Aspectos de Herança em uma Notação Orientada a Objetos Baseada em Redes de Petri

Sandro Alex Damasceno Costa

Dissertação submetida à Coordenação do Curso de Pós-Gradução em Informática da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Redes de Petri

Angelo Perkusich
(orientador)
Jorge Cesar Abrantes de Figueiredo
(orientador)

Campina Grande, Paraíba, Brasil

C837a Costa, Sandro Alex Damasceno
Aspectos de heranca em uma notacao orientada a objetos baseada em redes de petri / Sandro Alex Damasceno Costa. - Campina Grande, 1999.
85 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Redes de Petri 2. Engenharia de Software 3. Orientacao a Objetos 4. Dissertacao - Informatica I. Perkusich, Angelo II. Figueiredo, Jorge Cesar Abrantes de III. Universidade Federal da Paraiba - Campina Grande (PB) IV. Título

CDU 004.7(043)

ASPECTOS DE HERANÇA EM UMA NOTAÇÃO ORIENTADA A OBJETOS BASEADA EM REDES DE PETRI

SANDRO ALEX DAMASCENO COSTA

DISSERTAÇÃO APROVADA EM 25.02.1999

PROF. ANGELO PERKUSICH, D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Co-Orientador

PROF. EVANDRO DE BARROS COSTA, D.Sc
Examinador

PROF. JOSÉ REINALDO SILVA, Ph.D
Examinador

PROF. AUGUSTO CÉSAR ALVES SAMPAIO, Ph.D
Examinador

CAMPINA GRANDE – PB

Resumo

Neste trabalho, apresentamos uma discussão sobre os aspectos relacionados à incorporação do conceito de herança em uma notação formal, baseada em redes de Petri e em Orientação a Objetos. Esta notação tem por objetivo a especificação de sistemas distribuídos e concorrentes de software. Buscamos um tratamento mais adequado para a questão da herança, diferente do que é realizado por outras propostas que vêem na herança objetivos puramente práticos de aumento de produtividade. Consideramos em primeiro plano neste trabalho a importância de um mecanismo para a representação da especialização conceitual.

Abstract

This work presents a discussion related to aspects dealing with the inclusion of the inheritance concept in a formal notation, based on Petri nets and Object Oriented concepts. The objective of such notation is the specification of concurrent distributed software systems. In this work a more adequate approach to deal with inheritance is considered, opposed to those that consider only practical aspects of inheritance to increase productivity. The importance of a mechanism to represent the conceptual specialization is the main concern in this work.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, à minha família pelo completo apoio, pela compreensão e pelo carinho, fundamentais para a realizacao deste trabalho. Obrigado meus pais (Antonio e Marlene), meus irmãos (Léo e Mi) e meus avós (David, Judith e Julieta).

Agradeço aos meus orientadores pela confiança, pela paciência e pela amizade. Obrigado pela seriedade na orientação, pela qualidade das discussões e pelo empenho.

A Dalton, pelo fundamental incentivo e pelo inestimável apoio. Aprendi e tenho aprendido muito com nossas intermináveis conversas. Que algumas delas terminem em ações comprometidas com a realidade que nos cerca.

Aos colegas do grupo de Redes de Petri, pela amizade, pela paciência e pelas contribuições.

Aos colegas do mestrado.

Aos professores e funcionários.

Finalmente, gostaria de agradecer, de forma geral, a todas as pessoas com as quais convivi nestes dois anos em Campina Grande. Obrigado por me sentir querido e integrado nesta cidade da qual aprendi a gostar.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Dentre as técnicas utilizadas pela Engenharia de Software para a construção de sistemas, os métodos formais (matemáticos) têm o papel de ajudar a lidar com a complexidade inerente a estes sistemas, oferecendo rigor às descrições produzidas em cada fase do desenvolvimento [GUEZZI ET AL., 1991; ALAGAR & PERIYASAMY, 1998].

A princípio, métodos formais são utilizados para dar consistência ao levantamento de requisitos feito junto ao usuário final. Assim, ao obtermos uma especificação formal dos requisitos, descritos em linguagem natural, podemos eliminar uma série de deficiências, como por exemplo [MEYER, 1985]: redundância, incompletude, contradição, ambigüidade, entre outras. Quanto mais tardia a descoberta de tais deficiências, mais aumenta o custo total de desenvolvimento dos sistemas. Não se pretende que as notações formais substituam a linguagem natural no levantamento de requisitos, mas sim que sirvam de complemento a estas [MEYER, 1985].

Sem considerar uma metodologia específica para o desenvolvimento de sistemas complexos, podemos entender que este processo é dividido em fases e envolve várias pessoas [GUEZZI ET AL., 1991]. Chamamos genericamente de *descrição do sistema* o produto de cada uma destas fases. Chamamos de *especificação* a descrição do sistema obtida em cada fase intermediária e *implementação* o resultado final do desenvolvimento, descrito em alguma linguagem de programação, que possa ser traduzida automaticamente para código binário de alguma máquina. Portanto, uma implementação é necessariamente formal, uma vez que a linguagem utilizada tem sintaxe e semântica completamente definidas. Uma especificação, por sua vez, pode ser formal ou não.

Neste trabalho, chamamos de notação ou ferramenta formal a uma linguagem para especificação formal de sistemas.

Sendo a primeira das fases o levantamento dos requisitos e a última a implementação do sistema, as fases intermediárias devem aumentar os níveis de detalhe a cada modelo, de forma a garantir que a implementação obtida seja fiel aos requisitos levantados em comum acordo com o usuário final. Neste sentido, torna-se interessante a utilização de especificação formal como um *acordo* ou *contrato* firmado entre os responsáveis por fases vizinhas. Uma vez que é possível verificar formalmente se o produto de uma fase (uma especificação formal) implementa a especificação gerada na fase anterior, temos um instrumento poderoso para garantir a compatibilidade da implementação final com os requisitos iniciais.

Além dessas características, outro ganho obtido através do uso de notações formais é a possibilidade de análise automática dos modelos obtidos. Deste modo, podemos estudar propriedades dinâmicas dos sistemas bem antes da sua conclusão.

Porém, existem problemas na utilização das ferramentas formais por parte dos desenvolvedores, que esboçam resistência em lidar com os aspectos matemáticos envolvidos. Além disso, a compreensão dos modelos gerados fica restrita a quem domina o formalismo, o que limita o seu poder de comunicação.

Redes de Petri, como uma notação para especificação formal de sistemas (incluídos os sistemas de software), apresenta a característica de possuir uma representação gráfica rigorosa, porém simples e por vezes intuitiva, que permite a sua utilização sem que tenhamos que nos ater a detalhes matemáticos. As redes de Petri se prestam mais adequadamente à descrição e estudo de características concorrentes dos sistemas.

As chamadas redes de Petri de Alto Nível, com destaque para as redes Predicado/Transição (PrT-Nets) [GENRICH, 1987] e as redes de Petri Coloridas (CPN) [JENSEN, 1992], associam às características das redes de Petri clássicas uma linguagem para especificação dos dados dos sistemas. Desta forma, constituem notações mais adequadas para especificação dos sistemas, devendo a estrutura das redes de Petri ser utilizada para descrever o controle (principalmente concorrente) e a linguagem associada, os dados.

Porém as redes de Petri de Alto Nível carecem de mecanismos eficientes de mo-

dularização, o que dificulta bastante a sua utilização para a especificação de sistemas complexos. Nesse sentido, algumas pesquisas vêm sendo desenvolvidas com o objetivo de associar às redes de Petri o poder de estruturação e encapsulamento da Orientação a Objetos [BASTIDE, 1995; BALDASSARI ET AL., 1989; ENGELFRIET ET AL., 1990; BATTISTON & DE CINDIO, 1993; LAKOS, 1995a; LAKOS & KEEN, 1991; BUCHS & GUELFI, 1991; DENG ET AL., 1993].

Em [GUERRERO, 1997; GUERERRO ET AL., 1998; GUERRERO ET AL., 1997], foi proposta uma notação, denominada G-CPN, para especificação formal de sistemas distribuídos de software. G-CPN é inspirada no emprego dos conceitos de Orientação a Objetos presentes nas redes G-Nets [DENG ET AL., 1993; PERKUSICH & DE FIGUEIREDO, 1997], porém utilizando as redes de Petri Coloridas em lugar das redes de Petri Predicado/Transição.

Contudo, G-CPN não pode ser considerada uma notação orientada a objetos, pois não suporta o conceito de *herança*. Mesmo não havendo consenso sobre o que é necessário para que uma notação seja considerada como tal, as diversas abordagens sobre o assunto ([WEGNER, 1987], [BOOCH, 1994], e [RUMBAUGH ET AL., 1991], entre outros) tratam como essencial o suporte a herança. Embora, mais uma vez, não haja consenso sobre a definição de tal conceito nem como deve se dar o seu suporte.

Assim, vem sendo proposta uma notação derivada de G-CPN, chamada RPOO (Rede de Petri Orientada a Objetos) [COSTA ET AL., 1998b; DE MEDEIROS ET AL., 1998; COSTA ET AL., 1998a; GUERRERO, 1998b], que procura completa aderência à Orientação a Objetos, segundo os critérios definidos em [WEGNER, 1987]. As diferenças mais significativas que se propõe para RPOO com relação a G-CPN são as seguintes:

- redefinição dos mecanismos de comunicação entre objetos, optando-se por suportar uma única primitiva assíncrona, a partir da qual outras formas de comunicação possam ser elaboradas;
- substituição da linguagem funcional CPN-ML para descrição dos dados por uma abordagem baseada em especificação algébrica;
- incorporação do conceito de herança;

A incorporação do conceito de herança em qualquer notação deve ser guiada por

objetivos práticos ou conceituais. Estes objetivos não são necessariamente convergentes [TAIVALSAARI, 1996].

Objetivos de natureza prática nos levam ao conceito de herança relacionado a mecanismos de modificação incremental, que possibilitam que a descrição de uma (sub)classe possa utilizar a descrição de outra (super)classe. Através destes mecanismos, podemos gerar modelos mais compactos, nos quais, para uma subclasse, só precisamos descrever o que a diferencia da sua superclasse. Esperamos, com isso, simplificar a compreensão e a manutenção dos modelos. O conceito de herança entendido desta forma aumenta o poder de reutilização dos componentes de um modelo e, em conseqüência, tende a aumentar a produtividade do seu desenvolvimento.

Por sua vez, objetivos conceituais fazem-nos conceber herança como um mecanismo para representar a especialização conceitual, normalmente compreendida pelo relacionamento *é-um*. Relacionamentos caracterizados por estes mecanismos tendem a aumentar o conteúdo semântico dos modelos e a simplificar a sua compreensão [BOOCH, 1994].

Considerando um modelo de classes de um sistema obtido em uma determinada fase do seu desenvolvimento, uma das formas de inclusão de detalhes neste modelo pela fase seguinte pode ser a especialização das suas classes. A garantia de que uma classe B é subclasse de outra classe A deve ser suficiente para garantir que objetos de B substituam objetos de A no sistema tratado, sem necessidade de estudo do novo modelo como um todo.

1.1 Objetivos da Dissertação

Este trabalho tem como objetivo principal a investigação do conceito de herança, tanto com objetivos práticos quanto conceituais, na notação denominada RPOO.

Como existe uma certa discordância a respeito da importância e do significado do termo (herança) em Orientação a Objetos, este trabalho também se propõe a realizar um estudo sobre o assunto, tendo sempre como ponto de vista a sua utilidade com relação a uma notação para especificação formal de sistemas.

1.2 Escopo e Relevância

Dentro dos objetivos apresentados, este trabalho realiza um estudo sobre o conceito de herança em Orientação a Objetos. Este estudo serve de base para a discussão sobre a inclusão da herança em RPOO.

Escolhemos identificar herança, em primeiro plano, como um relacionamento hierárquico entre classes, no qual uma subclasse respeita alguns critérios de compatibilidade com relação à sua superclasse. Assim, dividimos a discussão sobre herança em RPOO segundo a natureza dos possíveis critérios a serem adotados: critérios sintáticos e critérios semânticos.

Os critérios sintáticos se referem a restrições impostas à descrição das subclasses. Neste trabalho, discutimos as restrições relacionadas à interface das classes RPOO. Formalizamos então o conceito de compatibilidade de assinatura entre classes de objetos RPOO.

Os critérios semânticos de compatibilidade, por sua vez, se referem a restrições sobre propriedades comportamentais das subclasses. Assim, utilizamos o conceito de comportamento observado externamente, introduzido por Milner [MILNER, 1980], para discutir o estabelecimento de critérios baseados ou no espaço de estados ou na seqüência de eventos dos objetos. Os critérios semânticos são discutidos tendo como objetivo garantir que objetos de subclasses possam ser manipulados como (substituir) objetos das suas respectivas superclasses.

A importância deste trabalho está no tratamento mais adequado à questão da herança em uma linguagem de especificação baseada em redes de Petri, principalmente com relação à discussão a respeito das restrições semânticas para estabelecimento de subclassificação. Assim, este trabalho busca contribuir para o desenvolvimento de notações cada vez mais adequadas à especificação formal de sistemas distribuídos de software.

1.3 Estrutura da Dissertação

No Capítulo 2, apresentamos a notação RPOO.

Discutimos alguns aspectos relacionados ao conceito de herança em Orientação a

Objetos no Capítulo 3. Apresentamos neste capítulo os possíveis tratamentos dados a herança, bem como alguns termos normalmente associados, como subclassificação e subtipificação.

No Capítulo 4, discutimos aspectos de herança associados a restrições sintáticas nas interfaces das classes.

No Capítulo 5, discutimos a utilização de propriedades semânticas das redes que descrevem o comportamento dos objetos para garantir níveis de substituição entre eles.

No Capítulo 6, apresentamos as conclusões e trabalhos futuros.

Capítulo 2

RPOO - Rede de Petri Orientada a Objetos

Neste capítulo, apresentamos RPOO (Rede de Petri Orientada a Objetos), uma ferramenta para especificação de sistemas de software concorrente e distribuído, baseada em redes de Petri e que incorpora os conceitos de Orientação a Objetos. Nesta ferramenta, os aspectos de notação oriundos das redes de Petri têm por objetivo prover meios para a descrição e estudo de características de concorrência dos sistemas. Por sua vez, os aspectos da Orientação a Objetos colaboram no tratamento da complexidade dos sistemas que se pretende descrever.

Não pretendemos apresentar a descrição detalhada de redes de Petri, mas apenas o suficiente para o bom entendimento e fácil leitura de RPOO. Assim, optamos por apresentar as definições formais relativas somente à sintaxe da notação. Incluímos estas definições de forma gradativa, à medida que vamos apresentando o caminho evolutivo que levou à sua elaboração.

O capítulo está dividido em duas seções. Na Seção 2.1, discutimos as redes de Petri segundo os avanços na tentativa de se obter notações com cada vez mais poder de expressão, partindo das chamadas redes clássicas até chegar às abordagens orientadas a objetos. Na Seção 2.2, definimos RPOO.

2.1 Redes de Petri

Redes de Petri são uma ferramenta matemática para especificação de sistemas, apropriadamente daqueles que apresentam características concorrentes, assíncronas, distribuídas, paralelas, não determinísticas, e/ou estocásticas [MURATA, 1989]. Através das redes de Petri, podemos descrever adequadamente tais características, que podem ser então estudadas por meio de verificação e/ou análise dos modelos.

Outra característica interessante das redes de Petri é possuir uma representação gráfica precisa e amigável, fazendo com que a sua utilização e entendimento sejam razoavelmente fáceis e, por vezes, intuitivos, sem perder em rigor formal. Desta forma, as redes de Petri constituem um poderoso meio de comunicação entre indivíduos envolvidos no processo de construção de sistemas.

A sintaxe de uma rede de Petri é geralmente compreendida de duas partes: uma *estrutura de rede* e *inscrições* associadas a esta estrutura. A semântica é definida por uma *regra de disparo*. Basicamente, para todas as classes existentes de redes de Petri a estrutura de rede pode ser definida da mesma forma. Estas classes se diferenciam pelas inscrições (tipos e linguagem utilizada) e/ou pela regra de disparo.

Uma estrutura de rede é um tipo de grafo composto por dois conjuntos disjuntos de nós (*lugares* e *transições*), no qual os arcos orientados conectam elementos de um conjunto a elementos do outro, denominado de *grafo bipartido dirigido*. Formalmente, uma estrutura de rede pode ser definida como segue.

Definição 2.1 (Estrutura de Rede) *Uma estrutura de rede de Petri é uma tripla* (P, T, F)*, na qual:*

- P *é um conjunto finito de lugares;*
- T *é um conjunto finito de transições;*
- $F \subseteq P \times T \cup T \times P$ *é uma relação de fluxo, que caracteriza os arcos;*
- $P \cap T = \emptyset$.

Em uma rede de Petri, as inscrições de uma estrutura correspondem a inscrições textuais, que podem estar associadas a lugares, transições e arcos. Os tipos de inscrições

que podem ser associadas a uma estrutura variam com as classes de redes de Petri. Também varia a linguagem na qual se baseiam.

Com relação ao comportamento dinâmico de uma rede de Petri, os estados de um sistema são caracterizados por distribuições de elementos pelos lugares, denominadas *marcações*. Cada marcação então associa determinados elementos, chamados *fichas*, aos lugares. O estado de um sistema varia com a ocorrência de eventos, modelados pelas transições. Uma regra de disparo portanto determina quais são as condições para uma transição estar *habilitada* a disparar (ocorrer) e quais são as conseqüências do seu disparo.

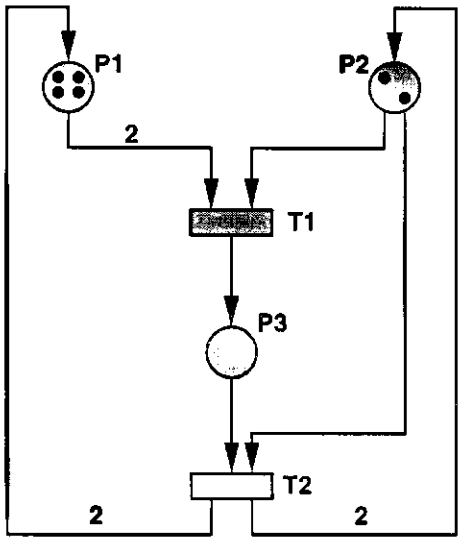

Figura 2.1: Exemplo de rede de Petri Lugar/Transição.

As classes de redes denominadas *redes de Petri clássicas* constituem as classes mais básicas. Elas são caracterizadas por suas fichas carregarem apenas informação binária. Como exemplo, temos as redes Elementares [THIAGARAJAN, 1987] e as redes Lugar/Transição [MURATA, 1989]. Vamos utilizar uma rede de Petri Lugar/Transição, para ilustrar o que já foi discutido até este ponto.

Temos na Figura 2.1 a representação gráfica de um exemplo de rede Lugar/Transição. Representamos os lugares por círculos e as transições por retângulos. Nas redes Lugar/Transição existem dois tipos de inscrições: a *inicialização* e o *peso dos arcos*. Com relação à semântica, a inicialização, também chamada de *marcação inicial*, determina o primeiro estado do sistema. Na Figura 2.1, vemos a inicialização

representada por marcas dentro dos lugares. Cada marca representa uma ficha sem informação. Neste exemplo, portanto, a marcação inicial associa 4 (quatro) fichas ao lugar $P1$, 2 (duas) ao lugar $P2$ e 0 (zero) ao lugar $P3$. O peso dos arcos, por sua vez, associa a cada arco um número natural diferente de 0 (zero). O peso dos arcos é determinante para indicar quando uma transição pode ocorrer e o que acontece quando da sua ocorrência. Na Figura 2.1, os arcos $(P1, T1)$, $(T2, P1)$ e $(T2, P2)$ têm peso 2 (dois). Os pesos dos demais arcos têm valor 1 (um) e são omitidos nessa representação gráfica.

Para descrevermos uma regra de disparo para redes Lugar/Transição vamos considerar os seguintes conceitos, relacionados a uma transição:

Lugares de entrada: lugares de onde partem os arcos (*arcos de entrada*) que atingem a transição;

Lugares de saída: lugares para onde partem os arcos (*arcos de saída*) que abandonam a transição.

Vamos considerar então a seguinte regra de disparo para redes Lugar/Transição [MURATA, 1989]. Considerando um determinado estado do sistema, caracterizado por uma marcação em uma rede, temos que:

- uma transição está habilitada para disparar se, para cada um de seus lugares de entrada existe associado *pelo menos* o número de fichas indicado pelo peso do arco de entrada respectivo;
- uma transição habilitada *pode* disparar;
- o disparo de uma transição habilitada provoca:
  - para cada um de seus lugares de entrada, a remoção do número de fichas indicado pelo peso do arco de entrada respectivo;
  - para cada um de seus lugares de saída, a adição do número de fichas indicado pelo peso do arco de saída respectivo.

Portanto, no exemplo da Figura 2.1, considerando a marcação inicial, somente a transição $T1$ está habilitada. O seu disparo retira 2 (duas) fichas do lugar $P1$ e 1 (uma) ficha do lugar $P2$ e adiciona 1 (uma) ficha ao lugar $P3$.

Com o objetivo de suportar a simulação de concorrência real no modelo, devemos considerar a noção de *passo habilitado*, composto por um conjunto de transições habilitadas, para o qual o disparo de uma das transições não desabilita as demais. Logo, as transições de um passo habilitado podem disparar simultaneamente (concorrentemente). Na Figura 2.1, para a marcação inicial, podemos ter dois disparos concorrentes da transição $T1$, que caracterizam assim um passo habilitado.

