Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Engenharia de Computação e Automação Industrial

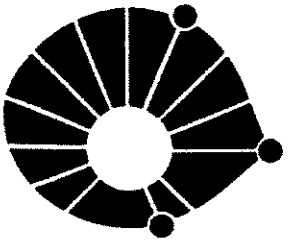
Rede de Agentes: Uma Ferramenta para o Projeto de Sistemas Inteligentes

José Antonio Sánchez Guerrero

Orientador: Prof. Dr. Ricardo Ribeiro Gudwin

Tese de Mestrado
Campinas – SP – Brasil
Fevereiro, 2000

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Engenharia de Computação e Automação Industrial



Rede de Agentes: Uma Ferramenta para o Projeto de Sistemas Inteligentes

José Antonio Sánchez Guerrero

Orientador: Prof. Dr. Ricardo Ribeiro Gudwin

Banca examinadora:

Prof. Dr. Ricardo Ribeiro Gudwin
DCA/FEEC/UNICAMP

Prof. Dr. José Reinaldo da Silva
Escola Politécnica/USP – São Paulo

Prof. Dr. Ivan Luiz Marques Ricarte
DCA/FEEC/UNICAMP

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação (FEEC) da Universidade Estadual de Campinas (UNICAMP), como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica.

Área de Concentração: Automação

Tese de Mestrado
Campinas – SP – Brasil
Fevereiro, 2000

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Sa55r

Sánchez Guerrero, José Antonio
Rede de agentes: uma ferramenta para o projeto de sistemas inteligentes / José Antonio Sánchez Guerrero.--Campinas, SP: [s.n.], 2000.

Orientador: Ricardo Ribeiro Gudwin.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Semiótica. 2. Inteligência artificial. 3. Sistemas inteligentes de controle. 4. Simulação (Computadores digitais). I. Gudwin, Ricardo Ribeiro. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

I never think of the future. It comes soon enough.
Albert Einstein

À minha mãe, minha filha, minha futura filha, minha esposa, minha irmã e em especial à memória de meu pai.

Resumo

Redes de objetos são ferramentas formais para a especificação e simulação de sistemas a eventos discretos, dotadas de características particularmente interessantes dentro do escopo dos chamados sistemas inteligentes, tais como capacidade de aprendizagem e adaptação. Redes de agentes são uma especialização das redes de objetos onde a capacidade de decisão fica distribuída ao longo dos objetos da rede, chamados então de agentes. Para tal, adotou-se um modelo específico de funções de seleção (que são genéricas na definição de rede de objetos), bem como um algoritmo geral que viabiliza a automatização e computabilidade destas funções. Redes de Agentes podem ser aplicadas em uma grande diversidade de contextos, desde a modelagem pura de sistemas complexos, com o propósito de análise, bem como na simulação e controle de tais sistemas. Particularmente, pode ser utilizada no contexto da engenharia de software, auxiliando nas etapas de análise, projeto e simulação de sistemas inteligentes. Com este trabalho desenvolveu-se uma ferramenta computacional (chamada ONtoolkit) que disponibiliza o uso das redes de agentes nas diferentes etapas de desenvolvimento de um sistema inteligente.

Palavras chaves: Semiótica Computacional, Sistemas Inteligentes e Simulação de Sistemas Inteligentes.

Abstract

Object Networks are formal tools for the specification and simulation of discrete event dynamical systems, equipped with characteristics particularly interesting under the scope of the so called intelligent systems, like learning and adaptation. Agent networks are a specialization of the object networks, where the decision capacity is distributed among the multiple objects on the network, which are called agents, in this case. Following this purpose, we adopted a specific model for the selection functions (which are generic in the original object network definition) and a general algorithm, which allows the automation and computability of such functions. Agent Networks can be applied to a great diversity of contexts, since the pure modelling of complex systems, with analysis purposes, passing through the simulation and control of such systems. Particularly, it can be used under the context of software engineering, as an aid to the analysis, design, and simulation of intelligent systems. In this work, we developed a computational tool (called ONtoolkit), which allows the use of agent networks in the different steps of development for an intelligent system.

Keywords: Computational Semiotic, Intelligent Systems, and Intelligent Systems Simulation.

Agradecimentos

Agradeço muito ao Professor Ricardo Ribeiro Gudwin pela orientação essencial ao desenvolvimento deste trabalho, pelo aprendizado propiciado e pelo convívio enriquecedor nos aspectos profissional e pessoal.

À minha esposa Diana Rosa, pela paciência, compreensão, carinho, amor e por me trazer alegria e realização nos momentos estressantes.

À minha família: minha mãe, minha filha, minha avó, minha irmã, minhas tias e meus primos Jorge e Yanet pela força, apoio e incentivo.

À minha irmã e companheira Lizet por todo o apoio, força, carinho, compreensão, amizade e por compartilhar todos os momentos bons e difíceis.

A meu amigo e colega Antônio S. R. Gomes por sua amizade e pelas grandes contribuições realizadas a este trabalho no dia a dia. Sem sua valiosa ajuda não teria sido possível a culminação com êxito deste trabalho.

Aos colegas Rodrigo Gonçalves, Mario Ernesto e Eduardo Fujito pelos comentários construtivos e sugestões.

A meus grandes amigos Lídice Romero, Luis A. Gurris e Miguel Angel, que a pesar da distância, sempre estão junto a mim.

A meus amigos Luis Alberto, Franklin, Maria Eugenia, Electo, Zaida, Vicente, Celeste, Wilson, Lino, Martica, Luis Mariano, Sahudy, Eduardo, Daynet, Juan C. H., Annabell, Raúl e Annia por ser uma comunidade cubana alegre.

A meus amigos Viviane, Cristina, Ester, Cida, Edmundo Spoto, Daniela, Thaís, Ellen, Raquel, Leandro, Rafael, Verena, Luiz Felipe, Míriam e Paulo pela amizade e esforço por fazer do Brasil minha segunda terra, especialmente a Rosa, Isolina e Luisa pela força e carinho de mãe.

Ao professor Fernando José Von Zuben pelas sugestões e comentários durante minha formação na pós-graduação.

A todos os amigos do L.C.A. pela companhia diária e pelo companheirismo, em especial a Magali e Pedro pela amizade de sempre.

Ao CNPq (Conselho Nacional de Desenvolvimento Cientifico e Tecnológico) por seu suporte financeiro, através de uma bolsa de estudos, que permitiu que eu pudesse me dedicar integralmente ao desenvolvimento deste trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Definições

CAPÍTULO 1. Introdução

A rápida evolução da tecnologia da computação permitiu o desenvolvimento e proliferação de novos tipos de sistemas dinâmicos, muitos deles de grande complexidade. Como exemplos de sistemas deste tipo, podemos citar as redes de computadores, sistemas de fabricação automatizados, sistemas de controle de tráfego (aéreo, rodoviário, ferroviário, etc), sistemas de fluxo de informação e sistemas inteligentes de um modo geral. Todas as atividades nestes sistemas acontecem pela ocorrência assíncrona de eventos discretos, onde alguns deles podem ou não ser controlados. Estas características, por si mesmas, nos conduzem a uma classe de sistemas muito particular, conhecida como "sistemas dinâmicos a eventos discretos" [Cassandras 93]. O arsenal matemático desenvolvido e/ou adaptado para tratar estes problemas permitiu o desenvolvimento de *frameworks* de modelagem, de ferramentas de *design*, novas técnicas de teste e procedimentos de controle sistemático para esta nova geração de sistemas complexos. Entre estas ferramentas, podem ser mencionados os Autômatos Finitos[Ullman 79] e as Redes de Petri [Murata 89], que permitem a modelagem, avaliação e teste de sistemas a eventos discretos, o que possibilita descrever as propriedades, limitações e implicações destes sistemas.

Os sistemas inteligentes são considerados um caso especial dos sistemas dinâmicos a eventos discretos. Só que as ferramentas existentes até hoje não fornecem as facilidades necessárias para a modelagem de sistemas deste tipo que exibam características especiais tais como a capacidade de adaptação e aprendizagem. Com a finalidade de suprir esta necessidade foi desenvolvida por Gudwin [Gudwin 96] a teoria dos sistemas de objetos (especificamente as redes de objetos), que propõe uma ferramenta formal para o *design* e modelagem dos sistemas inteligentes.

Uma disciplina emergente na modelagem de sistemas inteligentes, denominada Semiótica Computacional [Gudwin 99b], está sendo desenvolvida. Ela é utilizada para a construção de sistemas inteligentes autônomos capazes de executar comportamentos considerados inteligentes. Tais sistemas incluem módulos especializados para a percepção, modelo do mundo, julgamento de valores e geração de comportamento. Nesta disciplina, a rede de objetos é utilizada como ferramenta formal para a modelagem e estudos dos sistemas inteligentes.

As redes de objetos e os conceitos relacionados a ela, por força da abrangência a que se propunham destinar, foram colocados de forma muito genérica. Dentre os conceitos colocados de maneira bem genérica, talvez o caso mais crítico seja a definição da função de seleção, responsável direta pelo comportamento dinâmico das redes. Toda essa liberdade na escolha da função de seleção, impede a implementação de uma ferramenta computacional de uso amplo para as redes de objetos. Como a obtenção de ferramentas computacionais automáticas para o *design*, simulação e teste de sistemas inteligentes corresponde a uma das metas de se ter tal teoria, havia a necessidade que esse amplo espectro se fechasse, de modo

a permitir a construção de tal ferramenta. Para isso, seria necessário reformular algumas definições feitas sobre as redes de objetos e a criação de novos conceitos que permitissem a obtenção de uma ferramenta formal e computacional que fosse implementável. O ideal seria não sacrificar o poder de representação das redes de objetos, ou seja, seria manter as mesmas capacidades, potencialidades de representação e modelagem das redes de objetos. Estas considerações serviram de motivação para este trabalho. A proposta inicial era viabilizar a implementação de uma ferramenta computacional para as redes de objetos, efetuando as restrições necessárias, mas sem comprometer o potencial de representação das redes de objetos. Nos capítulos a seguir, apresenta-se a concretização deste trabalho, que acabou por resultar nas chamadas "Redes de Agentes", ou seja, uma classe específica de redes de objetos, que sem sacrificar significativamente seu poder de representação, permitiu a implementação computacional de uma ferramenta para modelagem e simulação de sistemas a eventos discretos baseada no paradigma das redes de objetos. Como conseqüência natural, procedeu-se à implementação dessa ferramenta, sendo que a mesma se encontra hoje operacional, sendo utilizada por todo um grupo de pesquisa na área de sistemas inteligentes.

Para uma maior compreensão e conveniência do leitor, este trabalho encontra-se estruturado da seguinte maneira:

- No capítulo 2 são apresentados os conceitos básicos envolvendo as redes de objetos. Esses incluem todas as definições necessárias para a compreensão formal e computacional deste paradigma. Inicia-se pela definição de objetos, passando-se por sistemas de objetos e por fim as redes de objetos, de uma maneira natural. As redes de objetos são então propostas como uma ferramenta teórico-formal para a modelagem de sistemas inteligentes. Também são colocados alguns exemplos da utilização das redes de objetos, com o objetivo de facilitar ao leitor a compreensão dos conceitos subjacentes a este paradigma. Apesar da maior parte deste capítulo ser inspirada no trabalho de Gudwin [Gudwin 96], uma série de definições foram reformuladas e outras acrescentadas, de modo a permitir os desenvolvimentos formais das redes de agentes, colocadas a seguir.

- No capítulo 3 introduz-se afinal as redes de agentes. Inicia-se com um preâmbulo, onde é feita uma introdução à teoria de agentes, que possibilitará situar o leitor no enfoque adotado para a terminologia utilizada. Logo a seguir são desenvolvidos os conceitos necessários para definir o que será denominado de rede de agentes. Após a definição deste novo conceito, apresenta-se um algoritmo que permite a obtenção de uma forma geral e uniforme para as funções de seleção - característica básica da rede de agentes. A descrição do algoritmo é seguida de um exemplo que ilustra a utilização do dito algoritmo. As principais contribuições formais introduzidas por esta tese, ou seja, as contribuições referentes ao aspecto conceptual do trabalho, são apresentadas neste capítulo.

- O capítulo 4 apresenta as contribuições práticas deste trabalho. É lá que se descreve a ferramenta computacional desenvolvida para implementar os conceitos mostrados no capítulo 3. A arquitetura básica da ferramenta é apreciada, seguida da explicação

dos passos necessários para que um usuário da ferramenta possa se servir da mesma para o *design* e simulação de sistemas, assim como as potencialidades disponíveis para sua utilização.

- No capítulo 5 apresentam-se dois exemplos da utilização do software desenvolvido, para a modelagem e simulação de sistemas inteligentes. Nesses exemplos tratou-se de utilizar a maioria dos recursos disponíveis na ferramenta computacional para eles também possam servir de guia ao usuário no futuro desenvolvimento de sistemas utilizando tal software.

- No capítulo 6 são colocadas as conclusões onde destacam-se as principais contribuições trazidas por este trabalho e a especulação sobre possíveis trabalhos futuros que permitam uma continuação do trabalho aqui desenvolvido, envolvendo o aperfeiçoamento da ferramenta apresentada no presente trabalho.

- Ao final, são apresentadas as referências bibliográficas citadas no texto.

CAPÍTULO 2. Redes de Objetos

2.1. Introdução

O desenvolvimento das Redes de Objetos está inspirado nas Redes de Petri [Murata 89] e a tecnologia orientada a objetos. As Redes de Petri constituem uma ferramenta gráfica e matemática de propósito geral para descrever relações existentes entre condições e eventos. As mesmas são consideradas como uma ferramenta adequada para análise e modelagem de sistemas caracterizados pela existência de concorrência, distribuição, paralelismo, com problemas de sincronização e não determinísticos. Por suas características gráficas podem ser utilizadas como uma ferramenta auxiliar no *design* de tais sistemas, contando com recursos de comunicação visual similar aos encontrados em fluxogramas ou diagramas de bloco. Como ferramenta matemática formal, são utilizadas para análise das propriedades e características do sistema modelado utilizando equações de estado, equações algébricas e outros modelos matemáticos.

Para estender o poder de representação das Redes de Petri e refinar sua funcionalidade, diferentes modelos foram desenvolvidos baseados em sua definição original, sendo essas extensões conhecidas como Redes de Petri de alto nível (*high-level Petri Nets*). Dentro desta categoria podemos citar as Redes de Petri do tipo Predicado-Transição (*Predicate-Transition Petri Nets*) [Genrich 81], Redes de Petri Coloridas (*Coloured Petri Nets*) [Jensen 90, Chistensen 92, Jensen 94, Jensen 97, Jensen 98] e, mais recentemente, as diferentes versões das Redes de Petri orientadas a objetos (*Object-Oriented Petri Nets*) [Janoušek 95a, Janoušek 95b, Janoušek 96, Ceška 96, Ceška 97a, Ceška97b, Janoušek 97, Newman 98, Vojnar 99a, Vojnar 99b]. Estas últimas combinam as potencialidades para a modelagem de sistemas das Redes de Petri com as potencialidades do *design* orientado a objeto para descrever sistemas grandes e complexos de maneira mais fácil e direta. Nenhuma destas abordagens viola a idéia original da estrutura das Redes de Petri, que é a existência de um conjunto de lugares, um conjunto de transições e um conjunto de arcos que conectam os lugares com as transições e vice-versa.

Outros autores tratarão de definir Redes de Petri com uma estrutura variável, como é o caso das Redes de Petri com *design* adaptativo (*Adaptative Design Petri Nets*) [Zhou 89], onde uma seqüência de Redes de Petri é gerada, dando a ilusão de uma rede que é adaptada por si mesma. Outro caso são as Redes de Petri auto-modificáveis (*Self-Modifying Petri Nets*) [Valk 78, Valk81]. Neste caso, os parâmetros dos arcos dependem do número de *tokens* em outros lugares. Todas estas extensões mantém a idéia original de se ter um número fixo de lugares e transições. Nestes casos as estruturas são intrinsecamente estáticas e o problema da utilização deste tipo de estrutura é sua incapacidade para modelar sistemas com aprendizagem e características adaptativas. Com o estudo dos sistemas inteligentes, a necessidade de representação do conhecimento e dos sistemas de tomada de decisões

baseados em regras, outros tipos de Redes de Petri apareceram. Um exemplo destes é a Rede de Petri Nebulosa (*Fuzzy Petri Nets*) [Pedrycs 94, Flenger 96].

As Redes de Objetos apresentam duas características que as distinguem quando são comparadas com as Redes de Petri. A primeira delas tem a ver com o aspecto estrutural. No enfoque das Redes de Objetos não existem transições. Só existe um conjunto de lugares e um conjunto de arcos que conecta estes lugares. A outra característica distintiva se refere ao aspecto funcional. Neste caso, os *tokens* são individualizados por objetos (isto já foi sugerido em algumas versões de Redes de Petri Orientadas a Objetos). Desta forma, os *tokens* não são mais todos iguais. Alguns objetos, chamados objetos ativos (aqueles que têm um número de métodos maior que zero), desempenham o papel de transições. Neste sentido os objetos que encontram-se nas Redes de Objetos podem agir às vezes como *tokens* (objetos passivos) e às veces como transições (objetos ativos). Como é possível a criação e destruição de objetos, podemos dizer que as Redes de Objetos têm um número variável de transições, o que não é permitido nas Redes de Petri pela sua própria definição. Por todos estes motivos, podemos afirmar que as Redes de Objetos não constituem um modelo de Rede de Petri estendido de alto nível, mas um modelo à parte, que apesar de sua inspiração em Redes de Petri e suas extensões, possui características que as diferenciam fundamentalmente das Redes de Petri.

A formulação das Redes de Objetos foi introduzida pela primeira vez por Gudwin [Gudwin 96, Gudwin98b], no contexto da análise, *design* e modelagem de sistemas inteligentes, tendo sido também utilizadas como uma ferramenta formal para o desenvolvimento da Semiótica Computacional [Gudwin 97a, Gudwin 97b, Gudwin 97c, Gudwin 99b] e simulação do ciclo semiótico (fluxo dos processos elementares estudados pela semiótica) em sistemas de computação. Na atualidade as Redes de Objetos têm sido utilizadas no contexto da inteligência computacional, computação flexível [Gudwin 97d, Gudwin98a] e computação com palavras [Gudwin 99a].

2.2. Objetos

A idéia de uma formulação matemática do conceito "objeto" está fortemente ligada ao conceito intuitivo e físico de objeto. Nesta seção primeiramente, analisamos uma definição conceitual de objetos, apresentamos algumas características dos objetos e em seguida passamos a uma definição formal de objeto, baseada na álgebra de conjuntos, como mostrado em [Gudwin 96].

Algumas das definições em [Gudwin 96] são reproduzidas aqui, sendo outras modificadas, de modo a tornar o material deste trabalho mais auto-contido.

2.2.1. O objeto conceptual

Nosso conceito de objeto é estritamente relacionado ao seu significado físico intuitivo. Um objeto pode ser definido ontologicamente como uma entidade do mundo real e

caracterizado por suas propriedades, que são catalogadas por meio de seus atributos [Wand 89]. A partir de um referencial de informações, é possível encontrar atributos que permitem distinguir os diferentes objetos. Estes atributos são utilizados para descrever os objetos.

Esta visão ontológica da idéia de objeto não considera que além de "existir" em um mundo real, os objetos também "atuam" sobre esse mundo real. Por esta razão um conceito matemático de objeto, além de descrevê-lo quanto ao seu aspecto existencial, também deve modelar o aspecto ativo destes objetos.

A conceptualização da idéia de objeto não pode, em princípio, ser feita de maneira independente. Apesar de podermos imaginar a existência de um objeto por si só, devemos considerar também sua capacidade de interação com diferentes objetos. Em outras palavras, para introduzir os principais conceitos sobre objetos, temos que nos referir a sistemas de objetos. Um sistema de objetos é um conjunto de entidades que existem e interagem entre si. Os componentes de um objeto, responsáveis pela interação, são mostrados na figura 2.1.

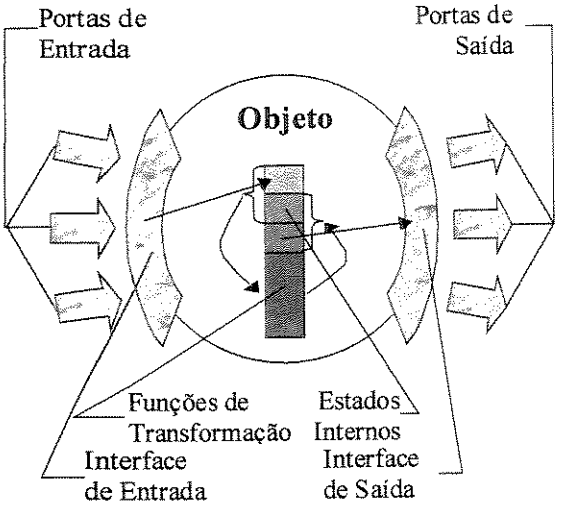


Figura 2.1 O objeto conceitual

Cada objeto ativo possui dois tipos de interface: uma interface de entrada e uma interface de saída (como é mostrado na figura 2.1). A interface de entrada é caracterizada por uma série de portas, chamadas de portas de entrada e a interface de saída é também composta por uma coleção de portas de saídas. Dentro de um objeto podemos encontrar seus estados internos. Estes estados são divididos em quatro regiões. A primeira região é uma cópia da interface de entrada, e a segunda região abrange as variáveis internas do objeto. A terceira região é uma cópia da interface de saída do objeto e a quarta e última região é um conjunto de funções de transformação (internas ao objeto). Objetos passivos não precisam das interfaces de entrada nem de saída. Isto é, objetos passivos não precisam da primeira e terceira região. Além disso, os objetos passivos, como seu nome indica, não têm atividades internas. Por isso, também não apresentam a quarta região. Em resumo, um objeto passivo só tem a região destinada a armazenar as variáveis de estado do objeto.

A interação entre objetos é regulada por um mecanismo chamado de ativação ou disparo, que é executado somente pelos objetos ativos. Neste mecanismo, o que caracteriza a primeira fase da ativação é que alguns objetos são conectados ao objeto ativo, através das portas de entrada (interface de entrada), começando assim o que é chamado de *fase de assimilação*. Nesta fase, o objeto ativo faz uma cópia dos estados internos dos objetos ligados a ele (pela interface de entrada) para seu estado interno (naquela região destinada a estes fins, ver figura 2.1). Depois da assimilação, os objetos conectados podem ser destruídos ou podem ser liberados de retorno ao sistema. Se eles são destruídos, então temos uma assimilação destrutiva (ou consumo). Caso contrário, temos uma assimilação não destrutiva. Na segunda fase do mecanismo de ativação, o objeto ativo usa uma das funções de transformação para mudar seus estados internos (lembremos que as interfaces de entrada e de saída são partes de seus estados internos). Esta fase é chamada de *fase de transformação*. Depois da fase de transformação, alguns dos estados internos do objeto ativo são copiados na interface de saída. Logo, outro conjunto de objetos é conectado às portas de saída (interface de saída), e os estados internos daqueles que estão presentes na interface de saída são mudados. Esta última fase é chamada *fase de geração* ou *fase de regeneração* e depende dos objetos que são ligados às portas de saída. Se os objetos conectados à interface de saída já existem, então este processo é chamado regeneração porque somente altera os estados internos dos objetos ligados. Porém, esta última fase, também pode criar um objeto novo, que não fazia parte do sistema de objetos original. Neste caso, a última fase cria este objeto novo, preenche seus estados internos com informação da interface de saída, e o libera para o sistema. Este processo é chamado geração. Todas estas fases são mostradas na figura 2.2.

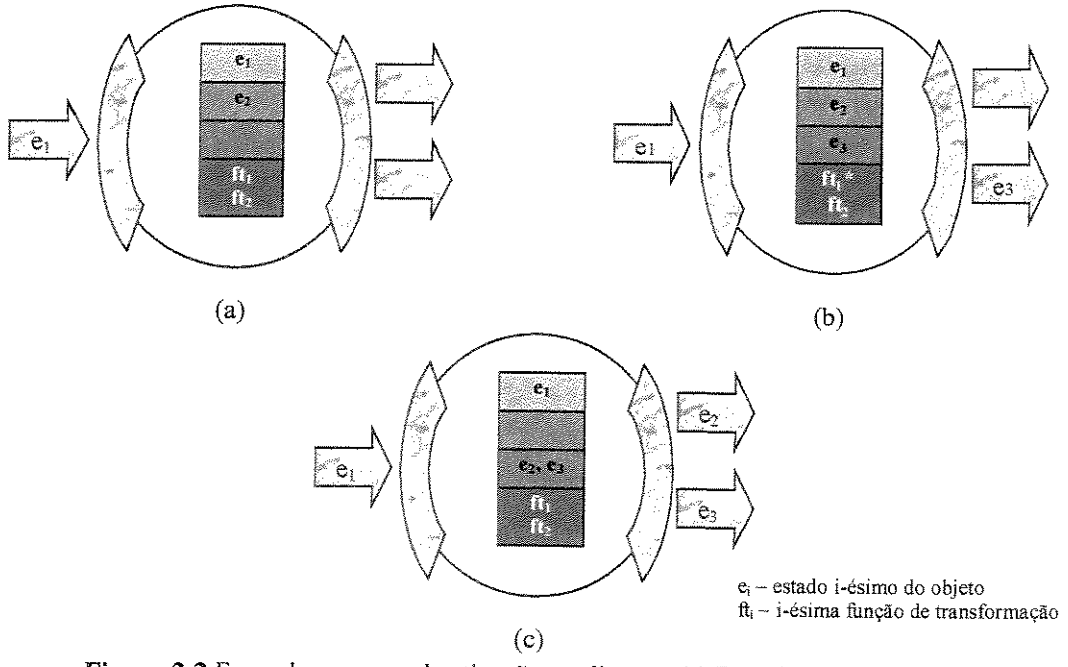


Figura 2.2 Fases do processo de ativação ou disparo. (a) Fase de assimilação. (b) Fase de transformação. (c) Fase de geração e/ou regeneração.

O mecanismo de ativação ou disparo pode permitir diferentes tipos de comportamento, como se mostra na figura 2.3. Neste exemplo, o objeto o_6 é o objeto ativo que executa o processo de disparo.

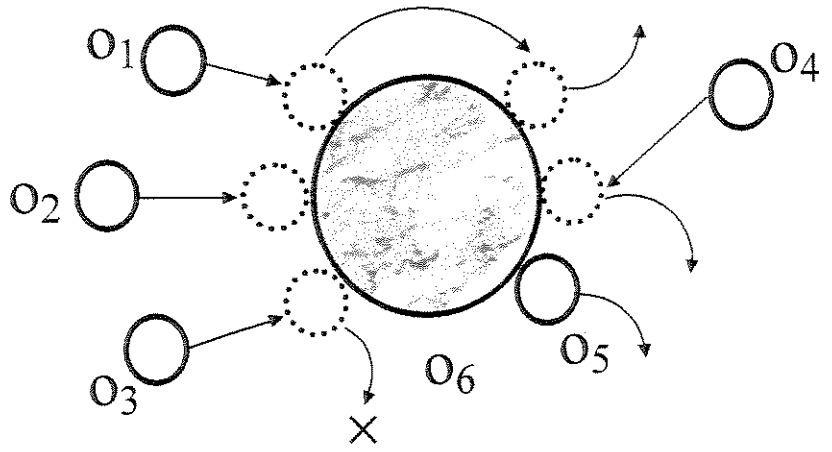


Figura 2.3 Interação entre objetos.

