)
- Reisig, Wolfgang. 1985. *Petri nets: an Introduction*. Springer-Verlag New York, Inc. (Citado nas págs. 17, 23, 24 e 199)
- Reisig, Wolfgang. 1998. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag. (Citado na pág. 150)
- Reisig, Wolfgang e Rozenberg, Grzegorz, editores. 1998a. *Lectures on Petri Nets I: Basic Models*. Number 1491 in Lecture Notes in Computer Science; Advances in Petri Nets. Springer-Verlag, Germany. (Citado nas págs. 17, 22 e 24)
- Reisig, Wolfgang e Rozenberg, Grzegorz, editores. 1998b. *Lectures on Petri Nets II: Applications*. Number 1492 in Lecture Notes in Computer Science; Advances in Petri Nets. Springer-Verlag, Germany. (Citado nas págs. 17, 22 e 24)
- Rice, John R. e Rosen, Saul. 2004. History of the Department of Computer Sciences at Purdue University. <http://www.cs.purdue.edu/history/history.html>, 2004. Consultado em 27 de Dezembro de 2004. (Citado na pág. 7)
- Roscoe, A. W., Hoare, C. A. R., e Bird, Richard. 1997. *The Theory and Practice of Concurrency*. Prentice Hall PTR. (Citado nas págs. 55 e 135)
- Ruby Home Page. 2004. Ruby Home Page. <http://www.ruby-lang.org/en/>, 2004. (Citado nas págs. 88, 102, 120 e 265)
- S. Schöf, Sonnenschein, M., e Wieting, R. 1995. Efficient Simulation of THOR Nets. In *Lecture Notes in Computer Science; Proceedings of the 16th International Conference in Application and Theory of Petri Nets (ICATPN'95), Turin, Italy*, págs. 412–431. Springer-Verlag, 1995. (Citado na pág. 41)
- s.a. 1969. Software Engineering: Report of a conference sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, 1969. A conferência realizou-se entre 7 e 11 de Ou-

- tubro de 1968 em Garmisch na Alemanha. A publicação está disponível em <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>. (Citado nas págs. 5 e 299)
- s.a. 1996. 3th Advanced Course on Petri Nets. <http://www.daimi.au.dk/PetriNets/education/acpn1996/>, 1996. (Citado na pág. 22)
- s.a. 2003a. Building a Better Bug-Trap. *The Economist, Science and Technology Quarter*. Também disponível em http://www.economist.com/displaystory.cfm?story_id=1841081. (Citado na pág. 22)
- s.a. 2003b. ISO/IEC JTC1/SC34 Web Server. <http://www.y12.doe.gov/sgml/sc34/sc34oldhome.htm>, 2003b. Consultado em 12 de Abril de 2005. (Citado na pág. 97)
- s.a. 2003c. RELAX NG homepage. <http://www.relaxng.org/>, 2003c. Consultado em 12 de Abril de 2005. (Citado nas págs. 97, 193 e 259)
- s.a. 2004a. 4th Advanced Course on Petri Nets. <http://www.acpn.de/>, 2004a. (Citado nas págs. 17 e 22)
- s.a. 2004b. CPN Tools homepage. <http://wiki.daimi.au.dk/cpntools>, 2004b. (Citado nas págs. 9, 14, 24, 27, 31, 39, 110, 120, 136, 146, 207 e 265)
- s.a. 2004c. Design/CPN homepage. <http://www.daimi.au.dk/designCPN/>, 2004c. (Citado nas págs. 9, 27, 39, 110, 120 e 136)
- s.a. 2004d. Petri Net Markup Language (PNML). <http://www.informatik.hu-berlin.de/top/pnml/about.html>, 2004d. (Citado nas págs. 51, 92, 93, 95 e 209)
- s.a. 2004e. Software and System Engineering – High-level Petri Nets – Part 1: Concepts, Definitions and Graphical Notation. <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38225&scopelist=>, 2004e. (Citado na pág. 210)
- s.a. 2005a. aspectj project. <http://eclipse.org/aspectj/>, 2005a. Consultado em 20 de Janeiro de 2005. (Citado na pág. 66)
- s.a. 2005b. javacc Project home. <https://javacc.dev.java.net/>, 2005b. (Citado na pág. 187)
- s.a. 2005c. Petri Nets Standards. <http://www.daimi.au.dk/PetriNets/standardisation/>, 2005c. Consultado em 3 de Fevereiro de 2005. (Citado nas págs. 51, 92 e 210)
- s.a. 2005d. Petri Nets Tool Database. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>, 2005d. (Citado nas págs. 9, 24, 110 e 208)
- SDL Forum Society. 2005. SDL Forum Society homepage. <http://www.sdl-forum.org/>, 2005. Também contem informação sobre Message Sequence Charts (MSCs). Consultado em 29 de Maio de 2005. (Citado nas págs. 1 e 2)
- Sebesta, Robert W. 1996. *Concepts of Programming Languages*. Addison-Wesley. (Citado na pág. 7)
- Sgroi, Marco, Lavagno, Luciano, e Sangiovanni-Vincentelli, Alberto. 2000. Formal Models for Embedded System Design. *IEEE Des. Test*, 17(2):14–27. (Citado na pág. 2)
- Shapiro, Stuart. 1997. Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering. *IEEE Annals of the History of Computing*, 19(1):20–54. (Citado nas págs. 3, 5, 6 e 300)
- Sharpe, Richard. 2004. Formal Methods Start to Add up Again. *Computing*. Também disponível em <http://http://www.computing.co.uk/print/1151896>. (Citado na pág. 22)
- Silva, Manuel. 1985. *Las Redes de Petri: en la Automática y la Informática*. Editorial AC, Madrid. (Citado nas págs. 38, 39 e 200)