2.1.1 Redes de Petri Coloridas

As redes de Petri clássicas, contudo, apresentam algumas limitações na modelagem de sistemas reais e para supri-las foram propostas algumas extensões, dentre as quais destacam-se: as redes Temporais [AJMONE MARSAN, 1989; MERLIN, 1979; RAMCHANDANI, 1974; SIFAKIS, 1980], que fornecem elementos para a modelagem de aspectos temporais quantitativos; e as redes de Alto Nível, que permitem a construção de estruturas mais compactas, através do aumento do poder de expressão.

A principal particularidade nas redes de Alto Nível é que suas fichas passam a carregar informações complexas e não apenas informações binárias. Assim, um lugar pode comportar fichas distintas umas das outras, desde que seus valores pertençam a um mesmo domínio de dados (ou tipo). Baseados nesta característica, podemos evitar a repetição de estruturas bastante similares nos modelos sem perder em clareza. Quanto maiores os sistemas, maiores os ganhos que tal tipo de *fatoração* pode proporcionar [JENSEN, 1992].

As redes de Petri de Alto Nível devem, portanto, suportar uma linguagem de manipulação de dados classificados em tipos para compor as inscrições das estruturas. Desta forma, é possível para o desenvolvedor concentrar a manipulação dos dados nas inscrições, fazendo com que a estrutura das redes descreva as características concorrentes dos sistemas. Na verdade, ao dispor uma linguagem para as inscrições tão poderosa (computacionalmente) quanto as redes de Petri, ou quanto qualquer linguagem de programação, as redes de Petri de Alto Nível deixam a cargo do desenvolvedor balancear

a descrição dos sistemas entre as inscrições e a estrutura [JENSEN, 1992]. Cabe a este desenvolvedor portanto ponderar a clareza da descrição obtida e do estudo que poderá realizar sobre esta.

Dentre as propostas de redes de Petri de Alto Nível, as mais importantes são as redes Predicado/Transição (PrT-Nets) [GENRICH, 1987] e as redes de Petri Coloridas (CPN - *Coloured Petri Nets*) [JENSEN, 1992]. As redes de Petri Coloridas apresentam a vantagem de serem hoje largamente utilizadas e de existirem métodos e ferramentas consolidados de análise, verificação e simulação, com destaque para a ferramenta Design CPN [DES, 1996].

As redes de Petri Coloridas são utilizadas como base para RPOO. Porém, utilizamos neste trabalho uma definição um pouco diferente da original para as redes de Petri Coloridas. Na definição original [JENSEN, 1992], é utilizada uma linguagem funcional, CPN-ML, para as inscrições. Com o objetivo de tornar as nossas definições o mais genéricas possível e de controlar em um número mínimo de definições todo o formalismo que envolve RPOO, utilizamos, ao invés de CPN-ML, termos de uma assinatura (como definida para especificações algébricas) como inscrições. Com isto, nos isentamos de lidar com todos os detalhes de uma linguagem determinada de inscrições, uma vez que este não é o foco deste trabalho. Além disto, uma abordagem baseada em especificações algébricas nos permite um nível abstrato mais apropriado para especificar dados do que através do uso de uma linguagem funcional, como CPN-ML [GUERRERO, 1998b]. Na prática, porém, para que uma rede de Petri Colorida seja efetivamente utilizada para descrever sistemas, deve-se utilizar uma linguagem com primitivas de mais alto nível para especificar os dados.

Apresentaremos então, a partir deste ponto, os conceitos necessários para a definição de redes de Petri Coloridas utilizada como base para RPOO.

Com relação às inscrições, portanto, para as definições encontradas no restante do trabalho, devemos assumir[1]:

- $\Sigma = \langle \mathcal{S}, \mathcal{O} \rangle$ é uma assinatura, na qual:
  - $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ é o conjunto finito de sortes (nomes de conjuntos);

[1] As definições detalhadas com relação a esta abordagem encontram-se no Apêndice A.

– $\mathcal{O} = \{o_1, o_2, \ldots, o_n\}$ é o conjunto finito de símbolos operacionais;

- $\mathcal{V}_\Sigma = V_{S_1} \cup V_{S_2} \cup \cdots \cup V_{S_n}$ é um conjunto de variáveis de Σ, formado pela união disjunta de conjuntos de variáveis indexados por $\mathcal{S}$, de modo a que cada variável de $\mathcal{V}_\Sigma$ esteja associada a um sorte S_i (seja do tipo S_i);

- $T_\Sigma = T_{S_1} \cup T_{S_2} \cup \cdots \cup T_{S_n}$ é o conjunto dos termos de Σ, considerando $\mathcal{V}_\Sigma$, formado pela união disjunta de conjuntos de termos indexados por $\mathcal{S}$, sendo que cada termo de T_Σ está associado a um sorte S_i;

- E_Σ é o conjunto de todas as equações sobre Σ;

Nas redes de Petri Coloridas podemos identificar quatro tipos de inscrições:

Domínios dos Lugares: também chamados de conjuntos de cores dos lugares, determinam os domínios aos quais devem pertencer as fichas contidas em cada lugar;

Guardas: associadas às transições, determinam condições que devem ser satisfeitas para que estas possam estar habilitadas[2];

Expressões dos Arcos: equivalentes aos pesos dos arcos das redes Lugar/Transição, determinam as fichas que são removidas dos (ou adicionadas aos) lugares associados, na ocorrência dos disparos;

Inicializações: associadas aos lugares, determinam quais fichas estarão presentes nestes lugares no seu estado inicial.

Utilizamos o conceito de multi-conjuntos para as definições das inscrições de uma rede de Petri Colorida. A família de funções $[X \to I\!N]$, chamada de multi-conjuntos de X, na qual $I\!N$ é o conjunto dos números naturais, é notada por X^{mc}. Intuitivamente, um multi-conjunto pode ser entendido como um conjunto no qual os elementos podem aparecer repetidos. Formalmente, podemos definir as inscrições em uma rede de Petri Colorida como segue [JENSEN, 1992].

[2]Veremos mais adiante que o conceito de habilitação nas redes de Petri Coloridas não está associado às transições isoladas.

Definição 2.2 (Inscrições de uma Rede Colorida) *Seja $N = \langle P, T, F \rangle$ uma estrutura de rede. As inscrições para esta estrutura, baseadas em Σ, compreendem a tupla $\langle c, g, e, i \rangle$, na qual:*

- $c : P \rightarrow \mathcal{S}$ *é uma função de cores, que determina os domínios dos lugares;*
- $g : T \rightarrow E_\Sigma$ *é uma função de guardas, que associa uma transição a uma equação sobre Σ (chamada guarda);*
- $e : F \rightarrow T_\Sigma^{mc}$ *é uma função de expressões de arcos, tal que:*
  - *se $f = \langle p, t \rangle \in F$ ou $f = \langle t, p \rangle \in F$ e $c(p) = S_i$ então $e(f) \in T_{S_i}^{mc}$;*
- $i : P \rightarrow T_\Sigma^{mc}$ *é uma função de inicialização para os lugares, tal que:*
  - *se $p \in P$ e $c(p) = S_i$ então $i(p) \in T_{S_i}^{mc}$.*

Na verdade, na Definição 2.2, a função de cores (c) associa cada lugar a um nome de conjunto (sorte) e não a um conjunto propriamente, como na definição original [JENSEN, 1992]. Como condições são na verdade expressões booleanas, utilizamos equações para representar a guarda de uma transição. Como veremos mais à frente, a avaliação destas equações é considerada para determinar a habilitação das transições a que estão associadas. As expressões dos arcos associam a cada arco um multi-conjunto de termos que será avaliado em um multi-conjunto de elementos do tipo do lugar associado ao arco. A inicialização associa aos lugares multi-conjuntos de elementos dos seus tipos. Assim, podemos ter associado a um lugar mais de uma ocorrência de um mesmo elemento. Para tanto, as expressões dos arcos e a marcação inicial devem respeitar as cores dos lugares associados.

A definição apresentada a seguir estabelece uma rede de Petri Colorida como uma entidade sintática, uma vez que é baseada em uma assinatura (Σ). Os elementos semânticos associados à rede (fichas e marcações) devem, portanto, estar relacionados a uma Σ-álgebra[3].

Definição 2.3 (Rede de Petri Colorida) *Uma rede de Petri Colorida é uma tripla $\langle \Sigma, N, I \rangle$, na qual:*

[3]Definição no Apêndice A.

- $N = \langle P, T, F \rangle$ *é uma estrutura de rede de Petri;*

- $I = \langle c, g, e, i \rangle$ *é uma inscrição para N, baseada na assinatura* Σ.

Na Figura 2.2 temos um exemplo de uma rede de Petri Colorida. Podemos notar que a estrutura é semelhante à estrutura do exemplo da Figura 2.1 (página 9). Conforme a Definição 2.3, a estrutura desta rede é composta por:

- $P = \{P1, P2, P3\}$

- $T = \{T1, T2\}$

- $F = \{(P1, T1), (P2, T1), (P2, T2), (P3, T2), (T1, P3), (T2, P1), (T2, P2)\}$

As inscrições na Figura 2.2 são representadas graficamente como segue. Os nomes dos domínios dos lugares começam por uma letra maiúscula e são grafados em itálico. Assim, temos os lugares $P1$ e $P3$ com o domínio $Cor1$ e o lugar $P2$ com o domínio $Cor2$. As guardas estão próximas às transições e são descritas entre colchetes. Embora as guardas sejam definidas formalmente como equações da assinatura, utilizamos uma grafia mais intuitiva para expressões booleanas. Neste exemplo, temos portanto somente uma guarda $(y = t)$, associada à transição $T1$. As expressões estão posicionadas próximas aos respectivos arcos. As inicializações estão sublinhadas e próximas aos respectivos lugares. Quando a inicialização de um lugar denotar 0 (zero) fichas, será omitida. A descrição de multi-conjuntos é bastante intuitiva. Por exemplo, a inicialização do lugar $P1$ indica que no estado inicial este lugar terá duas fichas com o valor a e duas com o valor b. Por sua vez, o lugar $P2$ será inicializado com uma ficha com o valor t e outra com o valor s.

Na parte inferior direita da Figura 2.2 temos o *nó de declaração* da rede. Temos aí declarados os domínios a que estão associadas as cores (tipos), bem como os tipos das variáveis utilizadas.

Chamamos de *variáveis da transição* o conjunto composto pelas variáveis presentes nas expressões dos arcos conectados a uma transição (arcos de entrada e de saída) e pelas variáveis presentes nas guardas. Uma *ligação*[4] relacionada a uma transição pode

[4]Tradução para *binding*.

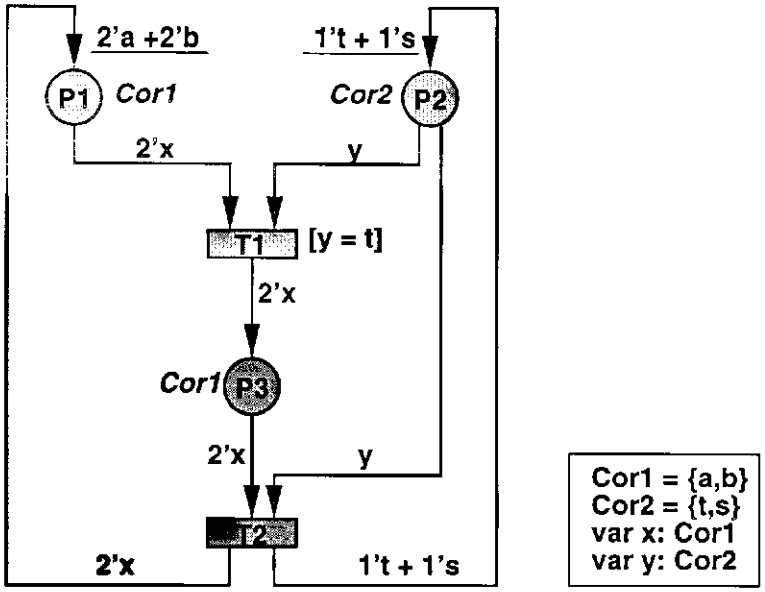

Figura 2.2: Exemplo de rede de Petri Colorida

ser compreendida intuitivamente como uma substituição das variáveis (desta transição) por valores dos seus respectivos domínios, desde que as guardas da transição fiquem válidas. Por exemplo, na Figura 2.2, temos que as variáveis da transição $T1$ são x e y. Considerando os tipos destas variáveis, temos que as ligações possíveis para a transição $T1$ são $\langle x = a, y = t\rangle$ e $\langle x = b, y = t\rangle$. As substituições $\langle x = a, y = s\rangle$ e $\langle x = b, y = s\rangle$ não são ligações para $T1$, pois invalidam a guarda $[y = t]$.

Um *elemento de ligação* é composto por uma transição e por uma ligação para esta. Dizemos que um elemento de ligação está habilitado se existem, nos lugares de entrada da transição respectiva, as fichas correspondentes às expressões dos arcos de entrada (considerando a substituição relativa das variáveis). Na Figura 2.2, para a marcação inicial, os elementos de ligação $\langle T1, \langle x = a, y = t\rangle\rangle$ e $\langle T1, \langle x = b, y = t\rangle\rangle$ estão habilitados. Um elemento de ligação habilitado *pode* disparar. O disparo de um elemento de ligação remove fichas dos lugares de entrada e acrescenta outras aos lugares de saída, relativas às avaliações dos respectivos arcos. No exemplo da Figura 2.2, para a marcação inicial, o disparo do elemento de ligação $\langle T1, \langle x = a, y = t\rangle\rangle$ remove duas fichas com valor a do lugar $P1$ (equivalente à avaliação da expressão $2'x$, do arco $(P1, T1)$) e uma ficha com valor t do lugar $P2$ (equivalente à avaliação da expressão y, do arco $(P2, T1)$) e acrescenta duas fichas com valor a ao lugar $P3$ (equivalente à

avaliação da expressão $2'x$, do arco $(T1, P3)$).

2.1.2 Redes de Petri e Orientação a Objetos

Apesar de possibilitar a construção de modelos mais compactos que os obtidos com as redes clássicas, a utilização de redes de Petri Coloridas pode ainda resultar em modelos difíceis de compreender e manipular (manter e estender). Nota-se que tais modelos são, muitas vezes, desnecessariamente complexos ou não descrevem satisfatoriamente os sistemas considerados [CHRISTENSEN & HANSEN, 1993]. Além disto, não existe um suporte adequado à reutilização de partes dos modelos. Um caminho a ser adotado com o objetivo de controlar a complexidade é dotar a notação da capacidade de modularização dos modelos, baseada em mecanismos de abstração significativos.

Exemplos importantes de mecanismos de abstração são aqueles que permitem a descrição de sistemas segundo níveis de detalhamento (ou abstração). Através de mecanismos desta natureza, os detalhes da descrição são introduzidos a cada novo nível, que (a princípio) pode ser compreendido independente dos demais [BOOCH, 1994].

Neste sentido, por exemplo, foram idealizadas as redes de Petri Coloridas Hierárquicas (HCPN - *Hierarchical Coloured Petri Nets*) [JENSEN, 1992]. A idéia nas HCPNs é possibilitar associar a uma transição (chamada *transição de substituição*) uma rede de Petri Colorida mais complexa, que seria o seu detalhamento. Esta proposta mostra-se contudo limitada, pois a abstração é garantida apenas a nível estrutural. Nas CPNs hierárquicas, só faz sentido o comportamento de todo o sistema, estudado a partir da rede plana (na qual as transições de substituição são substituídas por suas respectivas redes), sem níveis hierárquicos [LAKOS, 1997]. Seria desejável que as sub-redes relativas a cada nível hierárquico tivessem comportamento semelhante a uma CPN e, deste modo, fosse possível efetuar análise e verificação sobre elas, de forma isolada [LAKOS, 1997].

Além disso, devemos considerar que existem outras formas de abstração, que ajudam a capturar aspectos relevantes dos sistemas e a lidar com a complexidade, que não são hierárquicas.

A Orientação a Objetos oferece um conjunto de mecanismos de abstração que pretende representar estruturas importantes utilizadas pelo homem nos processos de com-

preensão. Assim, através da classificação, agregação e herança, entre outros, pretende-se decompor um sistema de forma natural para o entendimento humano [WEGNER, 1987].

Nesse contexto, algumas experiências têm sido realizadas com o objetivo de aplicar às redes de Petri de Alto Nível as estratégias de tratamento de complexidade da Orientação a Objetos. Estas experiências têm resultado em notações híbridas, baseadas nas duas teorias. Pretendendo compreender melhor tais notações, Bastide agrupou-as em duas categorias: "objetos dentro de redes de Petri" e "redes de Petri dentro de objetos" [BASTIDE, 1995]. Na primeira categoria, as fichas da rede podem ser objetos. Na segunda, as redes de Petri modelam os métodos responsáveis pelo comportamento de cada objeto.

Além de propiciar modelos melhor estruturados, os conceitos de Orientação a Objetos ainda potencializam a possibilidade de reutilização de trechos de descrição em outros contextos que não aqueles para os quais foram inicialmente planejados.

Como principais modelos de redes de Petri associados a orientação a objetos, podemos citar: as redes PROT [BALDASSARI ET AL., 1989] e sua ferramenta CASE PROTOB; os modelos POT e POP [ENGELFRIET ET AL., 1990]; as redes OBJSA [BATTISTON & DE CINDIO, 1993]; as redes cooperativas [BASTIDE ET AL., 1996]; as extensões OPN [LAKOS, 1995a] e LOOPN [LAKOS & KEEN, 1991]; CO-OPN [BUCHS & GUELFI, 1991]; as redes G-Nets [DENG ET AL., 1993; PERKUSICH & DE FIGUEIREDO, 1997].

Não é objetivo deste trabalho desenvolver uma análise exaustiva desses modelos. Um tal estudo pode ser encontrado em [GUERRERO, 1998a].

Como uma outra ferramenta que incorpora alguns conceitos do paradigma da Orientação a Objetos à teoria de redes de Petri, G-CPN foi proposta tendo como foco a especificação e modelagem de sistemas distribuídos de software [GUERRERO, 1997; GUERERRO ET AL., 1998; GUERRERO ET AL., 1997]. G-CPN incorpora os conceitos de orientação a objetos presentes nas redes G-Nets ao formalismo de redes de Petri Coloridas (CPN). G-CPN apresenta a capacidade de modelar sistemas através de módulos fracamente acoplados, a partir dos quais a semântica do sistema completo pode ser deduzida. Desta forma, os sistemas podem ser especificados de maneira incremental

[GUERRERO, 1997].

2.2 RPOO - Rede de Petri Orientada a Objetos

Nesta seção, apresentamos a notação denominada RPOO, como resultado da evolução de G-CPN.

O modelo conceitual de computação que norteia a notação é influenciado pela teoria de *Actors* [AGHA, 1986]. Assim, um sistema é concebido como uma colcção de objetos concorrentes entre si, que se comunicam através da troca assíncrona de mensagens. Cada objeto possui um identificador único no sistema, que é utilizado para destinar as mensagens. A coleção de objetos que caracteriza um sistema pode variar dinamicamente, com objetos podendo ser criados ou destruídos. A relação de conhecimento entre objetos de um sistema (objeto *a conhece* objeto *b*), também chamada *topologia de interconexão*, também é dinâmica.

Os objetos são agrupados em classes e encapsulam, além de estado e comportamento, linhas de controle. Isto implica na possibilidade de comportamento interno concorrente.

A especificação de um sistema em RPOO é composta da descrição das classes de todos os objetos que podem participar do sistema, acompanhada da declaração dos objetos iniciais. Este conjunto de classes deve atender a restrições (descritas mais à frente) que o caracterize como um *conjunto integrado de classes*.

2.2.1 Classes RPOO

A descrição de uma classe RPOO é composta de uma interface e um corpo. A interface pode ser entendida como uma abstração dos objetos da classe, considerando-se o que pode estar *acessível* a outros objetos. O corpo é a descrição do comportamento dos objetos da classe.

Em RPOO, uma interface é composta por um conjunto de nomes de atributos (que vão caracterizar o estado dos objetos) e métodos, seguidos de seus tipos[5]. O corpo de

[5]Na definição formal apresentada neste trabalho, são utilizados nomes de conjuntos (*sortes*) para especificar os tipos dos atributos e métodos.

uma classe RPOO, por sua vez, é formado por uma rede de Petri Colorida.

Para os conceitos definidos a partir deste ponto, devemos fazer as seguintes considerações:

- $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ é um conjunto finito de nomes de classes;
- $\mathcal{V}_\mathcal{C} = V_{C_1} \cup V_{C_2} \cup \cdots \cup V_{C_n}$ é um conjunto de variáveis indexadas por $\mathcal{C}$;
- $\Sigma' = \langle \mathcal{S}', \mathcal{O} \rangle$ é uma assinatura, na qual:
  - $\mathcal{S}' = \mathcal{S} \cup \mathcal{C}$ é a união de um conjunto finito de sortes (nomes de conjuntos) com o conjunto de nomes de classes;
- $\mathcal{M}$ é um conjunto de nomes de mensagens.

Ao basearmos as definições a seguir em Σ', tornamos possível a manipulação dos identificadores dos objetos nas inscrições relativas a uma classe RPOO.