Os objetos o_1, o_2, o_3 e o_4 são os objetos a ser assimilados no processo de ativação. Os objetos o_1 e o_4 são regenerados, e o objeto o_5 é gerado. Note que o_1 é, ao mesmo tempo, assimilado e regenerado. O objeto o_2, depois da assimilação, é liberado de retorno ao sistema, mas o_3 é destruído.

Para o controle do processo de ativação, existe uma função especial associada a cada objeto chamada de *função de seleção*. Esta função decide quais objetos vão estar conectados, ou vão fazer parte das portas de entrada, quais vão estar ligados às portas de saída e qual das funções de transformação vai ser usada no processo de ativação. A estratégia de controle em um sistema de objetos é ditada pelas funções de seleção.

A função de seleção tem algumas restrições. Estas restrições são concernentes com as exigências das funções de transformação, bem como com a prevenção de possíveis problemas envolvendo a sincronização no disparo dos objetos. Cada função de transformação (interna) exige um conjunto mínimo de objetos (de onde retira informações) para começar o procedimento de ativação. Deste modo, a função de seleção tem que considerar a disponibilidade simultânea de todos os objetos necessários para *habilitar* uma função de transformação. Os problemas de sincronização que podem aparecer são relativos aos múltiplos objetos ativos conectados a um mesmo objeto. Para uma conexão de assimilação deve-se garantir que só um dos objetos ativos executa uma assimilação destrutiva. Se algum dos objetos assimilados, também está sendo regenerado, este deve ser regenerado através de um só objeto ativo. Neste caso, não pode ser assimilado de forma destrutiva. Assim, deverá haver uma política global para a função de seleção, assegurando que essas restrições sejam satisfeitas. A função de seleção será estudada de maneira detalhada, oportunamente nesta tese.

2.2.2. Princípios da existência e interação dos objetos

Para uma boa interação entre os objetos constituindo um "sistema de objetos", assume-se os seguintes princípios:

1. Os objetos são únicos e identificados por seu nome.
2. Cada objeto possui um conjunto de atributos e/ou partes.
3. Um objeto pode possuir um conjunto de funções de transformação.
4. Um objeto do sistema pode assimilar (destrutivamente ou não) e/ou gerar outro objeto do sistema.
5. Os objetos podem ser classificados hierarquicamente em função de seus atributos e funções de transformação.
6. A interação entre objetos se limita à assimilação e à geração de novos objetos por objetos do sistema.

Os objetos são únicos e identificados por seu nome

Por esse princípio se afirma a existência e unicidade de um objeto do sistema. Assim, dois objetos podem ser iguais em quase tudo, mas devem ter um nome diferente, de modo que sejam facilmente identificáveis como dois objetos diferentes.

Cada objeto possui um conjunto de atributos e/ou partes

Com este princípio garantimos que cada objeto possa ser caracterizado. Os atributos (propriedades) são características extraídas de diferentes domínios que são atribuídas ao objeto. Esses domínios constituem um "referencial de informações" (*frame of reference*) para os objetos. Além dos atributos, os objetos podem ser constituídos por partes. As partes são outros objetos, que por sua vez podem ser constituídos por atributos e partes. Note que esta definição é recursiva. Para finalizar a recursividade, devem existir objetos que não contenham partes, ou seja, sejam definidos somente por atributos. Estes objetos podem ser chamados de objetos simples, objetos primitivos ou objetos-base.

Um objeto pode possuir um conjunto de funções de transformação

Este princípio caracteriza a interação de um objeto num sistema de objetos. Um objeto **pode** possuir um conjunto de funções de transformação. Caso um objeto possua uma ou mais funções de transformação, ele é denominado um **objeto ativo.** Caso não as possua, é chamado um **objeto passivo**. Uma função de transformação é uma função que tem por domínio o conjunto de atributos e partes dos objetos externos conectados à interface de entrada do objeto e o conjunto de atributos e partes internos ao objeto. O contradomínio de uma função de transformação será o conjunto de atributos e partes internos ao objeto, assim como também o conjunto de atributos e partes dos objetos externos conectados à interface de saída do objeto. Uma função de transformação, quando executada, poderá alterar os atributos internos do objeto, bem como os atributos externos dos objetos que são partes do objeto.

Um objeto do sistema pode assimilar (destrutivamente ou não) e/ou gerar outro objeto do sistema

Este princípio tem muita ligação com o princípio anterior (a possibilidade de existência de funções de transformação), definindo o caráter ativo do objeto no sistema. Um objeto ativo pode assimilar (destrutivamente ou não) objetos do sistema, ou seja receber informações de outros objetos do sistema, destruindo-os ou não, modificar as informações internas do objeto por meio de suas funções de transformação, e gerar novos objetos para o sistema. Do mesmo modo que no caso anterior, um objeto ativo **pode** assimilar e/ou gerar outros objetos. Não necessariamente o fará. Somente os objetos ativos assimilam e/ou geram novos objetos (a assimilação destrutiva de um objeto também é chamada de "consumo" deste objeto). Os objetos passivos simplesmente existem, podendo ser assimilados (consumidos) e gerados por objetos ativos.

Objetos ativos que assimilam objetos mas não podem gerar nenhum outro objeto, ou seja, cujo contradomínio de suas funções de transformação refere-se somente a atributos internos, são chamados de **objetos vertedouros**. Analogamente, objetos ativos que somente geram novos objetos, não assimilando nenhum outro objeto, ou seja, cujo domínio de suas funções de transformação refere-se somente a atributos internos, são denominados de **objetos fontes**.

Os objetos podem ser classificados hierarquicamente em função de seus atributos e funções de transformação

Por esse princípio cria-se uma taxonomia e classificação dos objetos. Essa classificação é corporificada pela idéia de classe (ou tipo). Assim, cria-se uma relação de equivalência entre objetos que possuam um mesmo referencial de informações, um mesmo conjunto de partes e um mesmo conjunto de funções de transformação. Uma classe ou tipo passa a ser representada por uma lista de domínios de atributos, partes e funções de transformação, sendo que cada objeto que possuir domínios semelhantes é dito pertencente a tal classe. Dois tipos de hierarquias são identificadas. A primeira corresponde a uma **hierarquia de partes**, ou seja, um objeto é parte de outro, ou tem outro objeto como uma de suas partes. A segunda é a **hierarquia de tipos**, ou seja, caso um objeto tenha os mesmos domínios de atributos e partes, e mesmas funções de transformação de uma classe, e além disso possua ainda outros domínios de atributos, partes e funções de transformação (objeto mais especializado), ele será um objeto de uma nova classe, que herda as características da classe anterior (chamada sua superclasse), além de incorporar novas características. Do mesmo modo, um objeto pode herdar características de mais de uma classe. Nesse caso, diz-se existir uma herança múltipla.

A interação entre objetos se limita à assimilação (destrutiva ou não) e à geração de novos objetos por objetos do sistema

Com este princípio, assume-se que a dinâmica de um sistema de objetos é definida pelo mecanismo de assimilação e geração de objetos por objetos ativos do sistema. Ou seja,

nenhum mecanismo adicional é necessário para que o sistema evolua dinamicamente no tempo.

Outros conceitos ou propriedades referentes a objetos foram levantados por diferentes autores. Dentre eles destacam-se:

- Os objetos são abstrações.
- Os objetos provêem serviços.
- Objetos clientes fazem requisições de serviços.
- Os objetos são encapsulados.
- As requisições identificam os métodos (nesse caso as funções de transformação) a serem utilizados.
- As requisições podem referenciar seus objetos de origem.
- Novos objetos podem ser criados.
- Métodos podem ser genéricos.
- Objetos podem ser classificados em termos de seus serviços.
- Objetos podem ter uma implementação comum.
- Objetos podem partilhar a implementação parcialmente.

Estas propriedades encontram-se mais direcionadas para modelar a idéia de objeto em programação, tendo sido analisadas extensivamente em [Gudwin 96]. Os princípios colocados anteriormente abrangem de maneira uniforme todas estas propriedades.

2.2.3. O objeto formal

Nesta seção faremos algumas definições preliminares, incluindo uma definição formal de objeto, além de de alguns conceitos relacionados a esta.

DEFINIÇÃO 2.1 – ÊNUPLAS

Sejam $q_1, q_2, \ldots, q_n$ elementos genéricos pertencentes aos conjuntos $Q_1, Q_2, \ldots, Q_n$ respectivamente.

Define-se uma **ênupla** como o agrupamento dos elementos $q_1, q_2, \ldots, q_n$ formando um único elemento composto denominado q. Para representar o agrupamento de n elementos, utiliza-se uma notação especial, onde os elementos são separados por vírgulas, e a ênupla é delimitada por parênteses, conforme a seguir:

$$q = (q_1, q_2, \ldots, q_n) \in Q_1 \times Q_2 \times \ldots \times Q_n$$

O nome ênupla se refere a um agrupamento genérico de n elementos. Para um número específico de elementos agrupados, utilizam-se denominações particulares. Assim uma ênupla com dois elementos é um par, três elementos uma tripla, quatro elementos uma quádrupla, etc.

Os elementos que integram uma ênupla, chamados de suas componentes, podem ser referenciados por seu índice na ênupla, de acordo com a ordem em que aparecem na mesma.

Observações:

- O conjunto gerado pelo produto cartesiano de n conjuntos corresponde ao conjunto de todas as ênuplas que têm como sua $n-ésima$ componente um elemento do $n-ésimo$ conjunto. Assim, os elementos do produto cartesiano de n conjuntos são ênuplas.
- Uma ênupla não é um conjunto. Se uma ênupla fosse um conjunto, uma ênupla com apenas um elemento, seria um conjunto unitário, e não um elemento. Uma ênupla com apenas um elemento é o próprio elemento.
- As componentes de uma ênupla podem também ser ênuplas. Ênuplas deste tipo são chamadas de **ênuplas complexas**. Por exemplo: $q=(q_1,(q_{21},q_{22},q_{23}),q_3,q_4)$. A qualquer momento, pode-se referenciar a ênupla (q_{21},q_{22},q_{23}) por seu nome composto q_2, neste caso a ênupla ficaria assim: $q=(q_1,q_2,q_3,q_4)$.

DEFINIÇÃO 2.2 – ARIDADE DE UMA ÊNUPLA

Seja uma ênupla $q=(q_1,q_2,\ldots,q_n)$. Define-se a **aridade** da ênupla q representada por $Ar(q)$, como o número de elementos que constituem a ênupla q.

Observação:

- Para o caso de ênuplas complexas, a aridade diz respeito à ênupla principal.

Exemplos: $q=(q_1,\ldots,q_n),\ Ar(q)=n$

$q=(a,b,c),\ Ar(q)=3$

$q=(a,(b,c),d),\ Ar(q)=3$

$q=((a,b,c),(d,(e,f),g)),\ Ar(q)=2$

DEFINIÇÃO 2.3 – ÍNDICE DE REFERÊNCIA

Para a localização de uma componente em uma ênupla, associa-se um índice de referência do elemento dentro da ênupla. Para o caso de uma ênupla simples s, o índice de referência consiste de um número i, $1\le i\le Ar(s)$. Para o caso de ênuplas complexas, o índice de referência será uma ênupla i, onde cada elemento i_k desta ênupla corresponde a um sub-índice dentro da ênupla de nível k. Cada sub-índice pode assumir valores entre 1 e a aridade da ênupla no nível k. O índice de referência pode ser utilizado também para a identificação do domínio do elemento.

Exemplos: $s=(a,b,c),\ S=S_A\times S_B\times S_C$

$i=1\rightarrow s_i=a,\ S_i=S_A$

$i=2\rightarrow s_i=b,\ S_i=S_B$

$i=3\rightarrow s_i=c,\ S_i=S_C$

$$c = (a,(b,(c,d),e),f),\ C = C_A \times (C_B \times (C_C \times C_D) \times C_E) \times C_F$$
$$i = 1 \rightarrow c_i = a,\ C_i = C_A$$
$$i = 2 \rightarrow c_i = (b,(c,d),e),\ C_i = C_B \times (C_C \times C_D) \times C_E$$
$$i = (2,1) \rightarrow c_i = b,\ C_i = C_B$$
$$i = (2,2) \rightarrow c_i = (c,d),\ C_i = C_C \times C_D$$
$$i = (2,2,2) \rightarrow c_i = d,\ C_i = C_D$$

DEFINIÇÃO 2.4 – FÓRMULA DE INDUÇÃO

Sejam:

- Uma ênupla $q = (q_1, q_2, \ldots, q_n)$
- Uma expressão k formada por meio da gramática a seguir, onde i corresponde a um índice de referência da ênupla q:

  $k \leftarrow [i]$

  $i \leftarrow i,i$

  $i \leftarrow [i,i]$

A expressão k é chamada de uma **fórmula de indução**.

Exemplos: $k = [i_1,[i_2,i_3,i_4],i_5]$

$k = [[i_1,i_2],[i_3,[i_4,i_5]]]$

$k = [i_1,i_2,i_3]$

DEFINIÇÃO 2.5 – INDUÇÃO DE UMA ÊNUPLA

Sejam:

- Uma ênupla $q = (q_1, q_2, \ldots, q_n)$, definida em $Q = Q_1 \times \cdots \times Q_n$
- Uma fórmula de indução k

Define-se a **indução** de q segundo k, como uma nova ênupla $q_{(k)}$, onde os elementos da ênupla original são agrupados seguindo-se a fórmula de indução k, substituindo os colchetes por parênteses e os índices de referência i_j pelos elementos originais q_{i_j} da ênupla q. O domínio $Q_{(k)}$ de $q_{(k)}$ pode também ser obtido seguindo-se a fórmula de indução k, omitindo-se os colchetes externos da fórmula, substituindo-se os colchetes internos por parênteses, as vírgulas por × e os índices i_j pelos sub-domínios originais Q_{i_j} de Q.

Exemplos: $q = (a,b,c,d),\ Q = Q_1 \times Q_2 \times Q_3 \times Q_4$

$k_1 = [1,3,4,2],\ q_{(k_1)} = (a,c,d,b),\ Q_{(k_1)} = Q_1 \times Q_3 \times Q_4 \times Q_2$

$k_2 = [4,1],\ q_{(k_2)} = (d,a),\ Q_{(k_2)} = Q_4 \times Q_1$

$k_3 = [1,[2,3],4]$, $q_{(k_3)} = (a,(b,c),d)$, $Q_{(k_3)} = Q_1 \times (Q_2 \times Q_3) \times Q_4$
$r = (a,(b,c),d)$, $R = R_1 \times (R_2 \times R_3) \times R_4$
$k_4 = [1,(2,1),(2,2),3]$, $r_{(k_4)} = (a,b,c,d)$, $R_{(k_4)} = R_1 \times R_2 \times R_3 \times R_4$
$k_5 = [3,2]$, $r_{(k_5)} = (d,(b,c))$, $R_{(k_5)} = R_4 \times (R_2 \times R_3)$
$k_6 = [3,2,(2,1)]$, $r_{(k_6)} = (d,(b,c),b)$, $R_{(k_6)} = R_1 \times (R_2 \times R_3) \times R_2$

DEFINIÇÃO 2.6 – SUB-ÊNUPLA

Seja q uma ênupla e k uma fórmula de indução, conforme anteriormente.

Uma ênupla $q_{(k)}$ formada pela indução de q segundo k é chamada de uma **sub-ênupla** de q, se cada índice que aparece na fórmula de indução k é um índice unário, aparecendo uma única vez na fórmula e a fórmula só possui um par de colchetes.

Exemplos: Os descritos na definição 2.5 referentes às fórmulas de indução k_1, k_2 e k_5.

DEFINIÇÃO 2.7 – RELAÇÃO

Sejam n conjuntos $R_1,\ldots,R_n$ e $R = \{(r_{i1},\ldots,r_{in})\}$, $i = 1,\ldots,m$, um conjunto de m ênuplas de aridade n, onde $n > 1$ e $\forall i \in \{1,\ldots,m\}$, $\forall k \in \{1,\ldots,n\}$, $r_{ik} \in R_k$.

O conjunto R, $R \subseteq R_1 \times \cdots \times R_n$ é dito ser uma **relação** em $R_1 \times \cdots \times R_n$.

O produto cartesiano $R_1 \times \cdots \times R_n$ é chamado de **universo** da relação

DEFINIÇÃO 2.8 – PROJEÇÃO DE UMA RELAÇÃO

Seja $R = \{r_i\}$, $r_i = (r_{i1},\ldots,r_{in})$ uma relação $n-$*ária* definida em $R_1 \times \cdots \times R_n$. Seja k uma fórmula de indução formada apenas por índices unários, $k = [k_1,k_2,\ldots,k_m]$, $k_i \in \{1,\ldots,n\}$, $k_i \neq k_j$, se $i \neq j$, $i = 1,\ldots,m$, $j = 1,\ldots,m$, $m \leq n$.

Define-se a **projeção** de R segundo k, $R \downarrow k$, ou alternativamente, $R_{(k)}$, como a relação obtida pela união de todas as ênuplas $r_{i(k)} = (r_{ik_1},\ldots,r_{ik_m})$, obtidas a partir da indução de cada ênupla de R segundo k:

$$R_{(k)} = \cup r_{i(k)}$$

Exemplos:

$A = \{1,2\}$ $B = \{a,b,c\}$ $C = \{\alpha,\beta,\gamma\}$. $R = \{(1,a,\beta),(2,c,\alpha),(2,b,\beta),(2,c,\beta)\}$
$k = [1,3]$, $R \downarrow k = R_{(k)} = \{(1,\beta),(2,\alpha),(2,\beta)\}$
$k = [3,2]$, $R \downarrow k = R_{(k)} = \{(\beta,a),(\alpha,c),(\beta,b),(\beta,c)\}$

Observações:

- Os elementos da ênupla que aparecem na sub-ênupla não necessariamente precisam aparecer na mesma ordem que na ênupla original.
- Sub-ênuplas de diferentes ênuplas de R que sejam iguais, aparecem somente uma vez na projeção. Com isso, o número de elementos da projeção de R será sempre menor ou igual ao número de elementos de R.

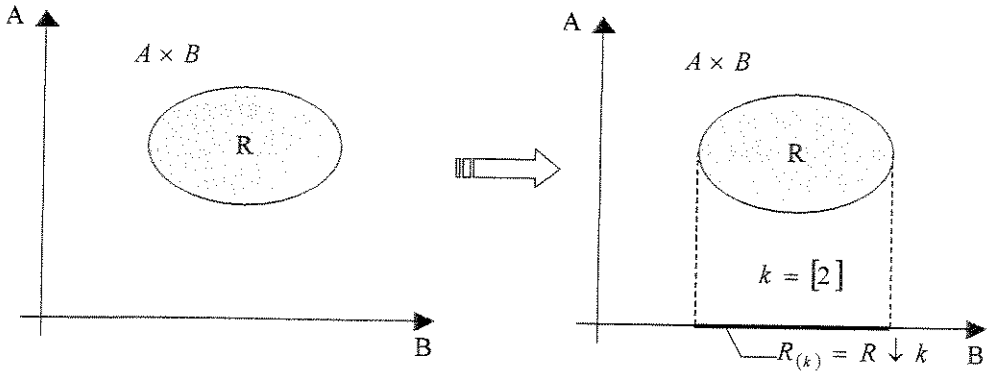


Figura 2.4 Exemplo de Projeção de uma Relação.

DEFINIÇÃO 2.9 – PROJEÇÃO LIVRE DE UMA RELAÇÃO

Sejam $R = \{r_i\}$, uma relação definida em U e k uma fórmula de indução.

Define-se a **projeção livre** de R em $U_{(k)}$, $R \downarrow U_{(k)}$, ou alternativamente, $R_{(k)}$ como a relação obtida pela união de todas as sub-ênuplas $r_{i(k)}$ de R, obtidas pela indução das ênuplas de R segundo k:

$$R_{(k)} = \cup r_{i(k)}$$

Observações:

- A projeção livre é uma generalização do conceito de projeção.
- Na projeção, somente as ênuplas induzidas por fórmulas formadas por índices unários são consideradas, o que implica em ênuplas definidas sobre as dimensões principais do domínio da relação.
- Para a projeção livre qualquer ênupla induzida pode ser utilizada. Isso implica que se a fórmula de indução for formada por índices unários que aparecem apenas uma vez, então a projeção livre torna-se uma projeção.

DEFINIÇÃO 2.10 – FÓRMULA DE EXTENSÃO

Sejam:

- Uma ênupla $q = (q_1, q_2, \ldots, q_n)$.
- Uma expressão k formada por meio da gramática a seguir, onde i corresponde a um índice de referência da ênupla q, e c corresponde a um conjunto:

$k \leftarrow [a]$
$a \leftarrow a, a$
$a \leftarrow [a, a]$
$a \leftarrow i$
$a \leftarrow c$

A expressão k é chamada de uma **fórmula de extensão**.

Exemplos: $k = [i_1, [i_2, c_1], c_2]$
$k = [c_1, i_1, c_2]$
$k = [i_1, [c_1, c_2], [i_2, i_3, c_3]]$

DEFINIÇÃO 2.11 –EXTENSÃO DE UMA ÊNUPLA

Sejam:

- Uma ênupla $q = (q_1, q_2, \ldots, q_n)$, definida em $Q = Q_1 \times \cdots \times Q_n$.
- $\{C_1, \ldots, C_m\}$, onde cada C_j é um conjunto.
- Uma fórmula de extensão k, contendo índices de referência da ênupla q e conjuntos $C_j \in \{C_1, \ldots, C_m\}$.

Define-se a **extensão** de q segundo k, como um conjunto de ênuplas $q^{(k)}$, formadas da seguinte maneira:

Passo 1: Seguindo-se a fórmula de extensão k, substitui-se os colchetes por parênteses, e os índices de referências i_j pelos elementos originais q_{i_j} da ênupla q.

Passo 2: A ênupla resultante do passo 1 é transformada em um conjunto de ênuplas, substituindo-se uma ênupla contendo um conjunto C_j em uma dada posição por um conjunto de ênuplas, de tal forma que cada uma delas contenha um diferente elemento de C_j na posição onde C_j se encontra originalmente. Este passo é repetido em todas as ênuplas geradas até que todos os conjuntos C_j sejam substituídos.

Exemplo: $q = (a, b, c)$, $A = \{1, 2\}$, $B = \{\alpha, \beta\}$, $k = [2, [1, A], B]$

Passo 1:
$[2, [1, A], B] \rightarrow (b, (a, A), B)$
Passo 2:
$(b, (a, A), B) \rightarrow \{(b, (a, 1), B), (b, (a, 2), B)\} \rightarrow$
$\{(b, (a, 1), \alpha), (b, (a, 1), \beta), (b, (a, 2), \alpha), (b, (a, 2), \beta)\}$
$q^{(k)} = \{(b, (a, 1), \alpha), (b, (a, 1), \beta), (b, (a, 2), \alpha), (b, (a, 2), \beta)\}$
$q = (a, b, c)$, $A = \{1, 2, 3\}$, $B = \{\alpha, \beta\}$, $k = [3, 1, A]$
Passo 1:
$[3, 1, A] \rightarrow (c, a, A)$

Passo 2:
$(c,a,A) \rightarrow \{(c,a,1),(c,a,2),(c,a,3)\}$
$q^{(k)} = \{(c,a,1),(c,a,2),(c,a,3)\}$

DEFINIÇÃO 2.12 – EXTENSÃO CILÍNDRICA DE UMA RELAÇÃO

Sejam:

- $R = \{r_i\}$, $r_i = (r_{i1},\ldots,r_{in})$ uma relação $n-$*ária* definida em $R_1 \times \cdots \times R_n$.
- k uma fórmula de extensão.

Define-se a **extensão cilíndrica** P de R segundo k, $P = R \uparrow k$, ou alternativamente, $R^{(k)}$ como a relação obtida pela união das extensões de todas as ênuplas de R segundo k:

$$R^{(k)} = \cup r_i^{(k)}$$

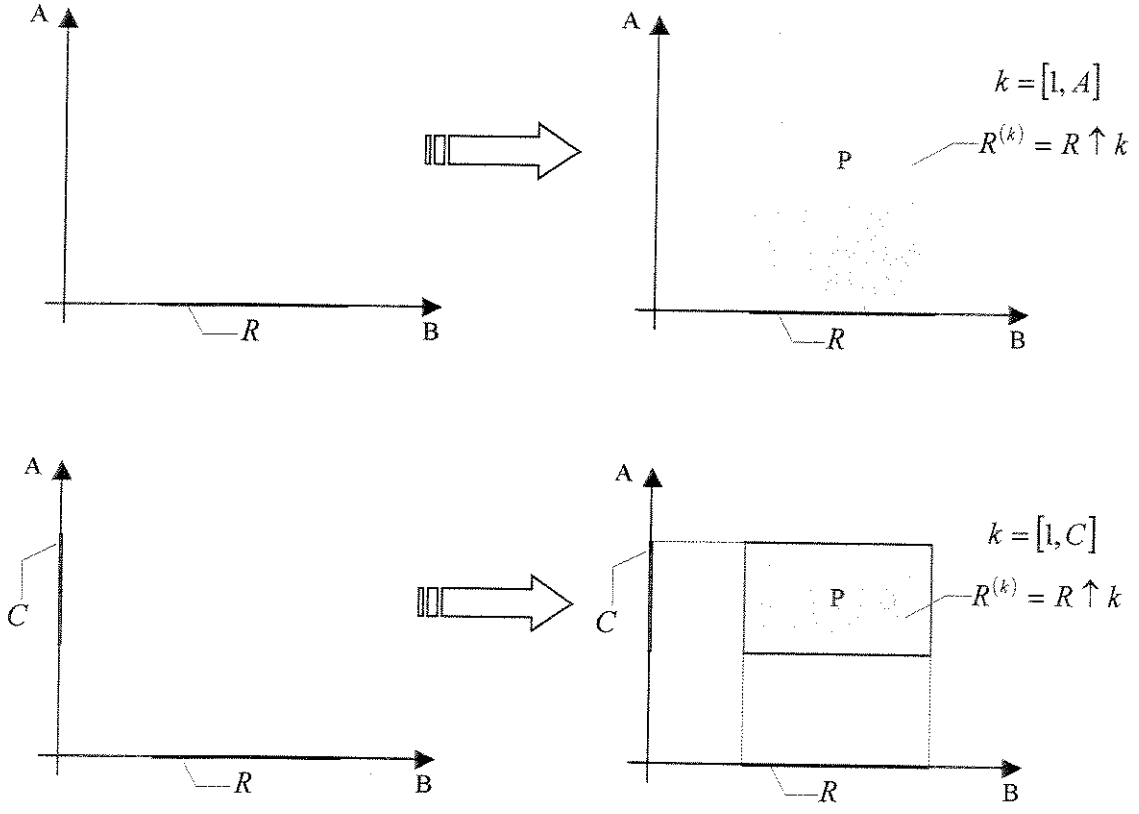


Figura 2.5 Exemplos de Extensão Cilíndrica de uma Relação.