- Silva, Manuel. 1993. *Practice of Petri Nets in Manufacturing*, capítulo Introducing Petri Nets, págs. 1–62. Chapman & Hall. ISBN 0 412 41230 6. (Citado nas págs. 17 e 24)
- Simons, Anthony J. H. e Graham, Ian. 1999. *Behavioral Specifications of Businesses and Systems*, capítulo 30 Things that go Wrong in Object Oriented Modelling with UML 1.3, págs. 237–257. Kluwer Academic Publishers. (Citado na pág. 179)
- Smith, Einar. 1993. Arbiter Behaviour and Petri Nets. *Fachberichte Informatik, Universität Koblenz-Landau*, 2/93. (Citado na pág. 86)
- Spinellis, Diomidis. 2003. On the Declarative Specification of Models. *IEEE Software*, 20(2):94–96. (Citado na pág. 2)
- Steimann, Friedrich, Gößner, Jens, e Mück, Thomas. 2003. On the key role of compositioning object-oriented modelling. In «UML» 2003 - *The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference San Francisco, CA, USA, October 2003 Proceedings*, LNCS 2863. Springer, 2003. (Citado na pág. 137)
- Störrle, Harald. 2003. Semantics of Interactions in UML 2.0. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), 28-31 October 2003, Auckland, New Zealand*, págs. 129–136. IEEE Computer Society, 2003. (Citado na pág. 6)
- Störrle, Harald e Hausmann, Jan Hendrik. 2005. Towards a Formal Semantics of UML 2.0 Activities. In *German Software Engineering Conference*, págs. 117–128, 2005. (Citado na pág. 6)
- Stroustrup, Bjarne. 1994. *The Design and Evolution of C++*. Addison-Wesley. (Citado na pág. 182)
- Stroustrup, Bjarne. 1997. *The C++ Programming Language*. Addison-Wesley. (Citado na pág. 125)
- SysML Partners. 2005. Systems Modeling Language (SysML). <http://www.sysml.org/>, 2005. Consultado em 29 de Maio de 2005. (Citado na pág. 4)
- Taivalsaari, Antero. 1996. On the Notion of Inheritance. *ACM Computing Surveys (CSUR)*, 28(3): 438–479. (Citado na pág. 137)
- Tarr, Peri, Ossher, Harold, Harrison, William, e Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software engineering*, págs. 107–119. IEEE Computer Society Press, 1999. (Citado na pág. 66)
- The Joint Task Force for Computing Curricula 2005. Computing Curricula 2005 The Overview Report including The Guide to Undergraduate Degree Programs in Computing. Disponível em http://campus.acm.org/public/comments/Draft_5-23-05.pdf, 2005. (Citado nas págs. 5 e 300)
- Valk, Rüdiger. 1996. *On Processes of Object Petri Nets*. Bericht FBI-HH-B-185/96, Fachbereich Informatik, Universität Hamburg. (Citado na pág. 42)
- Valk, Rüdiger. 2003. *Essential Features of Petri Nets*, capítulo 2, págs. 9–28. In Rüdiger Valk, Girault e Valk [2003]. (Citado na pág. 3)
- Valk, Rüdiger. 2004. Object Petri Nets. In Desel, Jörg, Reisig, Wolfgang, e Rozenberg, Grzegorz, editores, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science: Advances in Petri Nets*, págs. 819–848. Springer-Verlag, Berlin Heidelberg. Based on the lectures given at the 4th Advanced Course on Petri Nets held in Eichstätt, Germany in September 2004. (Citado nas págs. 31 e 42)
- Valmari, Antti. 1994. Compositional Analysis with Place-Bordered Subnets. In Valette, R., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, págs. 531–547. Springer-Verlag, 1994. (Citado na pág. 27)