Definição 2.4 (Interface de uma Classe) *Uma interface baseada em Σ' é uma tripla $\langle A, M, \tau \rangle$, na qual:*

- *A é um conjunto finito de nomes de atributos;*
- *M é um conjunto de nomes de métodos;*
- *$\tau : A \cup M \rightarrow \mathcal{S}$ é uma aplicação que a cada nome de atributo e de método faz corresponder um sorte;*
- *A e M são disjuntos, ou seja, $A \cap M = \emptyset$.*

É importante notarmos, pela definição anterior, que os métodos em RPOO não retornam qualquer valor. O sorte associado a cada um dos métodos refere-se ao tipo do seu argumento. Um argumento associado a um sorte composto (combinação de outros sortes) pode ser utilizado para descrever um método que recebe uma lista de argumentos.

Definição 2.5 (Classe) *Uma sêxtupla $\langle \mathcal{I}, \mathcal{N}, l^a, l^m, \eta, \mu \rangle$ é uma classe baseada em Σ' se e somente se as seguintes condições são satisfeitas:*

- $\mathcal{I} = \langle A, M, \tau \rangle$ *é uma interface baseada em* Σ';
- $\mathcal{N} = \langle \Sigma', N, I \rangle$ *é uma rede de Petri Colorida, denominada corpo da classe (considere* $N = \langle P, T, F \rangle$ *e* $I = \langle c, g, e, i \rangle$*);*
- $l^a : A \rightarrow P$ *é uma aplicação que a cada atributo associa um lugar da estrutura, respeitando os tipos, ou seja, para todo atributo* $a \in A$*, se* $p = l^a(a)$ *então* $c(p) = \tau(a)$;
- $l^m : M \rightarrow T \times \mathcal{V}_{\Sigma'}$ *é uma aplicação que a cada método associa um conjunto de elementos de ativação que respeita os tipos, ou seja, se* $\langle t, v \rangle \in l^m(m)$ *e* $v \in \mathcal{V}_{S_i}$ *então* $\tau(m) = S_i$;
- $\eta : T \rightarrow 2^{\mathcal{V}_C \times \mathcal{C}}$ *é uma aplicação que a cada transição associa um conjunto de inscrições de criação de objetos, tais que se* $\langle v, C_i \rangle \in \eta(t)$ *então* $v \in V_{C_i}$;
- $\mu : T \rightarrow 2^{\mathcal{V}_C \times \mathcal{M} \times T_{\Sigma'}}$ *é uma aplicação que a cada transição associa um conjunto de inscrições de mensagens.*

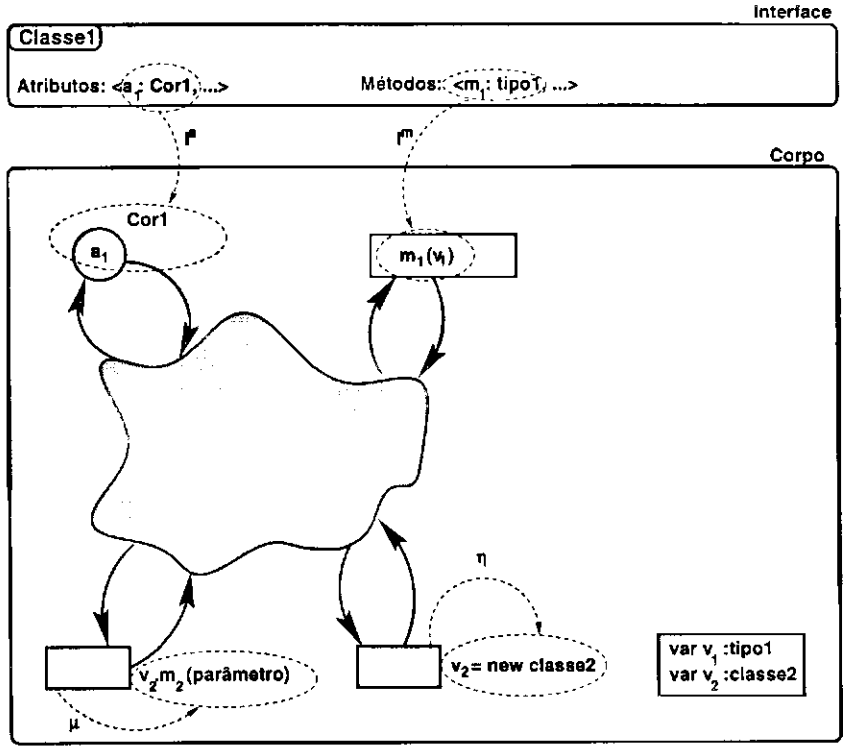

Figura 2.3: Classe RPOO

A Figura 2.3 ilustra o esquema de uma classe RPOO. Cada atributo na interface é relacionado a um lugar no corpo e seu tipo deve coincidir com o conjunto de cores do lugar. Esta relação, definida pela aplicação l^a, é descrita concedendo-se o nome do atributo ao respectivo lugar. Os métodos, por sua vez, são associados, cada um, a uma transição (com o mesmo nome), chamada *transição de ativação* e uma variável desta. O tipo desta variável deve coincidir com o tipo do método. Esta relação é definida pela aplicação l^m.

As inscrições de mensagens associadas às transições e definidas pela aplicação μ, determinam os pontos de *invocação* de métodos de outros objetos. Cada inscrição de mensagem é composta de: uma variável de classe, que vai ser ligada a um identificador de objeto (da respectiva classe); um nome de método, indicando qual o método a ser invocado; e um termo que define o parâmetro, ou informação, a ser transmitido com a mensagem. O fato do objeto receptor e do conteúdo das mensagens poderem ser descritos por variáveis faz com que eles possam ser determinados apenas no momento dos disparos.

A comunicação entre objetos se dá através da invocação/ativação de métodos. A invocação de um método acontece com o disparo de uma transição que possui uma inscrição de mensagem associada. Tal invocação disponibiliza para o objeto invocado uma ficha tendo como valor a informação a ser transmitida. Esta ficha será então ligada à variável associada ao método invocado. O equivalente a uma invocação numa rede de Petri Colorida convencional é ilustrado pela Figura 2.4. A ficha com a informação seria depositada em um lugar do qual parte um arco para a transição de ativação, cuja expressão é justamente composta pela variável relacionada ao método. Para tanto, o tipo da variável deve ser o mesmo do método. Consideramos também que o *ambiente* responsável por fazer a ligação entre uma invocação e uma ativação é confiável, ou seja, uma mensagem invocada *com certeza* habilitará as condições externas para a respectiva ativação no objeto destino.

As inscrições de criação de um novo objeto, definidas por η, são compostas de uma variável e de um nome de classe. O tipo da variável deve ser igual ao nome da classe. Uma vez disparada a transição equivalente, um novo objeto da classe determinada é criado e seu nome (identificador) é ligado à variável da classe. É interessante notarmos

que a possibilidade de criação de novos objetos por outros, aliada à possibilidade de descrever o receptor de uma mensagem através de uma variável, faz com que possamos descrever sistemas nos quais a topologia de interconexão é dinâmica.

O restante do comportamento da rede é semelhante ao já apresentado para uma rede de Petri Colorida.

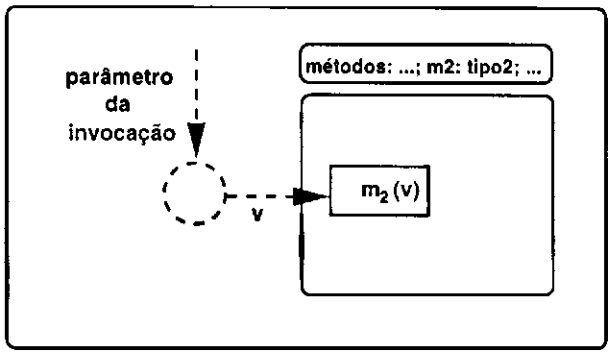

Figura 2.4: Semântica de uma invocação em RPOO

2.2.2 Sistemas RPOO

Para compor uma especificação de sistema em RPOO, um determinado conjunto de classes precisa atender a certas restrições, impostas com o objetivo de manter a consistência do modelo. Tais restrições devem levar em consideração dois aspectos:

1. para toda classe, cada inscrição de mensagem deve respeitar a interface da classe referenciada, ou seja, tal classe deve possuir o método descrito na mensagem e o tipo deste método deve ser o mesmo do termo passado como parâmetro;

2. as inscrições de criação de objetos devem referenciar classes que fazem parte da especificação do sistema.

Um tal conjunto de classes é denominado *conjunto integrado de classes.* Dado um conjunto de classes, a verificação, se este constitui um conjunto integrado, pode ser realizada estaticamente.

Um modelo de um sistema é então formado por um conjunto integrado de classes e uma declaração dos objetos que compõem a configuração inicial do sistema.

Vamos exemplificar um sistema RPOO através de uma possível solução orientada a objetos para o conhecido problema do *jantar dos filósofos*. Este problema foi inicialmente introduzido por Dijkstra em [DIJKSTRA, 1971]. Nossa solução é composta por duas classes de objetos: *garfos* e *filósofos*. Como sabemos deste problema, os filósofos são dispostos em uma mesa circular, devendo compartilhar os garfos com os vizinhos. Assim, um filósofo compartilha o garfo à sua direita com o vizinho da direita (para o qual se trata do garfo da esquerda) e o garfo da esquerda com o vizinho da esquerda (para o qual se trata do garfo da direita).

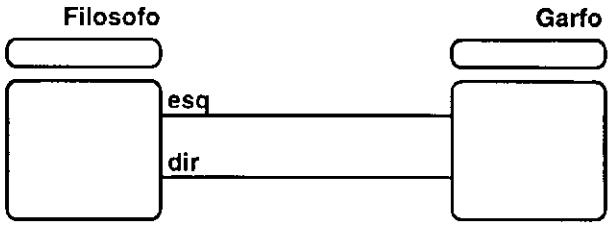

Figura 2.5: Relacionamentos entre a classe dos filósofos e a classe dos garfos

Na Figura 2.5 ilustramos os relacionamentos que estamos considerando entre as classes. Deste modo, cada filósofo *conhece* dois garfos: o da esquerda (esq) e o da direita (dir). Os garfos não precisam *conhecer* nenhum filósofo. Por *conhecer*, estamos querendo dizer, por exemplo, que os filósofos devem guardar os identificadores dos garfos vizinhos. Com relação a RPOO, estes identificadores devem ser mantidos em lugares específicos. Consideramos então que fazem parte dos atributos da classe dos filósofos. Para simplificar as descrições, omitimos a representação dos lugares responsáveis por estabelecer os relacionamentos entre classes. A simples menção de uma variável com o nome do atributo indica que esta sempre será ligada ao conteúdo do lugar omitido (ver Figura 2.6).

As descrições completas das classes dos garfos e dos filósofos encontram-se ilustradas respectivamente na Figura 2.7 e na Figura 2.8.

Um garfo deve sempre estar em um entre dois estados: *livre* ou *alocado*. Estes dois estados são descritos na rede pelos dois lugares de mesmo nome. A cor destes lugares é E, o que indica que as fichas contidas neles (todas iguais a e) não carregam informação, apenas importando a sua presença ou ausência [JENSEN, 1992]. Inicialmente um garfo está no estado *livre*, indicando que não está sendo utilizado por nenhum filósofo. Isto é

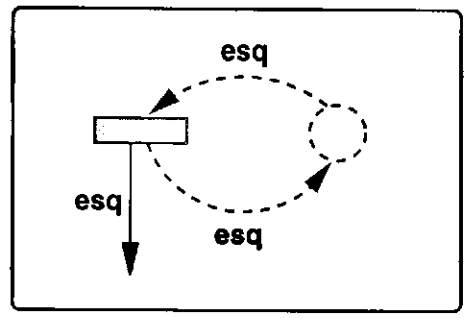

Figura 2.6: Lugares que estabelecem relacionamentos entre classes

descrito pela marcação inicial do lugar *livre*: $1'e$. Um garfo pode estar apto a receber dois tipos de mensagem: *aloca* e *libera*. Deste modo, estando o garfo livre, o envio de uma mensagem *aloca* para ele habilita o disparo do evento associado. Ocorrendo o disparo, o garfo envia uma mensagem de volta ao filósofo confirmando a alocação. O garfo passa então para o estado *alocado*. O envio de uma mensagem *libera* para ele habilita o disparo do evento associado, cujo disparo leva o garfo de volta ao estado *livre*.

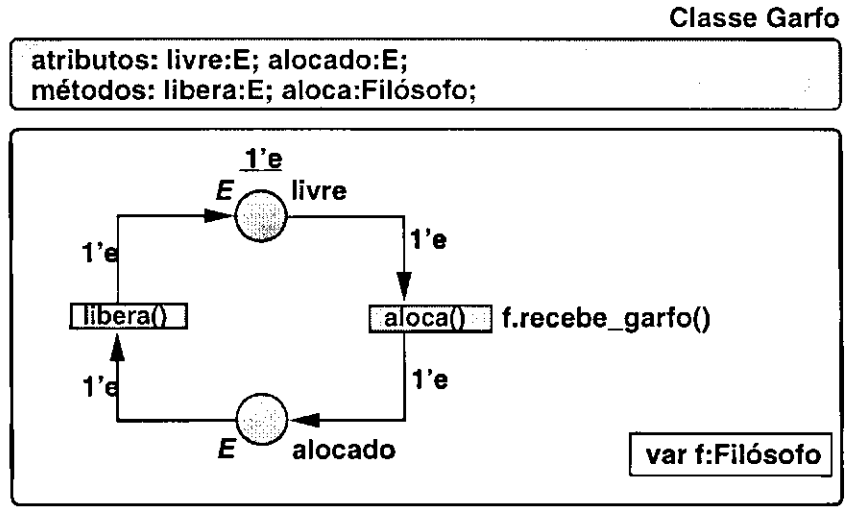

Figura 2.7: Descrição da classe dos garfos

Um filósofo, por sua vez, pode estar nos estados *pensando* ou *comendo*. Os lugares que representam estes estados também comportam fichas sem informação. A presença de uma única ficha com valor e em um desses lugares indica o estado do filósofo. Inicialmente um filósofo está no estado *pensando*. Para passar para o estado *comendo*, um filósofo deve conseguir alocar os garfos à sua esquerda e direita. As variáveis *esq* e *dir*

são sempre ligadas aos identificadores destes dois garfos. Portanto, uma vez no estado *pensando*, o evento *aloca_garfo* estará habilitado. O seu disparo faz com que sejam enviadas mensagens de alocação aos garfos. O disparo do evento *aloca_garfo* também retira a ficha do lugar *pensando* e adiciona duas fichas ao lugar *garfos_alocados*. As mensagens de confirmação de alocação enviadas de volta pelo garfo correspondem ao método *recebe_garfo*. Para cada confirmação aceita, uma ficha (sem informação) é depositada no lugar *garfos_recebidos*. Duas fichas neste lugar indicam que os dois garfos foram alocados e então habilitam o filósofo a comer. Isto se dá a partir do disparo do evento *começar_comer*, que remove as duas confirmações do lugar *garfos_recebidos* e adiciona uma no lugar *comendo*. O disparo do evento *libera_garfo* encerra a atividade de comer dos filósofos. As mensagens de liberação enviadas aos garfos não necessitam confirmação. Como a nossa solução introduz alguns estágios intermediários entre pensar e comer, podemos considerar que o filósofo está comendo quando o lugar *comendo* contém uma ficha. Caso contrário, ele está pensando.

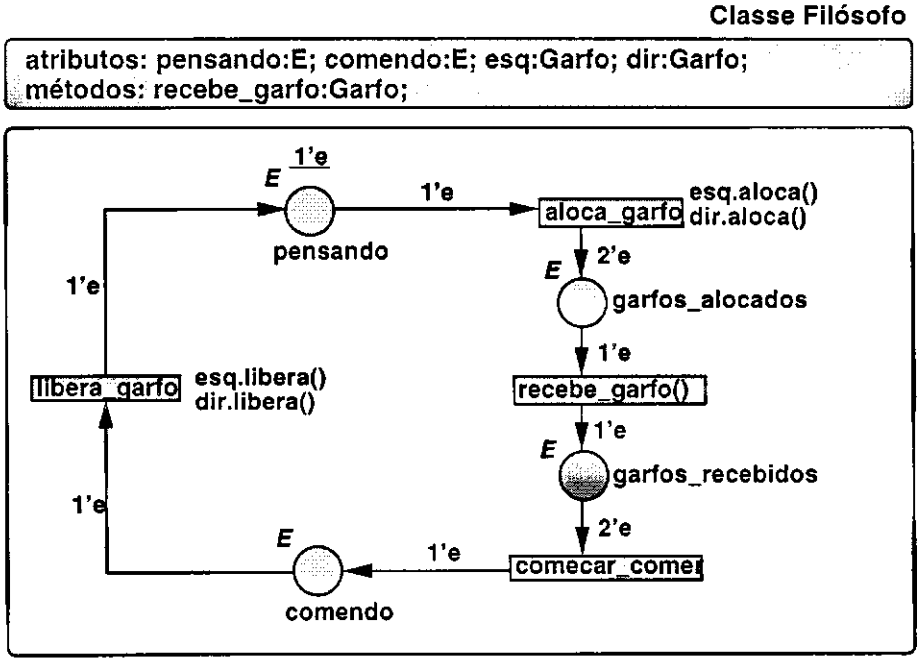

Figura 2.8: Descrição da classe dos filósofos

Podemos notar que as classes dos garfos e dos filósofos , conforme descritas nas Figuras 2.7 e 2.8, compõem um conjunto integrado de classes. Para completar a descrição do sistema, precisamos da declaração do seu estado inicial, formado pelos objetos que

compõem a primeira configuração. Por exemplo, um jantar com 5 (cinco) filósofos é formado por 5 (cinco) objetos da classe dos filósofos e (5) cinco objetos da classe garfo. Cada filósofo deve estar associado a um garfo à sua esquerda e outro à sua direita. Estas associações devem indicar uma disposição circular entre os objetos. Como a sintaxe para a declaração dos objetos não está definida, preferimos omitir qualquer declaração precisa da configuração inicial.

Capítulo 3

Aspectos de Herança

Neste capítulo, tratamos de aspectos relacionados ao conceito de herança que interessam a uma linguagem de especificação formal, caso de RPOO. É comum encontrarmos na literatura certa discordância na utilização de certos termos centrais para a compreensão desse conceito. Na verdade, a principal discordância diz respeito à noção do próprio termo *herança*, que ora é compreendido como um relacionamento entre classes, ora como um mecanismo eficiente de reutilização de descrições. Procuramos portanto aqui abordar estes aspectos, na tentativa de constituir um conjunto básico de conhecimentos necessários para suportar herança em RPOO.

O capítulo é organizado como segue. Na Seção 3.1 situamos herança como um conceito central em Orientação a Objetos, descrevendo os principais aspectos relacionados. Na Seção 3.2 discutimos os critérios de compatibilidade utilizados para restringir as hierarquias de herança. Na Seção 3.4 tratamos da possibilidade de aproximar a teoria de tipos de dados às classes. Na Seção 3.3 discutimos a herança como um mecanismo de modificação incremental com finalidades práticas de aumento de produtividade.

3.1 Sobre Herança

A Orientação a Objetos como paradigma para linguagens de programação teve início de fato no final dos anos 60 com o surgimento de Simula, uma linguagem que tinha como objetivo inicial a simulação de fenômenos do mundo real [TAIVALSAARI, 1996]. Como tal, Simula foi projetada de modo a que suas primitivas tivessem uma forte

ligação com os conceitos perceptíveis das realidades que se desejava capturar. Conceitos dos domínios de problemas passam então a ser capturados pela idéia de classes de objetos, estes incorporando estado (dados) e comportamento. Um programa orientado a objetos é então uma coleção de objetos, que por sua vez representam entidades reais ou imaginárias, cuja execução é o resultado da interação entre os seus objetos. Na verdade, estes conceitos, proporcionando a solução de problemas através de princípios como o encapsulamento (dados e processo) e a modularização, convergem com as idéias de máxima coesão e fraco acoplamento que dominaram as técnicas de desenvolvimento de programas na década de 60.

O paradigma de Orientação a Objetos[1] acabou por se estender para outras notações utilizadas nos processos de desenvolvimento de sistemas de software, constituindo assim metodologias de desenvolvimento orientadas a objeto (ex. Booch [BOOCH, 1994] e OMT [RUMBAUGH ET AL., 1991]). Duas das principais características destas metodologias são:

1. Maneira uniforme de tratar o problema ao longo de todo o desenvolvimento, causando uma diminuição do esforço nas passagens de fases (análise → projeto → implementação). O objetivo neste ponto é a estruturação de uma forma natural de proceder desde a concepção do problema até a solução implementada;

2. Suporte aos principais mecanismos de abstração utilizados pelo homem para capturar domínios de problemas: classificação/instanciação, especialização/generalização e agregação, entre outros.

Como um dos mais importantes mecanismos de abstração para entender um domínio de aplicação, a *especialização conceitual* permite relacionar conceitos aos pares através do relacionamento intuitivo *é-um*[2]. Deste modo, quando relacionamos os conceitos *macaco* e *mamífero* da seguinte forma:

um macaco é-um mamífero

[1]Segundo Wegner, uma notação é dita orientada a objeto se suporta os conceitos de objeto, classe e herança [WEGNER, 1987]. Embora bastante aceita, podemos encontrar outras concepções em [BOOCH, 1994] e [RUMBAUGH ET AL., 1991].

[2]Em [BRACHMAN, 1983], encontramos catalogadas algumas outras relações semânticas que têm sido representadas pelo conectivo *é-um*.

queremos expressar que se um indivíduo (elemento) é uma instância de *macaco* (corresponde ao conceito), então ele também é uma instância de *mamífero* e, dependendo do nível de abstração, pode ser visto (manipulado, entendido) como um ou como outro.