Exemplos: $A = \{1,2\}\ B = \{a,b,c\}\ C = \{\alpha,\beta,\gamma\}.\ R = \{(1,a),(2,c)\}$

$k_1 = [1,2,C],\ R \uparrow k_1 = P_1 = (1,a)^{(k_1)} \cup (2,c)^{(k_1)}$

$P_1 = \{(1,a,\alpha),(1,a,\beta),(1,a,\gamma)\} \cup \{(2,c,\alpha),(2,c,\beta),(2,c,\gamma)\}$

$P_1 = \{(1,a,\alpha),(1,a,\beta),(1,a,\gamma),(2,c,\alpha),(2,c,\beta),(2,c,\gamma)\}$

$$k_2 = [C,1,2],\ R \uparrow k_2 = P_2 = (1,a)^{(k_2)} \cup (2,c)^{(k_2)}$$
$$P_2 = \{(\alpha,1,a),(\beta,1,a),(\gamma,1,a)\} \cup \{(\alpha,2,c),(\beta,2,c),(\gamma,2,c)\}$$
$$P_2 = \{(\alpha,1,a),(\beta,1,a),(\gamma,1,a),(\alpha,2,c),(\beta,2,c),(\gamma,2,c)\}$$

Observação:

- Os elementos da ênupla que aparecem na sub-ênupla não necessariamente precisam aparecer na mesma ordem que na ênupla original.

DEFINIÇÃO 2.13 – JUNÇÃO DE RELAÇÕES

Sejam:

- $R = \{r_i\},\ r_i = (r_{i1},\ldots,r_{in})$ uma relação $n-$*ária* definida em $R_1 \times \cdots \times R_n$.
- $S = \{s_j\},\ s_i = (s_{j1},\ldots,s_{jm})$ uma relação $m-$*ária* definida em $S_1 \times \cdots \times S_m$.
- k_1 e k_2 duas fórmulas de extensão tais que:

$$R_1 \times \ldots \times R_n \uparrow k_1 = S_1 \times \ldots \times S_m \uparrow k_2 = P$$

Define-se a **junção** das relações R e S em P, denotada por $R * S|_P$ como sendo:

$$R * S|_P = (R \uparrow k_1) \cap (S \uparrow k_2)$$

Exemplos: $A = \{1,2\}\ B = \{a,b,c\}\ C = \{\alpha,\beta,\gamma\}$,

$$R \subset A \times B,\ R = \{(1,a),(2,c)\},$$
$$S \subset B \times C,\ S = \{(a,\alpha),(b,\beta)\}$$
$$k_1 = [1,2,C],\ k_2 = [A,1,2],\ P = A \times B \times C$$
$$R * S|_{A \times B \times C} = \{(1,a,\alpha)\}$$
$$k_1 = [1,2,B,C],\ k_2 = [A,B,1,2],\ P = A \times B \times B \times C$$
$$R * S|_{A \times B \times B \times C} = \{(1,a,a,\alpha),(1,a,b,\beta),(2,c,a,\alpha),(2,c,b,\beta)\}$$

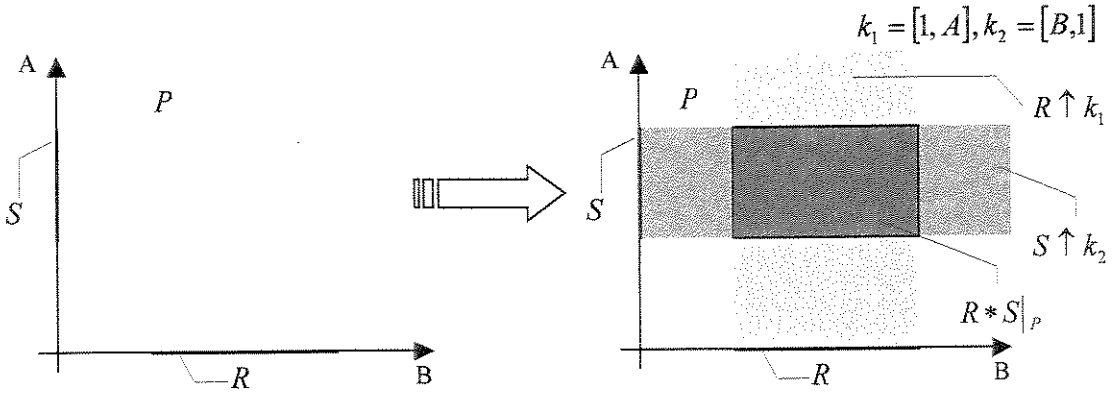


Figura 2.6 Exemplo de Junção de Relações.

DEFINIÇÃO 2.14 – VARIÁVEL

Sejam T um conjunto enumerável, com um elemento genérico t e $X \subseteq U$ um subconjunto de um universo U. A variável x de tipo X é uma função $x : T \to X$. Note que uma função é também uma relação, que por sua vez pode ser expressa por um conjunto. Sendo assim, $x \subset T \times X$.

Exemplos: $T = \{1,2,3\}$, $X = \{a,b,c\}$, $x(1) = a$, $x(2) = b$, $x(3) = c$
$x = \{(1,a),(2,b),(3,c)\}$
$T = \{1,2,3,\ldots\}$, $X = \{a,b,c\}$, $x(1) = a$, $x(2) = b$, $x(3) = c$, ...
$x = \{(1,a),(2,b),(3,c),\ldots\}$

DEFINIÇÃO 2.15 – VARIÁVEL COMPOSTA

Seja x uma variável de tipo X. Se os elementos de X são ênuplas, a variável x é chamada uma **variável composta** ou **estrutura**.

O valor de uma variável composta, em um determinado instante de tempo, será sempre uma ênupla. Os valores individuais de cada sub-elemento dessa ênupla podem ser obtidos, referenciados por seu índices na ênupla, também conhecidos como campos da variável. De um modo especial, se o conjunto X corresponde ao produto cartesiano de n conjuntos X_i, ou seja, $X = X_1 \times \cdots \times X_n$, cada campo da variável pode ser visto como uma projeção livre de x em $N \times X_i$, ou seja, é uma variável simples de tipo X_i.

Exemplo:
$N = \{1,2,3\}, X_1 = \{a,b\}, X_2 = \{c,d\}, X = X_1 \times X_2 = \{(a,c),(a,d),(b,c),(b,d)\}$
$x = \{(1,(a,c)),(2,(a,d)),(3,(a,d))\}$
$x \downarrow N \times X_1 = \{(1,a),(2,a),(3,a)\}$
$x \downarrow N \times X_2 = \{(1,c),(2,d),(3,d)\}$

DEFINIÇÃO 2.16 – VARIÁVEL DE CONJUNTO

Sejam T um conjunto enumerável, com um elemento genérico t e $X \subseteq U$ um subconjunto de um universo U. Define-se uma **variável de conjunto** de tipo X como uma função $x : T \to 2^X$.

Exemplos: $T = \{1,2,3\}$, $X = \{a,b,c,d\}$
$x = \{(1,\{a,b\}),(2,\{b,c,d\}),(3,\{a,c\})\}$
$T = \{1,2,3\}$, $X = \{a,b,c,d\}$, $X^2 = X \times X = \{(a,a),(a,b),(a,c),(a,d),\ldots\}$
x de tipo X^2, $x = \{(1,\{(a,b)\}),(2,\{(b,c),(a,d)\}),(3,\{(a,c),(c,d),(b,d)\})\}$

DEFINIÇÃO 2.17 – DESCRITOR DE UMA CLASSE

Define-se o **descritor de uma classe** C, $d(C)$ como sendo uma ênupla $a=(a_1,\ldots,a_n)$, onde cada a_i pode ser , ou um conjunto, ou um descritor de função.

Um **descritor de função** é uma ênupla $b=(b_1,b_2)$, utilizada para descrever o domínio e o contradomínio de uma função da seguinte forma:

- b_1 é uma ênupla que representa o domínio da função:
  $b_1=(b_{11},\ldots,b_{1k})$, tal que: $1\le b_{1i}\le n$, $b_{1i}\neq b_{1j}$ se $i\neq j$ e $a_{b_{1i}}$ é um conjunto ou $b_1=(0)$.
- b_2 é uma ênupla que representa o contradomínio da função:
  $b_2=(b_{21},\ldots,b_{2l})$, tal que: $1\le b_{2i}\le n$, $b_{2i}\neq b_{2j}$ se $i\neq j$ e $a_{b_{2i}}$ é um conjunto ou $b_2=(0)$.

Exemplos: Sendo V_1,V_2,V_3,V_4,V_5 conjuntos quaisquer, e C_1 e C_2 duas classes que se deseja representar, $d(C_1)=(V_1,V_2,((1),(2)))$, $d(C_2)=(V_1,V_2,V_3,((1,2),(5)),V_4,V_5,((6),(3)))$

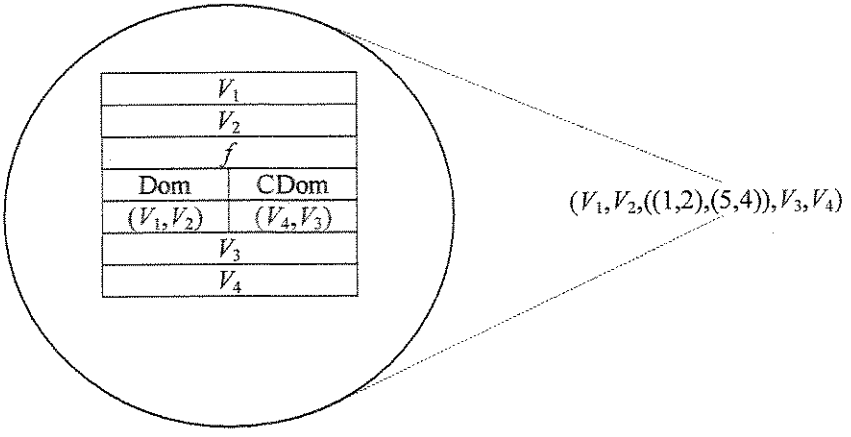


Figura 2.7 Exemplo de descritor de uma classe

Observe que um elemento da ênupla que pertence ao descritor de uma função pode ser (0), o que significa que essa função não tem domínio ou contradomínio. Esse tipo paradoxal de função (chamado aqui de *função* somente para manter a homogeneidade de terminologia) será utilizado na modelagem de objetos ativos do tipo fonte ou vertedouro.

DEFINIÇÃO 2.18 – CONCORDÂNCIA COM UM DESCRITOR DE UMA CLASSE

Uma ênupla $c=(c_1,\ldots,c_n)$ diz-se em **concordância com um descritor de uma classe** $a=(a_1,\ldots,a_m)$, denotando-se $c\lhd a$, se:

- $Ar(c)=n=m=Ar(a)$

- cada um dos elementos c_j de c segue o padrão definido pelos elementos a_j de a, ou seja:
  - se a_j é um conjunto, então $c_j \in a_j$
  - se a_j é um descritor de função $((b_{11}, b_{12}, \ldots, b_{1k}), (b_{21}, b_{22}, \ldots, b_{2l}))$, então c_j é uma função cujo domínio é $a_{b_{11}} \times a_{b_{12}} \times \cdots \times a_{b_{1k}}$ e o contradomínio é $a_{b_{21}} \times a_{b_{22}} \times \cdots \times a_{b_{2l}}$ e $a_{b_{11}}, a_{b_{12}}, \ldots, a_{b_{1k}}, a_{b_{21}}, a_{b_{22}}, \ldots, a_{b_{2l}}$ são conjuntos.

DEFINIÇÃO 2.19 – CLASSE

Uma **classe** C é o conjunto de todas as ênuplas $c_i = (c_{i1}, \ldots, c_{in})$, que estão em concordância com seu descritor de classe $d(C)$.

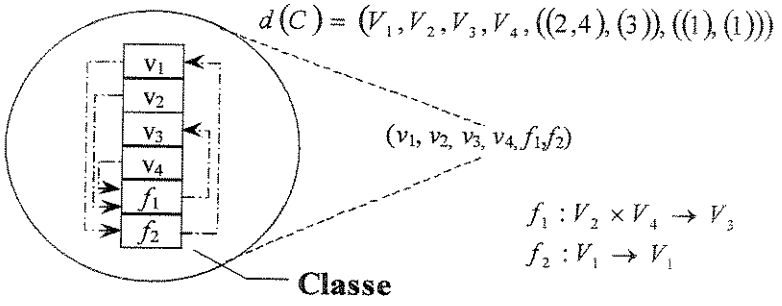


Figura 2.8 Representação de uma Classe

DEFINIÇÃO 2.20 – OBJETO

Seja C uma classe não vazia. Seja c uma variável de tipo C. A variável c é chamada então de um **objeto** da classe C.

Observações:

- A definição de um objeto pode encampar outros objetos. Uma vez que um objeto é uma variável cujos valores são ênuplas, que possuem elementos v_i pertencentes a conjuntos V_i, se esses conjuntos V_i forem por sua vez classes, então uma variável de V_i também será um objeto. Nesse caso dizemos que tal objeto é uma **parte** do outro objeto, dessa forma podemos ter objetos que são constituídos por partes, que por sua vez também são objetos.
- Se o descritor de uma classe $d(C)$ da classe C tem aridade 1 ($Ar(d(C)) = 1$) e esse elemento de $d(C)$ é um conjunto, então a ênupla se reduz a um único elemento. Assim, uma variável é também um objeto. Uma variável nestas condições é chamada de um **objeto primitivo**.

- Se o descritor de uma classe $d(C)$ da classe C tem aridade 0 ($Ar(d(C)) = 0$), então essa classe é chamada **classe vazia**, mas pela definição de objeto, não pode existir um objeto desta classe.

DEFINIÇÃO 2.21 – OBJETO GENÉRICO

Seja C uma classe não vazia. Seja c uma variável de conjunto de tipo C. A variável c é chamada então de um **objeto genérico** da classe C.

DEFINIÇÃO 2.22 – INSTÂNCIA DE UM OBJETO

Seja c um objeto de uma classe C. Define-se como a instância de um objeto em um instante n o valor de c nesse instante: $c(n)$. Lembrando-se que C é um conjunto de ênuplas, a instância de um objeto será um elemento de C, no caso, uma ênupla. Note que a instância de um objeto c em um instante n é um elemento de C.

DEFINIÇÃO 2.23 – SUPERCLASSE E SUBCLASSE

Seja C uma classe. Um conjunto D cujos elementos são sub-ênuplas dos elementos de C, todas pertencentes a um mesmo universo, de tal forma que para cada elemento em C corresponda um elemento em D, e D é uma classe, é chamado uma **superclasse** de C. Nesse caso, C é chamada de **subclasse** de D. Observe que uma classe pode ser definida a partir da definição de uma ou mais classes primitivas. Lembrando que uma classe é uma relação, uma classe pode ser gerada pela extensão cilíndrica de uma classe, pela junção de diversas classes ou mesmo pela extensão cilíndrica da junção de diversas classes. Em todos esses casos, as classes primitivas são superclasses da classe originada.

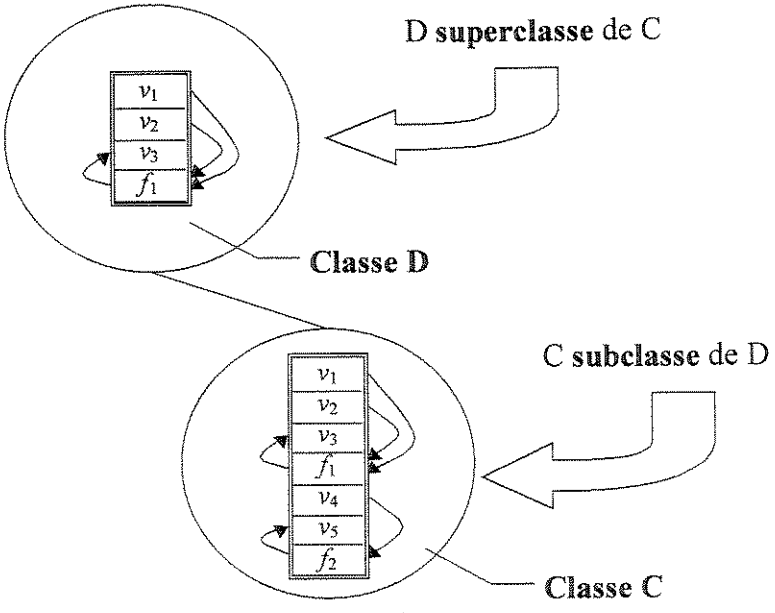


Figura 2.9 Exemplo de Superclasse e Subclasse.

PROPOSIÇÃO 2.1 – HIERARQUIA DE CLASSES

As definições de classe, projeção, extensão cilíndrica e junção induzem uma hierarquia entre as classes, onde a projeção de uma classe que também é uma classe corresponde a uma superclasse desta. De maneira reversa, a partir de uma classe e de sua extensão cilíndrica, obtém-se uma subclasse da classe original. Analogamente a junção de duas classes é uma subclasse de ambas classes originais. Note que qualquer classe é subclasse da classe vazia.

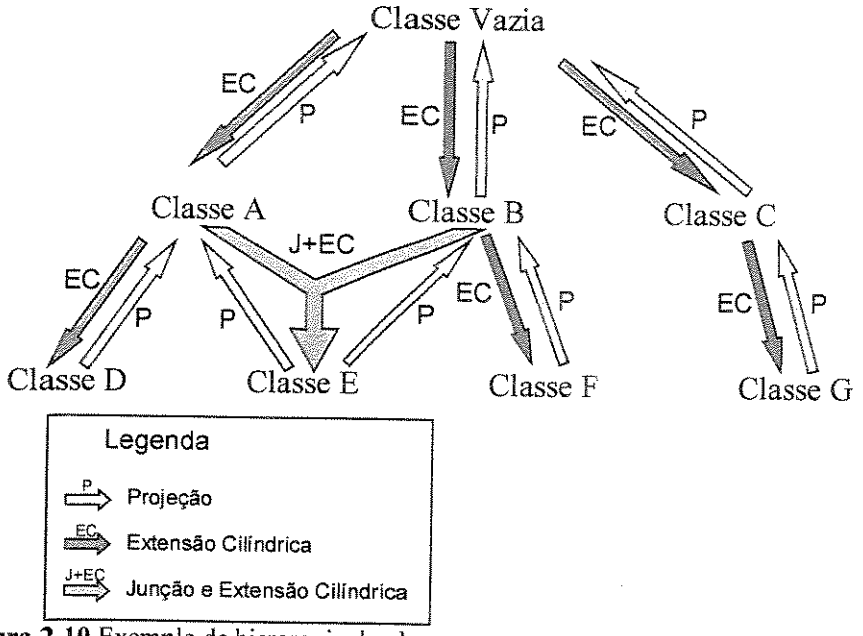


Figura 2.10 Exemplo de hierarquia de classes

DEFINIÇÃO 2.24 – PROJEÇÃO OBJÉTICA

Sejam

- c um objeto da classe C
- $g = [g_1, g_2, \ldots, g_o]$, $o \in \mathrm{N}$, uma fórmula de indução
- $g^* = [g_1^*, g_2^*, \ldots, g_o^*]$, $o \in \mathrm{N}$, uma expressão obtida através de g da seguinte maneira:

$$g_i \rightarrow (2, g_i) = g_i^*$$
$$(g_i) \rightarrow (2, g_i) = g_i^*$$

Define-se a **projeção objética** q de c segundo g, denotando-se $q = c \downarrow g$, como sendo o objeto obtido pela projeção livre de c segundo k, onde $k = [1, g^*]$.

Exemplo: $c = \{(n, (v_1(n), v_2(n), v_3(n)))\}$, $n \in N$

$$g = [1,3], g^* = [(2,1),(2,3)], k = [1, [(2,1),(2,3)]]$$
$$q = c \downarrow g = \{(n, (v_1(n), v_3(n)))\}, \; n \in N$$

$$c = \{(n,(v_1(n),(v_{21}(n),v_{22}(n)),v_3(n)))\},\ n \in N$$
$$g = [(2,1)],\ g^* = [[(2,2,1)]],\ k = [1,[(2,2,1)]]$$
$$q = c \downarrow g = \{(n,(v_{21}(n)))\},\ n \in N$$

DEFINIÇÃO 2.25 – SUB-OBJETO

Seja c um objeto de uma classe C e d um objeto de uma classe D, que é uma superclasse de C, ou seja, $D = C \downarrow X$. Se para todos os instantes n, as instâncias de d corresponderem às sub-ênuplas das instâncias de c, d é chamado de um **sub-objeto** de c. Matematicamente, d corresponde à projeção objética de c segundo X (que deve sempre existir, visto que D é superclasse de C), isto é, $d = c \downarrow X$.

Exemplo: $d = \{(n,(v_1(n),v_2(n)))\},\ n \in N$
$$c = \{(n,(v_1(n),v_2(n),v_3(n),v_4(n)))\},\ n \in N$$
$$X = [1,2],\ d = c \downarrow X$$

DEFINIÇÃO 2.26 – OBJETOS ATIVOS E PASSIVOS

Um objeto c de uma classe C é chamado de um **objeto ativo** se um dos componentes do descritor da classe $d(C)$ é um descritor de função; caso que não exista um componente do $d(C)$ que seja um descritor de função, então c é chamado de **objeto passivo**.

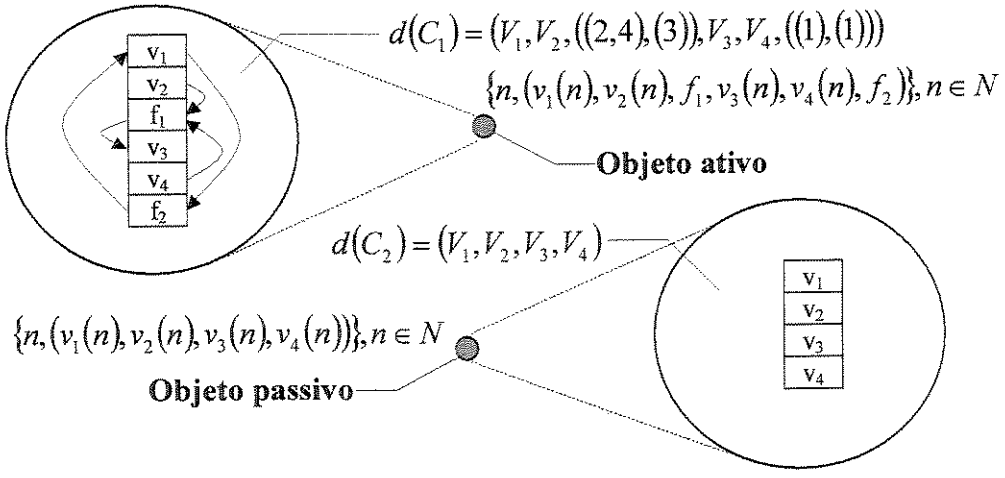


Figura 2.11 Exemplo de um objeto ativo e um objeto passivo

DEFINIÇÃO 2.27 – INTERFACE DE ENTRADA

Sejam:

- $d(C) = (a_1, a_2, \ldots, a_n)$ o descritor de uma classe C

- N_f o conjunto dos índices i, tais que, os a_i são descritores de função, da forma $a_i = \left(\left(b^i_{11},\ldots,b^i_{1K_i}\right),\left(b^i_{21},\ldots,b^i_{2L_i}\right)\right)$
- Os conjuntos P e Q formados por:

$$P = \bigcup_{i\in N_f}\bigcup_{j=1}^{k_i} b^i_{1j} \qquad \text{e} \qquad Q = \bigcup_{i\in N_f}\bigcup_{j=1}^{L_i} b^i_{2j}$$

- O conjunto $R = P - \{P \cap Q\}$
- $g = [g_1, g_2, \ldots, g_O]$ uma fórmula de indução, tal que, $\forall i \in \{1,\ldots,o\}$, $o \in \mathrm{N}$, $g_i \in R$ e $\forall x \in R$, $\exists g_i = x$
- c um objeto ativo de uma classe C e I uma superclasse de C, tal que, $C \downarrow g = I$

Define-se a **interface de entrada** i do objeto c, como o objeto gerado pela projeção objética de c segundo g, ou seja, $i = c \downarrow g$.

Exemplo: $d(C) = (V_1, V_2, V_3, V_4, V_5, ((1,2),(3)), ((2,4),(5,2)))$

$C = (v_1, v_2, v_3, v_4, v_5, f_1, f_2)$

$N_f = \{6,7\}$

$P = \{1,2,4\}$, $Q = \{3,5,2\}$, $R = \{1,4\}$

$g = [1,4]$

$C \downarrow g = (v_1, v_4)$

$c = \{(n, (v_1(n), v_2(n), v_3(n), v_4(n), v_5(n), f_1, f_2))\}$, $n \in N$

$i = c \downarrow g = \{(n, (v_1(n), v_4(n)))\}$, $n \in N$

$$d(C) = (V_1, V_2, V_3, ((1,2),(3)), V_4, V_5, ((4),(5)))$$

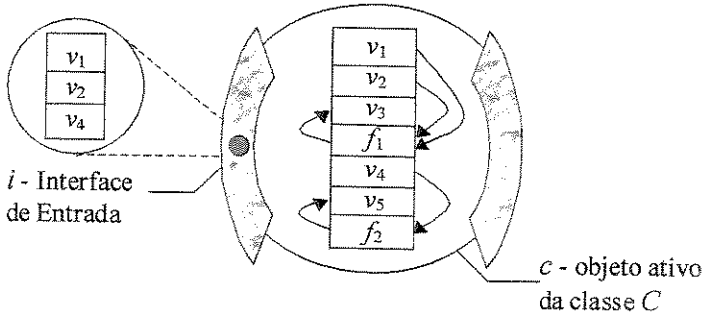


Figura 2.12 Exemplo de Interface de Entrada

DEFINIÇÃO 2.28 – INTERFACES DE ENTRADA ESPECÍFICAS A FUNÇÃO

Sejam:

- $d(C) = (a_1, a_2, \ldots, a_n)$ o descritor de uma classe C

- c um objeto ativo de uma classe C
- I^j uma superclasse de I e C
- N_f o conjunto dos índices i, tais que, os a_i são descritores de função, da forma $a_i = \left(\left(b^i_{11},\ldots,b^i_{1K_i}\right),\left(b^i_{21},\ldots,b^i_{2L_i}\right)\right)$, sendo $a_j = \left(\left(b^j_{11},\ldots,b^j_{1K_j}\right),\left(b^j_{21},\ldots,b^j_{2L_j}\right)\right)$ o descritor de função específico para a função j
- Os conjuntos P_j e Q formados por:

$$P_j = \bigcup_{n=1}^{k_j} b^j_{1n} \qquad \text{e} \qquad Q = \bigcup_{i\in N_f}\bigcup_{j=1}^{L_i} b^i_{2j}$$

- O conjunto $R_j = P_j - \{P_j \cap Q\}$
- $g^j = \left[g^j_1, g^j_2,\ldots,g^j_o\right]$ uma fórmula de indução, tal que, $\forall i \in \{1,\ldots,o\}$, $o \in \mathrm{N}$, $g^j_i \in R$ e $\forall x \in R$, $\exists g^j_i = x$

Define-se a **interface de entrada específica à função** j do objeto c, i^j como o objeto passivo obtido pela projeção objética de c segundo g^j, ou seja, $i^j = c \downarrow g^j$.