- van der Aalst, W.M.P. e Basten, T. 1997. Life-Cycle Inheritance - A Petri-Net-Based Approach. In *Proceeding of the 18th International Conference on Application and Theory of Petri Nets, Toulouse, France.*, págs. 62–81. Springer-Verlag, 1997. (Citado na pág. 139)
- van der Aalst, W.M.P. e van Hee, Kees. 2002. *Workflow Management: Models, Methods, and Systems*. MIT Press. (Citado na pág. 8)
- W3C. 2005. Extensible Markup Language (XML). <http://www.w3.org/XML/>, 2005. Consultado em 3 de Fevereiro de 2005. (Citado nas págs. 51, 80 e 93)
- Weber, Michael e Kindler, Ekkart. 2003. The Petri Net Markup Language. In Ehrig, H., Reisig, W., Rozenberg, G., e Weber, H., editores, *Petri Net Technology for Communication Based Systems*, volume 2472 of *LNCS*, págs. 124–144. Springer-Verlag. (Citado na pág. 51)
- Wegner, Peter. 1987. Dimensions of Object-Based Language Design. *ACM SIGPLAN Notices*, 22(12): 168–182. (Citado na pág. 139)
- Wieringa, Roel. 1998. A Survey of Structured and Object-oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, 30(4):459–527. (Citado na pág. 2)
- Wieringa, Roel. 2003. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann. (Citado nas págs. 162 e 163)
- Wirth, Niklaus. 1971. Program Development by Stepwise Refinement. *Communications of the ACM*, 14 (4):221–227. (Citado na pág. 36)
- Yakovlev, Alex, Gomes, Luis, e Lavagno, Luciano, editores. 2000. *Hardware Design and Petri Nets*. Kluwer Academic Publishers. (Citado na pág. 8)
- Zurawski, Richard e Zhou, MengChu. 1994. Petri Nets and Industrial Applications - a Tutorial. *IEEE Transactions on Industrial Electronics*, 41(6):567–583. (Citado nas págs. 17 e 24)

Glossário Português-Inglês

- abstracção:** abstraction, 36
- agente:** agent, 43
- agregação:** aggregation, 137
- apta:** enabled, 205
- ascendente:** bottom-up, 26, 36
- associação:** association, 137
- assunto transversal:** crosscutting concern, 67
- canal síncrono:** synchronous channel, 33
- casos de uso:** use cases, 163
- ciclo-fechado:** closed-loop, 200
- classificação:** classification, 133
- composição:** composition, 24, 137
- composição forte:** strong composition, 137
- composto:** composite, 137
- conjuntos de cores:** colour sets, 115
- desdobragem:** unfolding, 32
- desenho:** design, 42
- diagramas de colaboração:** collaboration diagrams, 178
- disparar:** fire, 3
- disposição:** layout, 52
- dobragem:** folding, 26
- entrecortar:** crosscut, 49, 66
- escalonador:** scheduler, 131
- estadogramas:** statecharts, 2
- generalização:** generalization, 139