A especialização conceitual estabelece uma hierarquia (relação de ordem) parcial entre conceitos. Em tal hierarquia, as propriedades comuns podem ser associadas aos conceitos de mais alto nível e então compartilhadas por seus descendentes, como ilustrado na Figura 3.1. Assim, as propriedades atribuídas aos mamíferos são ainda propriedades dos macacos.

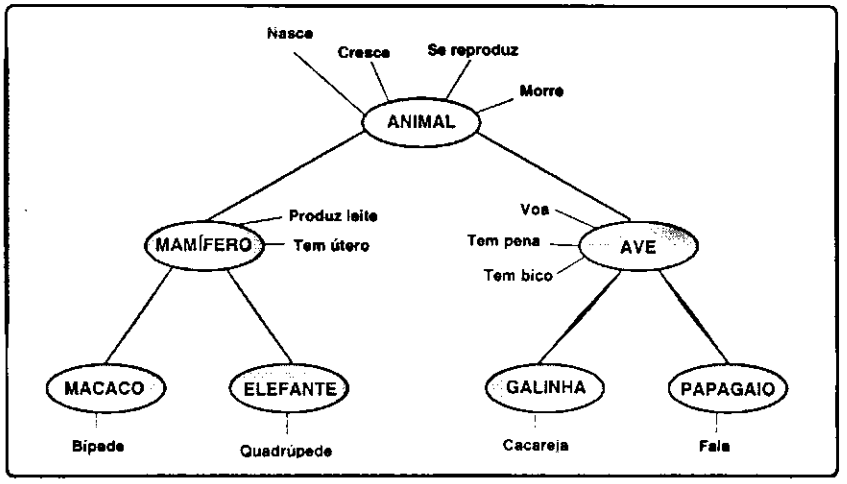

Figura 3.1: Especialização conceitual (as ligações entre os nós representam o relacionamento *é-um*)

A motivação inicial para a introdução do termo *herança* no contexto de Orientação a Objetos foi a criação de um mecanismo para capturar/representar a especialização conceitual [TAIVALSAARI, 1996; MADSEN ET AL., 1990]. Porém, *herança* pode significar também o *processo* de recepção de propriedades de um ancestral (antecedente). Herança pode ser então utilizado como um mecanismo eficiente de reutilização. Assim, pedaços de descrições de classes podem ser herdados por suas classes descendentes (subclasses), sem que existam quaisquer garantias a respeito da correspondência do relacionamento obtido com a especialização conceitual.

De fato, razões de natureza pragmática têm motivado a maioria das implementações do conceito de herança. A reutilização de descrições tem sido, na verdade, uma das grandes promotoras da Orientação a Objetos e tem impulsionado a sua di-

vulgação, devido às vantagens práticas oferecidas. Desta forma, o conceito de herança tem sido amplamente utilizado, difundido e suportado como uma forma eficiente de reaproveitamento.

Podemos, portanto, reconhecer os dois principais aspectos relacionados ao conceito de herança:

1. herança com fins de especificação: o conceito de herança é visto como uma relação entre entidades abstratas (classes), que pretende capturar o relacionamento *é-um*;

2. herança com fins de implementação: o conceito de herança é visto como um mecanismo de reaproveitamento de descrições.

Os dois aspectos tratados acima são naturalmente divergentes, uma vez que o enfoque dado a um deles prejudica o outro [PALSBERG & SCHWARTZBACH, 1994; TAIVALSAARI, 1996]. Tratar herança como uma representação da especialização conceitual torna muito difícil a tarefa de desenvolver mecanismos de reaproveitamento de descrições. Por sua vez, mecanismos que maximizem o reaproveitamento de descrições dificilmente respeitam o relacionamento *é-um*.

3.2 Critérios de Compatibilidade

Ignorando as divergências na sua utilização/utilidade em Orientação a Objetos, de maneira geral, a idéia de herança permite estruturar objetos (ou classes) em hierarquias, nas quais os descendentes compartilham ou reutilizam propriedades dos seus ancestrais [WEGNER, 1987]. Esta noção, conforme já visto, dá margem para a utilização do conceito de herança tanto para representar a especialização conceitual quanto para obter reaproveitamento de descrições.

Considerando como classes podem ser descritas e como pode ser determinado o comportamento de seus objetos, existem diversas propriedades que podem ser compartilhadas entre classes. Podemos agrupar estas propriedades em duas categorias:

1. propriedades sintáticas: percebidas diretamente sobre as descrições das classes;

2. propriedades semânticas: percebidas sobre alguma representação do comportamento das classes (dos objetos das classes).

Diferentes tipos de propriedades compartilhadas determinam diferentes hierarquias entre classes, que passamos a chamar de *hierarquias de herança* [WEGNER, 1987]. Uma hierarquia de herança para compartilhar descrições sintáticas, por exemplo código em um programa, não necessariamente garante o compartilhamento de comportamento. Somente sob condições específicas (que serão vistas mais à frente), preservação de propriedades estruturais podem garantir algum nível de preservação de comportamento. Desta forma, diferentes hierarquias devem ser representadas por relações diferentes. Contudo, estas distinções não são usualmente percebidas e assim a herança é algumas vezes vista como um mecanismo universal que sempre pode representar o compartilhamento de ambas as categorias de propriedades, sintáticas e semânticas. Uma possível causa para este equívoco é o uso constante de metáforas biológicas (classificação das espécies) nos textos sobre Orientação a Objetos [WEGNER, 1987; BACLAWSKI & BIPIN, 1994]. Em tal classificação, ambas as categorias de propriedades podem ser compartilhadas entre ancestrais e descendentes. Identificar, por exemplo, que *um macaco é um mamífero* (relacionamento conceitual) implica certo nível de compatibilidade genética (relacionamento estrutural).

Um *critério de compatibilidade* é caracterizado por um certo número de propriedades e pode ser sintático ou semântico, de acordo com a categoria de propriedades a que está associado. Os critérios de compatibilidade portanto estabelecem restrições para que as classes possam se relacionar, definindo assim hierarquias entre elas. Na Figura 3.2 podemos ver como diferentes critérios de compatibilidade estabelecem diferentes hierarquias entre um mesmo conjunto de classes.

Pretendendo estabelecer os níveis de compatibilidade possíveis entre classes, Wegner propõe a seguinte classificação (aqui apresentada do nível mais básico ao mais restritivo) [WEGNER, 1988]:

cancelamento: uma (sub)classe não possui obrigatoriamente qualquer nível de compatibilidade com sua superclasse, uma vez que existe total liberdade no processo de herança de descrições, através de cancelamento, redefinição e adição (tratados na Seção 3.3);

compatibilidade de nomes: uma (sub)classe preserva os nomes dos atributos e métodos

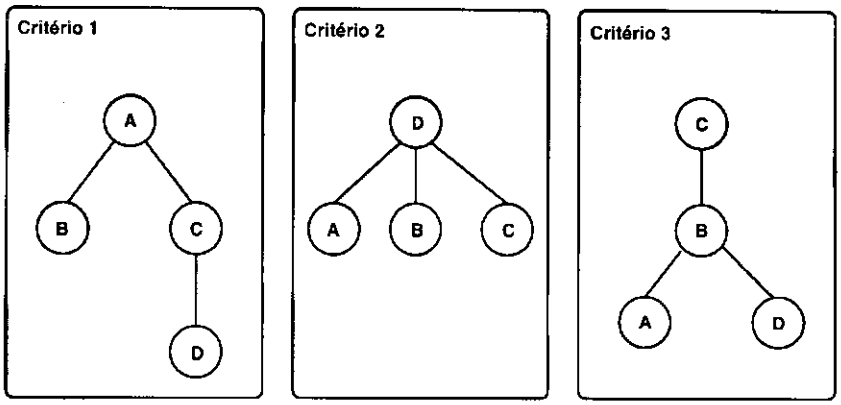

Figura 3.2: Hierarquias estabelecidas entre conjunto de classes por diferentes critérios de compatibilidade

presentes na sua superclasse, que podem contudo ser redefinidos;

compatibilidade de assinatura: a interface de uma (sub)classe preserva compatibilidade sintática completa com relação à interface de sua superclasse, sendo considerados, além dos nomes, os tipos relativos à assinatura;

compatibilidade de comportamento: uma (sub)classe preserva o comportamento de sua superclasse (de acordo com uma definição de equivalência semântica).

Para um dado critério de compatibilidade, duas abordagens podem ser consideradas:

1. Estabelecimento de um conjunto de mecanismos de modificação incremental sobre classes para gerar outras novas, preservando o critério;

2. Estabelecimento de técnicas de verificação do critério entre duas classes já descritas.

A primeira abordagem para herança tem uma preocupação pragmática e permite a descrição de novas classes de objetos a partir de outras já existentes [COOK ET AL., 1994; TAIVALSAARI, 1996; WEGNER, 1988]. Esta abordagem tem o papel de aumentar a produtividade no desenvolvimento de sistemas de software, uma vez que objetiva o aproveitamento de descrições. Contudo, quanto mais estrito é o critério de compatibilidade considerado, mais difícil é a tarefa de obter mecanismos de reutilização que

	Critérios Sintáticos	Critérios Semânticos
Mecanismos de Modificação	A	B
Verificação de Tipos	C	D

Tabela 3.1: Abordagens para herança vs. critérios de compatibilidade

o preserve [PALSBERG & SCHWARTZBACH, 1994]. Desta forma, critérios de compatibilidade baseados em propriedades sintáticas são mais apropriados neste caso. Esta abordagem é tratada com mais detalhes na Seção 3.3.

Com respeito à segunda abordagem, seu principal objetivo é aproximar o conceito de classes ao de tipos. Deste modo, uma dada noção de subclassificação (relativa a um critério de compatibilidade) passa a ser considerada como subtipificação[3]. A partir daí, torna-se possível expandir as verificações realizadas sobre os tipos em uma notação (*type checking*) para as classes. As relações entre subclassificação e subtipificação são tratadas na Seção 3.4.

Uma maneira interessante para entender os conceitos de herança discutidos, bem como suas aplicações, é obter o cruzamento dos critérios de compatibilidade com as abordagens para herança até aqui tratados. Uma síntese de tal cruzamento é apresentada na Tabela 3.1. As duas noções usualmente associadas aos mecanismos de modificação incremental e aos métodos de verificação estão representadas pelas células A e D respectivamente. As noções representadas pelas células B e C normalmente são ignoradas, mas podem assumir relevada importância em contextos específicos, como discutido mais adiante.

Devido ao fato dos mecanismos de modificação serem operadores sintáticos, é comum considerar que as hierarquias por eles construídas preservam, quando muito, compatibilidade sintática (representado por A). Porém, mecanismos de modificação que preservem propriedades semânticas (célula B) podem significar grande benefício na especificação de sistemas [WEGNER, 1988; VAN DER AALST & BASTEN, 1997]. Mecanismos que preservam compatibilidade sintática, contudo, são muito mais flexíveis que mecanismos que preservem compatibilidade semântica, promovendo portanto

[3]Tradução para *subtyping*.

muito mais a reutilização.

Os métodos de verificação, por sua vez, têm por objetivo controlar o comportamento das entidades verificadas. Por esta razão, tais métodos são normalmente associados a verificação de propriedades semânticas (célula D). Porém, em alguns casos, como considerado mais à frente, propriedades sintáticas podem ser aceitas como informações acerca do comportamento (célula C).

Neste trabalho, utilizamos o termo *herança* para designar relações parciais de ordem entre classes (tipos) de objetos, nas quais as classes de mais baixa ordem na hierarquia obtida (hierarquia de herança) são compatíveis (segundo um critério específico) com as suas classes ancestrais. Mais especificamente, denominamos *herança sintática* ou *herança semântica* dependendo da natureza do critério considerado.

3.3 Mecanismos de Modificação Incremental

A modificação incremental, como técnica, proporciona a concepção evolutiva de sistemas, na qual os elementos são definidos incrementalmente a partir de outros. Tal técnica tem se mostrado bastante eficiente não somente no desenvolvimento de sistemas de software como também em outros tipos de sistemas, como sistemas físicos e matemáticos [WEGNER, 1988]. Com relação aos sistemas de software, mecanismos de modificação incremental possibilitam a reutilização de descrições já existentes para descrever coisas novas. Um exemplo são as especificações recursivas, nas quais a solução para um problema com parâmetro n é utilizada para descrever a solução para um problema com parâmetro $n + 1$ [WEGNER, 1988].

3.3.1 Herança como Modificação Incremental

Caracterizar herança como um mecanismo de modificação incremental tem sido o principal enfoque dado na literatura e nas linguagens de programação orientadas a objetos [COOK ET AL., 1994; TAIVALSAARI, 1996; WEGNER, 1988]. Como tal, a herança pode ser definida do seguinte modo [TAIVALSAARI, 1996; WEGNER, 1988]:

$$R = P \oplus \Delta R$$

R e P são classes e ΔR é o *modificador*. Eles podem ser vistos como estruturas com um número finito de propriedades (atributos e métodos). As propriedades de ΔR podem ser ou *independentes* ou *sobrepostas* com relação às propriedades de P. As propriedades de R são portanto todas as de ΔR acrescentadas daquelas de P que não foram sobrepostas [WEGNER, 1988].

3.3.2 Auto-referências e Ligação Dinâmica

Simplesmente ter acesso a propriedades de outras classes pode ser reduzido apenas a invocações. Porém, o mecanismo de herança implica mais. Em essência, propriedades herdadas fazem mais parte das entidades que aquelas que são invocadas de outras [WEGNER, 1988].

Uma das características que diferenciam o mecanismo de herança de uma simples invocação é a *ligação dinâmica* de *auto-referências*[4] em tempo de execução ou simulação. Auto-referência é um suporte lingüístico que permite que um objeto e uma propriedade referenciados por ele somente sejam decididos dinamicamente. Tal decisão é feita de acordo com o contexto em que se deu a invocação, tornando as propriedades polimórficas [TAIVALSAARI, 1996]. Desta forma, é conseguida uma maior taxa de reutilização, evitando certas duplicações de código.

Se não forem consideradas as auto-referências e a respectiva ligação dinâmica (ou ligação tardia), o mecanismo de herança não passa de invocação [WEGNER, 1988].

3.3.3 Operações Disponíveis

Considerando a herança como um mecanismo de modificação incremental definido por $R = P \oplus \Delta R$, as seguintes operações podem ser realizadas por tal mecanismo [WEGNER, 1988; TAIVALSAARI, 1996]:

Cancelamento: através desta operação, uma subclasse pode herdar um subconjunto de propriedades da sua superclasse, ao invés do conjunto inteiro. Esta operação permite a escolha de determinadas propriedades que interessam na subclasse, através do descarte das não desejadas. Exemplo: é possível caracterizar uma

[4] Tradução para *self references*.

galinha como uma ave que não voa (a propriedade *voar* é cancelada). Na prática, o cancelamento de propriedades possibilita que se estabeleçam relacionamentos de herança com o único objetivo de aproveitar partes da descrição de uma classe, sem que esta tenha qualquer relação conceitual com a classe derivada. Exemplo: podemos estabelecer que *avião* é subclasse de *ave* só para aproveitar a propriedade *voar*, cancelando as demais. Uma tal prática, embora proporcione um alto grau de reaproveitamento, se realizada de forma descontrolada pode produzir descrições que façam pouco sentido [TAIVALSAARI, 1996].

Adição: através desta operação, uma subclasse pode possuir propriedades que não estão descritas na sua superclasse. Tal mecanismo proporciona a criação de classes, cujos objetos têm um propósito mais específico que os objetos da sua classe ancestral.

Redefinição: através desta operação, uma subclasse pode redefinir a descrição de propriedades herdadas da superclasse. Um mecanismo que inclua esta operação possibilita o detalhamento de propriedades ao longo da hierarquia de herança.

O que está em discussão aqui é o poder de expressão contra a clareza das estruturas hierárquicas resultantes. Implementações muito permissivas (que permitem cancelar, redefinir e adicionar propriedades livremente) com relação à subclassificação aumentam o poder de expressão da linguagem ao mesmo tempo em que reduzem a possibilidade de garantir algum nível de compatibilidade entre as classes associadas [WEGNER, 1987].

3.4 Classes e Tipos

Introduzimos nesta seção as noções básicas relacionadas aos conceitos *classe* e *tipo*. Os conceitos relacionados de *subclassificação* e *subtipificação*, normalmente associados ao conceito de herança, são também discutidos. Discutimos enfim as idéias envolvidas na aproximação destes dois conceitos.

3.4.1 Classes e Subclassificação

Em Orientação a Objetos, o conceito de classes serve para denominar um conjunto de objetos com a mesma estrutura interna [AMERICA & VAN DER LINDEN, 1990]. Na descrição de uma classe estão detalhados os elementos "visíveis" pelo ambiente externo aos objetos, chamados genericamente de *interface*, composta por atributos e métodos, bem como a especificação destes métodos (públicos e privados). Ao utilizarmos uma linguagem de programação orientada a objetos, esta especificação é a própria implementação dos métodos.

Chamamos subclassificação a qualquer relação de ordem parcial entre classes. Esta relação pode ser estabelecida *a priori*, através de um critério de compatibilidade, ou *a posteriori*, através de um mecanismo de modificação incremental. No primeiro caso, temos um conjunto pré-definido de classes e, para um dado critério de compatibilidade, a relação *é-compatível-com* define a subclassificação. No segundo caso, a relação de ordem é estabelecida à medida que as classes vão sendo descritas e corresponde à relação *é-definida-a-partir-de*. Neste caso, as classes relacionadas podem também respeitar um dado critério de compatibilidade. Seja portanto $\leq_s$ uma tal relação, dizemos que uma classe C_2 é subclasse de outra classe C_1 se $C_2 \leq_s C_1$.

3.4.2 Tipos e Subtipificação

O conceito de tipo de dados tem o papel de agrupar/classificar os elementos (dados), baseado no conjunto de operações permitidas sobre eles. Desta forma, em uma linguagem de programação que suporte tipos de dados, o tipo *inteiro*, por exemplo, agrupa um número finito de elementos (que representam um subconjunto finito dos números inteiros) e possuem em comum um conjunto finito de operações permitidas.

Considerando ainda uma possível relação entre os tipos, chamada subtipificação (que será tratada mais adiante), o objetivo geral dos tipos de dados é controlar o potencial uso dos elementos de dados [PALSBERG & SCHWARTZBACH, 1992]. Assim, especificar formalmente ou implementar, respeitando as operações permitidas pelos tipos de dados em uma notação, garante um certo grau de corretude[5] ao resultado. Além

[5]Tradução para *correctness*.

disto, é possível o desenvolvimento de técnicas que verificam se uma descrição obedece às restrições estabelecidas pelos tipos (*type checking*), sejam eles pré-definidos ou definidos pelo desenvolvedor. Outras vantagens, além da corretude, também atribuídas ao suporte a tipos são: legibilidade, garantias de segurança e eficiência [PALSBERG & SCHWARTZBACH, 1992; MADSEN ET AL., 1990].

Podemos entender os tipos, de maneira simplificada, como um conjunto de elementos que satisfazem um determinado predicado [PALSBERG & SCHWARTZBACH, 1992; PALSBERG & SCHWARTZBACH, 1994]. Deste modo, um elemento x pertence a um tipo T se satisfaz o predicado associado. Um elemento pode portanto pertencer a vários tipos, que necessariamente não possuem nenhuma relação entre si, desde que satisfaça aos respectivos predicados.

A relação de subtipificação, por sua vez, pode ser definida da seguinte forma: um tipo T_1 é *subtipo* de um tipo T_2 se todo elemento de T_1 for também elemento de T_2. Logo, a relação de subtipificação pode ser vista como uma relação de subconjunto entre conjuntos: $T_1 \subseteq T_2$.

Mais especificamente, com relação aos tipos de dados, o predicado que os define deve garantir que cada tipo seja uma coleção de elementos que compartilham o mesmo comportamento observável externamente [AMERICA & VAN DER LINDEN, 1990]. Desta forma, a subtipificação deve garantir que elementos de um (sub)tipo podem sempre ser manipulados como elementos de seus supertipos [CARDELLI & ABADI, 1996]. Wegner chama esta característica de *princípio da substituição*[6] (visto no Capítulo 5).

3.4.3 Subclassificação como Subtipificação

Considerando os benefícios de suportar tipos de dados em uma notação, parece natural o esforço em expandir a teoria dos tipos a classes de objetos [PALSBERG & SCHWARTZBACH, 1992; COOK ET AL., 1994; CARDELLI & ABADI, 1996; WEGNER, 1988; AMERICA & VAN DER LINDEN, 1990]. Neste sentido, é necessário aproximar os conceitos de subclassificação e subtipificação.

Como as definições de subclassificação e subtipificação dependem respectivamente das definições de classe e tipo, um primeiro passo é associar a cada classe um tipo.

[6]Tradução para *principle of substutability*.

Desta forma, todos os objetos de uma classe pertencem a pelo menos este tipo básico. Uma hierarquia de subtipificação é portanto uma hierarquia de subclassificação que preserva o princípio da substituição, ou seja, uma hierarquia na qual os objetos de uma (sub)classe podem substituir objetos da sua superclasse, pois preservam o seu comportamento (observável exteriormente).

O suporte a uma subclassificação que se aproxime do conceito de subtipificação em uma notação orientada a objetos permite que o comportamento dos objetos seja detalhado ao longo do desenvolvimento [AMERICA & VAN DER LINDEN, 1990]. Uma subclasse (que nesse caso determina também um subtipo) descreve objetos com comportamento mais detalhado que os objetos da sua superclasse, mas que dependendo da abstração podem ser vistos como objetos desta. Além disto, a proteção dada pela subtipificação garante alguns invariantes com relação ao comportamento dinâmico dos sistemas especificados, como por exemplo ausência de erros em tempo de execução (*run-time errors*) em uma implementação [PALSBERG & SCHWARTZBACH, 1992].