Exemplo: $d(C) = (V_1,V_2,V_3,V_4,V_5,((1,2),(3)),((2,4),(5,2)))$

$C = (v_1,v_2,v_3,v_4,v_5,f_1,f_2)$

$N_f = \{6,7\}$

$a^1 = ((1,2),(3))$ o descritor da função f_1

$a^2 = ((2,4),(5,2))$ o descritor da função f_2

$P_1 = \{1,2\}$, $Q = \{3,5,2\}$, $R_1 = \{1\}$

$P_2 = \{2,4\}$, $Q = \{3,5,2\}$, $R_2 = \{4\}$

$g^1 = [1]$

$g^2 = [4]$

$C \downarrow g^1 = (v_1)$

$C \downarrow g^2 = (v_4)$

$c = \{(n,(v_1(n),v_2(n),v_3(n),v_4(n),v_5(n),f_1,f_2))\}$, $n \in N$

$i^1 = c \dot{\downarrow} g^1 = \{(n,(v_1(n)))\}$, $n \in N$

$i^2 = c \dot{\downarrow} g^2 = \{(n,(v_4(n)))\}$, $n \in N$

Observações:

- Para um objeto c da classe C, tendo a classe, m funções, então existem m interfaces de saída específicas à função.
- Cada i^j é um sub-objeto de i e de c.

DEFINIÇÃO 2.29 – INTERFACE DE SAÍDA

Sejam:

- $d(C)=(a_1,a_2,\ldots,a_n)$ o descritor de uma classe C
- N_f o conjunto dos índices i, tais que, os a_i são descritores de função, da forma $a_i=\left(\left(b^i_{11},\ldots,b^i_{1K_i}\right),\left(b^i_{21},\ldots,b^i_{2L_i}\right)\right)$
- Os conjuntos P e Q formados por:

$$P=\bigcup_{i\in N_f}\bigcup_{j=1}^{k_i} b^i_{1j} \qquad \text{e} \qquad Q=\bigcup_{i\in N_f}\bigcup_{j=1}^{L_i} b^i_{2j}$$

- O conjunto $R=Q-\{Q\cap P\}$
- $p=[p_1,p_2,\ldots,p_r]$ uma fórmula de indução, tal que, $\forall i\in\{1,\ldots,r\}$, $r\in \mathrm{N}$, $p_i\in R$ e $\forall x\in R$, $\exists\, p_i=x$
- c um objeto ativo de uma classe C e O uma superclasse de C, tal que, $C\downarrow p=O$

Define-se a **interface de saída** o do objeto c, como o objeto gerado pela projeção objética de c segundo g, ou seja, $o=c\dot{\downarrow}p$.

Exemplo: $d(C)=(V_1,V_2,V_3,V_4,V_5,((1,2),(3)),((2,4),(5,2)))$

$C=(v_1,v_2,v_3,v_4,v_5,f_1,f_2)$

$N_f=\{6,7\}$

$P=\{1,2,4\}$, $Q=\{3,5,2\}$, $R=\{3,5\}$

$p=[3,5]$

$C\downarrow p=(v_3,v_5)$

$c=\{(n,(v_1(n),v_2(n),v_3(n),v_4(n),v_5(n),f_1,f_2))\}$, $n\in N$

$o=c\dot{\downarrow}p=\{(n,(v_3(n),v_5(n)))\}$, $n\in N$

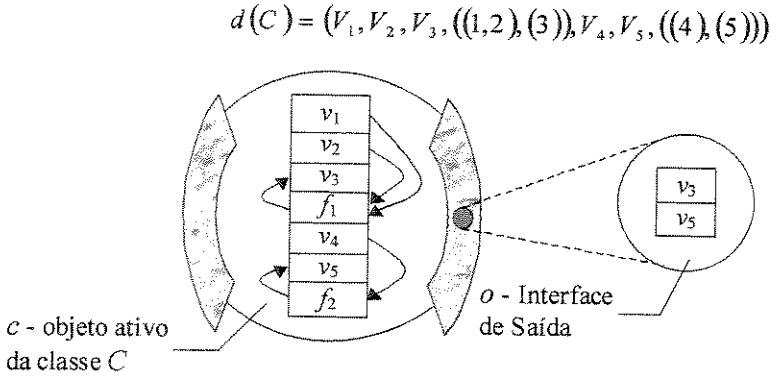


Figura 2.13 Exemplo de Interface de Saída

DEFINIÇÃO 2.30 – INTERFACES DE SAÍDA ESPECÍFICAS A FUNÇÃO

Sejam:

- $d(C)=(a_1,a_2,\ldots,a_n)$ o descritor de uma classe C
- c um objeto ativo de uma classe C
- I^j uma superclasse de I e C
- N_f o conjunto dos índices i, tais que, os a_i são descritores de função, da forma $a_i=\left((b^i_{11},\ldots,b^i_{1K_i}),(b^i_{21},\ldots,b^i_{2L_i})\right)$, sendo $a_j=\left((b^j_{11},\ldots,b^j_{1K_j}),(b^j_{21},\ldots,b^j_{2L_j})\right)$ o descritor de função específico para a função j
- Os conjuntos P e Q_j formados por:

$$P=\bigcup_{i\in N_f}\bigcup_{j=1}^{k_i} b^i_{1j} \qquad \text{e} \qquad Q_j=\bigcup_{n=1}^{L_j} b^j_{2n}$$

- O conjunto $R_j=Q_j-\{Q_j\cap P\}$
- $p^j=\left[p^j_1,p^j_2,\ldots,p^j_r\right]$ uma fórmula de indução, tal que, $\forall i\in\{1,\ldots,r\}$, $r\in\mathrm{N}$, $p^j_i\in R$ e $\forall x\in R$, $\exists\, p^j_i=x$

Define-se a **interface de saída específica à função** j do objeto c, o^j como o objeto passivo obtido pela projeção objética de c segundo p^j, ou seja, $o^j=c\downarrow p^j$.

Exemplo: $d(C)=(V_1,V_2,V_3,V_4,V_5,((1,2),(3)),((2,4),(5,2)))$

$C=(v_1,v_2,v_3,v_4,v_5,f_1,f_2)$

$N_f=\{6,7\}$

$a^1=((1,2),(3))$ o descritor da função f_1

$a^2=((2,4),(5,2))$ o descritor da função f_2

$P=\{1,2,4\}$, $Q_1=\{3\}$, $R_1=\{3\}$

$P=\{1,2,4\}$, $Q_2=\{5,2\}$, $R_2=\{5\}$

$p^1=[3]$

$p^2=[5]$

$C\downarrow p^1=(v_3)$

$C\downarrow p^2=(v_5)$

$c=\{(n,(v_1(n),v_2(n),v_3(n),v_4(n),v_5(n),f_1,f_2))\}$, $n\in N$

$o^1=c\,\dot{\downarrow}\,p^1=\{(n,(v_3(n)))\}$, $n\in N$

$o^2=c\,\dot{\downarrow}\,p^2=\{(n,(v_5(n)))\}$, $n\in N$

Observações:

- Para um objeto c da classe C, tendo a classe, m funções, então existem m interfaces de entrada específicas à função.

- Cada o^j é um sub-objeto de o e de c.

DEFINIÇÃO 2.31 – EXISTÊNCIA DE UM OBJETO

Um objeto c é dito existir em um instante n, se a função que mapeia as instâncias de c em C é definida para $n \in N$.

DEFINIÇÃO 2.32 – GERAÇÃO E CONSUMO DE OBJETOS

Um objeto é dito **gerado** em um instante n, se ele não existe em n e existe em $n+1$. Um objeto é **consumido** em n, se ele existe em n e não existe em $n+1$.

DEFINIÇÃO 2.33 – ESCOPO HABILITANTE DE UMA FUNÇÃO

Sejam:

- $d(C)=(a_1,a_2,\ldots,a_m)$ o descritor de uma classe C
- um objeto ativo c da classe C
- f_j a j-ésima função de transformação interna da classe C
- i^j a interface de entrada específica à função f_j e $g^j=\lfloor g_1^j,\ldots,g_o^j \rfloor$, $o \in \mathrm{N}$, a fórmula de indução que gera i^j a partir de $c \downarrow g^j$
- um conjunto $B=\{0,1\}$

Um **escopo habilitante** $H(c,n,j)$ para a função f_j do objeto c no instante n, $n \in \mathrm{N}$, será qualquer conjunto de ênuplas $\{(h_t,b_t)\}$, $t=1,\ldots,o$, $o \in \mathrm{N}$, onde tem-se satisfeito que h_t é um objeto do tipo $a_{g_t^j}$ e $b_t \in B$ é um valor indicando se o objeto h_t será ($b_t=1$) ou não ($b_t=0$) consumido no disparo de c.

Observe-se que podem existir mais de um escopo habilitante para uma mesma combinação c, n, j.

Para o conjunto formado por todos os possíveis escopos habilitantes de c, n, j, utiliza-se a notação $\overline{H}(c,n,j)$. Observe-se que $\overline{H} \subseteq 2^{C \times B}$.

DEFINIÇÃO 2.34 – ESCOPO GERATIVO DE UMA FUNÇÃO

Sejam:

- $d(C)=(a_1,a_2,\ldots,a_m)$ o descritor de uma classe C
- um objeto ativo c da classe C
- f_j a j-ésima função de transformação interna da classe C
- o^j a interface de saída específica à função f_j. e $p^j=\lfloor p_1^j,\ldots,p_r^j \rfloor$, $r \in \mathrm{N}$, a fórmula de indução que gera o^j a partir de $c \downarrow p^j$

Um **escopo gerativo** $S(c,n,j)$ para a função f_j do objeto c no instante n, $n \in \mathrm{N}$, será qualquer conjunto de objetos $\{S_u\}$, $u=1,\ldots,r$, $r \in \mathrm{N}$,onde se satisfaça que S_u é um objeto do tipo $a_{p_u^j}$.

Observe-se que podem existir mais de um escopo gerativo para uma mesma combinação c, n, j.

Para o conjunto formado por todos os possíveis escopos gerativos de c, n, j, utiliza-se a notação $\overline{S}(c,n,j)$. Observe-se que $\overline{S} \subseteq 2^C$.

DEFINIÇÃO 2.35 – HABILITAÇÃO DE UM OBJETO ATIVO

Um objeto ativo c de uma classe C é dito **habilitado** em n, se todos os objetos pertencentes a um escopo habilitante de uma de suas funções f existem em n. A função f é dita estar habilitada em n.

DEFINIÇÃO 2.36 – DISPARO DE UM OBJETO ATIVO

Sejam:

- $d(C)=(a_1,a_2,\ldots,a_m)$ o descritor de uma classe C
- um objeto c de uma classe C
- a instância de c em n, $c(n)=(c_1(n),\ldots,c_m(n))$
- f_j uma função de transformação de c em n, habilitada por um escopo habilitante $H(c,n,j)=\{(h_t,b_t)\}$
- um escopo gerativo $S(c,n,j)=\{s_u\}$ para f_j, tal que, $(s_u,1)$ não pertença ao escopo habilitante de nenhum outro objeto para o instante n
- i^j a interface de entrada específica à função f_j e $g^j=\lfloor g_1^j,\ldots,g_o^j \rfloor$ a fórmula de indução que gera i^j a partir de $c \downarrow g^j$
- o^j a interface de saída específica à função f_j e $p^j=\lfloor p_1^j,\ldots,p_r^j \rfloor$ a fórmula de indução que gera o^j a partir de $c \downarrow p^j$

O disparo do objeto ativo c no instante n corresponde a:

- determinação da instância de c no instante $n+1$, em função das instâncias de c e h_t no instante n:

$$c_i(n+1)=\begin{cases} c_i(n), \text{ se } i \notin \{g_1^j,\ldots,g_o^j\} \text{ e } i \notin \{p_1^j,\ldots,p_r^j\} \\ h_{g_i^j}(n), \text{ se } i \in \{g_1^j,\ldots,g_o^j\} \text{ e } i \notin \{p_1^j,\ldots,p_r^j\} \\ f(w_1,\ldots,w_b), \text{ se e } i \in \{p_1^j,\ldots,p_r^j\} \end{cases}$$

$$\text{onde } w_q = \begin{cases} h_{g_q^j}(n), \text{ se } i \in \{g_1^j, \ldots, g_o^j\} \\ c_q(n), \text{ se } i \notin \{g_1^j, \ldots, g_o^j\} \end{cases}$$

- o consumo, no instante n, dos objetos h_t, $(h_t, b_t) \in H$, tais que $b_t = 1$.
- a agregação, no instante n dos objetos contidos em S e inexistentes em n.
- a determinação do valor das instâncias dos objetos em S, para o instante $n+1$, em função da instância de c no instante $n+1$:

$$s_u(n+1) = c_u(n+1), \forall u \in \{p_1^j, \ldots, p_r^j\}$$

Observe-se que no disparo de um objeto ativo existem duas relações que devem ser verificadas:

- do valor dos objetos no instante $n-1$, com a escolha do escopo habilitante e
- do valor dos objetos no instante $n+1$, com a determinação do escopo gerativo

2.3. Sistema de Objetos

Um sistema de objetos é um conjunto de objetos (ou objetos genéricos) que estão associados uns aos outros de tal forma que cada instância destes objetos em um instante determinado, está em função das instâncias de todos os objetos no instante anterior, ou seja, que os valores das instâncias dos objetos (e mesmo sua existência em diferentes instantes) devem estar associados entre eles de acordo com as leis de disparo e geração (ou regeneração) de objetos. A determinação dos objetos que interagirão a cada instante, é feita por uma função (que chamaremos de função de seleção), que definirá os escopos habilitantes e escopos gerativos para cada disparo.

2.3.1. O Sistema de Objetos Formal

DEFINIÇÃO 2.37 – SISTEMA DE OBJETOS

Sejam:

- $\Sigma = \{C_i\}$ um conjunto de classes
- c_i objetos de classe C_i, $i = 1, \ldots, \delta$, $\delta > 0$
- $C = \bigcup_i c_i$
- $\Theta_i = \{0, \ldots, m_i\}$, onde m_i é o número de funções de transformação do objeto c_i
- $B = \{0,1\}$
- $\gamma(c_i, n)$, $0 \le i \le \delta$, $\delta > 0$, uma **função de seleção** $\gamma : C \times N \to \overline{H} \times \overline{S} \times \Theta_i$, a qual para cada objeto c_i, no instante n, seleciona um escopo habilitante H_i, um escopo gerativo S_i e o índice da função de transformação (ou função interna) a ser executada pelo objeto. Esta seleção tem que cumprir as seguintes restrições:
  - $\forall (c,b) \in H_i$, se $b = 1$, $(\forall k \neq i)((c,1) \notin H_k)$

- $\forall c \in S_i$, $(\forall k \neq i)(c \notin S_k)$ e $(\forall k)((c,1) \notin H_k)$

Se c_i é um objeto passivo ou, para um determinado n, $\nexists H_i \neq \phi$ ou $\nexists S_i \neq \phi$ então $\gamma(c_i,n) = (\phi,\phi,0)$. O terceiro índice 0 indica que nenhuma função de transformação será executada, ou seja, o objeto não será disparado.

Um **sistema de objetos** Ω é uma ênupla (Σ, C, γ), tal que:

- os objetos c_i sejam funções definidas em um mesmo N
- para $n = 0$, exista pelo menos um $\gamma(c_i,n) = (H_i,S_i,j)$ com objeto c_i definido
- para $n > 0$, todos os objetos ativos c_i com $\gamma(c_i,n) \neq (\phi,\phi,0)$, ou seja com $\gamma(c_i,n) = (H_i,S_i,j)$ sejam disparados, conforme o escopo habilitante H_i, e o escopo gerativo S_i, utilizando sua função de transformação j
- para $n > 0$, todos os objetos c_i existentes em n com suas instâncias $(n+1)$ não afetados pelo item anterior sejam regenerados: $c_i(n+1) = c_i(n)$

Na figura 2.14 mostra-se um exemplo da interdependência entre os objetos, que caracteriza a um sistema de objetos. Nesta figura, cada objeto é representado como uma seqüência de retângulos, onde cada retângulo corresponde a uma instância temporal do objeto. Para os instantes onde o objeto não é definido, não é desenhado nenhum retângulo. Os valores das instâncias nos instantes de tempo sucessivos são determinadas a partir dos valores das instâncias nos instantes anteriores. Os disparos são representados na figura com setas. Para o caso apresentado nesta figura, temos que do instante 1 para o instante 2, não houve disparo, somente regeneração da instância do objeto 1. Do instante 2 para o instante 3, houve um disparo de um objeto (não identificado na figura), utilizando o objeto 1 como escopo habilitante e como escopo gerativo ao objeto 2. Como resultado deste disparo ocorrido, temos a geração do objeto 2, definindo-se o valor de sua instância para o instante 3 e a regeneração do objeto 1 para o instante 3, pois ele não foi consumido (assimilado destrutivamente) neste processo. Do instante 3 para o instante 4, não houve disparo, mas tão somente a regeneração das instâncias dos objetos 1 e 2. Do instante 4 para o instante 5, houve dois disparos de dois objetos (não representados na figura). Um deles utiliza como escopo habilitante o objeto 2, por meio de uma assimilação não destrutiva, e como escopo gerativo o objeto 4. O outro utiliza como escopo habilitante os objetos 1 (em assimilação destrutiva) e 2 (em assimilação não destrutiva), e como escopo gerativo o objeto 2. Como resultado destes disparos, temos o consumo da instância do objeto 1, a regeneração do objeto 2 e a geração do objeto 4. Do instante 5 para o instante 6, não houve disparo, ocorrendo a regeneração das instâncias dos objetos 2 e 4. Do instante 6 para o instante 7, houve um disparo de um objeto (não representado na figura) que tem como escopo habilitante os objetos 2 e 4 (consumindo estes) e como escopo gerativo o objeto 3, criando-se uma instância do objeto 3 para o instante 7. Do instante 7 em diante, até o instante 10 (instante final), apenas ocorre a regeneração da instância do objeto 3.

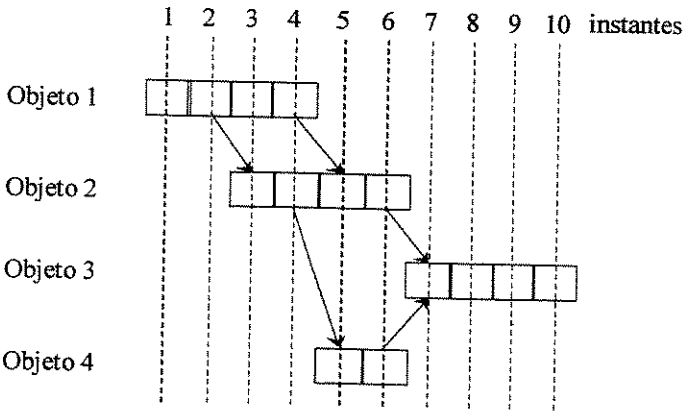


Figura 2.14 Exemplo de um Sistema de Objetos

Uma propriedade desejável para um sistema de objetos é sua **computabilidade.** Uma vez que um sistema de objeto é computável, pode-se determinar para qualquer instante $n \in N$, o valor das instâncias dos objetos existentes em n, além da inexistência, em n, dos demais objetos do sistema. Entretanto, a natureza recursiva de um sistema de objetos não garante por si só que este seja computável. Para garantir isso, são necessárias algumas condições adicionadas, dadas pelo teorema 3.1 definido por Gudwin em [Gudwin 96].

2.4. Redes de Objetos

Uma rede de objetos é um tipo especial de sistema de objetos, na qual são incluídas restrições adicionais concernentes às interações entre objetos. Para distinguir as redes de objetos dos sistemas de objetos, assumamos a existência de lugares e arcos cujo papel é similar aos usados no contexto das redes de Petri.

Algumas restrições são colocadas às redes de objetos que fazem delas um tipo especial de sistemas de objetos. Uma delas é que os objetos estarão associados a lugares. Assim, diz-se que cada objeto, em um instante de tempo, encontra-se em um determinado lugar. Cada lugar só pode ser ocupado por objetos do mesmo tipo. Assim, para cada lugar, está associada uma classe e dois conjuntos de arcos, os arcos de entrada e de saída que conectam os diferentes lugares. Os objetos em um lugar só podem interagir com os objetos que estão nos lugares conectados a ele através dos arcos. Para cada lugar, existe um conjunto (que pode ser vazio) de lugares conectados a ele através dos arcos de entrada. Estes lugares são chamados de portas de entrada do lugar. Analogamente, para cada lugar existe um conjunto (que pode ser vazio) de lugares conectados a ele através dos arcos de saída, chamados de portas de saída do lugar. Para cada campo da interface de entrada/saída haverá uma porta (lugar) de entrada/saída associada.

Lembremos que existem objetos passivos (que só contém informações) e objetos ativos (que podem ou não conter informações - e além disso possuem uma ou mais funções de

transformação). Neste sentido, podemos dizer que existem lugares ativos ou passivos, se os objetos que estão nesses lugares são ativos ou passivos, respectivamente.

As redes de objetos podem ser representadas de forma gráfica. Neste caso, os lugares são representados por circunferências e os arcos por setas. Os lugares passivos são indicados por círculos, os lugares ativos por círculos duplos e as instâncias dos objetos por *tokens* (bolinhas pretas), como se mostra na figura 2.15.

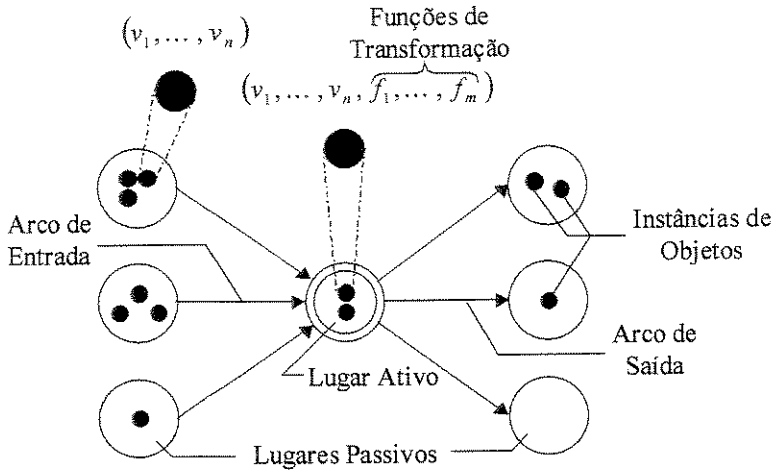


Figura 2.15 Exemplo de uma Rede de Objetos

Note que , diferentemente das redes de Petri, os tokens são instâncias de objetos que possuem individualidade ou personalidade. Em redes de objetos, tokens não são somente marcas nos lugares, mas dizem respeito a objetos com atributos e eventualmente funções de transformação. Além disso, tokens representando objetos ativos executam o papel das transições. Estas, podem ser móveis e mutáveis. Isso dá um grande poder de representação para uma rede de objetos, já que permite modelar sistemas que não são adequadamente modelados por redes de Petri (mesmo as mais sofisticadas), como por exemplo, sistemas adaptativos.

Como nos sistemas de objetos, o comportamento dinâmico de uma rede de objeto é caracterizado pelo disparo de objetos ativos. O disparo de um objeto ativo corresponde à geração de novas instâncias de objetos nos lugares diretamente conectados ao lugar onde o objeto ativo encontra-se, através dos arcos de saída. Para que um disparo de um objeto ativo seja bem sucedido, esse objeto primeiramente deve ter um escopo habilitante, ou seja, um conjunto de instâncias de objetos disponíveis em suas portas de entrada, que habilitem uma de suas funções de transformação. Para a seleção de um escopo habilitante, é utilizada a função de seleção, que escolhe das instâncias de objetos disponíveis, aqueles que serão utilizados no disparo. Depois do disparo, as instâncias de objetos podem ser colocadas em uma ou mais portas de saída do objeto. Isso é feito, também, pela função de seleção. As instâncias dos objetos utilizados como escopo habilitante podem ou não ser destruídos para o próximo instante de tempo.

2.4.1. A Rede de Objetos Formal

DEFINIÇÃO 2.38 – REDE DE OBJETOS

Sejam:

- $\Sigma = \{C_i\}$ um conjunto de classes
- $C = \{c_i\}$ um conjunto de objetos, onde c_i são objetos de classe C_i, $C_i \in \Sigma$, $0 \le i \le \delta$, $\delta > 0$.
- $\Pi = \{\pi_i\}$ um conjunto de lugares π_i
- $A = \{a_i\}$ um conjunto de arcos a_i
- η uma função de nó $\eta : A \to \Pi \times \Pi$
- ξ uma **função de localização** $\xi : N \times C \to \Pi$, que associa para cada objeto $c \in C$, em um instante n, um lugar Π
- $F(\pi)$ um mapeamento $\Pi \to 2^{\Pi}$, definido por $F(\pi) = \cup \pi_k$ onde $k \in K$, $K = \{k \mid \exists a_j \in A \text{ tal que } \eta(a_j) = (\pi_k, \pi)\}$
- $V(\pi)$ um mapeamento $\Pi \to 2^{\Pi}$, definido por $V(\pi) = \cup \pi_k$ onde $k \in K$, $K = \{k \mid \exists a_j \in A \text{ tal que } \eta(a_j) = (\pi, \pi_k)\}$
- $\Xi = \Xi(\pi)$ um mapeamento de classes $\Pi \to \Sigma$, de tal forma que $\forall \pi \in \Pi$ para cada **campo** ou componente v_i da interface de entrada de objetos da classe $\Xi(\pi)$, sendo v_i um objeto de classe C, $\exists \pi_k$, $\pi_k \in F(\pi)$, tal que $\Xi(\pi_k) = C$, e para campo v_i da interface de saída de objetos da classe $\Xi(\pi)$, sendo v_i um objeto de classe C, $\exists \pi_k$, $\pi_k \in V(\pi)$, tal que $\Xi(\pi_k) = C$
- i^i a interface de entrada de um objeto de classe $\Xi(\pi_i)$ e ∂_i o número de campos de i^i, ou seja, $\partial_i = Ar(i^i)$
- o^i a interface de saída de um objeto de classe $\Xi(\pi_i)$ e ρ_i o número de campos de o^i, ou seja, $\rho_i = Ar(o^i)$
- $fpi = \{fpi_i\}$ um conjunto de funções de atribuição de portas de entrada, onde cada $fpi_i : \{1, \ldots, \partial_i\} \to A$ é uma **função de atribuição de portas de entrada** para o objeto que encontra-se no lugar π_i
- $fpo = \{fpo_i\}$ um conjunto de funções de atribuição de portas de saída, onde cada $fpo_i : \{1, \ldots, \rho_i\} \to A$ é uma **função de atribuição de portas de saída** para o objeto que encontra-se no lugar π_i
- $\gamma = \{\gamma(c_i, n)\}$ o conjunto das funções de seleção, onde cada $\gamma(c_i, n)$ é uma função de seleção para o objeto c_i. Estas funções de seleção tem as seguintes restrições:
  - **R1**: $\forall (c,b) \in H_i$ (escopo habilitante para o objeto c_i), $\xi(n,c) = \pi$, $\pi \in F(\xi(n, c_i))$. Se $b = 1$, $(\forall k \neq i)((c,1) \notin H_k)$

- **R2**: $\forall c \in S_i$ (escopo gerativo para o objeto c_i), $\xi(n+1,c)=\pi$, $\pi \in V(\xi(n,c_i))$. $(\forall k \neq i)(c \notin S_k)$ e $(\forall k)((c,1) \notin H_k)$

Define-se uma **rede de objetos** $\Re$ como uma ênupla da seguinte forma:

$$\Re = (\Sigma, C, \gamma, \Pi, \Xi, A, \eta, fpi, fpo, \xi),$$

onde as seguintes restrições são cumpridas:

- um sistema de objetos $\Omega = (\Sigma, C, \gamma)$ deve ser caracterizado
- para cada objeto $c_i \in C$ que tenha uma função de transformação f disparada no instante n, estando esse objeto no instante n em um lugar $\pi = \xi(n,c_i)$, os objetos s_i^k do escopo gerativo S_i indicado por $\gamma(c_i,n)$ devem ter uma função de localização definida por $\xi(n+1,s_i^k)=\pi^k$, onde π^k deve ser tal que $\eta(fpo_\pi(k'))=(\pi,\pi^k)$ e k' é o índice do $k-ésimo$ campo da interface de saída específica à função f de c_i referenciado na interface de saída de c_i.
- para cada objeto $c_i \in C$ que tenha uma função de transformação f disparada no instante n, estando esse objeto no instante n em um lugar $\pi = \xi(n,c_i)$, os objetos h_i^k do escopo habilitante H_i indicado por $\gamma(c_i,n)$ devem ter uma função de localização definida por $\xi(n,h_i^k)=\pi^k$, onde π^k deve ser tal que $\eta(fpi_\pi(k'))=(\pi^k,\pi)$ e k' é o índice do $k-ésimo$ campo da interface de entrada específica à função f de c_i referenciado na interface de entrada de c_i.