guarda: guard, 30

iterador: iterator, 76

jogador de marcas: token player, 206

linguagem de inscrições: inscription language, 80

localidade: locality, 3

lugares de fusão: fusion places, 147

lugares de substituição: substitution places, 27

lugares porto: port places, 39, 147

macro: macro, 37

nós de invocação: invocation nodes, 41

padrão: pattern, 165

pedido: request, 113

PNML estruturado: Structured PNML, 95

PNML modular: Modular PNML, 95

predicado-transição: predicate-transition, 30

programação generativa: generative programming, 66

pronta: ready, 204

redes de Petri objecto: object Petri nets, 31, 42

redes dentro de redes: nets-within-nets, 42

refinamento: refinement, 24, 36

sistemas embutidos ou embebidos: embedded systems, 4

substitution transition: transição de substituição, 39

transições de substituição: substitution transitions, 27

vinculação dinâmica: dynamic binding, 119, 139

vínculo: binding, 116

rede de Petri referência: reference nets, 42

Glossário Inglês-Português

- abstraction:** abstracção, 36
- agent:** agente, 43
- aggregation:** agregação, 137
- association:** associação, 137
- binding:** vínculo, 116
- bottom-up:** ascendente, 26, 36
- classification:** classificação, 133
- closed-loop:** ciclo-fechado, 200
- collaboration diagrams:** diagramas de colaboração, 178
- colour sets:** conjuntos de cores, 115
- composite:** composto, 137
- composition:** composição, 24, 137
- crosscut:** entrecortar, 49, 66
- crosscutting concern:** assunto transversal, 67
- design:** desenho, 42
- dynamic binding:** vinculação dinâmica, 119, 139
- embedded systems:** sistemas embutidos ou embebidos, 4
- enabled:** apta, 205
- fire:** disparar, 3
- folding:** dobragem, 26
- fusion places:** lugares de fusão, 147
- generalization:** generalização, 139
- generative programming:** programação generativa, 66
- guard:** guarda, 30
- inscription language:** linguagem de inscrições, 80

invocation nodes: nós de invocação, 41

iterator: iterador, 76

layout: disposição, 52

locality: localidade, 3

macro: macro, 37

Modular PNML: PNML modular, 95

nets-within-nets: redes dentro de redes, 42

object Petri nets: redes de Petri objecto, 31, 42

pattern: padrão, 165

port places: lugares porto, 39, 147

predicate-transition: predicado-transição, 30

ready: pronta, 204

reference nets: rede de Petri referência, 42

refinement: refinamento, 24, 36

request: pedido, 113

scheduler: escalonador, 131

statecharts: estadogramas, 2

strong composition: composição forte, 137

Structured PNML: PNML estruturado, 95

substitution places: lugares de substituição, 27

substitution transitions: transições de substituição, 27

synchronous channel: canal síncrono, 33

token player: jogador de marcas, 206

transição de substituição: substitution transition, 39

unfolding: desdobragem, 32

use cases: casos de uso, 163

Sobre as Citações

As citações de José Eduardo Agualusa, no início da Introdução e da Conclusão, foram retiradas da primeira estória do livro "Estranhões e Bizarrocos [estórias para adormecer anjos]" das Publicações Dom Quixote.

A citação de Manuel António Pina, no início do Capítulo 2 (pág. 21) é de um belíssimo e extraordinário livro de histórias para crianças com o título "Histórias que me contaste tu", editado pela Assírio & Alvim.

A frase de Mark Twain parafraseada nos agradecimentos é apenas(!) mais uma das suas conhecidas frases. No original parece que foi: "I have never let my schooling interfere with my education."

Finalmente, apresentam-se as versões originais e proveniência das citações traduzidas do inglês:

- As frases de Tom DeMarco no início da Introdução (pág. 1):

"The one document that we found ourselves using most was Erna's Petri net. It showed how all the pieces of the puzzle were related and how they were obliged to interact. (...). One of my colleagues, Jut Kodner, observed that the diagram was a better spec than the spec." [DeMarco, 2002, pág. 522].