Podemos perceber a relação entre a noção de subtipificação discutida e a especialização conceitual. O relacionamento *é-um* indica uma forte ligação entre as formas como elementos das categorias relacionadas podem ser manipulados (visão externa), e não necessariamente as suas semelhanças estruturais (visão interna).

Porém, exigir que uma hierarquia de subclassificação preserve o princípio da substituição, com a noção associada de compatibilidade de comportamento, é forte e restrita demais em muitas situações práticas [WEGNER, 1988]. Como foi dito na Seção 3.2, construir mecanismos de modificação incremental para este tipo de hierarquia é extremamente custoso, principalmente se são considerados sistemas interativos. Noções de comportamento muito poderosas podem até mesmo impossibilitar a verificação de tipos [AMERICA & VAN DER LINDEN, 1990]. Alguns autores argumentam também que o processo de concepção de objetos, no qual objetos mais detalhados são definidos a partir de outros já descritos, precisa de liberdade na evolução que a subtipificação não fornece [TAIVALSAARI, 1996; WEGNER, 1988].

No restante do trabalho daremos importância a hierarquias de subclassificação que pretendem preservar o comportamento entre as classes relacionadas. A preservação do

comportamento pode acontecer através de restrições sintáticas (herança sintática) ou semânticas (herança semântica).

Capítulo 4

Herança Sintática em RPOO

Neste capítulo, trataremos das questões relacionadas à herança de propriedades sintáticas entre classes RPOO. Consideramos neste trabalho o conceito de herança associado principalmente a relacionamentos entre as classes, estabelecidos pelo respeito a determinadas restrições, chamadas de critérios de compatibilidade. No caso da herança sintática tratada neste capítulo, os critérios são observados sobre as interfaces das classes. Através de uma tal observação podemos determinar se uma classe é subclasse de outra (com relação a uma restrição específica).

Seria possível também tratar o conceito de herança como um mecanismo de modificação incremental que gera subclasses a partir de uma classe. Consideramos apenas que, como este mecanismo deve ser específico para um critério, este deve ser estabelecido antes. Apesar disto, fazemos neste capítulo algumas considerações sobre possíveis mecanismos de herança.

Na Seção 4.1, discutimos os aspectos relacionados ao conceito de herança que são considerados neste capítulo. Algumas observações sobre o estabelecimento de mecanismos de herança sintática são feitas na Seção 4.2. Na Seção 4.3, apresentamos a compatibilidade de assinatura entre classes (entre suas interfaces) como uma restrição sintática à subclassificação. Na Seção 4.4, expomos algumas conclusões.

4.1 Critérios Sintáticos de Compatibilidade

É bastante comum encontrarmos na literatura o conceito de herança tratado como um mecanismo poderoso de modificação incremental, que aumenta o poder de expressão de notações orientadas a objeto [WEGNER, 1988; COOK ET AL., 1994; BOOCH, 1994]. Neste caso, o conceito de herança é fortemente ligado às ações subentendidas pelo verbo *herdar*. Mais especificamente: uma ação de herdar permite que descrições de (sub)classes aproveitem (herdem) as descrições prévias das respectivas superclasses.

A ênfase neste aspecto do conceito de herança se deve ao fato da importância prática de mecanismos de modificação incremental, não somente no desenvolvimento de sistemas de software, como em vários campos da atividade humana. Reutilizar descrições (ou especificações) segura e eficazmente é uma das principais medidas para um aumento de produtividade no desenvolvimento de algum projeto. Pedaços de soluções já avaliados podem assim ser utilizados para compor novas soluções. Mecanismos de modificação incremental podem ser mecanismos eficientes de reutilização de descrições.

Porém, devemos estar cientes que este é apenas um aspecto relacionado ao conceito de herança como tratado neste trabalho. Importante, mas não o único. Achamos mais adequado tratar o conceito de herança de forma mais abrangente e o consideramos associado aos relacionamentos caracterizados pela expressão *herda-de*. Tais relacionamentos podem ser identificados como hierarquias sobre um conjunto de classes, nas quais as subclasses preservam (ou herdam) algumas propriedades das superclasses. Estas propriedades preservadas podem ser sintáticas (relacionadas às descrições das classes) ou semânticas (relacionadas aos seus comportamentos). Portanto, restrições sintáticas ou semânticas associadas a um relacionamento específico de herança devem garantir que a respectiva hierarquia, chamada neste trabalho de hierarquia de herança, preserve as propriedades desejadas. A estas restrições, assim como fez Wegner [WEGNER, 1988] e conforme o Capítulo 3, chamamos de critérios de compatibilidade.

Enfocamos, portanto, com relação ao conceito de herança, as restrições (critérios de compatibilidade) que caracterizam as hierarquias. Estas restrições têm por objetivo oferecer garantias sobre o comportamento das classes. Mais especificamente, buscamos restrições que garantam a substituição de objetos das superclasse por objetos das

subclasses. Logo, as hierarquias de herança vistas desta forma são passíveis de estudo.

Deixamos para segundo plano a questão das hierarquias serem estabelecidas por mecanismos de modificação incremental ou não. A principal razão para isto é o fato de que os mecanismos de modificação incremental devem possibilitar a descrição de classes a partir de outras, preservando as restrições associadas. Assim, o mecanismo é diretamente dependente do critério de compatibilidade. Portanto, segundo este ponto de vista, os critérios de compatibilidade devem ser estabelecidos prioritariamente.

Vale ressaltar que o fato de tratarmos o estabelecimento de mecanismos de modificação incremental em segundo plano não significa que o consideremos de menor importância. Ao optarmos por enfatizar o princípio da substituição como norteador das hierarquias de herança, sabemos ser este aspecto muito restrito e, conforme apontou Lakos [LAKOS, 1995b], altamente inibidor do aproveitamento de descrições. Entendemos simplesmente que, se devem existir restrições sobre as diferenças das subclasses em relação as suas superclasses, estas restrições precisam ser primeiro definidas para que então se estabeleçam os mecanismos que as preservem. Mecanismos que permitam descrever classes a partir de outras sem observar quaisquer restrições são equivalentes a operações de cópia seguidas de alterações aleatórias das descrições.

Por considerarmos o estabelecimento de critérios de compatibilidade como central para a discussão de herança, dividimos este conceito segundo a natureza das restrições a que pode estar submetida: herança sintática e herança semântica. Neste capítulo, tratamos da herança sintática, ou seja, das hierarquias de classes nas quais as subclasses preservam propriedades puramente sintáticas com relação à superclasse.

Como discutido mais adiante no Capítulo 5, para a especificação de sistemas distribuídos e concorrentes, restrições puramente sintáticas sobre as interfaces dificilmente garantem a substituição adequada entre objetos de classes relacionadas. Porém, dentro de determinadas limitações no comportamento dos sistemas como um todo, os critérios de compatibilidade discutidos neste capítulo são bastante relevantes. Além disto, a compatibilidade de assinatura (definida mais à frente) é utilizada no Capítulo 5 como base para critérios semânticos de compatibilidade.

Antes de discutirmos os critérios sintáticos de compatibilidade, vamos definir, na seção seguinte, algumas operações básicas que podem fazer parte de mecanismos de mo-

dificação incremental que preservam propriedades sobre a interface dos objetos. Deve-se notar que este é um primeiro passo e pode constituir a base para o desenvolvimento de mecanismos mais poderosos.

4.2 Mecanismos de Herança Sintática em RPOO

Embora estejamos tratando herança como um relacionamento que preserva determinadas restrições, vamos fazer nesta seção algumas considerações sobre o estabelecimento de mecanismos de herança em RPOO, mais especificamente mecanismos de herança sintática. Os mecanismos de herança são mecanismos de modificação incremental que auxiliam projetistas de sistemas na descrição de classes a partir de outras, preservando as restrições associadas (critérios de compatibilidade).

Na verdade, o que se busca com os mecanismos de herança é um suporte poderoso à descrição da especialização entre classes. Ao especializar uma (super)classe é natural que tenhamos como base a sua descrição e a partir dela, através de alterações, alcancemos a descrição da subclasse.

Assim, como visto na Seção 3.3.1, a herança como modificação incremental pode ser definida pela expressão $R = P \oplus \Delta R$, na qual a descrição de uma subclasse (R) é obtida a partir da descrição da sua superclasse (P), sofrendo alterações, caracterizadas pelo modificador ΔR. Desta forma, compreende duas partes:

1. cópia da descrição da superclasse;

2. alterações sobre a cópia feita, respeitando as possíveis restrições associadas ao mecanismo.

Embora conceitualmente um mecanismo de herança realize uma cópia da descrição das superclasses, as linguagens de programação, por exemplo, utilizam operações mais sofisticadas, que evitam duplicação de código através de técnicas de resolução de nomes (incluindo, se for o caso, as peculiaridades da herança múltipla) e de ligação dinâmica das auto-referências [TAIVALSAARI, 1996]. Isto caracteriza um dos principais benefícios atribuídos aos mecanismos de herança. Ao evitar a duplicação das descrições, estes mecanismos simplificam a tarefa de manutenção, pois os pedaços de descrição herdados

encontram-se concentrados em um só lugar, existindo métodos para o estabelecimento de ligações simbólicas. A ligação dinâmica de auto-referências faz com que mecanismos de herança se diferenciem de simples chamadas de procedimento [TAIVALSAARI, 1996]. Porém essas técnicas simplesmente cumprem o papel de simular a cópia da descrição, com vantagens de manutenção.

Deste modo, conceitualmente, podemos considerar que são realizadas cópias efetivas e que cada classe possui a sua própria descrição. Não precisamos assim nos ater ao funcionamento de quaisquer dessas técnicas. É claro que, devido à sua importância, elas devem ser consideradas no desenvolvimento de ferramentas de software responsáveis por editar, analisar ou verificar descrições de sistemas RPOO. Algumas destas técnicas encontram-se detalhadas em [TAIVALSAARI, 1996] e representam boa parte da propagada economia de implementação obtida com as linguagens de programação orientadas a objetos.

A segunda parte dos mecanismos de herança, as alterações que preservam as restrições associadas, constitui-se a mais complexa. Isto se deve ao fato de que tais alterações devem respeitar as restrições impostas pelo relacionamento de herança pretendido. Dependendo do nível das restrições, o estabelecimento destas alterações pode até ser inviabilizado.

Estas alterações na verdade caracterizam a motivação para o estabelecimento de mecanismos de herança. Isto porque, embora a verificação das restrições possa ser realizada automaticamente (considerando ser decidível se uma classe respeita uma restrição) e *a posteriori*, esta verificação pode exigir demasiado esforço computacional [VAN DER AALST & BASTEN, 1997]. Assim, deseja-se fornecer ao projetista um poderoso recurso para a obtenção das hierarquias de herança: uma operação de transformação que implemente as restrições.

Essas alterações são especificamente ligadas aos critérios de compatibilidade a serem preservados pelo mecanismo. Com relação às restrições sintáticas sobre as interfaces das classes, tratadas na Seção 4.3 deste capítulo, o estabelecimento das restrições sobre as alterações que as preservem já está relativamente consolidado e restringe-se à utilização combinada das três operações seguintes, discutidas na Seção 3.3.3, sobre os métodos e atributos dos objetos [WEGNER, 1988; TAIVALSAARI, 1996]:

1. cancelamento

2. adição

3. redefinição

Nas sub-seções seguintes, discutimos cada uma dessas operações no contexto de RPOO. Basicamente, as operações de cancelamento e adição têm aqui um tratamento semelhante ao dado pelas linguagens de programação orientadas a objeto [TAIVALSAARI, 1996], sendo operações que alteram a interface de uma classe, seguida das relativas alterações no seu corpo. Por sua vez, a redefinição possui a peculiaridade de considerar que são realizadas também alterações no corpo das classes que não têm relação direta com a interface.

Evitamos exibir a formalização de tais operações, por considerarmos que isto não ajudaria na compreensão.

Devemos considerar a definição de classe presente na Seção 2.2.1.

4.2.1 Cancelamento

Consideramos que o cancelamento em RPOO pode ser aplicado tanto a atributos quanto a métodos da interface de uma classe. O corpo de uma classe só é alterado especificamente nos pontos de contato com a interface.

Desse modo, dados uma classe, um conjunto de atributos e um conjunto de métodos a serem cancelados, o cancelamento produz outra classe, baseada na primeira, porém com a exclusão dos atributos e métodos determinados. Em consequência, são eliminadas as relações destes atributos e métodos com os seus tipos, bem como as suas relações com o corpo da classe (relação com um lugar, no caso dos atributos, e com uma transição e uma variável, no caso dos métodos).

Na Figura 4.1 e na Figura 4.2 temos respectivamente a Classe A e a Classe B, esta obtida a partir da primeira pelo cancelamento do atributo $a2$ e do método $m2$. A sintaxe utilizada na legenda da Figura 4.2 é bastante simples:

$$nova\_classe = cancela(classe\_original, \{at1; at2; ...\}, \{mt1; mt2; ...\})$$

na qual *nova_classe* é a classe resultante do cancelamento dos atributos {at1;at2;...} e dos métodos {mt1;mt2;...} da classe *classe_original.*

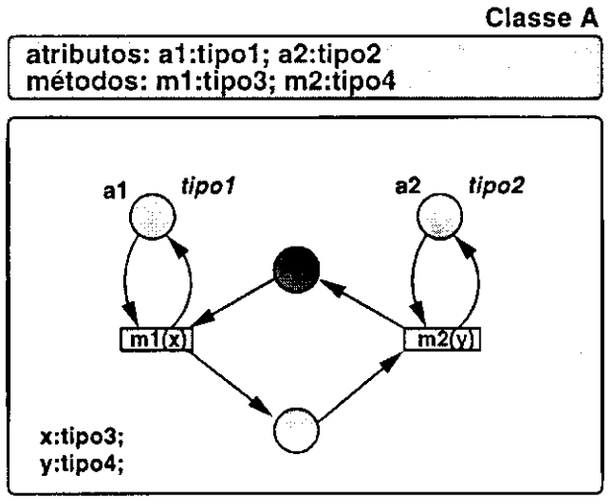

Figura 4.1: Classe A

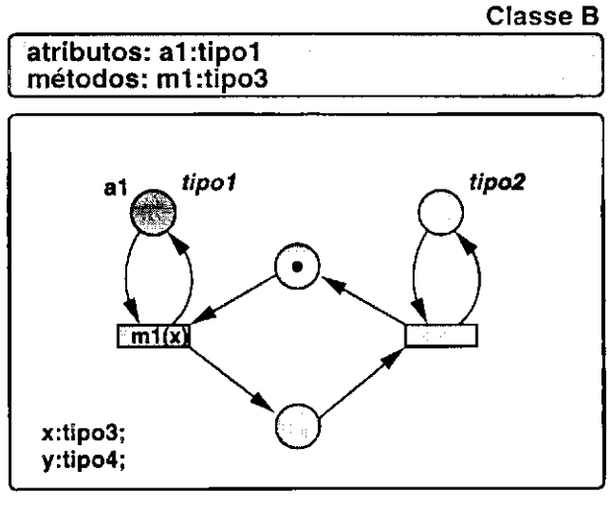

Figura 4.2: Classe B = cancela(Classe A, {a2}, {m2})

4.2.2 Adição

Assim como o cancelamento, a adição produz uma classe a partir de outra através da alteração da interface e das consequentes alterações no corpo.

Dados uma classe, um conjunto de atributos e um conjunto de métodos a serem adicionados, acompanhados dos seus tipos e da definição das variáveis que irão se

nos tipos associados (dos parâmetros e de retorno). Porém, em RPOO o comportamento dos objetos de uma classe é descrito por uma única rede de Petri Colorida, que pode ter como estrutura um grafo conexo ou não-conexo. Assim, redefinir parte deste comportamento (relacionado ao processamento de alguma mensagem recebida) consiste em alterar parte da rede.

Portanto, podemos considerar a existência de dois tipos de operações de redefinição de propriedades em RPOO, que podem estar relacionadas:

1. à interface. Neste caso, a operação não inclui nem exclui nada. Simplesmente altera determinados tipos de atributos ou métodos, com as consequentes alterações das cores dos lugares associados aos atributos e dos tipos das variáveis associadas aos métodos.

2. ao corpo. Neste caso, a operação permite uma quase completa redefinição da rede de Petri Colorida que compõe o corpo da classe: criação/eliminação de lugares, transições e arcos, bem como alteração nas suas respectivas inscrições. As restrições ficam por conta dos lugares e das transições que estão relacionados à interface que não podem ser alterados.

A Figura 4.4 ilustra a Classe D, resultado de operações de redefinição no corpo da Classe C (Figura 4.3). Por esta operação consistir basicamente da edição livre das redes, optamos por não detalhar a sua sintaxe, que seria muito extensa. Poderíamos contudo dividí-la em pequenas operações do tipo *incluir_lugar* ou *excluir_arco* com o objetivo de simplificar a notação.

Podemos considerar finalmente que a Classe D foi obtida a partir da Classe A (Figura 4.1), através de um mecanismo que compreende as operações de cancelamento, adição e redefinição, sem quaisquer restrições.

4.3 Restrições sobre a Interface

Nesta seção tratamos das restrições sintáticas que interessam ser estabelecidas sobre subclasses RPOO. De acordo com o que é discutido na Seção 3.2, Wegner aponta

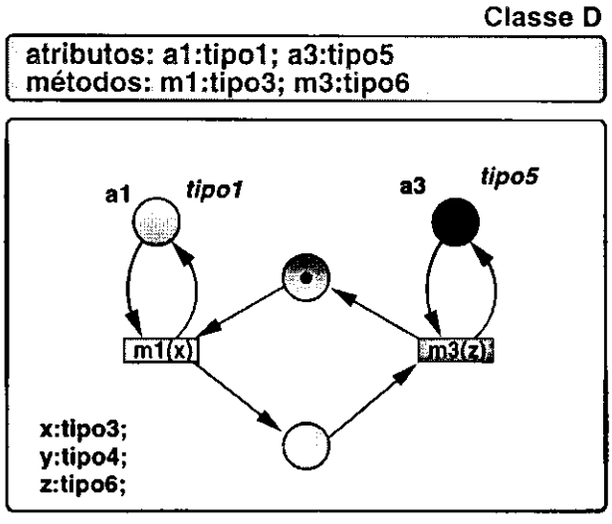

Figura 4.4: Classe D: redefinição do corpo da classe C

como importantes critérios sintáticos de compatibilidade, e que são garantidos sobre as interfaces das classes, os seguintes [WEGNER, 1988]:

1. cancelamento
2. compatibilidade de nomes
3. compatibilidade de assinatura

Com relação a esses critérios, somente consideraremos a compatibilidade de assinatura, no contexto de RPOO. As razões para não tratarmos dos demais critérios são as que seguem.

O critério chamado de cancelamento, embora seja considerado desta forma, não se refere propriamente a um critério de compatibilidade, uma vez que se baseia na herança de propriedades através de uma total liberdade nas adição, cancelamento (daí o nome) e redefinição de propriedades. Construída desta forma, não é possível garantir que uma subclasse preserve qualquer propriedade (seja qual for) da sua superclasse, uma vez que *todas* as propriedades desta podem ser canceladas e adicionadas outras novas na criação daquela. Portanto quaisquer duas classes podem estar relacionadas por esse critério (e em qualquer sentido). Acreditamos que tal critério é considerado com o objetivo de abarcar algumas implementações de mecanismos de herança em linguagens de programação orientadas a objeto. Nestas implementações, não é garantido

qualquer compatibilidade entre superclasse e subclasse, com a intenção de aumentar as possibilidades de reaproveitamento de código. Contudo, devido ao fato do cancelamento não constituir um critério passível de análise/verificação, este não é considerado neste trabalho.

Quanto à compatibilidade de nomes, este critério não apresenta os problemas do anterior, pois não permite que métodos ou atributos sejam cancelados nas subclasses. Porém, ao exigir que somente os nomes dos métodos e atributos sejam preservados, este critério não garante a substituição. Por exemplo, o conteúdo de uma mensagem m_1, endereçada a um objeto de uma classe, não necessariamente seria tratado por um objeto de uma subclasse sua, pois os tipos desta mensagens nas duas classes não necessitariam ser compatíveis.

Dessa forma, tratamos da compatibilidade de assinatura em RPOO, que apesar de não garantir completamente a substituição em modelos de computação interativos é utilizada como base para definição de restrições mais poderosas (ver Capítulo 5).

Compatibilidade de Assinatura em RPOO

Consideraremos nesta seção as restrições impostas à interface de subclasses RPOO para que estas possam preservar a compatibilidade de assinatura com relação às suas superclasses. Primeiro, vamos discutir a relação entre *interface* e *assinatura* em RPOO.

Conforme visto na Seção 2.2.1, a interface de uma classe RPOO é composta por:

- um conjunto de nomes de atributos (A);
- um conjunto de nomes de métodos (M);
- uma aplicação (τ) que relaciona cada nome de atributo ou método a um tipo (a um sorte, de acordo com a definição encontrada naquela seção).

Embora sejam denominados de maneira semelhante, o conceito de assinatura, retirado do mundo das especificações algébricas e utilizado no Capítulo 2 para definir as inscrições das redes, e o conceito de assinatura de uma interface tratado nesta seção são diferentes. Como definimos neste trabalho a interface das classes baseada apenas

em nomes (atributos, métodos, conjuntos), este conceito é semelhante ao conceito de assinatura das classes.