Uma classe importante de redes de objetos são as redes de objetos computáveis. Este tipo de rede pode ser determinada iterativamente, a partir de uma seqüência de redes de objetos $\Re_0$, $\Re_1$, ... , onde cada uma das redes $\Re_i$ contém um número finito de objetos. A rede de objetos inicial da seqüência, $\Re_0$, tem uma denominação especial, chamada de **núcleo da rede** [Gudwin 96].

2.5. Exemplos

2.5.1. Rede de Petri Colorida

Como um primeiro exemplo, mostraremos como uma rede de objetos é capaz de representar uma rede de Petri Colorida (mostrada na figura 2.16).

Neste exemplo, para que a transição T seja habilitada devem existir dois *tokens* no lugar S e um *token* no lugar B. A transição T coloca um *token* de tipo P do lugar B para o lugar C.

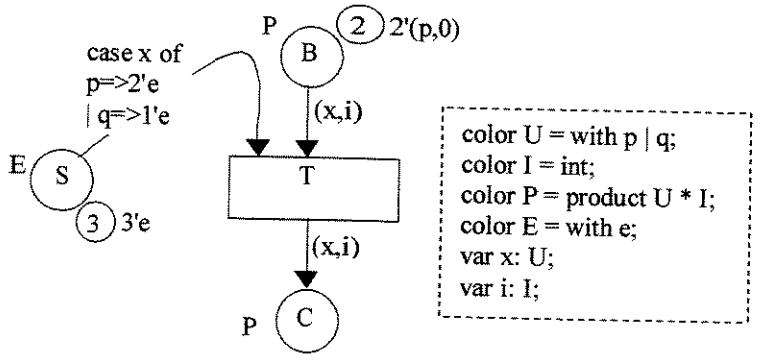


Figura 2.16 Rede de Petri Colorida de exemplo.

A rede de objetos que modela e representa essa rede de Petri colorida está na figura 2.17.

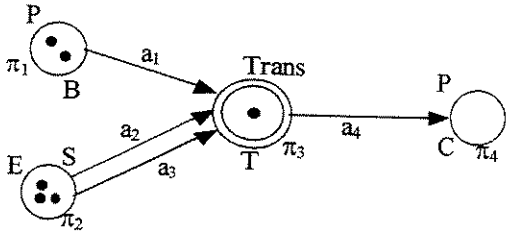


Figura 2.17 Rede de objetos correspondente à rede de Petri Colorida.

Sejam a classe $E = \{(v_1)\}$, onde $v_1 \in \{e, \_\}$, a classe $P = \{(v_2, v_3)\}$, onde $v_2 \in \{p, q, \_\}$ e $v_3 \in N$. A classe ativa T está definida por: $T = \{(v_4, v_5, v_6, v_7, f_1)\}$, onde $v_4 \in E$, $v_5 \in E$, $v_6 \in P$ e $v_7 \in P$. As variáveis v_4, v_5 e v_6 correspondem à interface de entrada e v_7 corresponde à interface de saída. A função de transformação $f_1 : E \times E \times P \rightarrow P$ é uma função condicional, que assimila destrutivamente dois objetos da classe E (do lugar S), caso o valor do atributo de um objeto da classe P no lugar B seja $v_2 = p$ ou apenas um objeto da classe E, caso o valor do atributo seja $v_2 = q$, transferindo o objeto da classe P do lugar B para o lugar C.

2.5.1.1. Núcleo da rede de objeto

$\Re_0=(\Sigma,\Pi,\Xi,A,\eta,fpi,fpo,\mathcal{C}^0,\xi^0,\gamma^0)$
$\Sigma=\{P,E,T\}$
$\Pi=\{\pi_1,\pi_2,\pi_3,\pi_4\}$
$\Xi=\{(\pi_1,P),(\pi_2,E),(\pi_3,T),(\pi_4,P)\}$
$A=\{a_1,a_2,a_3,a_4\}$
$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_2,\pi_3)),(a_3,(\pi_2,\pi_3)),(a_4,(\pi_3,\pi_4))\}$
$fpo_{\pi 3}(1)= a_4$, $fpi_{\pi 3}(1)= a_1$, $fpi_{\pi 3}(2)= a_2$, $fpi_{\pi 3}(3)= a_3$

$\mathcal{C}^0$= {c_1, c_2, c_3, c_4, c_5, c_6}
Nu_P= (_,0) é um valor *default*, $Nu_P \in P$
Nu_E= (_) é um valor *default*, $Nu_E \in E$
c_1={(0,(p,0))}
c_2={(0,(p,0))}
c_3={(0,(e))}
c_4={(0,(e))}
c_5={(0,(e))}
c_6={(0,((e),(e),(p,0),Nu_P, f_1))}
ξ^0={(0,c_1,π_1), (0,c_2,π_1), (0,c_3,π_2), (0,c_4,π_2), (0,c_5,π_2), (0,c_6,π_3)}
$\gamma^0{}_1(0)$=({ (c_1,1), (c_3,1), (c_4,1)},{c_7},1)

2.5.1.2. Seqüências da rede de objeto

$\Re_0$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^0$,ξ^0,γ^0)
Σ={*P*,*E*,*T*}
Π={π_1,π_2,π_3,π_4}
Ξ={(π_1,*P*),(π_2,*E*),(π_3,*T*),(π_4,*P*)}
A={a_1,a_2,a_3,a_4}
η={(a_1,(π_1,π_3)),(a_2,(π_2,π_3)),(a_3,(π_2,π_3)),(a_4,(π_3,π_4))}
$fpo_{\pi 3}(1)$= a_4, $fpi_{\pi 3}(1)$= a_1, $fpi_{\pi 3}(2)$= a_2, $fpi_{\pi 3}(3)$= a_3

$\mathcal{C}^0$= {c_1, c_2, c_3, c_4, c_5, c_6}
Nu_P= (_,0) é um valor *default*, $Nu_P \in P$
Nu_E= (_) é um valor *default*, $Nu_E \in E$
c_1={(0,(p,0))}
c_2={(0,(p,0))}
c_3={(0,(e))}
c_4={(0,(e))}
c_5={(0,(e))}
c_6={(0,((e),(e),(p,0),Nu_P, f_1))}
ξ^0={(0,c_1,π_1), (0,c_2,π_1), (0,c_3,π_2), (0,c_4,π_2), (0,c_5,π_2), (0,c_6,π_3)}
$\gamma^0{}_1(0)$=({ (c_1,1), (c_3,1), (c_4,1)},{c_7},1)

A figura 2.17 mostra a rede de objetos neste instante.

$\Re_1$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^1$,ξ^1,γ^1)
Σ={*P*,*E*,*T*}
Π={π_1,π_2,π_3,π_4}

$\Xi=\{(\pi_1,P),(\pi_2,E),(\pi_3,T),(\pi_4,P)\}$
$A=\{a_1,a_2,a_3,a_4\}$
$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_2,\pi_3)),(a_3,(\pi_2,\pi_3)),(a_4,(\pi_3,\pi_4))\}$
$fpo_{\pi 3}(1)= a_4$, $fpi_{\pi 3}(1)= a_1$, $fpi_{\pi 3}(2)= a_2$, $fpi_{\pi 3}(3)= a_3$

$\mathcal{C}^1= \{c_1,c_2,c_3,c_4,c_5,c_7\}$
$Nu_P= (\_,0)$ é um valor *default*, $Nu_P \in P$
$Nu_E= (\_)$ é um valor *default*, $Nu_E \in E$
$c_1=\{(0,(p,0))\}$
$c_2=\{(0,(p,0)), (1,(p,0))\}$
$c_3=\{(0,(e))\}$
$c_4=\{(0,(e))\}$
$c_5=\{(0,(e)), (1,(e))\}$
$c_6=\{(0,((e),(e),(p,0),Nu_P, f_1)), (1,(Nu_E, Nu_E, Nu_P, (p,0), f_1))\}$
$c_2=\{(1,(p,0))\}$
$\xi^1= \xi^0 \cup \{(1,c_2,\pi_1), (1,c_5,\pi_2), (1,c_6,\pi_3), (1,c_7,\pi_4)\}$
$\gamma^1{}_1(1)=(\phi,\phi,0)$

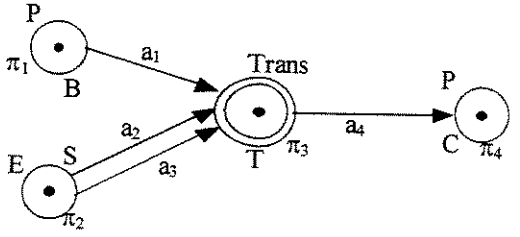


Figura 2.18 Rede de objetos correspondente à rede de Petri Colorida no instante 1.

Neste exemplo ilustrativo a rede termina neste passo, já que não se pode formar outro escopo habilitante para a função de transformação do objeto ativo. Este não pode mais ser disparado, portanto.

2.5.2. Processando Conhecimento

Imagine-se o seguinte sistema de processamento de conhecimento (neste caso, um conjunto de conhecimentos remáticos [Gudwin 96]). Sejam:

- N o conjunto dos números inteiros e R o conjunto dos números reais.
- Uma classe C_1 representando instâncias de conhecimentos remáticos, dada por $C_1 = \{(v_1, v_2)\}$, seguindo o descritor de classe $d(C_1) = (V_1, V_2)$. Considere $v_1 \in N$, correspondendo a um *flag* que indica o tipo de conhecimento remático, como se mostra a seguir:
  0 - conhecimento sensorial
  1 - conhecimento designativo

2 - conhecimento prescritivo

e $v_2 \in R^n$ correspondendo a um conjunto de atributos que caracterizam o conhecimento remático (neste caso, está aqui somente para constar - seu valor não é importante).

- Uma classe C_2 representando instâncias de objetos do tipo sensor, dada por $C_2 = \{(v_3, v_4, f_1)\}$, seguindo o descritor de classe $d(C_2) = (V_3, V_4, ((3),(3,4)))$, onde $v_3 \in N$ corresponde a um *timer* indicando o instante de tempo, $v_4 \in C_1$, corresponde à interface de saída e f_1 é uma função de transformação $f_1 : N \rightarrow N \times C_1$ que a cada instante de tempo atualiza o *timer* interno e gera um objeto do tipo C_1 correspondente a um conhecimento sensorial (*flag* = 0).
- Uma classe C_3 representando instâncias de objetos do tipo atuador, dada por $C_3 = \{(v_5, v_6, f_2)\}$, seguindo o descritor de classe $d(C_3) = (V_5, V_6, ((5,6),(5)))$, onde $v_5 \in N$ corresponde a um *timer* indicando o instante de tempo, $v_6 \in C_1$ corresponde à interface de entrada e f_2 é uma função de transformação $f_2 : N \times C_1 \rightarrow N$ que a cada instante de tempo atualiza o *timer* interno e assimila destrutivamente um objeto do tipo C_1 correspondente a um conhecimento prescritivo (*flag* = 2).
- Uma classe C_4 representando instâncias de objetos do tipo conhecimento argumentativo, dado por $C_4 = \{(v_7, v_8, f_3, f_4)\}$, seguindo o descritor de classe $d(C_4) = (V_7, V_8, ((7),(8)),((7),(8)))$, onde $v_7 \in C_1$ corresponde à interface de entrada e $v_8 \in C_1$ corresponde à interface de saída , e f_3 é uma função de transformação $f_3 : C_1 \rightarrow C_1$, que assimila destrutivamente objetos de conhecimentos rematicos, do tipo sensorial (*flag* = 0), gerando objetos de conhecimentos rematicos do tipo conhecimento designativo (*flag* = 1), e f_4 é uma função de transformação $f_4 : C_1 \rightarrow C_1$, que assimila destrutivamente objetos de conhecimentos rematicos, do tipo designativo (*flag* = 1), gerando objetos de conhecimentos rematicos do tipo conhecimento prescritivo (*flag* = 2).

2.5.2.1. Núcleo da rede de objeto

$\Re_0$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^0$,ξ^0,γ^0)
Σ={C_1,C_2,C_3,C_4}
Π={π_1,π_2,π_3,π_4}
Ξ={(π_1,C_2),(π_2,C_3),(π_3,C_1),(π_4,C_4)}
A={a_1,a_2,a_3,a_4}
η={(a_1,(π_1,π_3)),(a_2,(π_3,π_4)),(a_3,(π_4,π_3)),(a_4,(π_3,π_2))}
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$

$\mathcal{C}^0$= {c_1, c_2, c_3, c_4}
Nu=(0,r) é um valor *default*, Nu $\in$ C_1

$c_1=\{(0,(0,Nu,f_1))\}$
$c_2=\{(0,(0,Nu,f_2))\}$
$c_3=\{(0,(Nu,Nu,f_3,f_4))\}$
$c_4=\{(0,(Nu,Nu,f_3, f_4))\}$
$\xi^0=\{(0,c_1,\pi_1), (0,c_2,\pi_2), (0,c_3,\pi_4), (0,c_4,\pi_4)\}$
$\gamma^0{}_1(0)=(\phi,\{c_5\},1)$ $\gamma^0{}_2(0)=(\phi,\phi,0)$
$\gamma^0{}_3(0)=(\phi,\phi,0)$ $\gamma^0{}_4(0)=(\phi,\phi,0)$

A seguir mostra-se o grafo que representa a rede de objeto descrita pelo núcleo da rede descrito acima.

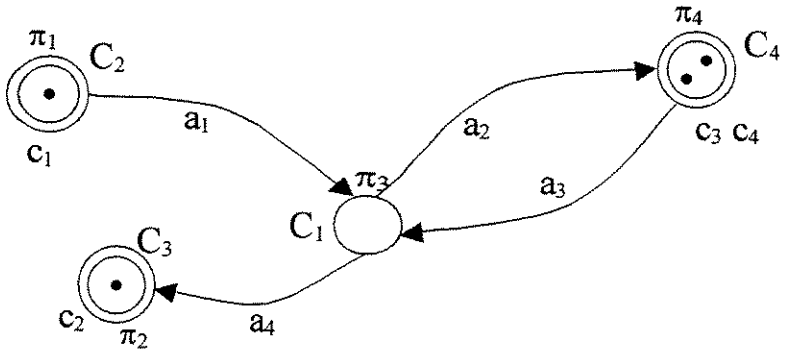


Figura 2.19 Rede de objetos para o processamento de conhecimento.

2.5.2.2. Algumas instâncias da seqüência

$\Re_0=(\Sigma,\Pi,\Xi,A,\eta,fpi,fpo,\mathcal{C}^0,\xi^0,\gamma^0)$
$\Sigma=\{C_1,C_2,C_3,C_4\}$
$\Pi=\{\pi_1,\pi_2,\pi_3,\pi_4\}$
$\Xi=\{(\pi_1,C_2),(\pi_2,C_3),(\pi_3,C_1),(\pi_4,C_4)\}$
$A=\{a_1,a_2,a_3,a_4\}$
$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_3,\pi_4)),(a_3,(\pi_4,\pi_3)),(a_4,(\pi_3,\pi_2))\}$
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$

$\mathcal{C}^0= \{c_1, c_2, c_3, c_4\}$
Nu=(0,r) é um valor *default*, Nu $\in C_1$
$c_1=\{(0,(0,Nu,f_1))\}$
$c_2=\{(0,(0,Nu,f_2))\}$
$c_3=\{(0,(Nu,Nu,f_3,f_4))\}$
$c_4=\{(0,(Nu,Nu,f_3, f_4))\}$
$\xi^0=\{(0,c_1,\pi_1), (0,c_2,\pi_2), (0,c_3,\pi_4), (0,c_4,\pi_4)\}$

$\gamma^0{}_1(0)=(\phi,\{c_5\},1)$ $\gamma^0{}_2(0)=(\phi,\phi,0)$
$\gamma^0{}_3(0)=(\phi,\phi,0)$ $\gamma^0{}_4(0)=(\phi,\phi,0)$

A grafo da rede de objetos no instante 0 mostra-se na figura 2.19

$\Re_1=(\Sigma,\Pi,\Xi,A,\eta,fpi,fpo,\mathcal{C}^1,\xi^1,\gamma^1)$
$\Sigma=\{C_1,C_2,C_3,C_4\}$
$\Pi=\{\pi_1,\pi_2,\pi_3,\pi_4\}$
$\Xi=\{(\pi_1,C_2),(\pi_2,C_3),(\pi_3,C_1),(\pi_4,C_4)\}$
$A=\{a_1,a_2,a_3,a_4\}$
$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_3,\pi_4)),(a_3,(\pi_4,\pi_3)),(a_4,(\pi_3,\pi_2))\}$
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$

$\mathcal{C}^1= \{c_1,c_2,c_3,c_4,c_5\}$
Nu=(0,r) é um valor *default*, Nu $\in C_1$
$c_1=\{(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1))\}$
$c_2=\{(0,(0,Nu,f_2)), (1,(1,Nu,f_2))\}$
$c_3=\{(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4))\}$
$c_4=\{(0,(Nu,Nu,f_3, f_4)), (1,(Nu,Nu,f_3,f_4))\}$
$c_5=\{(1,(0,r_1))\}$
$\xi^1= \xi^0 \cup \{(1,c_1,\pi_1), (1,c_2,\pi_2), (1,c_3,\pi_4), (1,c_4,\pi_4) , (1,c_5,\pi_3)\}$
$\gamma^1{}_1(1)=(\phi,\{c_6\},1)$ $\gamma^1{}_2(1)=(\phi,\phi,0)$
$\gamma^1{}_3(1)=(\{(c_5,1)\},\{c_7\},3)$ $\gamma^1{}_4(1)=(\phi,\phi,0)$

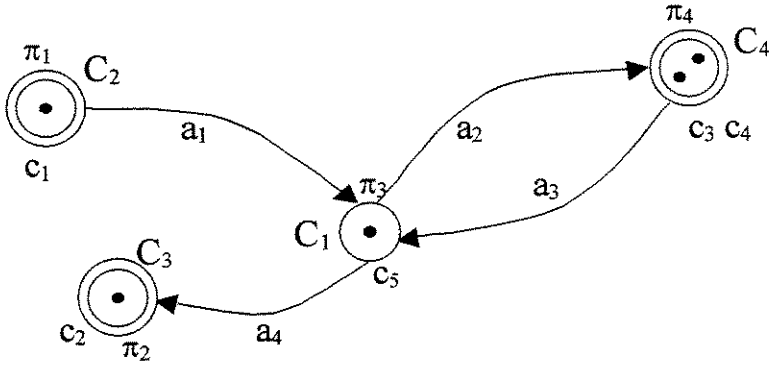


Figura 2.20 Rede de objetos para o processamento de conhecimento, no instante 1.

$\Re_2=(\Sigma,\Pi,\Xi,A,\eta,fpi,fpo,\mathcal{C}^2,\xi^2,\gamma^2)$
$\Sigma=\{C_1,C_2,C_3,C_4\}$
$\Pi=\{\pi_1,\pi_2,\pi_3,\pi_4\}$
$\Xi=\{(\pi_1,C_2),(\pi_2,C_3),(\pi_3,C_1),(\pi_4,C_4)\}$
$A=\{a_1,a_2,a_3,a_4\}$

$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_3,\pi_4)),(a_3,(\pi_4,\pi_3)),(a_4,(\pi_3,\pi_2))\}$
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$

$\mathcal{C}^2= \{c_1,c_2,c_3,c_4,c_5,c_6,c_7\}$
Nu=(0,r) é um valor *default*, $Nu \in C_1$
$c_1=\{(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1)), (2,(2,(0,r_2),f_1))\}$
$c_2=\{(0,(0,Nu,f_2)), (1,(1,Nu,f_2)), (2,(2,Nu,f_2))\}$
$c_3=\{(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,((0,r_1),(1,s_1),f_3,f_4))\}$
$c_4=\{(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,(Nu,Nu,f_3,f_4))\}$
$c_5=\{(1,(0,r_1))\}$
$c_6=\{(2,(0,r_2))\}$
$c_7=\{(2,(1,s_1))\}$
$\xi^2= \xi^1 \cup \{(2,c_1,\pi_1), (2,c_2,\pi_2), (2,c_3,\pi_4), (2,c_4,\pi_4), (2,c_6,\pi_3), (2,c_7,\pi_3)\}$
$\gamma^2{}_1(2)=(\phi,\{c_5\},1)$ $\gamma^2{}_2(2)=(\phi,\phi,0)$
$\gamma^2{}_3(2)=(\{(c_6,1)\},\{c_8\},3)$ $\gamma^2{}_4(2)=(\{(c_7,1)\},\{c_9\},4)$

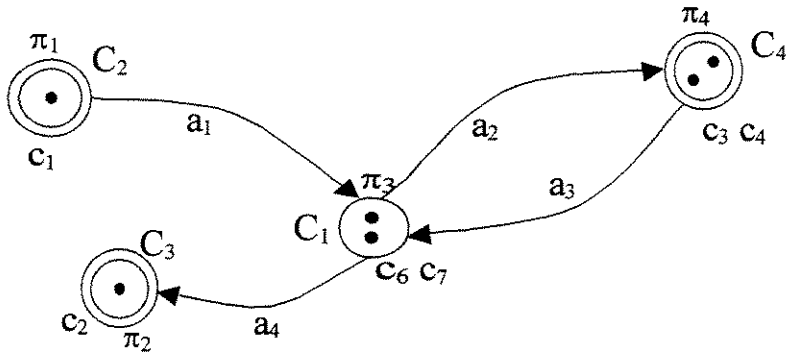


Figura 2.21 Rede de objeto para o processamento de conhecimento, no instante 2

$\Re_3=(\Sigma,\Pi,\Xi,A,\eta,fpi,fpo,\mathcal{C}^3,\xi^3,\gamma^3)$
$\Sigma=\{C_1,C_2,C_3,C_4\}$
$\Pi=\{\pi_1,\pi_2,\pi_3,\pi_4\}$
$\Xi=\{(\pi_1,C_2),(\pi_2,C_3),(\pi_3,C_1),(\pi_4,C_4)\}$
$A=\{a_1,a_2,a_3,a_4\}$
$\eta=\{(a_1,(\pi_1,\pi_3)),(a_2,(\pi_3,\pi_4)),(a_3,(\pi_4,\pi_3)),(a_4,(\pi_3,\pi_2))\}$
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$
$\mathcal{C}^3= \{c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8,c_9\}$
Nu=(0,r) é um valor *default*, $Nu \in C_1$
$c_1=\{(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1)), (2,(2,(0,r_2),f_1)), (3,(3,(0,r_3),f_1))\}$
$c_2=\{(0,(0,Nu,f_2)), (1,(1,Nu,f_2)), (2,(2,Nu,f_2)), (3,(3,(2,q_1),f_2))\}$
$c_3=\{(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,((0,r_1),(1,s_1),f_3,f_4)), (3,((0,r_2),(1,s_2),f_3,f_4))\}$

c_4={(0,(Nu,Nu,f_3, f_4)), (1,(Nu,Nu,f_3,f_4)), (2,(Nu,Nu,f_3,f_4)), (3,((1,s_1),(2,q_1),f_3,f_4))}
c_5={(1,(0,r_1)), (3,(0,r_3))}
c_6={(2,(0,r_2))}
c_7={(2,(1,s_1))}
c_8={(3,(1,s_2))}
c_9={(3,(2,q_1))}
$\xi^3 = \xi^2 \cup$ {(3,c_1,π_1), (3,c_2,π_2), (3,c_3,π_4), (3,c_4,π_4), (3,c_5,π_3), (3,c_8,π_3), (3,c_9,π_3)}
$\gamma^3{}_1(3)$=(ϕ,{c_6},1) $\gamma^3{}_2(3)$=({(c_9,1)},ϕ,2)
$\gamma^3{}_3(3)$=({(c_5,1)},{c_7},3) $\gamma^3{}_4(3)$=({(c_8,1)},{c_{10}},4)

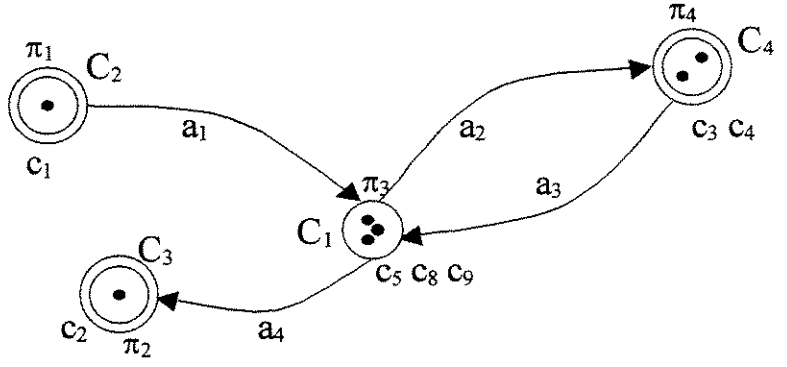


Figura 2.22 Rede de objeto para o processamento de conhecimento, no instante 3

$\Re_4$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^4$,ξ^4,γ^4)
Σ={C_1,C_2,C_3,C_4}
Π={π_1,π_2,π_3,π_4}
Ξ={(π_1,C_2),(π_2,C_3),(π_3,C_1),(π_4,C_4)}
A={a_1,a_2,a_3,a_4}
η={(a_1,(π_1,π_3)),(a_2,(π_3,π_4)),(a_3,(π_4,π_3)),(a_4,(π_3,π_2))}
$fpo_{\pi 1}(1)$= a_1, $fpi_{\pi 2}(1)$= a_4, $fpi_{\pi 4}(1)$= a_2, $fpi_{\pi 4}(2)$= a_2, $fpo_{\pi 4}(1)$= a_3, $fpo_{\pi 4}(2)$= a_3