- A origem do termo engenharia de software (pág. 5):

"The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering." [s.a., 1969]

- A definição de Engenharia de Software da *Joint Task Force for Computing Curricula 2005* (pág. 5):

"While computer science (like other sciences) focuses on creating new knowledge, software engineering (like other engineering disciplines) focuses on rigorous me-

thods for designing and building things that reliably do what they're supposed to." [The Joint Task Force for Computing Curricula 2005, 2005]

- Sobre as diferentes visões de cientistas e engenheiros (pág. 6):

"Clearly, though, the stance people took with regard to testing versus formal verification was at least partially a function of how they perceived themselves. Self-perceived scientists might develop a very different view than self-perceived engineers. Where you sit sometimes determines where you stand." [Shapiro, 1997]

- Sobre o fosso entre a Matemática e os domínios de aplicação (pág. 8):

"The conceptual gap between application domains and mathematics must be bridged by building mathematical models of the application domains. Within an appropriate model, formal language is extended to include the vocabulary and relationships on the domain. The lack of appropriate models, on the other hand, constitutes a large barrier to the use of formal methods in an application domain." [Hall et al., 1996]

- O comentário de Deutsch sobre linguagens visuais (pág. 23):

"Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?" [Lakin, 1998].

- As frases de Michael Jackson sobre estruturação hierárquica (pág. 49):

"Hierarchical Structuring is one way of separating concerns. Usually it doesn't work well and some kind of parallel structure is better. The separation on coloured picture into cyan, magenta, yellow, and black overlays is a better metaphor for description structuring than the hierarchical assembly structure of parts in manufacturing. Hierarchical structure is the sand on which Top-Down methods are built." [Jackson, 1995, pág. 4]

- As frases de Michael Jackson sobre linguagens gráficas (pág. 91):

"For a graphic description you also have an extra task: deciding how to place the symbols. This can be an unwelcome complication, especially if you fell sure that there shouldn't be any crossings." [Jackson, 1995, pág. 89]

- Frederick Brooks sublinhando a importância da ocultação de informação (pág. 109):

"Parnas was right, and I was wrong. I am now convinced that information hiding, today often embodied in object-oriented programming, is the only way of raising the level of software design." [Brooks, 1995, pág. 272]

- Novamente Frederick Brook, agora salientando a utilidade do encapsulamento e da reutilização de módulos (pág. 112):

"Most of what we hope to gain from object-oriented programming derives in fact from the first step, module encapsulation, plus the idea of prebuilt libraries of modules or classes that are designed and tested for reuse." [Brooks, 1995, pág. 273]

- Sinais e operações segundo a especificação da UML 1.5 (pág. 134):

"Several kinds of requests exist between instances (e.g., sending a signal and invoking an operation). The former is used to trigger a reaction in the receiver in an asynchronous way and without a reply, while the latter applies an operation to an instance, which can be either done synchronously or asynchronously and may require a reply from the receiver to the sender. Other kinds of requests are used for example to create a new instance or to delete an already existing instance." [OMG, 2003, pág. 2-110].

- Generalização segundo a especificação da UML 1.5 (pág. 139):

"Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information." [OMG, 2003, págs. 3-86].

- Composição versus herança, segundo o "GoF Book" (pág. 142):

"Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behaviour will depend on their interrelationships instead of being defined in one class." [Gamma et al., 1995, pág. 19].

- E a frase de David Gries sobre abstracção (pág. 184):

"By abstraction we mean the act of singling out a few properties of an object for further use or study, omitting from consideration other properties that don't concern us for the moment." [Gries, 1981].

Às vezes, ouvia contar uma história sobre umas índias na Bolívia, acho que era na Bolívia, que não gostavam do feitio da cabeça com que lhes nasciam os filhos. Então, punham-lhes umas talas, para eles terem uma cabeça "apresentável" em sociedade, e só quando a cabeça se aproximava do formato de um cubo, então é que as mães bolivianas ficavam satisfeitas. Hoje, as pessoas dizem: oh!, felizmente acabaram essas brutalidades, acabou essa porcaria toda! Mas na verdade não acabou, porque, hoje, quando uma pessoa faz um curso e consegue alcançar o doutoramento, em geral sai de lá com a cabeça cúbica.

Agostinho da Silva *in*

"A Última Conversa", Editorial Notícias, 1995.

Esta dissertação foi apoiada no âmbito da medida n.º 5, acção 5.3, "Formação avançada de docentes do ensino superior", integrada no eixo n.º 3, Sociedade de Aprendizagem da Intervenção Operacional da Educação (PRODEP III).