A compatibilidade de assinatura tem por objetivo garantir que mensagens enviadas a um objeto de uma classe possam ser tratadas por objetos de suas subclasses. Em um modelo de computação sequencial, este critério de compatibilidade pode garantir a substituição entre objetos.

O primeiro ponto a ser garantido é a preservação dos nomes de atributos e métodos. Logo, os nomes de atributos e métodos de uma classe devem estar presentes nas interfaces das suas subclasses. O segundo ponto diz respeito aos tipos dos métodos e atributos[1]. A idéia mais imediata seria garantir que eles fossem preservados também. Porém, isto seria restringir demais as possibilidades de redefinição das características das interfaces. Como veremos a seguir, existem determinadas alterações que ainda asseguram o tratamento de mensagens endereçadas à superclasse. Encontramos na literatura dois estilos de redefinição de funções normalmente aplicados à compatibilidade de assinatura [TAIVALSAARI, 1996; WEGNER, 1988; LAKOS, 1995b]: *covariante* e *contravariante.*

O estilo covariante de redefinição de funções se baseia na constatação prática de que, ao especializarmos um conceito, é natural que restrinjamos os domínios dos seus atributos e consequentemente das operações. Deste modo, segundo Lakos, este estilo é o que apresenta mais interesse do ponto de vista prático [LAKOS, 1995b]. Assim, por exemplo, ao especializarmos o conceito *pessoa* no conceito *aposentado*, parece natural a restrição do domínio do atributo *idade* de (0..120) para (65..120). A Figura 4.5 ilustra uma descrição para as classes Pessoa e Aposentado, com esta especializando aquela segundo o estilo covariante. Além do atributo idade, é descrita também a operação *altera_idade*, que troca a idade antiga por uma outra, enviada por outro objeto.

Com base na Figura 4.5, contudo, podemos observar que o estilo covariante apresenta problemas para a substituição. Por exemplo, uma mensagem *altera_idade(30)* é perfeitamente tratável por um objeto da classe Pessoa, mas não pode ser aceita sem problemas por um objeto da classe Aposentado, pois não pertence ao domínio(tipo) do

[1]Conforme discutido no Capítulo 2, o tipo de um método é o tipo do parâmetro que é transmitido pela método (ou mensagem).

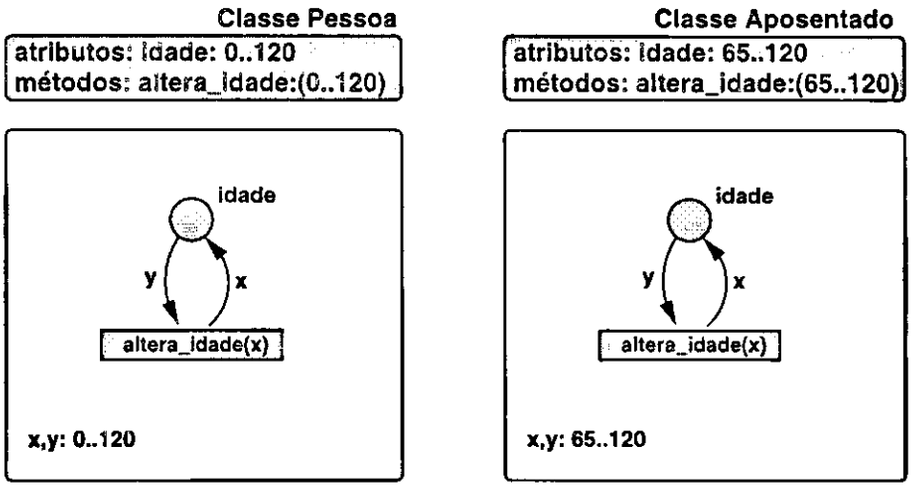

Figura 4.5: Classes Pessoa e Aposentado

seu atributo idade.

O estilo contravariante de redefinição de funções resolve esse problema, ao considerar na especialização de conceitos que os domínios dos atributos podem ser expandidos, assim como também os domínios dos parâmetros dos métodos. Segundo o estilo contravariante, somente os tipos de retorno dos métodos são restringidos (são subtipos dos tipos de retorno dos métodos das superclasses).

Como nosso objetivo com as restrições é tentar garantir a substituição entre objetos, optamos pelo estilo contravariante para definir a compatibilidade de assinatura em RPOO. Com relação a esse estilo, não consideramos as observações sobre os tipos de retornos dos métodos por não estarmos tratando de mensagens síncronas em RPOO.

Antes de estabelecer a compatibilidade de assinatura, precisamos definir uma relação entre os elementos de $\mathcal{S}$ (conjunto de nomes de sortes de uma assinatura Σ). Sejam S_i e S_j dois nomes de conjuntos pertencentes a $\mathcal{S}$. Dizer que S_j é *subtipo* de S_i (notado por $S_j \leq_s S_i$) implica que possíveis domínios referenciados por estes nomes (D_{S_j} e D_{S_i} respectivamente) devem obedecer à seguinte relação:

$$D_{S_j} \subseteq D_{S_i}$$

Assim, definimos a compatibilidade de assinatura entre classes como segue.

Definição 4.1 (Compatibilidade de Assinatura) *Sejam k_i e k_j duas classes e ${}^{k_i}I = \langle {}^{k_i}A, {}^{k_i}M, {}^{k_i}\tau \rangle$ e ${}^{k_j}I = \langle {}^{k_j}A, {}^{k_j}M, {}^{k_j}\tau \rangle$ suas respectivas interfaces. k_i apresenta*

compatibilidade de assinatura com relação a k_j (ou seja, k_i é subclasse de k_j, segundo a compatibilidade de assinatura) se e somente se:

- ${}^{k_i}A \supseteq {}^{k_j}A$
- ${}^{k_i}M \supseteq {}^{k_j}M$
- $\forall a \in {}^{k_j}A$: ${}^{k_j}\tau(a) \leq_s {}^{k_i}\tau(a)$
- $\forall m \in {}^{k_j}M$: ${}^{k_j}\tau(m) \leq_s {}^{k_i}\tau(m)$

Desse modo, temos que a interface de uma subclasse deve preservar todos os nomes dos atributos e métodos da sua superclasse. Além disto, os atributos preservados na subclasse devem ter como tipo supertipos dos atributos da sua superclasse. O mesmo deve ser garantido para os tipos dos métodos preservados. Assim, estamos estabelecendo o estilo contravariante de redefinição, com o objetivo de garantir que objetos de uma subclasse recebam mensagens destinadas a objetos de sua subclasse.

4.4 Conclusões

Apresentamos neste capítulo considerações sobre o estabelecimento de restrições sintáticas que possam vir a garantir a substituição entre objetos de classes distintas. Na verdade, discutimos aspectos relacionados a restrições sobre a interface dos objetos, definindo por fim a compatibilidade de assinatura como a mais adequada com objetivos de substituição. Porém, no contexto de sistemas tratados neste trabalho, a importância deste critério de compatibilidade é o de compor a base para restrições mais eficientes, que considerem aspectos dinâmicos dos objetos, assunto do Capítulo 5.

Com relação a restrições sintáticas, contudo, parece-nos relevante considerar também, além das propriedades sobre a interface, propriedades estruturais sobre as redes que modelam o comportamento dos objetos. Por existir uma série de propriedades já estudadas sobre as estruturas das redes de Petri e que informam sobre propriedades semânticas, como as invariantes de lugar e transição [MURATA, 1989], estas propriedades podem ser consideradas para dar garantias de substituição mais poderosas. Além

disto, o esforço computacional para verificar tais propriedades é menor do que verificar propriedades sobre o grafo de ocorrência das redes.

Capítulo 5

Herança Semântica em RPOO

Neste capítulo tratamos dos relacionamentos entre classes que preservam propriedades comportamentais, chamados genericamente neste trabalho de herança semântica. Embora seja possível pensar em mecanismos de auxílio a projetistas de sistemas que incrementalmente construam classes a partir de outras [VAN DER AALST & BASTEN, 1997], preservando tais propriedades, o que estamos considerando no contexto deste trabalho é a sua verificação *a posteriori*. Assim, dado um conjunto previamente definido de classes e um critério de compatibilidade comportamental, queremos saber como estas classes se relacionam segundo este critério. O objetivo geral é garantir certos níveis de substituição entre objetos de classes relacionadas.

O capítulo é dividido como segue. Na Seção 5.1, abordamos as principais características de uma noção de comportamento para sistemas interativos. Na Seção 5.2 discutimos as possibilidades de estabelecimento de critérios semânticos de compatibilidade entre classes RPOO. Nesta seção, fazemos considerações sobre critérios definidos sobre o espaço de estados das redes de Petri que descrevem o comportamento das classes e sobre as possíveis seqüências de eventos ativados por estas redes. Finalmente, na Seção 5.3 apresentamos algumas conclusões.

5.1 Comportamento em Sistemas Interativos

Como discutido no Capítulo 2, RPOO é uma ferramenta para especificação formal de sistemas distribuídos abertos [AGHA, 1986; AGHA ET AL., 1993], nos quais a topologia

de interconexão é naturalmente dinâmica. Portanto, com relação a RPOO, é importante a definição de mecanismos que considerem, para efeito de verificação ou análise, os contextos nos quais os componentes de tais sistemas aparecem. Assim, torna-se relevante uma representação adequada da especialização conceitual (relacionamento *é-um*), através da qual é possível estabelecer uma hierarquia entre os componentes (na verdade, entre classes de componentes) que garanta que os membros de níveis inferiores podem substituir seus descendentes (princípio da substituição). Em outras palavras, é preciso garantir que objetos de uma (sub)classe possam ser manipulados também como objetos da sua superclasse. Uma relação desta natureza estabelece uma hierarquia (relação de ordem), pois garante a substituição em somente um sentido.

Uma relação de ordem que preserva o princípio da substituição se aproxima do conceito de subtipificação (tipos de dados) e pode trazer para uma notação orientada a objetos um poder de verificação semelhante ao obtido sobre os tipos de dados nas linguagens de programação [PALSBERG & SCHWARTZBACH, 1994; COOK ET AL., 1994; CARDELLI & ABADI, 1996].

Embora a herança sintática (discutida no Capítulo 4) já apresente como objetivo controlar o comportamento externo dos objetos, notamos que somente através da observação de propriedades semânticas podemos garantir um controle mais adequado para a categoria de sistemas de software considerada neste trabalho.

Assim como foi feito para herança sintática, a herança semântica é discutida em termos de critérios de compatibilidade. A diferença é que os critérios agora considerados são definidos sobre propriedades semânticas. Quanto mais restritivos forem os critérios considerados, menores serão as possibilidades de relacionamento entre as classes e mais difícil será (se for o caso) o estabelecimento de mecanismos de modificação incremental que os preservem.

Central para a discussão de critérios de compatibilidade comportamentais é uma noção adequada de comportamento para os componentes a serem considerados (objetos). Uma vez estabelecida uma tal noção de comportamento, poderemos então definir níveis de compatibilidade que garantam que objetos de uma (sub)classe podem aparecer em contextos onde são esperados objetos da sua (super)classe. Uma relação de equivalência seria suficiente para garantir a substituição. Porém, para representar a

especialização conceitual só precisamos assegurar a substituição em um sentido. Portanto, assim como são definidas noções de equivalência entre redes [POMELLO ET AL., 1992], estamos interessados em noções de compatibilidade. Tais noções de compatibilidade devem permitir certo nível de expansão, porém restringindo o comportamento das subclasses com relação ao comportamento das suas superclasses. Desta forma, podemos relacionar classes de objetos de modo a que as subclasses representem especializações das superclasses. Estas subclasses podem detalhar algumas funcionalidades das superclasses ou mesmo acrescentar outras novas.

Em uma computação seqüencial, na qual as operações podem ser vistas como seqüências finitas de ações sem efeitos colaterais, podemos simplificar o entendimento do comportamento das entidades através de conjuntos de entrada e saída (modelo funcional) [POMELLO ET AL., 1992]. Assim, considerando o comportamento de um sistema orientado a objetos sob uma perspectiva puramente seqüencial, podemos caracterizar o comportamento dos objetos simplesmente através de restrições sintáticas (notadamente sobre suas assinaturas). Algumas linguagens de programação e ferramentas para a especificação formal utilizam esta simplificação para caracterizar compatibilidades comportamentais entre classes relacionadas por herança [LAKOS, 1995a; LAKOS, 1995b]. Porém, ao considerarmos as características interativas da computação orientada a objetos concorrente, a compreensão das operações sob um ponto de vista puramente funcional não é adequada [POMELLO ET AL., 1992]. Os métodos dos objetos, além de poderem provocar a alteração dos seus estados, podem ter o seu fluxo de execução interrompido por interações com outros objetos no sistema, ou eventualmente por linhas paralelas de controle existentes no próprio objeto. Além disto, a disponibilidade da ativação destes métodos pode variar com o estado dos objetos. Desta forma, é necessário fazer novas considerações com o objetivo de definir uma noção de comportamento mais apropriada.

Milner estabeleceu a noção de um observador externo para caracterizar a equivalência de comportamento em um sistema interativo (equivalência por observação) [MILNER, 1980]. A partir daí, numerosas noções de equivalência de comportamento para sistemas interativos e concorrentes foram propostas, considerando como observável ou a *seqüência de eventos* (ações) por eles produzida ou seu *espaço de estados*

[POMELLO ET AL., 1992].

Na seção seguinte, vamos fazer considerações sobre a observação de seqüência de eventos e de espaço de estados em redes de Petri Coloridas, com o objetivo de estabelecer critérios semânticos de compatibilidade para classes RPOO.

5.2 Critérios Semânticos de Compatibilidade em RPOO

Considerando que o comportamento das classes em RPOO é descrito através de redes de Petri Coloridas, as propriedades semânticas destas classes devem ser observadas de acordo com tal formalismo. Logo, com relação às redes de Petri coloridas, seqüência de eventos e espaço de estados representam diferentes visões sobre o grafo de ocorrência [JENSEN, 1992]. Regras de compatibilidade podem então ser estabelecidas sobre estas visões, baseadas em propriedades semânticas, tais como: alcançabilidade, limitação, vivacidade, reversibilidade (e estado inicial), cobertura e persistência, entre outras [MURATA, 1989].

Para tanto, vamos fazer considerações sobre o grafo de ocorrência como caminho para o estudo das propriedades comportamentais das classes (das redes que compõem a descrição das classes).

5.2.1 Grafo de Ocorrência

Informalmente, um grafo de ocorrência de uma rede de Petri Colorida é um grafo no qual os nós são as marcações alcançáveis pela rede a partir de uma marcação inicial (que é o nó de partida do grafo) e os arcos são as ocorrências dos elementos de ligação que provocam as mudanças de estado [JENSEN, 1992].

Grafos de ocorrência para redes de Petri Coloridas (na verdade, para qualquer classe de redes de Petri) podem com muita facilidade se tornar extremamente grandes, aumentando assim a complexidade computacional envolvida no processo de estudá-las [JENSEN, 1992]. Além disto, se nas redes estiverem representados, por exemplo, contadores, o conjunto de marcações alcançáveis torna-se infinito. Técnicas de redução para

as redes de Petri Coloridas foram propostas com o objetivo de resultar em grafos mais manipuláveis, sem que haja perda das principais características dos sistemas [JENSEN, 1992].

Dentre as diversas propriedades que podem ser observadas sobre o grafo de ocorrência devemos escolher alguns aspectos relevantes para estabelecer garantias de substituição. Para a classe de sistemas distribuídos e concorrentes considerada neste trabalho, o que interessa são as interações dos objetos com o mundo exterior, sendo de menor relevância os passos internos que não alteram estas relações. Deste modo, vamos considerar os critérios de compatibilidade que procuram preservar características das interações dos objetos com o ambiente externo.

A interação dos objetos com o ambiente externo acontece através da troca de mensagens: mensagens aceitas do ambiente e mensagens enviadas a outros objetos. Além disto, objetos podem enviar mensagens de criação de outros objetos. Partindo desta consideração, vamos identificar um nível de compatibilidade *mínimo* para que o princípio da substituição seja assegurado.

Com relação às mensagens recebidas, é desejável que os objetos de uma subclasse sejam aptos a tratar *pelo menos* as mesmas mensagens tratadas pelos objetos das suas superclasses. Além disto, temos que considerar também os tipos dos conteúdos das mensagens. Assim, devemos garantir que subclasses preservem a compatibilidade de assinatura (conforme visto no Capítulo 4) com relação às superclasses (considerações sobre a disponibilidade destas mensagens são feitas mais adiante). Neste contexto, portanto, fica expresso que as subclasses podem acrescentar novas funcionalidades ou mesmo especializar aquelas das suas superclasses, mas não podem suprimi-las.

Com relação às mensagens enviadas e à criação de novos objetos, fazemos considerações mais adiante, ficando o nível mínimo de compatibilidade associado à compatibilidade de assinatura, a partir da qual discutimos as demais. Estudamos então algumas possíveis abordagens para compor a herança semântica, levando-se em conta a interação dos objetos com o ambiente externo. Consideramos portanto:

1. as variações nas mensagens habilitadas ao longo do espaço de estados, aqui chamado de *dinâmica de interfaces*;

2. as possíveis seqüências de eventos observáveis externamente na dinâmica de cada objeto.

5.2.2 Critérios Sobre a Evolução Dinâmica das Interfaces

Uma marcação em uma rede de Petri define o seu estado. Portanto, o espaço de estados de uma CPN é o conjunto de marcações alcançáveis a partir da marcação inicial. Com relação ao grafo de ocorrência, o espaço de estados é definido pelos seus nós.

Devido ao fato de estarmos interessados no controle do comportamento externo dos objetos, não nos preocuparemos com critérios de compatibilidade que considerem todo o espaço de estados.

Particularmente, existe uma abstração especial sobre o espaço de estados que é bastante relevante dentro do contexto de RPOO: a *evolução dinâmica das interfaces.* Em RPOO, embora as interfaces dos objetos representem as mensagens que podem ser recebidas, no ciclo-de-vida destes objetos o conjunto real de serviços disponíveis (chamado *estado da interface*) muda dinamicamente. Ou seja, é possível que, para alguns estados alcançáveis, determinadas mensagens, embora invocadas por outros objetos, não possam ser ativadas.

Por exemplo, considerando a rede da classe garfo (Figura 2.5, Capítulo 2), os objetos desta classe, a cada momento, estão aptos a receber mensagens *aloca* ou *libera.* Nunca acontece de um garfo estar apto a ativar estas duas mensagens ao mesmo tempo. Se uma mensagem *aloca* chegar para um determinado garfo no estado *alocado*, ela não poderá ser ativada até que uma mensagem *libera* seja invocada e ativada. Poderíamos estar interessados em assegurar que especializações da classe garfo preservassem esta característica (entre outras) para que fosse permitida a substituição.

A comparação entre classes, portanto, busca relacionar como o conjunto de métodos habilitados varia durante o ciclo de vida dos objetos.

Por exemplo, vamos considerar as classes A e B, descritas na Figura 5.1 e na Figura 5.2 respectivamente[1]. Se considerarmos que *tipo*1′ é supertipo de *tipo*1, *tipo*2′ é

[1]Os exemplos apresentados ao longo deste capítulo contêm uma simplificação na descrição das redes de Petri. Embora estejamos tratando de redes de Petri Coloridas, as fichas nestes exemplos não *carregam* informação. Com isto, os grafos de ocorrência também são simplificados, pois os arcos que

supertipo de *tipo*2, e assim sucessivamente, temos claramente que a classe B preserva compatibilidade de assinatura com relação à classe A. Porém, se observarmos, mesmo superficialmente, como se comportam as interfaces dos objetos das duas classes, podemos notar que algumas propriedades não são preservadas. Na classe A, as habilitações das mensagens $m1$, $m2$ e $m3$ dependem causalmente umas das outras, o que faz com que elas nunca estejam habilitadas ao mesmo tempo. Ainda na classe A, temos que em qualquer instante ou a mensagem $m1$ ou a mensagem $m4$ está habilitada, nunca as duas ao mesmo tempo. Na classe B, contudo, nenhuma destas propriedades é preservada. Nos dois únicos estados alcançados por sua rede associada, as mensagens $m1$, $m2$ e $m3$ ou estão todas habilitadas ou nenhuma está. Além disto, podemos observar que a ativação da mensagem $m4$ nunca está habilitada em objetos da classe B, pois em *nenhum* estado das suas redes haverá fichas nos lugares a e b (seus lugares de entrada) simultaneamente. Além destas, outras incongruências poderiam ainda ser observadas.

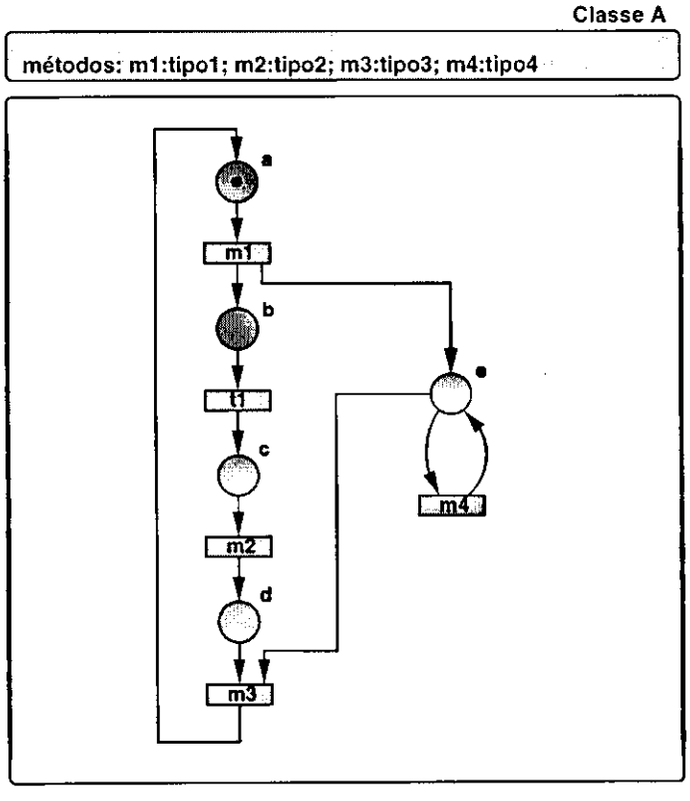

Figura 5.1: Descrição da classe RPOO A

representam as mudanças de estado são denotados simplesmente pelos nomes das transições e não por um elemento de ligação (transição + ligação).