$\mathcal{C}^4$= {c_1,c_2,c_3,c_4,c_5 ,c_6,c_7,c_8,c_9,c_{10}}
Nu=(0,r) é um valor *default*, Nu $\in$ C_1
c_1={(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1)), (2,(2,(0,r_2),f_1)), (3,(3,(0,r_3),f_1)), (4,(4,(0,r_4),f_1))}
c_2={(0,(0,Nu,f_2)), (1,(1,Nu,f_2)), (2,(2,Nu,f_2)), (3,(3,(2,q_1),f_2)), (4,(4,(2,q_2),f_2))}
c_3={(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,((0,r_1),(1,s_1),f_3,f_4)), (3,((0,r_2),(1,s_2),f_3,f_4)), (4,((0,r_3),(1,s_3),f_3,f_4))}
c_4={(0,(Nu,Nu,f_3, f_4)), (1,(Nu,Nu,f_3,f_4)), (2,(Nu,Nu,f_3,f_4)), (3,((1,s_1),(2,q_1),f_3,f_4)), (4,((1,s_2),(2,q_2),f_3,f_4))}
c_5={(1,(0,r_1)), (3,(0,r_3))}
c_6={(2,(0,r_2)), (4,(0,r_4))}

c_7={(2,(1,s_1)), (4,(1,s_3))}
c_8={(3,(1,s_2))}
c_9={(3,(2,q_1))}
c_{10}={(4,(2,q_2))}
$\xi^4 = \xi^3 \cup$ {(4,c_1,π_1), (4,c_2,π_2), (4,c_3,π_4), (4,c_4,π_4), (4,c_6,π_3), (4,c_7,π_3), (4,c_{10},π_3)}
$\gamma^4{}_1(4)$=(ϕ,{c_5},1) $\gamma^4{}_2(4)$=({(c_{10},1)},ϕ,2)
$\gamma^4{}_3(4)$=({(c_6,1)},{c_8},3) $\gamma^4{}_4(4)$=({(c_7,1)},{c_9},4)

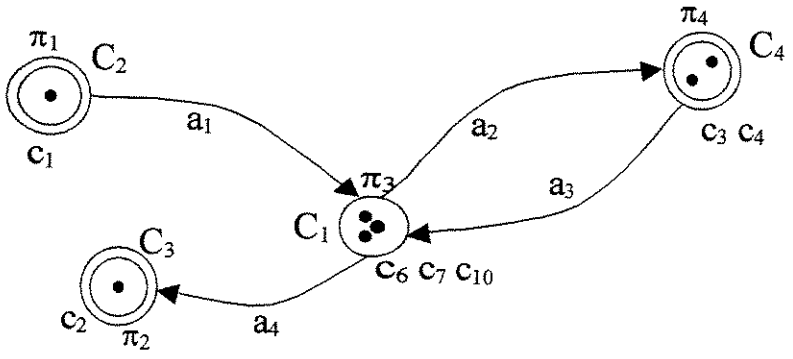


Figura 2.23 Rede de objeto para o processamento de conhecimento, no instante 4

$\Re_5$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^5$,ξ^5,γ^5)
Σ={C_1,C_2,C_3,C_4}
Π={π_1,π_2,π_3,π_4}
Ξ={(π_1,C_2),(π_2,C_3),(π_3,C_1),(π_4,C_4)}
A={a_1,a_2,a_3,a_4}
η={(a_1,(π_1,π_3)),(a_2,(π_3,π_4)),(a_3,(π_4,π_3)),(a_4,(π_3,π_2))}
$fpo_{\pi 1}(1)$= a_1, $fpi_{\pi 2}(1)$= a_4, $fpi_{\pi 4}(1)$= a_2, $fpi_{\pi 4}(2)$= a_2, $fpo_{\pi 4}(1)$= a_3, $fpo_{\pi 4}(2)$= a_3

$\mathcal{C}^5$= {c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8,c_9,c_{10}}
Nu=(0,r) é um valor *default*, Nu $\in$ C_1
c_1={(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1)), (2,(2,(0,r_2),f_1)), (3,(3,(0,r_3),f_1)), (4,(4,(0,r_4),f_1)), (5,(5,(0,r_5),f_1))}
c_2={(0,(0,Nu,f_2)), (1,(1,Nu,f_2)), (2,(2,Nu,f_2)), (3,(3,(2,q_1),f_2)), (4,(4,(2,q_2),f_2)), (5,(5,(2,q_3),f_2))}
c_3={(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,((0,r_1),(1,s_1),f_3,f_4)), (3,((0,r_2),(1,s_2),f_3,f_4)), (3,((0,r_2),(1,s_2),f_3,f_4)), (4,((0,r_3),(1,s_3),f_3,f_4)), (5,((0,r_4),(1,s_4),f_3,f_4))}
c_4={(0,(Nu,Nu,f_3, f_4)), (1,(Nu,Nu,f_3,f_4)), (2,(Nu,Nu,f_3,f_4)), (3,((1,s_1),(2,q_1),f_3,f_4)), (4,((1,s_2),(2,q_2),f_3,f_4)), (5,((1,s_3),(2,q_3),f_3,f_4))}
c_5={(1,(0,r_1)), (3,(0,r_3)), (5,(0,r_5))}
c_6={(2,(0,r_2)), (4,(0,r_4))}
c_7={(2,(1,s_1)), (4,(1,s_3))}

c_8={(3,(1,s_2)), (5,(1,s_4))}
c_9={(3,(2,q_1)), (5,(2,q_3))}
c_{10}={(4,(2,q_2))}
$\xi^5 = \xi^4 \cup$ {(5,c_1,π_1), (5,c_2,π_2), (5,c_3,π_4), (5,c_4,π_4), (5,c_5,π_3), (5,c_8,π_3), (5,c_9,π_3)}
$\gamma^5{}_1(5)$=(ϕ,{c_6},1) $\gamma^5{}_2(5)$=({(c_9,1)},ϕ,2)
$\gamma^5{}_3(5)$=({(c_5,1)},{c_7},3) $\gamma^5{}_4(5)$=({(c_8,1)},{c_{10}},4)

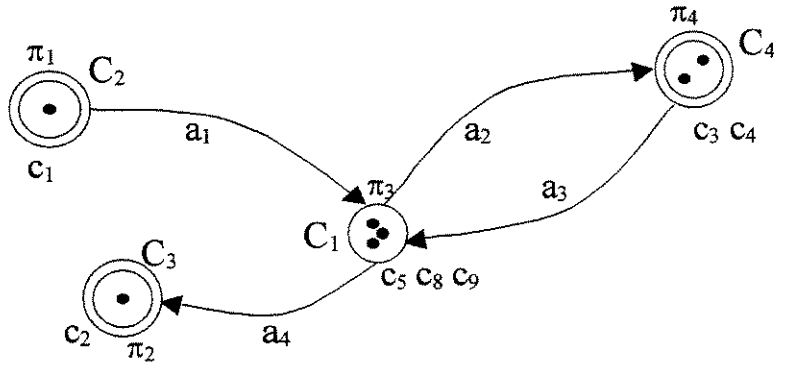


Figura 2.24 Rede de objeto para o processamento de conhecimento, no instante 5

$\Re_6$=(Σ,Π,Ξ,A,η,fpi,fpo,$\mathcal{C}^6$,ξ^6,γ^6)
Σ={C_1,C_2,C_3,C_4}
Π={π_1,π_2,π_3,π_4}
Ξ={(π_1,C_2),(π_2,C_3),(π_3,C_1),(π_4,C_4)}
A={a_1,a_2,a_3,a_4}
η={(a_1,(π_1,π_3)),(a_2,(π_3,π_4)),(a_3,(π_4,π_3)),(a_4,(π_3,π_2))}
$fpo_{\pi 1}(1)= a_1$, $fpi_{\pi 2}(1)= a_4$, $fpi_{\pi 4}(1)= a_2$, $fpi_{\pi 4}(2)= a_2$, $fpo_{\pi 4}(1)= a_3$, $fpo_{\pi 4}(2)= a_3$

$\mathcal{C}^6$= {c_1,c_2,c_3,c_4,c_5 ,c_6,c_7,c_8,c_9,c_{10}}
Nu=(0,r) é um valor *default*, Nu $\in$ C_1
c_1={(0,(0,Nu,f_1)), (1,(1,(0,r_1),f_1)), (2,(2,(0,r_2),f_1)), (3,(3,(0,r_3),f_1)), (4,(4,(0,r_4),f_1)),
(5,(5,(0,r_5),f_1)), (6,(6,(0,r_6),f_1))}
c_2={(0,(0,Nu,f_2)), (1,(1,Nu,f_2)), (2,(2,Nu,f_2)), (3,(3,(2,q_1),f_2)), (4,(4,(2,q_2),f_2)),
(5,(5,(2,q_3),f_2)), (6,(6,(2,q_4),f_2))}
c_3={(0,(Nu,Nu,f_3,f_4)), (1,(Nu,Nu,f_3,f_4)), (2,((0,r_1),(1,s_1),f_3,f_4)), (3,((0,r_2),(1,s_2),f_3,f_4)),
(3,((0,r_2),(1,s_2),f_3,f_4)), (4,((0,r_3),(1,s_3),f_3,f_4)), (5,((0,r_4),(1,s_4),f_3,f_4)),
(6,((0,r_5),(1,s_5),f_3,f_4))}
c_4={(0,(Nu,Nu,f_3, f_4)), (1,(Nu,Nu,f_3,f_4)), (2,(Nu,Nu,f_3,f_4)), (3,((1,s_1),(2,q_1),f_3,f_4)),
(4,((1,s_2),(2,q_2),f_3,f_4)), (5,((1,s_3),(2,q_3),f_3,f_4)) , (6,((1,s_4),(2,q_4),f_3,f_4))}
c_5={(1,(0,r_1)), (3,(0,r_3)), (5,(0,r_5))}
c_6={(2,(0,r_2)), (4,(0,r_4)), (6,(0,r_6))}
c_7={(2,(1,s_1)), (4,(1,s_3)), (6,(1,s_5))}

c_8={(3,(1,s_2)), (5,(1,s_4))}
c_9={(3,(2,q_1)), (5,(2,q_3))}
c_{10}={(4,(2,q_2)), (6,(2,q_4))}
$\xi^6 = \xi^5 \cup$ {(6,c_1,π_1), (6,c_2,π_2), (6,c_3,π_4), (6,c_4,π_4), (6,c_6,π_3), (6,c_7,π_3), (6,c_{10},π_3)}
$\gamma^6{}_1(6)$=(ϕ,{c_5},1) $\gamma^6{}_2(6)$=({(c_{10},1)},ϕ,2)
$\gamma^6{}_3(6)$=({(c_6,1)},{c_8},3) $\gamma^6{}_4(6)$=({(c_7,1)},{c_9},4)

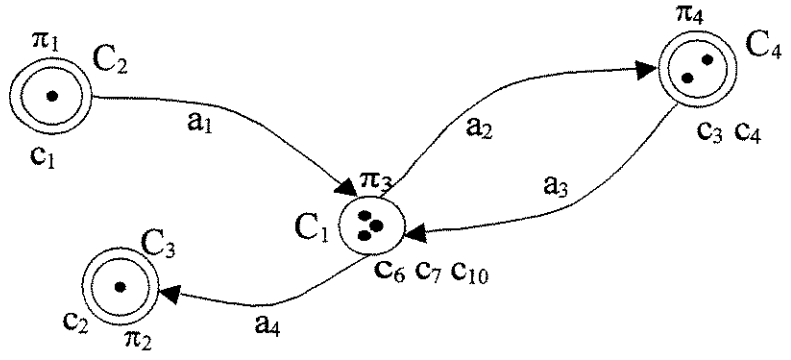


Figura 2.25 Rede de objeto para o processamento de conhecimento, no instante 6

Observe que neste caso, a função de seleção foi arbitrada ponto a ponto. Um possível algoritmo geral para obter o valor da função de seleção a cada instância é o seguinte:

Em cada instância um objeto da classe C_2 esta gerando um objeto da classe C_1 que representa um conhecimento sensorial (*flag* = 0). Sempre que exista um objeto da classe C_1 que represente um conhecimento sensorial (*flag* = 0) , um dos objetos da classe C_4 é disparado, assimilando destrutivamente o objeto com o conhecimento sensorial e gerando um objeto da classe C_1 que represente um conhecimento designativo (*flag* = 1). Sempre que exista um objeto representando um conhecimento designativo, um dos objetos da classe C_4 é disparado, assimilando destrutivamente o objeto representando o conhecimento designativo, gerando um objeto da classe C_1 que represente um conhecimento prescritivo (*flag* = 2). Sempre que exista um objeto representando um conhecimento prescritivo, será disparado um objeto da classe C_3 que assimila destrutivamente o objeto com o conhecimento prescritivo.

2.5.3. *Uso das redes de objetos para modelar o aprendizado de uma Rede Neural (RN) utilizando um Algoritmo Genético (GA).*

Neste exemplo [Guerrero 98, Guerrero 99a] mostraremos uma rede de objeto desenvolvida para modelar o mecanismo de aprendizado (através de um algoritmo genético), de uma rede neural desenvolvida para a aproximação da função $f(x,y,z) = e^{-z} \cdot (sin(x+y))^2$ no seguinte intervalo $-10\pi \le x \le 10\pi$, $-10\pi \le y \le 10\pi$ e $-1 \le z \le 1$.

Na figura 2.26 mostra-se o princípio de funcionamento do algoritmo genético utilizado para o aprendizado da rede neural desenvolvida para a aproximação da função $f(x,y,z)$ a qual se mostra na figura 2.27.

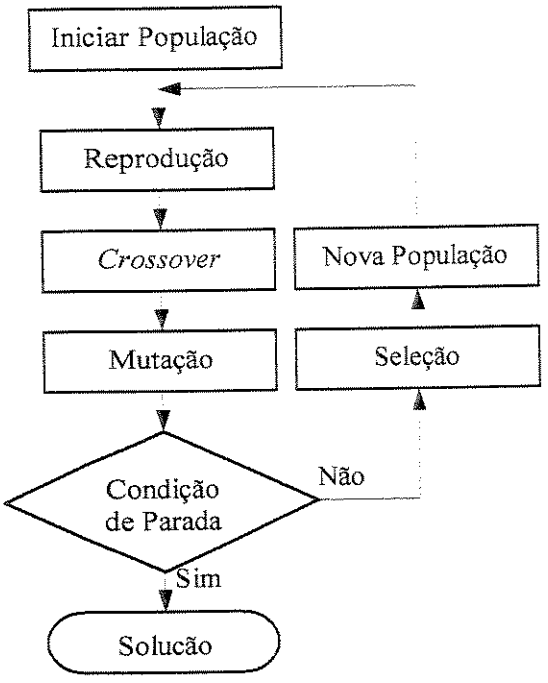


Figura 2.26 Princípio de funcionamento do GA utilizado para o aprendizado da RN.

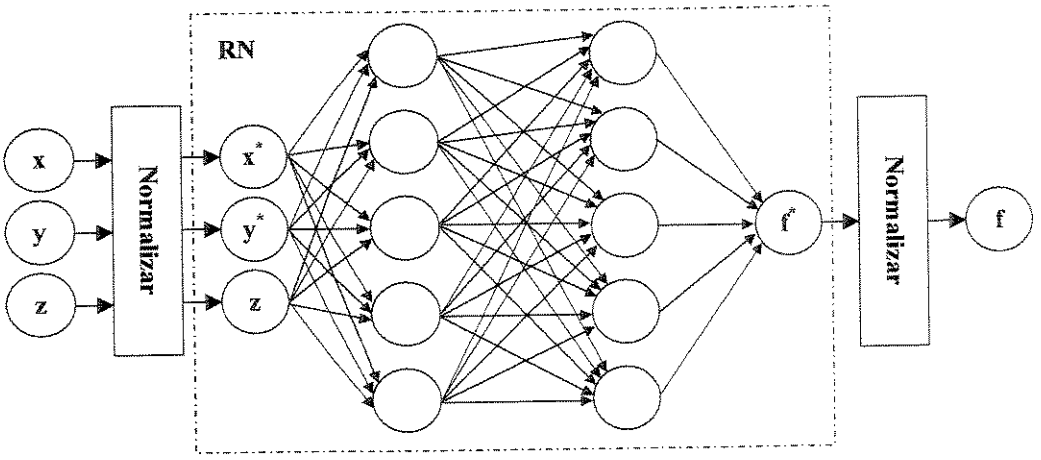


Figura 2.27 Estrutura da Rede Neural desenvolvida para aproximação da função *f(x,y,z)*.

A seguir mostramos a rede de objetos que modela o aprendizado da rede neural através do algoritmo genético.

Legenda	
GCT	Gerador de amostras.
GPI	Gerador de Redes Neurais.
Tr	Avaliador de Redes Neurais.
Cd	Codificador para cromossomos do GA.
SN	Operação de Seleção Natural para o GA
OpG	Operadores Genéticos para o GA.
Dcd	Decodificador de cromossomos do GA em matrizes de pesos para o treinamento da RN.
Solução	Representa a matrizes de pesos ótima para o trabalho da RN.
Bfrs	Representa um *buffer*

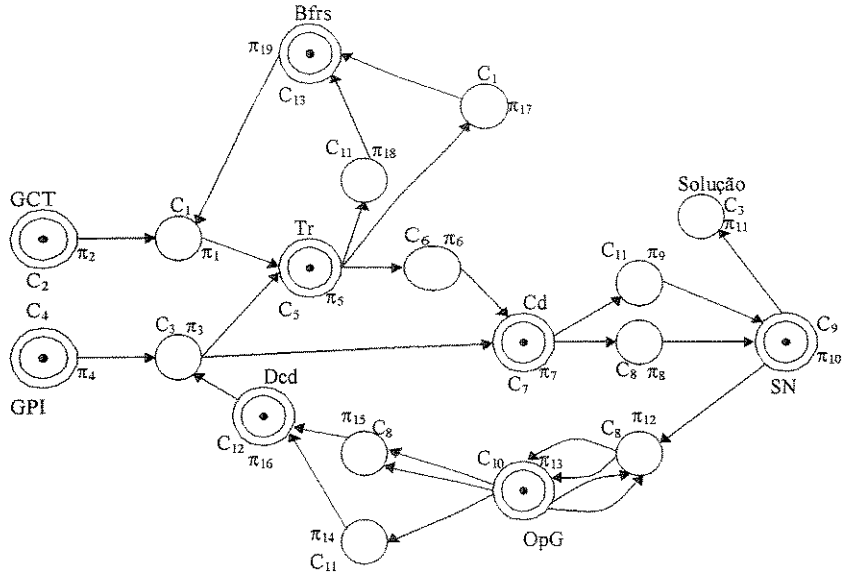


Figura 2.28 Rede de Objeto para modelar o aprendizado da RN utilizando-se um GA.

2.5.3.1. Definição das diferentes classes da rede de objetos

Classe C_1 – Classe de amostras.

$d(C_1) = (V_1, V_2, V_3, V_4)$ descritor da classe C_1

$C_1 - \{(v_1, v_2, v_3, v_4)\}$

onde:

$v_1 \in \Re$, representa o valor de x

$v_2 \in \Re$, representa o valor de y

$v_3 \in \Re$, representa o valor de z

$v_4 \in \Re$, representa o valor da função a ser aproximada $f(x,y,z)$

Classe C_2 – Classe dos geradores de amostra

$d(C_2) = (V_5, V_6, ((5),(5,6)))$ descritor da classe C_2

$C_2 - \{(v_5, v_6, f_1)\}$

onde:

$v_5 \in N$, representa um contador interno

$v_6 \in C_1$, corresponde à interface de saída de f_1 - representa o conjunto de treinamento da RN.

$f_1: N \rightarrow N \times C_1$, é uma função que gera um objeto da classe C_1 e atualiza o contador interno até que o contador interno seja igual ao tamanho do conjunto de treinamento (em nosso caso 500).

Classe C_3 – Classe de Redes Neurais

$d(C_3) = (V_7, V_8, V_9, Id)$ descritor da classe C_3

$C_3 - \{(v_7, v_8, v_9, id)\}$

onde:

$v_7 \in \Re^4 \times \Re^5$, representa a matriz dos pesos entre a entrada e a primeira camada oculta

$v_8 \in \Re^5 \times \Re^5$, representa a matriz dos pesos entre a primeira camada oculta e a segunda camada oculta

$v_9 \in \Re^5 \times \Re$, representa a matriz dos pesos entre a segunda camada oculta e a saída.

$id \in N$, representa um identificador.

Classe C_4 – Classe de Geradores de Redes Neurais

$d(C_4) = (V_{10}, V_{11}, ((10),(10,11)))$ descritor da classe C_4

$C_4 - \{(v_{10}, v_{11}, f_2)\}$

onde:

$v_{10} \in N$, representa um contador interno

$v_{11} \in C_3$, corresponde à interface de saída de f_2, representando uma rede neural.

$f_2: N \rightarrow N \times C_3$, é uma função que gera um objeto da classe C_3 e atualiza o contador interno, até que o contador interno seja igual ao tamanho da população (em nosso caso 3000).

Classe C_5 – Classe dos avaliadores de Redes Neurais

$$d(C_5) = \begin{pmatrix} V_{12}, V_{13}, V_{14}, V_{tp1}, V_{tp2}, V_{tp3}, V_{sal1}, V_{sal2}, \\ ((12,13),(tp1, tp2, tp3)), ((tp1, tp2, tp3),(sal1, sal2)) \end{pmatrix}$$ descritor da classe C_5

$C_5 - (v_{12}, v_{13}, v_{14}, v_{tp1}, v_{tp2}, v_{tp3}, v_{sal1}, v_{sal2}, f_3, f_4)$

onde:

$v_{12} \in C_1$ e $v_{13} \in C_3$, correspondem à interface de entrada de f_3, representando respectivamente uma amostra e uma rede neural.

$v_{14} \in C_6$ e $v_{tp1} \in \Re$, $v_{tp2} \in \Re$, $v_{tp3} \in \Re$, correspondem às interfaces de saída de f_3, correspondendo à uma avaliação da rede, frente à amostra.

$v_{sal1} \in C_6$ e $v_{sal2} \in C_{11}$, correspondem às interfaces de saída da função f_4

f_3: $C_1 \times C_3 \rightarrow C_1 \times \Re \times \Re \times \Re$, é uma função que faz a avaliação da RN, atualizando as variáveis internas e movendo o objeto da classe C_1.

f_4: $\Re \times \Re \times \Re \rightarrow C_6 \times C_{11}$, é uma função que gera um objeto da classe C_6 e um objeto da classe C_{11}.

Classe C_6 – Classe das avaliações

$d(C_6) = (V_{15}, V_{16}, Id)$ descritor da classe C_6

$C_6 - (v_{15}, v_{16}, id)$

onde:

$v_{15} \in \Re^3$ que tem a seguinte forma, (ErroMax, Erro_Med, ErroQM), e representa os diversos erros, erro máximo, erro médio e erro quadrático médio, respectivamente; obtidos pelo avaliados de RN.

$v_{16} \in \Re$, corresponde a uma relação do ErroQM e o ErroMax, representando a função objetivo *(fitness function)* para o GA.

$id \in N$, representa um identificador.

Classe C_7 – Classe dos codificadores

$d(C_7) = (V_{17}, V_{18}, V_{19}, V_{20}, V_{21}, ((19,18,17),(20,17)),((17),(17,21)))$ descritor da classe C_7

$C_7 - \{(v_{17}, v_{18}, v_{19}, v_{20}, v_{21}, f_5, f_6)\}$

onde:

$v_{17} \in N$ representa a um contador interno

$v_{18} \in C_6$ e $v_{19} \in C_3$, correspondem à interface de entrada de f_5.

$v_{20} \in C_8$, corresponde à interface de saída de f_5, representando um cromossomo.

$v_{21} \in C_{11}$, corresponde à interface de saída de f_6, representando um *flag* relacionado ao tamanho da população (0 se o contador interno é igual ao tamanho da população, 1 se o contador interno é maior que o tamanho da população)

f_5: $C_3 \times C_6 \times N \rightarrow C_8 \times N$, é uma função que assimila destrutivamente um objeto da classe C_3 e um objeto da classe C_6 com atributo de identificador *(id)* iguais, gerando um objeto da classe C_8, ou seja, um cromossomo da população, e atualizando o contador interno.

f_6: $N \rightarrow N \times C_{11}$, é uma função que gera um objeto da classe C_{11} e reseta o contador interno para 0.

Classe C_8 – Classe dos cromossomos

$d(C_8) = (V_{22}, V_{23}, V_{24}, V_{fl})$ descritor da classe C_8

$C_8 - \{(v_{22}, v_{23}, v_{24}, v_{fl})\}$

onde:

$v_{22} \in \Re^{50}$, representa um vetor que codifica os pesos da RN.

$v_{23} \in \Re^3$, representa uma tripla que contém os erros associados à avaliação da RN.

$v_{24} \in \Re$, representa o valor da função objetivo *(fitness function)* para o cromossomo.

$v_{fl} \in N$, representa um *flag* (0 indica que não se fez nada com ele, 1 que se fez uma cópia e 2 que se fez o *crossover*) utilizado pelo operadores genéticos.

Classe C_9 – Classe dos selecionadores naturais

$d(C_9) = (V_{25}, V_{26}, V_{27}, V_{28}, V_{29}, ((26,25),(25)),((27),(29)),((27),(29)),((27),(28)))$ descritor da classe C_9

$C_9 - \{(v_{25}, v_{26}, v_{27}, v_{28}, v_{29}, f_7, f_8, f_9, f_{10})\}$

onde:

$v_{25} \in N$ representa um contador interno (quantidade de iterações do GA).

$v_{26} \in C_{11}$ corresponde à interface de entrada da função f_7.

$V_{27} \in C_8$ corresponde à interface de entrada das funções f_8, f_9 e f_{10}.

$v_{28} \in C_3$, corresponde à interface de saída da função f_{10}.