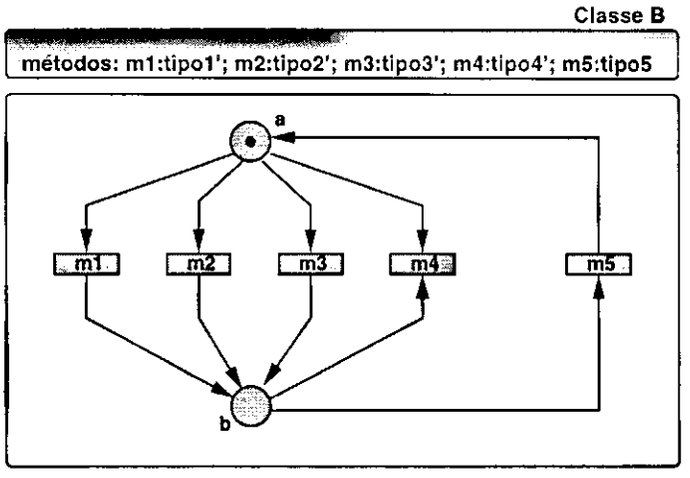

Figura 5.2: Descrição da classe RPOO B

Na Figura 5.3, temos um detalhamento da dinâmica da interface dos objetos da classe A. À esquerda, temos o grafo de ocorrência da rede associada. À direita, temos uma abstração deste grafo de ocorrência, a dinâmica da interface, na qual caracterizamos os estados pelas mensagens habilitadas, ao invés de caracteriza-las pelas marcações dos lugares. As propriedades mencionadas acima sobre a classe A podem ser facilmente observadas: em todos os estados ou a mensagem $m1$ ou a mensagem $m4$ estão habilitadas e as mensagens $m1$, $m2$ e $m3$ nunca estão habilitadas ao mesmo tempo, nem mesmo duas delas.

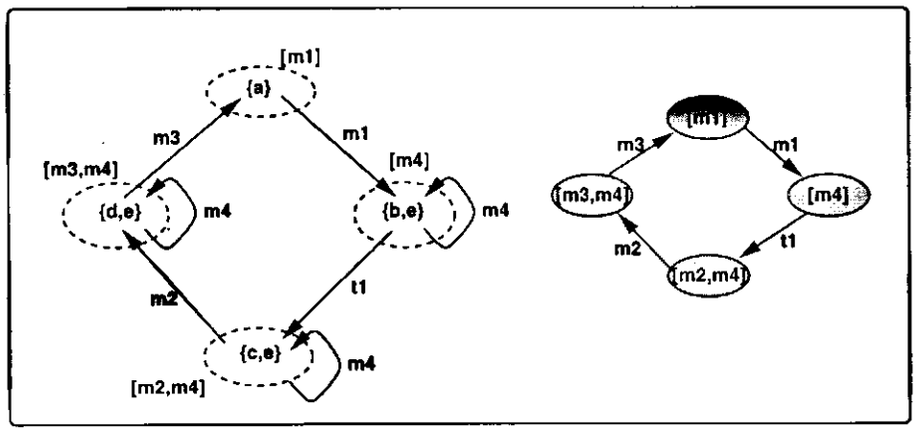

Figura 5.3: Grafo de ocorrência e dinâmica de interfaces dos objetos da classe A

Vamos considerar, informalmente, o seguinte critério de compatibilidade, chamado de Critério 1. Uma classe é compatível a outra com relação a este critério se preserva:

- compatibilidade de assinatura;
- pares de mensagens cujos elementos nunca estão habilitados ao mesmo tempo;
- conjuntos de mensagens que possuem sempre um e somente um de seus elementos habilitados.

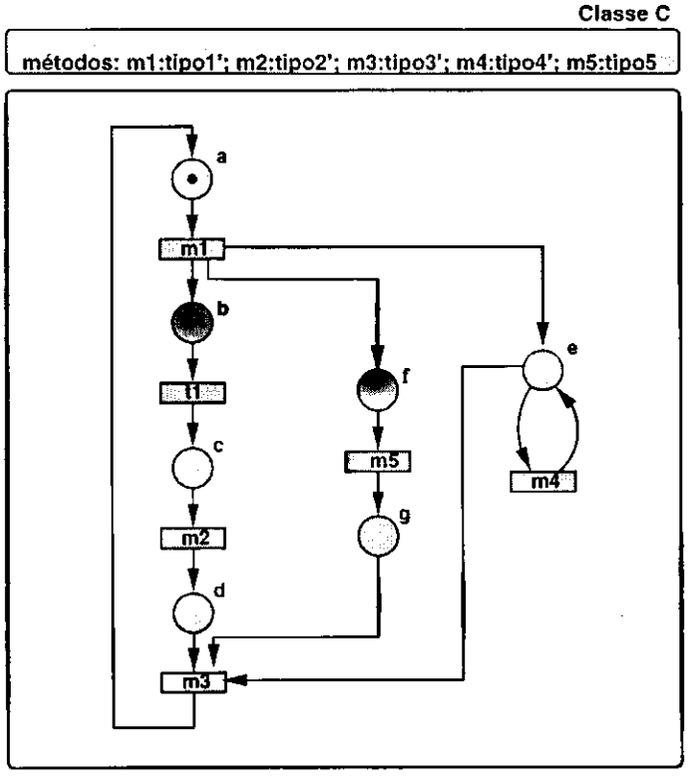

Figura 5.4: Descrição da classe RPOO C

Na Figura 5.4, temos a descrição da classe C. Se considerarmos que os tipos das mensagens desta classe são supertipos dos tipos das mensagens da classe A, temos que a classe C preserva compatibilidade de assinatura com relação a esta classe. Através da dinâmica da interface dos objetos da classe C apresentada na Figura 5.5, temos que os pares de mensagens cujas ativações nunca estão habilitadas ao mesmo tempo na classe A ($\{m1, m2\}$, $\{m1, m3\}$, $\{m1, m4\}$ e $\{m2, m3\}$) são preservados como tais na classe C e que, também nesta classe, temos que sempre os métodos $m1$ ou $m4$ podem ser ativados. Portanto, a classe C é compatível à classe A com relação ao Critério 1, o que é graficamente ilustrado pela Figura 5.6.

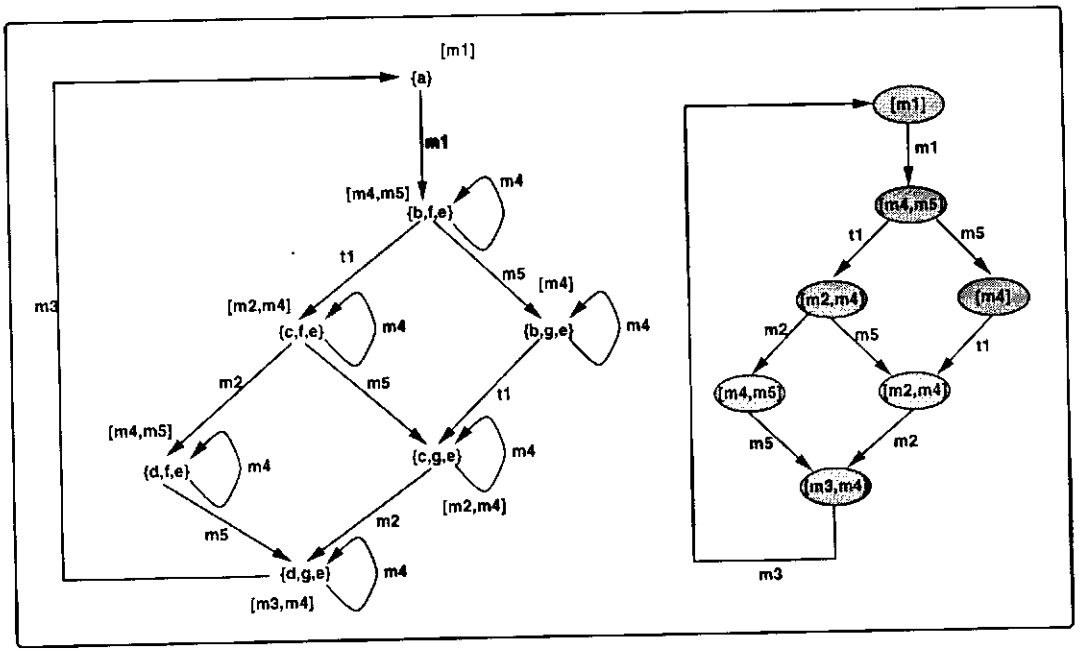

Figura 5.5: Grafo de ocorrência e dinâmica de interfaces dos objetos da classe C, descrita na Figura 5.3

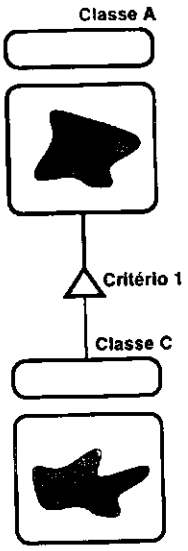

Figura 5.6: Relação entre as classes A e C

Podemos ainda estabelecer outros critérios que levem em consideração as relações entre as interfaces, o que refletiria relações (seqüência, concorrência, conflito) entre as ativações das mensagens. Por exemplo, o Critério 1 não exige a preservação da relação de seqüência entre as mensagens $m1$, $m2$ e $m3$ existente na classe A.

5.2.3 Critérios sobre Seqüências de Ações Observáveis Externamente

Uma outra possibilidade no estabelecimento de compatibilidade entre classes é centrarmos o foco nas ações observáveis externamente. Neste nível, portanto, não nos interessamos pelas ações que definem apenas o comportamento interno dos objeto.

Nas redes de Petri, as ações estão associadas ao disparo de transições. Especificamente em RPOO, as ações observáveis externamente são:

1. o envio de uma mensagem, pois este gera uma nova mensagem no ambiente;
2. a recepção de uma mensagem, pois esta elimina uma mensagem do ambiente;
3. a criação de um novo objeto no ambiente (sistema);

O que nos interessa neste ponto é definir critérios de compatibilidade relacionados à preservação de determinadas propriedades sobre seqüências destas ações.

Assim como é caracterizada a equivalência baseada em ações observáveis entre sistemas [POMELLO ET AL., 1992], os níveis de compatibilidade discutidos nesta seção tentam garantir certas propriedades entre as seqüências de ações (ou eventos), especificamente seqüências de envios e recepções de mensagens e criação de novos objetos.

Se considerarmos, por exemplo, a rede que descreve o comportamento da classe Filósofo (Figura 2.5, Capítulo 2), notaremos que os objetos desta classe produzem sempre a seguinte seqüência de eventos observáveis externamente: solicitam os garfos da direita e esquerda, recebem os garfos, liberam os garfos, novamente solicitam os garfos, e assim por diante. Podemos querer garantir que esta característica seja preservada nas subclasses da classe Filósofo. É importante notarmos neste ponto que, preservando esta característica, nós estamos impondo certas restrições ao comportamento dos objetos das subclasses da classe Filósofo quanto à seqüência de eventos. Porém, estamos

liberando outras formas de detalhamento, como a inclusão de outras funcionalidades, desde que não interfiram na propriedade em questão.

Vamos considerar que cada transição seja rotulada pelo evento que dispara[2]. As propriedades sobre as seqüências de eventos de uma rede podem então ser definidas sobre a linguagem produzida pelo disparo das transições, considerando que cada transição, ao ser disparada, gera como saída o seu rótulo. As transições que não são associadas a nenhuma ação externa produzem como saída a cadeia vazia. Porém, não é objetivo deste trabalho fazer considerações sobre as possíveis linguagens produzidas por redes de Petri.

De maneira simplificada, podemos estar interessados em garantir desde seqüências simples até níveis de compatibilidade que façam considerações sobre concorrência e relações causais entre eventos.

Como exemplo, vamos voltar a considerar as classe A e C, descritas respectivamente na Figura 5.1 e na Figura 5.4 da Seção 5.2.2. Naquela seção, foi definido o critério de compatibilidade chamado Critério 1, segundo o qual a classe C é subclasse da classe A. Vamos supor um modelo de sistema composto por um objeto da classe A e um objeto de uma classe que o envie somente seqüências das mensagens $m1$, $m2$ e $m3$, como por exemplo a classe D descrita na Figura 5.7. As mensagens invocadas pelo objeto da classe D poderão então ser ativadas pelo objeto da classe A sempre nesta ordem[3]: uma mensagem $m2$ após uma mensagem $m1$, uma mensagem $m3$ após uma mensagem $m2$ e uma mensagem $m1$ após uma mensagem $m3$ (com exceção da primeira mensagem $m1$ ativada). Uma vez que o objeto da classe D espera estar enviando mensagens para um objeto da classe A, uma mensagem $m5$ nunca será enviada[4].

Vamos imaginar agora um modelo de sistema semelhante ao anterior (estamos considerando, para ambos modelos de sistema o modelo de classes da Figura 5.8), com

[2]Com o objetivo de simplificar as idéias aqui tratadas, consideramos cada transição associada a somente um evento. Em RPOO contudo, podemos ter qualquer número finito de eventos associados a uma transição. Além disso, em um contexto de redes de Petri Coloridas, deveríamos considerar como rotuladas as ligações (transições + ligações) [LAKOS, 1995b].

[3]Mais uma simplificação: as únicas ações observáveis externamente presentes nesse exemplo são recepções de mensagens.

[4]Na verdade, isto poderia ser identificado estaticamente, através de uma verificação sintática.

Vamos considerar, informalmente, o seguinte critério de compatibilidade, chamado de Critério 2. Uma classe é compatível a outra com relação a este critério se preserva:

- compatibilidade de assinatura;
- seqüencias simples de eventos;

Poderíamos considerar ainda passos e processos com o objetivo de garantir características relacionadas respectivamente a concorrência e a relações causais entre as ações[POMELLO ET AL., 1992].

Podemos observar que a classe E, decrita na Figura 5.9 preserva o Critério 2 com relação à classe A. Embora a ativação da mensagem $m5$ possa levar um objeto da classe E a um estado morto, se objetos desta classe forem tratados como objetos da classe A (substituição) nunca terão estas mensagens invocadas. Desta forma, a classe E preservaria o comportamento observável externamente da classe A.

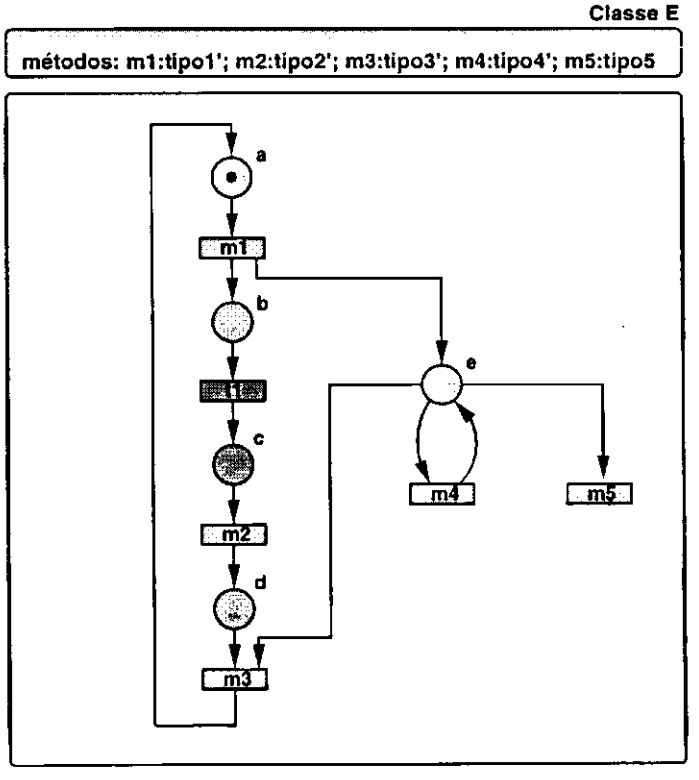

Figura 5.9: Descrição da classe RPOO E

Além dos aspectos aqui abordados, podemos ainda estabelecer preservação de relações entre eventos, como seqüências, conflitos e concorrência.

5.3 Conclusões

Apresentamos neste capítulo idéias relacionadas ao estabelecimento de restrições no comportamento dinâmico de objetos de subclasses, no contexto de herança em RPOO. Estas restrições têm por objetivo assegurar a substituição de objetos de uma determinada classe por objetos de suas subclasses. Embora a especialização pressuponha, em determinado nível, a expansão do comportamento (ou mesmo seu detalhamento) ao longo da hierarquia, algumas restrições são necessárias sobre esta expansão, objetivando garantias de substituição.

Os critérios de compatibilidade discutidos neste capítulo (Critério 1 relacionado à dinâmica da interface e Critério 2 relacionado às seqüências de eventos) foram apresentados informalmente. Isto porque eles são utilizados neste capítulo de maneira simplesmente ilustrativa, com o objetivo de representar as diversas possibilidades de critérios que possam vir a ser estabelecidos. Eles não representam critérios com relevância comprovada ou mesmo discutida, ao contrário dos critérios apresentados no Capítulo 4. Pela mesma razão, não discutimos aspectos de complexidade envolvidos.

Capítulo 6

Conclusões

Neste trabalho, discutimos o conceito de herança no contexto de uma notação para especificação formal baseada em redes de Petri, denominada RPOO (Rede de Petri Orientada a Objetos). Percebemos que este conceito pode ser utilizado ou para designar um mecanismo que possibilite descrever classes a partir de outras ou para designar um mecanismo de representação da especialização conceitual, presente nos domínios das aplicações.

Vimos que nem sempre é possível conciliar essas duas formas de conceber mecanismos de herança. Ao tentar potencializar a reutilização de descrições, podemos estabelecer mecanismos de herança que gerem classes sem qualquer relação com suas subclasses. Por outro lado, critérios de compatibilidade muito restritos para caracterizar subclassificação podem dificultar a tarefa de reutilizar descrições.

A herança tratada como um mecanismo de modificação incremental, principalmente por uma parte considerável das linguagens de programação, tem sua relevância nas técnicas avançadas de simulação da cópia do código. Parece-nos interessante a utilização da expressão *cópia simbólica* para se referir a estas técnicas, dissociando-a do termo herança. Assim, tais técnicas poderiam ser utilizadas com mais facilidade em outros contextos.

Dessa forma, optamos por associar herança, em primeiro plano, a relacionamentos que pretendam representar a especialização conceitual. Vimos que a especialização conceitual é uma abstração poderosa utilizada pelo homem para compreender e capturar determinados domínios. Assumimos que um mecanismo que represente a especialização

deve assegurar a substituição de objetos de uma classe por objetos de suas subclasses. Além disto, um mecanismo dessa natureza pode constituir um instrumento poderoso para a especialização dos modelos como um todo.

Dividimos o estudo da herança em RPOO segundo a natureza das propriedades observadas para a manutenção de compatibilidade de subclasses com relação a suas respectivas superclasses. Assim, examinamos a herança sintática e a herança semântica em RPOO.

Com relação à herança sintática, observamos as possibilidades de restrições sobre as interfaces das classes de objetos, com destaque para a compatibilidade de assinatura. Discutimos o fato de níveis de compatibilidade que consideram somente a interface dos objetos não serem adequados para garantir a substituição de objetos em um ambiente interativo de computação. Para tanto, deveríamos estabelecer restrições que levassem em conta propriedades dinâmicas dos objetos. Isto nos levou a propor a herança semântica para RPOO.

6.1 Trabalhos Futuros

A conclusão deste trabalho nos permite identificar alguns encaminhamentos futuros no sentido de completar a inclusão do conceito de herança em RPOO. Outros trabalhos são ainda necessários com o objetivo de tornar RPOO uma notação para uso efetivo na especificação de sistemas.

Em primeiro lugar, é preciso estudar em detalhes possíveis restrições no comportamento das classes que garantam efetivamente a substituição. Assim, devemos considerar principalmente as propriedades semânticas das redes que descrevem o comportamento das classes. Para cada conjunto de restrições considerado, devemos também estudar métodos de verificação de compatibilidade entre classes. Devemos então examinar a complexidade algorítmica destes métodos.

Assim, poderíamos utilizar as técnicas de cópia simbólica para construir classes umas a partir das outras e depois, através de métodos de análise, identificar os relacionamentos de subclassificação existentes.

Porém, como esta verificação pode ser computacionalmente muito custosa, uma

vez que pode exigir a construção dos grafos de ocorrência das redes, restam-nos dois caminhos possíveis para uma utilização mais eficaz da herança em RPOO:

1. desenvolver mecanismos de modificação incremental que construam classes umas a partir de outras, preservando determinadas restrições de comportamento;

2. utilizar análise estrutural das redes para garantir propriedades comportamentais.

A viabilidade de desenvolvimento de mecanismos de modificação incremental que preservem restrições de comportamento deve ser estudada, pois pode significar um grande aumento de produtividade durante o processo de especificação dos sistemas.

Porém, podemos estar interessados em garantir a compatibilidade mesmo entre classes que não foram descritas umas a partir das outras. Dessa forma, devemos investigar formas de garantir restrições de comportamento a um menor custo computacional. As redes de Petri possuem um conjunto consolidado de propriedades estruturais (como invariantes de lugar e transição, por exemplo) que se caracterizam por antecipar propriedades dinâmicas sem a necessidade da geração de grafos de ocorrência. A utilização de tais propriedades pode significar um enorme ganho na verificação de relações de subclassificação. Porém, a verificação de muitas destas propriedades, originalmente estudadas sobre redes de Petri clássicas, ainda não foram desenvolvidas para as redes de Petri de Alto Nível, como é o caso das redes coloridas [JENSEN, 1992].

Com o objetivo de validar as propostas de restrições que garantam a substituição segundo sua utilidade prática, necessitamos aplicá-las na modelagem de alguns sistemas, reais ou hipotéticos.

Com relação a RPOO, os três principais encaminhamentos no sentido de torná-la efetivamente utilizável são:

1. suporte a primitivas para comunicação de mais alto nível entre os objetos;

2. desenvolvimento de métodos de análise composicional, que componham os resultados de análises realizadas sobre as classes individualmente para obter resultados sobre o sistema completo;

3. desenvolvimento de uma ferramenta de suporte a edição, análise (incluindo simulação) e verificação dos modelos.

Bibliografia

[DES, 1996] *Design CPN, version 3.0.* University of Aarhus, Aarhus, Denmark, 1996.

[AGHA ET AL., 1993] Agha, Gul, Frolund, Svend, Kim, Wooyoung, Panwar, Rajendra, Patterson, Anna, Sturman, Daniel, 'Abstraction and modularity mechanisms for concurrent computing.' Gul Agha, Peter Wegner, Akinori Yonezawa, editores, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press: MIT Press, 1993.

[AGHA, 1986] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, Massachusetts: MIT Press, 1986.

[AJMONE MARSAN, 1989] Ajmone Marsan, M., 'Stochastic Petri Nets: An Elementary Introduction.' *Advances in Petri Nets 1989*, volume 424 de *Lecture Notes in Computer Science*, Springer-Verlag, 1989.

[ALAGAR & PERIYASAMY, 1998] Alagar, V. S., Periyasamy, K., *Specification of Software Systems.* Springer-Verlag, 1998.

[AMERICA & VAN DER LINDEN, 1990] America, Pierre, van der Linden, Frank, 'A parallel object-oriented language with inheritance and subtyping.' Norman Meyrowitz, editor, *OOPSLA/ECOOP'90*, volume 25 de *Sigplan Notices*, 1990.

[BACLAWSKI & BIPIN, 1994] Baclawski, Kenneth, Bipin, Indurkhya, 'The notion of inheritance in object-oriented programming.' *Communications of the ACM*, volume 37, num. 9, pp. 118, 1994.

[BALDASSARI ET AL., 1989] Baldassari, M., Bruno, G., Russi, V., Zompi, R., 'PROTOB a hierarchical object-oriented CASE tool for distributed systems.' C. Ghezzi,

J.A. McDermid, editores, *ESEC'89*, volume 387 de *Lecture Notes in Computer Science*, pp. 425–445, Springer-Verlag, 1989.

[BASTIDE, 1995] Bastide, R., 'Approaches in unifying Petri nets and the object-oriented approach.' *Proceedings of the First International Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, junho 1995.

[BASTIDE ET AL., 1996] Bastide, R., Lakos, C., Palanque, P., 'A cooperative Petri net editor.' Case Study Proposal for the 2nd Workshop on Objetc-Oriented Programming and Models of Concurrency.

[BATTISTON & DE CINDIO, 1993] Battiston, E., de Cindio, F., 'Class orientation and inheritance in modular algebraic nets.' *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, pp. 717–723, Le Touquet, France: IEEE, 1993.

[BOOCH, 1994] Booch, G, *Object-Oriented Analysis and Design.* Object-Oriented Software Engineering, Santa Clara, California, USA: Benjamin/Cummings Publishing Company, 2° edição, 1994.

[BRACHMAN, 1983] Brachman, Ronald J., 'What IS-A is and isn't:an analysis of taxonomic links in semantic networks.' *Computer*, volume 16, num. 10, pp. 30, 1983.

[BUCHS & GUELFI, 1991] Buchs, D., Guelfi, N., 'CO-OPN: A concurrent object oriented petri net approach.' *Proceedings of the 12th International Conference on the Application and Theory of Petri Nets*, Lecture Notes in Computer Science, Gjern, Denmark: Springer Verlag, 1991.

[CARDELLI & ABADI, 1996] Cardelli, Luca, Abadi, Martín, *A Theory of Objects.* Monographs in Computer Science, Springer-Verlag, 1996.

[CHRISTENSEN & HANSEN, 1993] Christensen, S., Hansen, N. D., 'Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs.' *Proceedings of 14th International Conference on Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, pp. 186 – 205, Springer Verlag, 1993.

[COOK ET AL., 1994] Cook, William R., Hill, Walter L., Canning, Peter S., 'Inheritance is not subtyping.' Carl A. Gunter, John C. Mitchell, editores, *Theoretical Aspects of OO Programming - Types, Semantics, and Language Design*, Foundations of Computing Series, MIT Press, 1994.

[COSTA ET AL., 1998a] Costa, S.A.D., Guererro, D.D.S., de Figueiredo, J.C.A., Perkusich, A., 'Aspectos de herança em uma ferramenta de modelagem de sistemas baseada em redes de petri.' *Anais do SBES'98, Simpósio Brasileiro de Engenharia de Software*, pp. 297–312, Maringá, PR, outubro 1998.

[COSTA ET AL., 1998b] Costa, S.A.D., Guererro, D.D.S., de Figueiredo, J.C.A., Perkusich, A., 'Inheritance issues in object-oriented petri net modeling.' *Proc. of IEEE Int. Conf. on Systems Man and Cybernetics*, pp. 196–201, San Diego, USA, outubro 1998.

[DE MEDEIROS ET AL., 1998] de Medeiros, A.K.A., Guererro, D.D.S., de Figueiredo, J.C.A., Perkusich, A., 'An object-oriented petri-net modeling tool and abstraction mechanisms for cooperative systems.' *Proc. of IEEE Int. Conf. on Systems Man and Cybernetics*, pp. 172–177, San Diego, USA, outubro 1998.

[DENG ET AL., 1993] Deng, Y., Chang, S.K., de Figueiredo, J.C.A., Perkusich, A., 'Integrating software engineering methods and petri nets for the specification and prototyping of complex software systems.' M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 de *Lecture Notes in Computer Science*, pp. 206 - 223, Chicago, USA: Springer-Verlag, junho 1993.

[DIJKSTRA, 1971] Dijkstra, E. W., 'Hierarchical ordering of sequencial processes.' *Acta Informatica 1*, pp. 115–138, 1971.

[ENGELFRIET ET AL., 1990] Engelfriet, J., Leih, G., Rozenberg, G, 'Net-based description of parallel object-based systems, or POTs and POPs.' J.W. de Bakker, W.P. de Roever, G. Rozenberg, editores, *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, pp. 229–273, Springer-Verlag, 1990.

[GENRICH, 1987] Genrich, H.J., 'Predicate/Transition nets.' W. Brauer, W. Reisig, G. Rozemberg, editores, *Petri Nets: Central Models and Their Properties*, volume 254 de *Lecture Notes in Computer Science*, pp. 207–247, Springer-Verlag, 1987.

[GUERERRO ET AL., 1998] Guererro, D.D.S., de Figueiredo, J.C.A., Perkusich, A., 'An object-based modular cpn approach: its application to the specification of a cooperative editing environment.' *Advances on Petri Nets: Object Orientation and Models of Concurrency, Lecture Notes in Computer Science*, 1998.

[GUERRERO, 1998a] Guerrero, Dalton D. S., *Orientação a Objetos e Modelos de Redes de Petri.* Relatório técnico, Coordenação de Pós-graduação em Engenharia Elétrica - COPELE/UFPB, Campina Grande, PB, abril 1998.

[GUERRERO, 1998b] Guerrero, Dalton D. S., *Uma Linguagem Orientada a Objetos Baseada em Redes de Petri para Especificação de Sistemas Distribuídos Concorrentes.* Relatório técnico, Coordenação de Pós-graduação em Engenharia Elétrica - COPELE/UFPB, Campina Grande, PB, julho 1998.

[GUERRERO, 1997] Guerrero, Dalton Dario Serey, *Sistemas de Redes de Petri Modulares Baseadas em Objetos.* Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, 1997.

[GUERRERO ET AL., 1997] Guerrero, D.D.S., de Figueiredo, J.C.A., Perkusich, A., 'Object-based high-level petri nets as a formal approach to distributed information systems.' *Proc. of IEEE Int. Conf. on Systems Man and Cybernetics*, pp. 3383–3388, Orlando, USA, outubro 1997.

[GUEZZI ET AL., 1991] Guezzi, Carlo, Jazayeri, Mehdi, Mandrioli, Dino, *Fundamentals of Software Engineering.* Prantice-Hall International Editions, 1991.

[JENSEN, 1992] Jensen, K., *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use.* EACTS – Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[LAKOS, 1997] Lakos, C., 'On the abstraction of coloured Petri nets.' Pierre Azéma, Gianfranco Balbo, editores, *Proceedings of the 18th International Conference on*

Applications and Theory of Petri Nets, volume 1248 de *Lecture Notes in Computer Science*, p. 42, Springer-Verlag, 1997.

[LAKOS & KEEN, 1991] Lakos, C.A., Keen, C.D., 'Modelling layered protocols in LO-OPN.' *Proceeding of Fourth International Workshop on Petri Nets and Performance Models*, Melbourne, Australia, 1991.

[LAKOS, 1995a] Lakos, Charles, 'From coloured Petri nets to object Petri nets.' *Proceedings of the 16th International Conference on Applications and Theory of Petri Nets*, Turin, Italy, junho 1995.

[LAKOS, 1995b] Lakos, Charles, 'Pragmatic inheritance issues for object Petri nets.' *TOOLS Pacific 1995*, pp. 309–321, Melbourne, Australia, 1995.

[MADSEN ET AL., 1990] Madsen, Ole L., Magnusson, Boris, Moller-Pedersen, Birger, 'Strong typing of object-oriented languages revisited.' Norman Meyrowitz, editor, *OOPSLA/ECOOP'90*, volume 25 de *Sigplan Notices*, 1990.

[MERLIN, 1979] Merlin, P.M., 'Specification and Validation of Protocols.' *IEEE Transactions on Communications*, volume COM-27, num. 11, pp. 1671–1680, novembro 1979.

[MEYER, 1985] Meyer, Bertrand, 'On formalism in specifications.' *IEEE Software*, volume 2, num. 1, pp. 6–26, janeiro 1985.

[MILNER, 1980] Milner, Robin, *A Calculus of Communicating Systems*, volume 92 de *Lecture Notes in Computer Science*. Berlin, Heidelberg - Germany: Springer-Verlag, 1980.

[MURATA, 1989] Murata, Tadao, 'Petri nets: Properties, analysis and applications.' *Proc. of the IEEE*, volume 77, num. 4, pp. 541–580, abril 1989.

[PALSBERG & SCHWARTZBACH, 1992] Palsberg, Jens, Schwartzbach, Michael I., 'Three discussions on object-oriented typing.' *ACM SIGPLAN OOPS Messenger*, volume 3, num. 2, pp. 31, 1992.

[PALSBERG & SCHWARTZBACH, 1994] Palsberg, Jens, Schwartzbach, Michael I., 'Static typing for object-oriented programming.' *Science of Computer Programming*, volume 23, num. 1, pp. 19, 1994.

[PERKUSICH & DE FIGUEIREDO, 1997] Perkusich, A., de Figueiredo, J.C.A., 'G-nets: A petri net based approach for logical and timing analysis of complex software systems.' *Journal of Systems and Software*, volume 39, num. 1, pp. 39–59, 1997.

[POMELLO ET AL., 1992] Pomello, L, Rozenberg, G., Simone, C., 'A survey of equivalence notions for net based systems.' G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 de *Lecture Notes in Computer Science*, pp. 410–472, Springer-Verlag, 1992.

[RAMCHANDANI, 1974] Ramchandani, C., *Analysis of Asynchronous Concurrent Systems by Petri Nets.* Relatório Técnico Project MAC-TR120, M.I.T., Cambridge, MA, 1974.

[RUMBAUGH ET AL., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modelling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[SIFAKIS, 1980] Sifakis, J., 'Performance evaluation of systems using nets.' *Net Theory and Applications*, volume 84 de *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[TAIVALSAARI, 1996] Taivalsaari, Antero, 'On the notion of inheritance.' *ACM Computing Surveys*, volume 28, num. 3, pp. 438, 1996.

[THIAGARAJAN, 1987] Thiagarajan, P.S., 'Elementary net systems.' W Brauer, W. Reisig, G. Rozenberg, editores, *Petri-Nets: Central Models and their Properties, Proc. of and Advance Course on Petri Nets*, volume 254 de *Lecturte Notes in Computer Science*, pp. 26–59, Springer-Verlag, 1987.

[VAN DER AALST & BASTEN, 1997] van der Aalst, W. M. P., Basten, T., 'Life-cycle inheritance - a petri-net-based approach.' Pierre Azéma, Gianfranco Balbo, edito-

res, *Application and Theory of Petri Nets 1997*, volume 1248 de *Lecture Notes in Computer Science*, pp. 62 – 81, Springer-Verlag, junho 1997.

[WEGNER, 1988] Wegner, P., 'Inheritance as an incremental modification mechanism, or what like is and isn't like.' *Proceedings of ECOOP'88 - European Conference on Object Oriented Programming*, volume 322 de *Lecture Notes in Computer Science*, pp. 55–77, Oslo, Norway: Springer-Verlag, 1988.

[WEGNER, 1987] Wegner, Peter, 'The object-oriented classification paradigm.' Bruce Shriver, Peter Wegner, editores, *Research Directions in Object-Oriented Programming*, p. 479, The MIT Press, 1987.

Apêndice A

Tipos Abstratos de Dados

A.1 Definições Sintáticas

Definição A.1 (Assinatura) *Uma assinatura é um par* $\Sigma = \langle \mathcal{S}, \mathcal{O} \rangle$ *onde* $\mathcal{S}$ *é um conjunto finito de* sorts *ou nomes de conjuntos e* $\mathcal{O}$ *é um conjunto finito de símbolos de operações.* $\mathcal{S}$ *e* $\mathcal{O}$ *são disjuntos.*

Definição A.2 (Aridade de um Operacional) *Dada uma assinatura* $\Sigma = \langle \mathcal{S}, \mathcal{O} \rangle$, *a aridade de um símbolo de operação* $o \in \mathcal{O}$ *é definida como um elemento de* $\mathcal{S}^* \times \mathcal{S}$. *Símbolos cujas aridades sejam elementos de* $\emptyset \times \mathcal{S}$ *são denominados constantes.*

- Nas definições seguintes, quando uma assinatura é considerada, subentende-se que as aridades de cada um dos operacionais é dada.

Definição A.3 (Variáveis de uma Assinatura) *Um conjunto de variáveis* $\mathcal{V}_\Sigma$ *de uma assinatura* $\Sigma = \langle \mathcal{S}, \mathcal{O} \rangle$ *é a união disjunta de uma família de conjuntos de variáveis indexada por* $\mathcal{S}$. *Assim, podemos usar a notação:* $\mathcal{V}_\Sigma = V_{S_1} \cup V_{S_2} \cup \cdots \cup V_{S_n}$.

- Se uma variável v pertence ao conjunto de variáveis cujo índice é S_i, ou seja, $v \in V_{S_i}$, dizemos que o tipo de v é S_i.

Definição A.4 (Termos de uma Assinatura) *Sejam uma assinatura* $\Sigma = \langle \mathcal{S}, \mathcal{O} \rangle$ *e um conjunto de variáveis* $\mathcal{V}_\Sigma = V_{S_1} \cup V_{S_2} \cup \cdots \cup V_{S_n}$. *O conjunto de termos da*

assinatura é definido como a união disjunta de uma família de conjuntos de termos indexados por $\mathcal{S}$:

$$T_\Sigma = T_{S_1} \cup T_{S_2} \cup \cdots \cup T_{S_n}$$

onde cada T_{S_i} é definido indutivamente por:

1. *todo elemento $v \in V_{S_i}$ é elemento de T_{S_i};*
2. *se $\langle\langle S_1, S_2, \ldots, S_n\rangle, S_i\rangle$ é a aridade de $o_i \in \mathcal{O}$ e t_1, t_2, $\ldots$, t_n são termos de T_{S_1}, T_{S_2}, $\ldots$, T_{S_n} respectivamente, então $o_i t_1 t_2 \ldots t_n$ é elemento de T_{S_i}.*
3. *apenas expressões geradas pelas regras 1 e 2 acima são termos de T_{S_i}.*

- Um termo $ot_1t_2\ldots t_n$ será denotado por $o(t_1, t_2, \ldots, t_n)$;
- Se um termo t pertence ao conjunto de termos cujo índice é S_i, ou seja, $t \in T_{S_i}$, então dizemos que o tipo de t é S_i.
- Nas definições seguintes, se uma assinatura e um conjunto de variáveis são consideradas, deve-se subentender o conjunto de termos induzidos.

Definição A.5 (Equação) *Seja Σ uma assinatura e $\mathcal{V}_\Sigma$ um conjunto de variáveis. Uma equação sobre Σ é um par ordenado de termos $\langle t_{esq}, t_{dir}\rangle$ do mesmo tipo: $t_{esq}, t_{dir} \in T_{S_i}$. O conjunto de todas as equações sobre uma assinatura será denotado por E_Σ.*

A.2 Definições Semânticas

Nas definições a seguir assumem-se:

- uma assinatura $\Sigma = \langle \mathcal{S}, \mathcal{O}\rangle$, onde:
  - $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ é o conjunto de *sorts* (S_i é o elemento típico);
  - $\mathcal{O} = \{o_1, o_2, \ldots, o_n\}$ é o conjunto de símbolos operacionais (o_i é o elemento típico);
  - as aridades dos operacionais são subentendidas;

- $\mathcal{V}_\Sigma = \{V_1, V_2, \ldots, V_n\}$ um conjunto de variáveis para Σ.

Definição A.6 (Σ-Álgebra) *Diz-se que uma álgebra $\langle D, O\rangle$ é uma Σ-álgebra, se e somente se:*

1. $D = \{D_{S_1}, D_{S_2}, \ldots, D_{S_n}\}$ *é um conjunto de domínios indexado por* $\mathcal{S}$;
2. $O = \{o_{o_1}, o_{o_2}, \ldots, o_{o_n}\}$ *é um conjunto de operações indexado por* $\mathcal{O}$;
3. *todas as operações da álgebra respeitam a aridade dos respectivos símbolos operacionais, ou seja, se a aridade de* $o_i \in \mathcal{O}$ *é* $\langle\langle S_1, S_2, \ldots, S_n\rangle, S_i\rangle$ *então o operador* $o_{o_i} \in O$ *deve ser definido como* $o_{o_i} : D_{S_1} \times D_{S_2}, \ldots, D_{S_n} \rightarrow D_{S_i}$.

Definição A.7 (Atribuição de Variáveis) *Seja $\langle D, O\rangle$ uma Σ-álgebra. Uma atribuição de variáveis é uma aplicação total de $\mathcal{V}_\Sigma$ em D que respeita os* sorts, *ou seja, variáveis de tipo S_i somente podem assumir valores do domínio D_{S_i}:*

$$a : \bigcup_{S_i \in \mathcal{S}} (V_{S_i} \Rightarrow D_{S_i})$$

Definição A.8 (Avaliação de Termos) *Seja a uma atribuição de variáveis. A avaliação de um termo t em função de a, denotada por $[\![t]\!]_a$, é definida indutivamente por:*

1. *se t é uma variável, então*

$$[\![t]\!]_a = a(t)$$

2. *se t é da forma $o_i(t_1, t_2, \ldots, t_n)$, onde $t_1, t_2, \ldots, t_n$ são termos e o_i é um símbolo operacional, então*

$$[\![t]\!]_a = o_{o_i}([\![t_1]\!]_a, [\![t_2]\!]_a, \ldots, [\![t_n]\!]_a)$$

Definição A.9 (Validade de uma Equação) *Uma equação $t_{esq} = t_{dir}$, sobre Σ, é válida em uma Σ-álgebra, se e somente se sob qualquer atribuição de variáveis a, $[\![t_{esq}]\!]_a = [\![t_{dir}]\!]_a$.*

A.3 Especificação de Tipos Abstratos de Dados

Definição A.10 (Especificações) *Uma especificação algébrica é um par* $\langle \Sigma, \mathcal{E} \rangle$ *onde:*

- Σ *é uma assinatura;*
- $\mathcal{E}$ *é um conjunto de equações sobre* Σ.
- obviamente subentendem-se, nesta definição, as aridades dos operacionais de Σ e o conjunto de variáveis que dá origem aos termos.

Definição A.11 (Modelos) *Uma* Σ*-álgebra é um modelo para uma especificação* $\langle \Sigma, \mathcal{E} \rangle$ *se e somente se todas as equações de* $\mathcal{E}$ *são válidas.*