$v_{29} \in C_8$, corresponde à interface de saída das funções f_8 e f_9.

f_7: $C_{11} \times N \rightarrow N$, é uma função que assimila destrutivamente um objeto da classe C_{11} e incrementa o contador interno.

f_8: $C_8 \rightarrow C_8$, é uma função que assimila destrutivamente a objeto da classe C_8 e faz uma cópia do objeto da classe C_8.

f_9: $C_8 \rightarrow C_8$, é uma função que assimila destrutivamente a objeto da classe C_8 e faz uma seleção com 30% dos melhores indivíduos e 70% dos piores indivíduos da população.

f_{10}: $C_8 \rightarrow C_3$, é uma função que assimila destrutivamente um objeto da classe C_8, faz uma escolha do melhor indivíduo da população e gera um objeto da classe C_3, que por sua vez representa a solução do problema.

Classe C_{10} – Classe dos operadores genéticos

$$d(C_{10}) = \begin{pmatrix} V_{30}, V_{31}, V_{32}, V_{33}, V_{34}, V_{35}, V_{36}, V_{37}, ((31),(30,33,35)), \\ ((31,32),(30,33,34,35,36)),((31),(30,33)), \\ ((30),(37)) \end{pmatrix}$$ descritor da classe C_{10}

$C_{10} - \{(v_{30}, v_{31}, v_{32}, v_{33}, v_{34}, v_{35}, v_{36}, v_{37}, f_{11}, f_{12}, f_{13}, f_{14})\}$

onde:

$v_{30} \in N$ representa um *flag* interno (0 ativa o operador reprodução, 1 ativa o operador *crossover*, 2 ativa o operador mutação e 3 indica o fim das operações genéticas).

$v_{31} \in C_8$ e $v_{32} \in C_8$ correspondem às interfaces de entrada das funções f_{11}, f_{12} e f_{13}.

$v_{33} \in C_8$, $v_{34} \in C_8$, $v_{35} \in C_8$ e $v_{36} \in C_8$ correspondem às interfaces de saída das funções f_{11}, f_{12} e f_{13}.

$v_{37} \in N$, corresponde à interface de saída da função f_{14}.

f_{11}: $C_8 \rightarrow N \times C_8 \times C_8$, é uma função que faz uma cópia do objeto da classe C_8, atualiza o *flag* interno do objeto C_8 e o transporta de lugar. Além disso, atualiza o valor do *flag* interno para "1". Esta função representa o operador genético *reprodução*.

f_{12}: $C_8 \times C_8 \rightarrow N \times C_8 \times C_8 \times C_8 \times C_8$, é uma função que faz um *crossover* uniforme de dois objetos da classe C_8 (indivíduos) selecionados utilizando a técnica *RouletteWheel*, atualiza o valor do *flag* interno para "2" e atualiza o flag interno dos objetos da classe C_8

selecionados, transportando-os de lugar. Esta função representa o operador genético *crossover*.

f_{13}: $C_8 \rightarrow N \times C_8$, é uma função que assimila destrutivamente o objeto da classe C_8, faz mutações no melhor indivíduo e faz uma atualização no valor do *flag* interno para "3". Esta função representa o operador genético *mutação*.

f_{14}: $N \rightarrow N$, é uma função que gera um objeto da classe C_{11} e atualiza o *flag* interno para "0".

Classe C_{11} – Classe dos *flags* de Controle

$d(C_{11}) = (V_{38})$ descritor da classe C_{11}

$C_{11} - \{(v_{38})\}$

onde:

$v_{38} \in N$ representa um *flag* para a ativação de algumas funções de transformação de alguns objetos (classe C_9 e classe C_{12}).

Classe C_{12} – Classe dos decodificadores

$d(C_{12}) = (V_{39}, V_{40}, V_{41}, ((40,39),(41)))$ descritor da classe C_{12}

$C_{12} - \{(v_{39}, v_{40}, v_{41}, f_{15})\}$

onde:

$v_{39} \in N$ e $v_{40} \in C_8$ correspondem à interface de entrada da função f_{15}.

$V_{41} \in C_3$, corresponde à interface de saída da função f_{15}.

f_{15}: $C_8 \times N \rightarrow C_3$, é uma função que assimila destrutivamente o objeto da classe C_8 e gera objetos da classe C_3. Caso não existam mais objetos da classe C_8, então assimila destrutivamente o objeto da classe C_{11}.

Classe C_{13} – Classe dos *buffers*

$d(C_{13}) = (V_{42}, V_{43}, V_{44}, ((42,43),(44)))$ descritor da classe C_{13}

$C_{13} - \{(v_{42}, v_{43}, v_{44}, f_{16})\}$

onde:

$v_{42} \in C_1$ e $v_{43} \in C_{11}$ corresponde à interface de entrada das função f_{16}.

$V_{44} \in C_3$, corresponde à interface de saída da função f_{16}.

f_{16}: $C_1 \times C_{11} \rightarrow C_1$, é uma função que assimila destrutivamente o objeto da classe C_1 e faz uma cópia dele. Quando não houver mais nenhum objeto da classe C_1 para ser assimilado, assimila destrutivamente o objeto da classe C_{11}.

2.5.3.2. Definindo a função de seleção para a rede de objetos definida.

Um algoritmo geral para a dinâmica deste sistema (ou seja, sua função de seleção) é o seguinte:

O objeto da classe C_2 gera um conjunto de objetos da classe C_1 que são colocados em π_1, criando desta maneira o conjunto de treinamento da Rede Neural. O objeto da classe C_4 gera um objeto da classe C_3.

O objeto da classe C_5 assimila um objeto da classe C_3 e vai assimilando destrutivamente os objetos da classe C_1 e efetuando a avaliação da RN, atualizando sempre as variáveis internas e gerando uma cópia do objeto C_1, assimilado destrutivamente para π_{17}. Quando não existam mais objetos em π_1, o objeto da classe C_5 gera um objeto da classe C_{11} para π_{18} e um objeto da classe C_6 para π_6.

O objeto da classe C_7 assimila destrutivamente os objetos da classe C_3 e C_6 que encontram-se em π_3 e π_6 respectivamente e que tenham o atributo de identificador *(id)* iguais, gerando o objeto da classe C_8, até que não existam mais objetos em π_6 e π_3. Então, caso a quantidade de objetos gerados para π_8 seja igual ao tamanho da população o objeto da classe C_7 gera um objeto da classe C_{11} com valor 0. Se a quantidade de objetos é maior do que o tamanho da população, então o objeto da classe C_7 gera um objeto da classe C_{11} com valor 1, que é colocado em π_9.

O objeto da classe C_9 tem um comportamento mais complexo. Seu regime de trabalho é o seguinte. Se o contador interno deste objeto é igual ao número de iterações máximo (condição de parada do AG), então este gera um objeto da classe C_3 que representa a solução do problema. Caso ele não seja igual, e existe um objeto da classe C_{11} em π_9 com valor 0, então executa-se a função de transformação f_7, que faz uma cópia do objetos da classe C_8 (que encontram-se em π_8) em π_{12}. Se o objeto da classe C_{11} tem como valor 1, então a função de transformação que se ativa é f_8. Esta, faz uma seleção compreendendo 30% dos melhores indivíduos e 70% dos piores indivíduos, que são colocados em π_{12}. Em caso de não existir objetos em π_8, o objeto da classe C_9, só assimila o objeto da classe C_{11} em π_9 e incrementa o contador interno que representa a condição de parada do GA.

O objeto da classe C_{10} em π_{13} representa os operadores genéticos (*reprodução*, *crossover* e *mutação*). Os mesmos são ativados da seguinte maneira: um objeto da classe C_{10} é ativado quando existe um objeto da classe C_8 em π_{12}. Se o *flag* interno é 0, então ativa-se o operador *reprodução* que faz uma cópia do objeto C_8 que se encontra em π_{12}, colocando-o em π_{15}. Re-envia os objetos da classe C_8 que estão em π_{12} para o próprio π_{12}, atualizando o atributo *id* para "1", indicando que esse objeto já foi copiado. Quando não existam mais objetos em π_{12} com atributo "0", então o *flag* interno é atualizado com o valor "1". Quando o *flag* interno tem valor "1", ativa-se o operador *crossover*. O mesmo assimila dois indivíduos baseados na técnica *RouletteWheel* e faz um *Crossover* uniforme de seus genes, colocando estes novos indivíduos em π_{15}. Além disso, re-envia os objetos utilizados de π_{12} para o próprio π_{12}, atualizando o atributo *id* com valor "2" e atualizando o *flag* interno com valor "2".

Se o *flag* interno é 2, ativa-se o operador *mutação*, que assimila destrutivamente os objetos da classe C_8 em π_{12}, seleciona o melhor indivíduo e faz mutações em seus genes

aleatoriamente, gerando novos indivíduos para π_{15}. Isso ocorre até que não existam mais objetos em π_{12}, gerando um objeto da classe C_{11} que é colocado em π_{14}.

O objeto da classe C_{12} em π_{16} é ativado com a existência de objetos da classe C_8 em π_{15} e C_{11} em π_{14}. Este objeto converte o vetor que representa o cromossomo (objeto da classe C_8) em matrizes de pesos (objetos da classe C_3), para que sejam utilizadas pela RN em sua nova avaliação.

O objeto da classe C_{13} é ativado de duas maneiras diferentes. Enquanto existir objetos da classe C_1 em π_{17} e um objeto da classe C_{11} em π_{18}, este assimila não destrutivamente o objeto em π_{18} e destrutivamente um dos objetos de π_{17}, fazendo uma cópia deste que é enviada para π_1. Quando não existir mais objetos da classe C_1 em π_{17}, ele então assimila destrutivamente o objeto da classe C_{11} em π_{18}.

2.6. Resumo

Neste capítulo, introduziu-se os fundamentos para a teoria dos objetos. Estes fundamentos são basicamente os introduzidos por Gudwin [Gudwin 96], enriquecidos por uma re-estruturação na concepção original, onde acrescentou-se algumas definições adicionais, necessárias para os desenvolvimentos seguintes, quando introduziremos as redes de agentes. Iniciou-se pela definição conceitual de objeto, seguida de uma definição formal para objetos. Em seguida, foram definidos os chamados sistemas de objetos e por fim as redes de objetos, caracterizadas como um tipo especial de sistema de objeto. Encerrou-se o capítulo com alguns exemplos ilustrativos de redes de objetos.

CAPÍTULO 3. Agentes e Redes de Agentes

3.1. Introdução

Este capítulo apresenta uma descrição da tecnologia de agentes (uma das áreas de maior crescimento em tecnologia da informação, na atualidade) [Jennings 98], seguida da formalização e definição das redes de agentes - um tipo especial de redes de objetos que incorpora uma série de conceitos utilizados em teoria de agentes. Genericamente, entende-se por teoria de agentes como sendo uma tecnologia de desenvolvimento de sistemas computacionais, baseada em uma metáfora para solução de problemas que utiliza entidades autônomas que cooperam e coordenam suas atividades de modo a obter um objetivo desejado. Uma das razões para a popularidade desta tecnologia é que esta constitui-se de uma maneira natural e intuitiva para a conceptualização e modelagem de diversos tipos de problemas. Outra razão é o conjunto de estruturas e processos úteis e poderosos gerados pela tecnologia de agentes para o *design* e construção de aplicações de *software* complexas.

O aspecto mais importante deste capítulo é a apresentação das rede de agentes como uma especificação das rede de objetos, as quais possibilitam a criação de agentes complexos mediante a interação dos diferentes micro-agentes básicos que compõem uma rede de agente.

3.2. O que é um agente?

O maior problema na definição do termo **agente** é que o mesmo é um vocábulo extensamente usado por muitas pessoas que trabalham em áreas estreitamente relacionadas. Wooldridge e Jennings [Wooldridge 95] consideram um desafio tentar produzir uma definição simples e universal que seja aceita. Eles colocam que pode-se distinguir duas noções gerais para o termo **agente. A** primeira delas, também chamada de noção fraca, é relativamente menos discutida. A segunda, chamada de noção forte, é normalmente mais discutida entre os pesquisadores da área.

3.2.1. Uma noção fraca para agentes

Segundo Wooldridge e Jennings [Wooldridge 95], uma noção fraca do termo agente é aquela que o utiliza para denotar qualquer *hardware* ou sistema de computação baseado em *software*, que apresente as seguintes propriedades:

- **autonomia**: os agentes operam sem a intervenção direta dos humanos ou outros agentes, além de ter algum tipo de controle de suas ações e estados internos;

- **habilidade social**: os agentes interagem com outros agentes (e possivelmente com humanos) através de algum tipo de linguagem de comunicação de agentes (ACL);
- **reatividade**: os agentes percebem seu ambiente, o qual pode ser o mundo real, um usuário via uma interface gráfica de usuário (GUI), uma coleção de outros agentes, a INTERNET, ou talvez a combinação de alguns destes ou de todos respondendo de forma oportuna às mudanças que ocorrem neste ambiente;
- **pró-atividade** *(pro-activeness)*: os agentes não simplesmente reagem em resposta ao ambiente, mas têm a capacidade de exibir condutas baseadas em metas, tomando a iniciativa em relação a suas próprias ações.

3.2.2. Uma noção forte para agentes

Para alguns pesquisadores, particularmente aqueles que trabalham na área da Inteligência Artificial, o termo agente deve ter um significado mais específico que o adotado pela noção fraca. Em sentido geral, estes pesquisadores descrevem um agente como um sistema de computação que, além de apresentar as propriedades identificadas na noção fraca, deve ser definido ou implementado utilizando-se conceitos que usualmente são aplicáveis aos seres humanos. Por exemplo, é bastante comum na Inteligência Artificial caracterizar os agentes usando noções aplicáveis à mente humana, tais como conhecimentos, crenças, intenções e obrigações.

3.2.3. Outros atributos para agentes

Em algumas ocasiões, outros atributos são discutidos no contexto de agentes. Como exemplo, temos:

- **mobilidade**: é a habilidade de um agente para se transportar entre máquinas participantes de uma rede de computadores;
- **veracidade**: é a suposição que um agente não comunicará informações falsas de maneira intencional;
- **benevolência**: é a suposição de que os agentes não têm objetivos contraditórios, e que todo agente tentará sempre responder ao que lhe é perguntado;
- **racionalidade**: é a suposição de que o agente sempre agirá para alcançar suas metas, e nunca agirá contra seus objetivos, pelo menos na medida em que suas crenças o permitam.

3.2.4. Outras definições de agente

Outros pesquisadores da área, tem desenvolvido definições interessantes para o termo "agente", muitas delas até conflitantes. Dentre outras podemos citar as seguintes:

- "Um agente é qualquer coisa que possa perceber um ambiente por meio de sensores e atuar no mesmo por meio de atuadores" (Russel e Norvig) [Russell 95]

- "Agentes inteligentes realizam continuamente três funções: percepção das condições dinâmicas de um ambiente, ação de modo a afetar condições do ambiente e raciocínio para interpretar percepções, realizar inferências e determinar ações" (Barbara Hayes-Roth - Stanford) [Franklin 96]
- "Agentes autônomos são sistemas computacionais que habitam um ambiente complexo e dinâmico, sensoreiam e atuam autonomamente neste ambiente, realizando desta maneira uma série de metas e tarefas para as quais foram projetados" (Pattie Maes - MIT Media Lab) [Franklin 96]
- "Agentes inteligentes são entidades de software que realizam um conjunto de operações em nome de um usuário ou outro programa com certo grau de independência ou autonomia, e desta maneira empregam algum conhecimento ou representação das metas e/ou desejos do usuário" (IBM's Intelligent Agent Strategy) [Franklin 96]
- "Um agente autônomo é um sistema que é parte de um ambiente, estando situado dentro dele, e sente e age sobre este ambiente, no tempo, de acordo com seus próprios propósitos, de modo a alterar o que sentirá no futuro" (Stan Franklin e Art Graesser) [Franklin 96]

3.3. Arquitetura de agentes

Um tópico importante relacionado à tecnologia de agentes diz respeito às diferentes arquiteturas que podem ser idealizadas para a implementação de agentes. Entende-se por uma arquitetura de agentes como o conjunto de especificações e técnicas utilizadas para a definição funcional dos agentes. Nesta seção, descreveremos alguns dos conceitos relacionados a arquiteturas de agentes. Primeiramente daremos duas definições extraídas do trabalho de Wooldridge e Jennings [Wooldridge 95].

Maes define uma arquitetura de agentes como:

> "Uma metodologia particular para a construção de agentes. Especifica como ... os agentes podem ser decompostos na construção de um conjunto de módulos (componentes) e como estes módulos podem interagir entre si. O conjunto total de módulos e suas interações deve especificar como os dados dos sensores e o estado interno do agente serão utilizados para determinar as ações realizadas pelo agente... e seu futuro estado interno. Uma arquitetura envolve técnicas e algoritmos que suportem esta metodologia."

Kaelbling considera uma arquitetura de agentes como:

> "Uma coleção específica de módulos de software (ou hardware), tipicamente designados por caixas com setas que indicam os dados e o fluxo de controle entre os módulos. Uma visão mais abstrata de uma arquitetura é uma metodologia geral para projetar a decomposição em módulos particulares direcionados a tarefas particulares."

3.3.1. Enfoques clássico: Arquiteturas deliberativas

Pode-se definir um agente deliberativo ou arquitetura deliberativa de agente, como um agente que contenha uma representação explícita de um modelo simbólico do mundo. Neste, as decisões (por exemplo, as ações a serem executadas) são determinadas através de um raciocínio lógico (ou pelo menos pseudo-lógico), baseado em padrões de "*matching*" e manipulação simbólica. A idéia de agentes deliberativos baseados puramente em um raciocínio lógico é extremamente tentadora. Para fazer com que o agente esteja de acordo com alguma teoria de agentes, poderíamos ingenuamente supor que é suficiente dar-lhe uma representação lógica desta teoria e fazer com que o agente realize alguma prova de teorema. Se tratarmos de construir um agente por esta via, teríamos pelo menos dois problemas importantes a serem resolvidos:

- Problema de tradução (*the transduction problem*): como traduzir o mundo real em uma descrição simbólica exata e adequada, que por sua vez, possa ser útil.
- Problema de representação e/ou raciocínio (*the representation/reasoning problem*): como armazenar estruturas lógicas que correspondam a uma representação simbólica de processos e entidades complexas do mundo e como fazer com que os agentes realizem algum raciocínio sobre estas estruturas, obtendo resultados que possam ser úteis.

Apesar do imenso volume de trabalhos que estes problemas tem gerado, muitos pesquisadores aceitam a hipótese de estarmos ainda longe de vermos estes problemas resolvidos, lembrando que a lógica de primeira ordem não é nem sequer decidível e extensões modais a ela (incluindo representações de crenças, desejos, tempo e outras) tendem a ser altamente não-decidíveis. Talvez o mais problemático para a Inteligência Artificial Simbólica é que muitos dos algoritmos de interesse para a manipulação simbólica são intratáveis (parece difícil construir algoritmos úteis de manipulação simbólica que tenham a garantia de terminar em um limite de tempo fixo aceitável e com resultados úteis). Um exemplo da abordagem clássica são os agentes planejadores.

3.3.1.1. Agentes planejadores

Desde o início dos anos 70, os adeptos da abordagem "*planning*" dentro da Inteligência Artificial estiveram estreitamente preocupados com o *design* de agentes artificiais. De fato, parece razoável dizer que a maioria das inovações no *design* de agentes vem desta comunidade. Determinar planos é essencialmente uma tarefa de programação automática: verificar se uma ação (ou conjunto de ações), quando executada(s), vai resultar na obtenção de algum objetivo desejado. Dentro da comunidade da Inteligência Artificial Simbólica, existe a predisposição em aceitar que qualquer agente artificial deve possuir algum tipo de sistema de planejamento como componente central. Talvez o mais conhecido sistema planejador é o STRIPS. Este sistema assume uma descrição simbólica do mundo, do objetivo desejado, e um conjunto de ações, caracterizadas por suas pré-condições e pós-condições associadas. O STRIPS tenta encontrar uma seqüência de ações que possa levar ao sucesso de alguma meta, usando uma análise simples de verificação de sucesso (*simple*

means-ends analysis). Esta análise envolve essencialmente um *matching* das pós-condições das ações com o objetivo desejado. O algoritmo de planejamento STRIPS era muito simples e teve sua ineficiência comprovada para problemas com complexidade moderada. Outras técnicas foram desenvolvidas para melhorar este algoritmo, tais como os planejadores não lineares e os planejadores hierárquicos. Porém em meados dos anos 80, alguns resultados teóricos mostravam que essas técnicas de refinamento poderiam não ser úteis em qualquer sistema de tempo restrito. Apesar destas dificuldades, várias tentativas têm sido realizadas para construir agentes nos quais o componente primário é um algoritmo planejador. Os resultados obtidos nesta área tiveram uma profunda influência sobre as pesquisas posteriores feitas sobre os algoritmos planejadores. Talvez mais do que qualquer outra coisa, os fracassos obtidos tenham levado alguns dos pesquisadores a questionarem, em sua totalidade, o paradigma da Inteligência Artificial Simbólica, e por conseguinte conduzido-os a trabalhar com enfoques alternativos que serão apresentados a seguir.

3.3.2. Enfoques alternativos: Arquiteturas reativas

A rejeição e o questionamento da viabilidade do paradigma da Inteligência Artificial Simbólica conduziu ao que geralmente é conhecido como sendo as arquiteturas reativas. Estas não incluem nenhum tipo de modelo simbólico central do mundo e não utilizam raciocínio simbólico complexo.

3.3.2.1. Linguagens de comportamento

Woldridge e Jennings[Wooldridge 95] consideram Rodney Brooks, pesquisador do MIT, como um dos principais críticos da noção de agência da Inteligência Artificial Simbólica, motivado pelas deficiências dos enfoques da Inteligência Artificial para construir mecanismos de controle para robôs móveis autônomos. Brooks [Brooks 86] resumiu uma arquitetura alternativa para construir agentes, chamada de arquitetura da suposição (*subsumption architecture*). Ele propôs em trabalhos posteriores [Brooks 90, Brooks 91a, Brooks 91b] três teses chaves:

- Comportamento inteligente pode ser gerado sem as representações explícitas do tipo que a Inteligência Artificial propõe.
- Comportamento inteligente pode ser gerado sem um raciocínio abstrato e explícito do tipo que a Inteligência Artificial propõe.
- Inteligência é uma propriedade emergente de certos sistemas complexos.

Brooks também identificou duas idéias básicas que tiveram influência em suas pesquisas:

- *situatedness* e *embodiment*: inteligência "real" está localizada no mundo e não em sistemas incorpóreos tais como provadores de teoremas ou sistemas especialistas.

- *Inteligência* e *emergência*: comportamento "inteligente" surge como resultado de uma interação de um agente com seu ambiente. Também, a inteligência está no olho do observador e não é uma propriedade isolada nem inata.

3.3.2.2. Outras arquiteturas reativas

Outros trabalhos foram desenvolvidos para explorar alternativas para o paradigma da Inteligência Artificial planejadora. Entre eles podemos citar os seguintes:

- Sistema PENGI: Jogo de computador desenvolvido por Agre & Chapman. Agre tinha observado que a maioria das atividades diárias são uma "rotina", ou seja, envolvem somente pequenas adaptações sobre atividades que já são conhecidas. Novas tarefas, uma vez aprendidas, podem ser incorporadas como novas rotinas, passando a incorporar o conjunto de atividades conhecidas. Agre propôs que uma arquitetura de agente eficiente pode ser baseada nos "argumentos de funcionamento". Em resumo, a idéia é que a maioria das decisões são rotinas, as quais podem ser codificadas em uma estrutura de baixo nível (tais como um circuito digital), que somente necessitará ser atualizada periodicamente, por exemplo para resolver novos tipos de problemas. [Agre 87].
- Arquitetura de Rede de Agente: Pattie Maes desenvolveu uma arquitetura de agentes na qual um agente é definido como um conjunto de módulos que competem entre si [Maes 91]. Estes módulos tem uma pequena semelhança com a arquitetura de suposição de Brooks. Cada módulo é especificado pelo projetista, em termos de suas pré e pós-condições e um nível de ativação, o qual oferece um valor real que indica a relevância de um módulo em uma situação particular. Quanto maior for o nível de ativação de um módulo, mais provável é a sua influência na conduta do agente. Uma vez especificado, o conjunto de módulos que competem são compilados dentro de uma rede de ativação distribuída (*spreading activation network*), onde os módulos são conectados uns aos outros segundo suas pré e pós-condições. Quando um agente está executando alguma tarefa, diferentes módulos podem ser mais ativos que outros, em determinadas situações, o que permite que estes possam ser executados. O resultado dessa execução pode ser um comando para uma unidade atuadora ou um incremento no nível de ativação de outro módulo conectado a ele. (OBS.: Nossa definição para redes de agentes não encontra qualquer relação com a rede de agentes de Pattie Maes).

3.3.3. Arquiteturas híbridas

Muitos pesquisadores sugeriram que nem um enfoque completamente deliberativo, nem um completamente reativo é apropriado para a construção de agentes. Discute-se então, o caso dos sistemas híbridos, abordagem que associa ambos enfoques. Um enfoque óbvio é construir um agente com dois (ou mais) sub-sistemas, um deles deliberativo (que contenha um modelo simbólico do mundo, desenvolva planos e tome decisões) e outro reativo (que seja capaz de reagir aos eventos que ocorrem no ambiente sem envolver raciocínios complexos). Com freqüência o componente reativo é colocado com algum tipo de

precedência sobre o componente deliberativo. Desta forma, ele pode oferecer uma resposta rápida a eventos importantes do ambiente. Este tipo de estrutura leva naturalmente à idéia de uma arquitetura em camadas. Nesta arquitetura, os sub-sistemas de controle de agentes são organizados dentro de uma hierarquia, na qual as camadas mais altas lidam com as informações em níveis crescentes de abstração. Assim, por exemplo, as camadas mais baixas poderiam mapear diretamente os dados dos sensores para as saídas dos atuadores sem nenhum tratamento prévio, enquanto as camadas superiores fazem o tratamento das tarefas de longo prazo. O principal problema nesta arquitetura é que o tipo de estrutura de controle deverá ser embutida nos sub-sistemas de agentes para manipular as interações entre as diferentes camadas. Dois bons exemplos desta arquitetura são as máquinas de Touring desenvolvidas por Ferguson e descritas por Wooldridge e Jennings [Wooldridge 95] e InteRRaP (figura 3.1) [Brenner 98]. Outro exemplo das arquiteturas híbridas é o *Procedural Reasoning System* (PRS) .

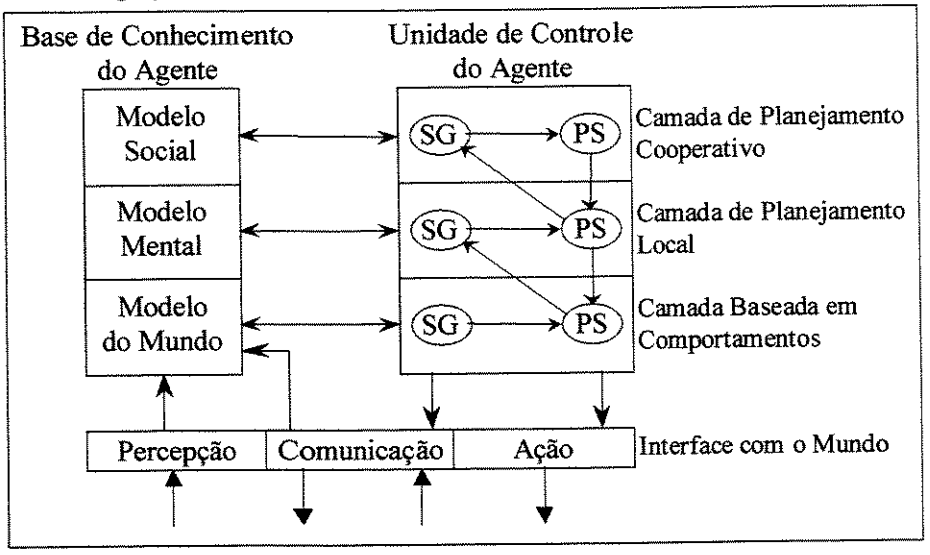


Figura 3.1 Arquitetura híbrida InteRRaP.

3.4. Tipologia de Agentes

Existem diferentes critérios para a classificação de agentes. Por exemplo, os agentes podem ser classificados por sua mobilidade, ou seja, sua habilidade em mover-se por diferentes nós de uma rede. Segundo este conceito os agentes podem ser classificados como agentes estáticos ou agentes móveis. Outra possível classificação pode ser feita segundo sua arquitetura. Assim, eles poderiam ser classificados como deliberativos ou reativos (também conhecidos na literatura como reflexivos).

Os agentes também podem ser classificados segundo os diferentes atributos que possam idealmente exibir. Neste sentido, Hyacinth Nwana [Nwana 96a, Nwana96b, Ndumu 97, Nwana 98], em seus trabalhos, descreve uma classificação prática dos agentes (baseado em alguns casos no que são os agentes, e outros no papel que eles executam), resultando na seguinte classificação:

- **Agentes colaborativos** (*Collaborative Agents*): agentes geralmente estáticos, grandes e de "grânulo grosso", sobre os quais há ênfase na autonomia e cooperação com outros agentes para executar tarefas em prol de seus proprietários, em ambientes multi-agente abertos ou de tempo limitado. Eles podem ter aprendizado, mas este atributo não é geralmente de maior importância em sua operação. Para coordenar suas atividades, eles podem realizar algum tipo de negociação para alcançar acordos mutuamente aceitáveis.[Nwana 96c]
- **Agentes de interface** (*Interface Agents*): suportam e fornecem uma ajuda pró-ativa, geralmente para um usuário utilizando um programa de aplicação complexo. Este tipo de agente enfatiza sua autonomia e capacidade de aprendizado para executar as tarefas em nome de seus proprietários. Uma metáfora usada para definir os agentes de interface, é que são *assistentes pessoais* os quais estão colaborando com o usuário no mesmo ambiente de trabalho. Sua cooperação com outros agentes, se existe, é tipicamente limitada para responder às consultas.[Maes 94, Lashkari 94, Koda 96]
- **Agentes móveis** (*Mobile Agents*): processos de software com capacidade de movimentar-se através das redes de longo alcance (*WANs, Wide Area Networks*), como é o caso da WWW (*World Wide Web*), interagindo com *hosts* externos, executando tarefas em nome de seus proprietários e retornando a sua origem com o resultado das tarefas executadas. Estas tarefas ou obrigações podem ser as mais diversas possíveis, desde fazer uma reserva de vôo até manipular uma rede de telecomunicações.
- **Agentes de informação** (*Information Agents*): administradores de informação WWW pró-ativos, dinâmicos, adaptativos e cooperativos que executam o papel de administradores, manipuladores ou coletores de informação de qualquer recurso distribuído.[Petrie 96]
- **Agentes reativos ou reflexivos** (*Reactive Agents*): agentes que não possuem internamente modelos simbólicos de seus ambientes, embora respondam de maneira "estímulo–resposta" ao estado atual do ambiente no qual são colocados.
- **Agentes híbridos** (*Hybrid Agents*): agentes cuja constituição é uma combinação de duas ou mais filosofias.
- **Sistemas de agentes heterogêneos** (*Heterogeneous Agent Systems*): algum software baseado em agentes que combine dois ou mais agentes das categorias descritas acima.

Outra classificação dos agentes que pode ser colocada é a seguinte:

- **Agentes Reativos ou Reflexivos**: apresentam um processo único de percepção-ação (reflexo). A ação é uma função direta das entradas dos sensores. Esta função pode ser qualquer conjunto de regras condição-ação (*fuzzy* ou binária), uma rede neural ou uma função matemática. (figura 3.2).

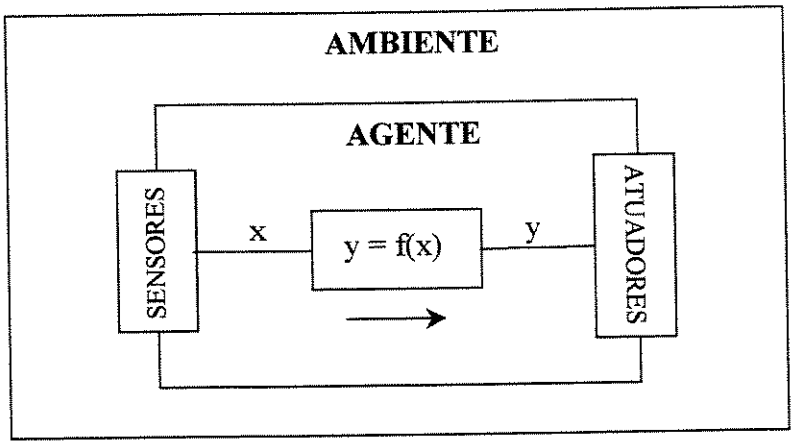


Figura 3.2 Exemplo de um agente reativo ou reflexivo.

- **Agentes Comportamentais**: apresentam processos independentes de percepção e ação. Neste caso, a percepção é um processo independente, controlado pelo módulo de percepção. Contém um conjunto pré-definido de comportamentos, que são selecionados dependendo da percepção. O cumprimento de suas tarefas depende de uma especificação adequada do conjunto de comportamentos disponíveis. (figura 3.3).

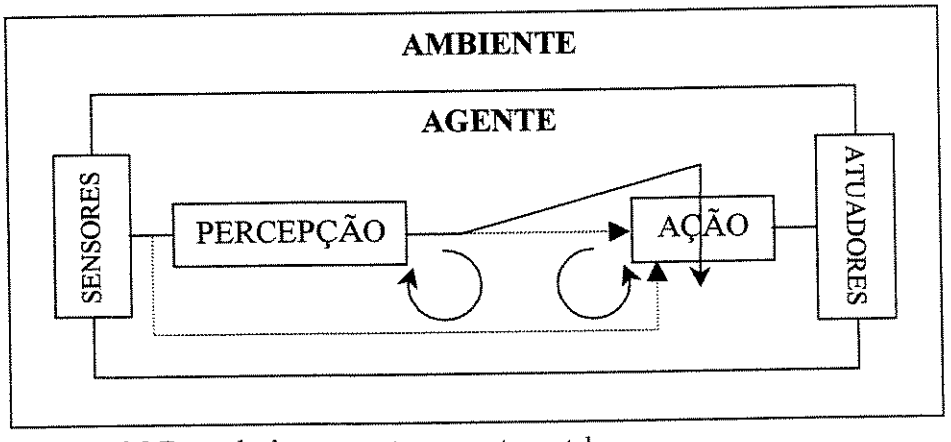


Figura 3.3 Exemplo de um agente comportamental.

- **Agentes Planejadores**: apresentam processos independentes de percepção e ação, possuem internamente uma representação simbólica do mundo (modelo do mundo, que pode ser conhecido "a priori" ou pode ser aprendido por meio da interação com o ambiente, através dos mecanismos de percepção). A ação não é gerada diretamente pela percepção, devido à existência de um mecanismo de geração de comportamentos, capaz de elaborar predições (usando o modelo do mundo) de como o mundo se comportará diante de possíveis ações tomadas pelo agente, além de um mecanismo de geração de planos (conjunto de ações a serem tomadas pelo agente) que permitirão ao agente escolher o plano que melhor satisfizer os objetivos do sistema e executá-lo efetivamente. (figura 3.4).

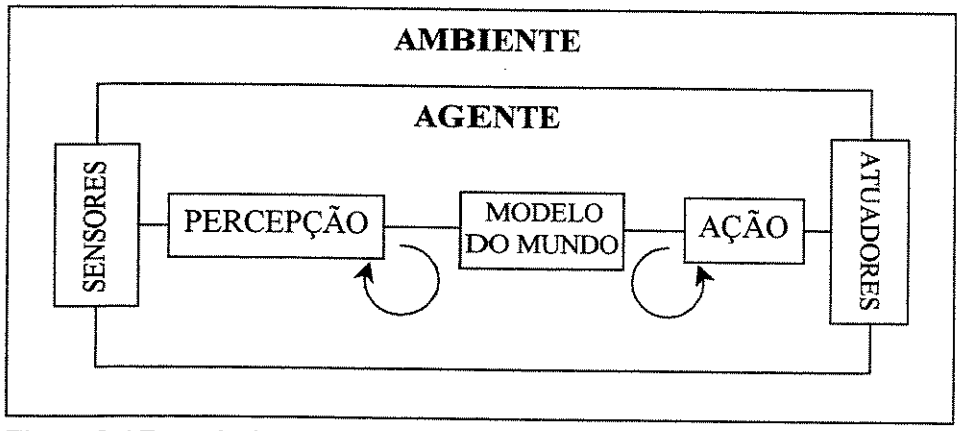


Figura 3.4 Exemplo de um agente planejador.

- **Agentes Emocionais**: muito parecido com os agentes planejadores, só acrescido de um sistema de valores (emoções - por exemplo: medo, desejo, dor, alegria) que permitirá avaliar internamente se os objetivos do agente estão sendo cumpridos. Também podem ser utilizadas de modo a influir no planejamento de futuras ações, além de que podem ser atribuídas a estados atuais ou a previsões de estados futuros. (figura 3.5).

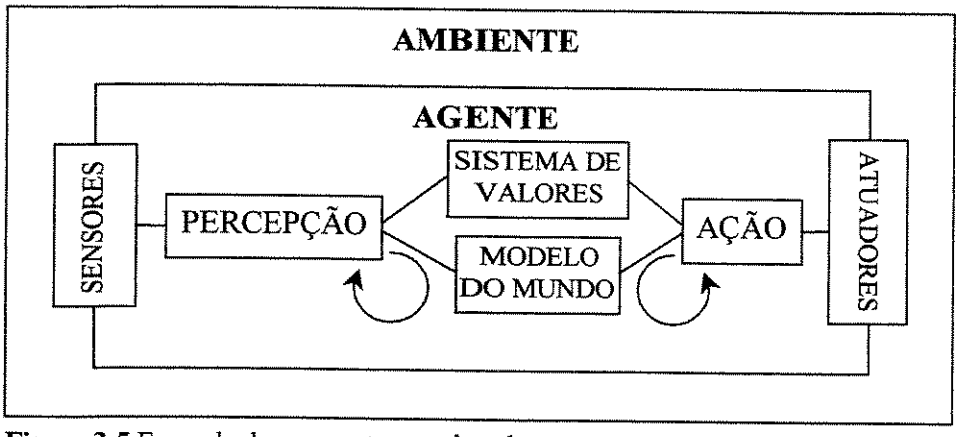


Figura 3.5 Exemplo de um agente emocional.

- **Agentes Comunicativos**: possuem um canal de comunicação direta. Somente há sentido em se falar deste tipo de agentes em sistemas multi-agentes. Precisam de uma linguagem de comunicação de agente (ACL, *Agent Comunication Language*) para a colaboração ou cooperação entre agentes. (figura 3.6).

- **Agentes Semióticos**: parecidos com os agentes comunicativos, mas não precisam de canais de comunicação direta, visto que o próprio ambiente é o canal de comunicação que será usado por eles para a colaboração ou cooperação entre si. Este tipo de agente precisa de mecanismos de percepção e ação muito mais sofisticados que nos casos anteriores, pois eles terão que interpretar e gerar signos para o ambiente. (figura 3.7).

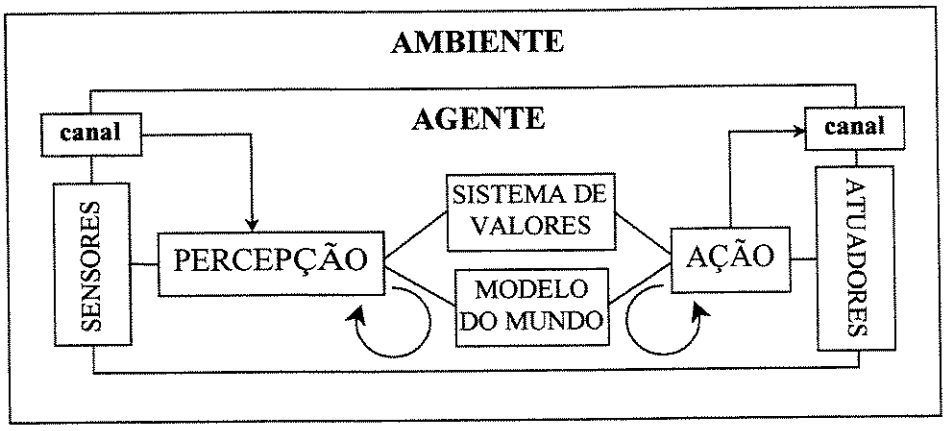


Figura 3.6 Exemplo de um agente comunicativo.

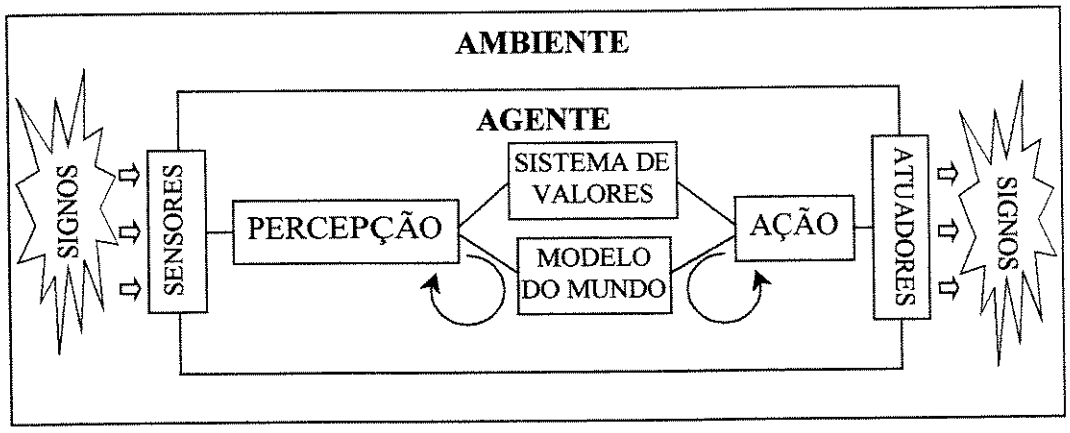


Figura 3.7 Exemplo de um agente semiótico.

3.5. Linguagens de agentes

Com o desenvolvimento da tecnologia de agentes, uma grande variedade de ferramentas de software encontram-se disponíveis para o *design* e construção de sistemas baseados em agentes. O número emergente de protótipos de linguagens de agentes é um sinal de que a tecnologia de agentes está sendo amplamente utilizada e que muitas aplicações baseadas em agentes estão sendo desenvolvidas.

Wooldridge e Jennings [Wooldridge 95] escreveram a respeito:

> "por uma linguagem de agente, entendemos um sistema que nos permita programar *hardware* ou *software* de sistemas de computação em termos de alguns dos conceitos desenvolvidos pelos teóricos de agentes. Como mínimo, esperamos que tal linguagem inclua alguma estrutura correspondente a um agente"

Como a questão "*O que é um agente?*" é muito polêmica e não existe um consenso (e possivelmente nunca exista) com respeito a esta questão, alguns pesquisadores consideram uma linguagem como linguagem de agentes e outros podem não considerá-la da mesma

maneira. O que é certo, é que elas prestam-se em diferentes graus para diferentes tipos de definições e aplicações de agentes.

Tipo(s) de agente	Classe de linguagem	Exemplo(s)
Agentes colaborativos	Linguagens Actor	Actors
	Linguagens de programação orientada a agentes	Agent-0 PLACA
Agentes de interface Agentes de Informação Agentes móveis	Linguagens Scripting	TCL/Tk Safe-TCL, Safe-Tk
		Java
		TELESCRIPT Active web tools
		Python
		Obliq
		APRIL
Agentes reativos	Linguagens reativas	RTA/ABLE

Tabela 3.1 Algumas linguagens de desenvolvimento de diferentes aplicações baseadas em agentes.[Nwana 96d]

A tabela 3.1 apresenta algumas linguagens para o desenvolvimento de diferentes aplicações baseadas em agentes. Nesta tabela não são apresentadas linguagens apropriadas para o desenvolvimento de sistemas multi-agentes colaborativos, visto que em geral, os agentes colaborativos são muito mais complexos que os agentes de interface, de informação ou móveis. Além de linguagens, eles precisam de arquiteturas ou plataformas para sua construção. Como alternativa eles podem ser construídos completamente usando alguma linguagem de terceira geração, como é o caso do C++, Java, Smalltalk ou Prolog.

Até agora temos utilizado a terminologia "linguagem de agentes" para referenciar a linguagem utilizada na criação dos agentes. É importante destacar que existe um outro enfoque para esta terminologia, onde utiliza-se "linguagem de agentes" para referenciar a linguagem que dois ou mais agentes utilizam para comunicar-se entre si, quando interagindo em um ambiente comum. Esta terminologia é usual dentro do contexto de sistemas multi-agentes, sendo que as "linguagens de agentes" são utilizadas para o intercâmbio de conhecimento a ser compartilhado entre os agentes que cooperam e/ou colaboram em sistemas deste tipo. Dois tipos de linguagens de agentes são destacados na literatura: as chamadas Linguagem de Comunicação de Agentes (ACL[1])[Nwana 96d, Singh 98, Labrou 99] e as chamadas Linguagens de Conteúdo (CL[2])[FIPA 99].

As ACL são utilizadas para deixar explícito o ato comunicativo relacionado à mensagem, ou seja, destaca-se o ato comunicativo pretendido pelo agente quando de sua comunicação com seu interlocutor. Alguns exemplos desta linguagem são o KQML(*Knowledge Query and Manipulation Language*) e o FIPA-ACL(*Foundation for*

[1] ACL – Agents Comunication Language
[2] CL – Content Language

Intelligent Physical Agents - Agents Comunication Language). As CL, por sua vez, são utilizadas para expressar o conhecimento que se deseja compartilhar com o destinatário da mensagem. Exemplos de linguagens deste tipo são o KIF(*Knowledge Interchange Format*), FIPA-CLL(*FIPA Content Language Library*), FIPA-SL(*FIPA Semantic Language*), FIPA-RDF (*FIPA Resource Description Framework*), FIPA-CCL(*FIPA Constraint Choice Language*) e o FIPA-KIF(*FIPA Knowledge Interchange Format*).

3.6. Objetos e Agentes

A engenharia de software baseada em agente é com freqüência comparada à programação orientada a objetos, em que os agentes, assim como os objetos, compartilham algumas propriedades tais como: encapsulamento, herança (com certa freqüência) e fornecem uma interface baseada em mensagens para suas estruturas de dados internas e seus métodos (ou algoritmos). Embora exista uma distinção importante: na programação orientada a objetos, o significado de uma mensagem pode ser diferente de um objeto para outro (princípio do polimorfismo). Na engenharia de software baseada em agentes, os agentes utilizam uma linguagem comum, que tem uma semântica independente dos agentes, ou seja, os agentes devem ter uma linguagem de comunicação (ACL) comum de modo que todos possam se entender.

Outra diferença entre objetos e agentes é que um objeto tem uma postura passiva diante do mundo. Ou seja, um objeto é uma entidade do mundo que somente recebe mensagens, efetuando um comportamento em resposta a elas. Por outro lado, um agente tem uma postura ativa, ou seja, é uma entidade do mundo que possui um ciclo de vida e que durante esse ciclo de vida estará adquirindo continuamente informações do mundo, através da busca ativa por mensagens que se encontrem no ambiente em que está inserido.

As semelhanças que tem os objetos e os agentes, possibilitam que algumas linguagens orientadas a objetos, tais como, Smalltalk, C++ ou Java se prestem para a construção de sistemas de agentes.

3.7. Rede de Agentes

Uma rede de agentes é um tipo especial de rede de objetos, na qual são colocadas restrições concernentes à definição da função de seleção. O que caracteriza uma rede de agentes frente a uma rede de objetos é que, ao contrário da rede de objetos, que não institui nenhuma política para a função de seleção, na rede de agentes temos uma política de função de seleção única e distribuída ao longo da rede. Esta política padroniza o mecanismo de seleção, exigindo do usuário somente uma função de utilidade que discrimine entre os objetos disponíveis para consumo, qual seria o mais útil.

Para ilustrar a diferença entre uma rede de agentes e uma rede de objetos, daremos um exemplo a seguir onde pode-se perceber esta diferença. Imagine-se que queremos modelar o comportamento na tomada de decisão em uma faculdade para dois departamentos (**D1** e

D2) quanto ao uso de suas verbas. **VE1** representa a verba específica do departamento **D1**, **VE2** a verba específica do departamento **D2** e **VG** a verba global disponível da faculdade para a compra de computadores. Suponha que o departamento **D1** já tenha utilizado toda sua verba específica, possuindo agora somente disponível a verba global da faculdade, que tanto pode ser utilizada para um departamento como para o outro, mas não por ambos. Como seria simulado este problema ?

Para que uma rede de objetos simule de forma correta esta situação faz-se necessária uma política de seleção, que permita decidir como serão consumidas estas verbas pelos departamentos na compra dos computadores. A definição desta política é necessária, pois diferentes políticas levarão a diferentes estados finais para o sistema. Caso a verba global **VG** seja alocada para **D1**, tanto **D1** quanto **D2** poderão comprar computadores. Caso ela seja alocada para **D2**, somente **D2** comprará novos computadores. Portanto, poderia parecer mais justo que a verba fosse alocada para **D1**. Entretanto, suponha que **D1** seja um departamento mal administrado, e já tenha gasto sua verba específica de maneira pouco planejada. Neste caso, seria inadequado privilegiar um departamento nestas condições, favorecendo o imediatismo na tomada de decisões. Uma das formas possíveis de resolver esse dilema é avaliar as produções científicas (**PC1**, produção científica do departamento **D1** e **PC2**, produção científica do departamento **D2**) de cada departamento e tomar este valor como critério de seleção na hora de decidir qual departamento utilizaria a verba global da faculdade.

A figura 3.8 mostra esse problema sendo resolvido por uma rede de objetos e por uma rede de agentes. No caso (a), o critério de seleção não aparece explicitamente na estrutura da rede, exigindo um algoritmo externo que viabilize sua implementação. Na rede de agentes (b), não há a necessidade de um algoritmo externo para a função de seleção, visto que existe um mecanismo que determina a função de seleção de forma única para qualquer problema a ser modelado. As produções científicas aparecem então como parâmetros do modelo, e nenhum critério de decisão externo é necessário.

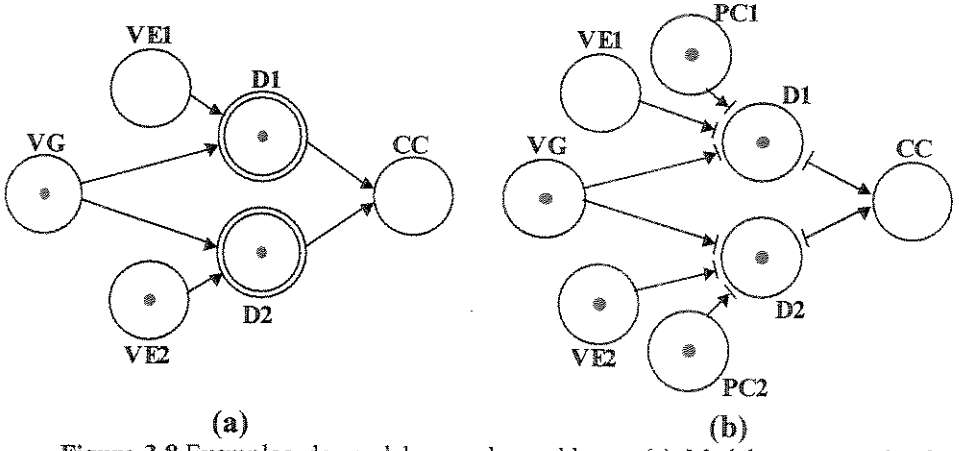


Figura 3.8 Exemplos de modelagem do problema. (a) Modelagem através de rede de objetos. (b) Modelagem através de rede de agentes.

Esse exemplo ilustra as principais características de ambos os modelos. Na rede de objetos, têm-se maior flexibilidade, permitindo-se que a função de seleção seja qualquer. Entretanto, o preço a se pagar é a possível ausência de elementos importantes (o critério de

decisão) colocados explicitamente no modelo desenvolvido, exigindo mecanismos complementares. Na rede de agentes, todas as informações importantes devem ser incluídas e explicitadas. A vantagem é que nenhum mecanismo adicional necessita ser incorporado, ou seja, a rede é auto-contida em seu funcionamento. Vale sempre a pena relembrar que uma rede de agentes é uma especialização de uma rede de objetos, ou seja, ela não deixa de ser uma rede de objetos. Uma rede de agentes pode, entretanto, ser automatizada. Ou seja, é possível a criação de uma ferramenta automática de simulação para redes de agentes, o que nem sempre é possível para qualquer rede de objetos.

3.7.1. Rede de Agentes Formal

DEFINIÇÃO 3.1 – REDE DE AGENTES

Seja $\Re = (\Sigma, \Pi, \gamma, \Xi, A, \eta, fpi, fpo, C, \xi)$ uma rede de objetos. Define-se uma rede de agentes $\overline{\Re}$ como a rede de objetos $\Re$, onde o conjunto de funções de seleção γ, além de cumprir as restrições impostas pela própria definição de rede de objetos (definição 2.38) é construída da seguinte forma:

$$\gamma(c,n) = (H_k(c,n,\theta_k), S_k(c,n,\theta_k), \theta_k)$$

onde:

- $k = \arg \max_{i \ s.a.\ restrições} (g(H_i(c,n,\theta_i),\theta_i))$, onde θ_i é o índice de uma possível função de transformação interna para o objeto c.
- $g : H_i \times \Theta_i \rightarrow R$ é uma função de avaliação que determina um "grau de interesse" para um possível escopo habilitante. Esta função expressa a "utilidade" ou "desejo" de se utilizar um determinado escopo habilitante no disparo do objeto c no instante n.
- as restrições referenciadas acima permitem a escolha de um escopo habilitante (dentre os possíveis escopos habilitantes) que não esteja em conflito com os escopos habilitantes escolhidos por outros objetos da rede.

As diferentes restrições ou especificações impostas aos sistemas de objetos permitem realizar uma classificação quanto as classes dos mesmos. A figura 3.9 mostra uma caracterização das classes dos sistemas de objetos.

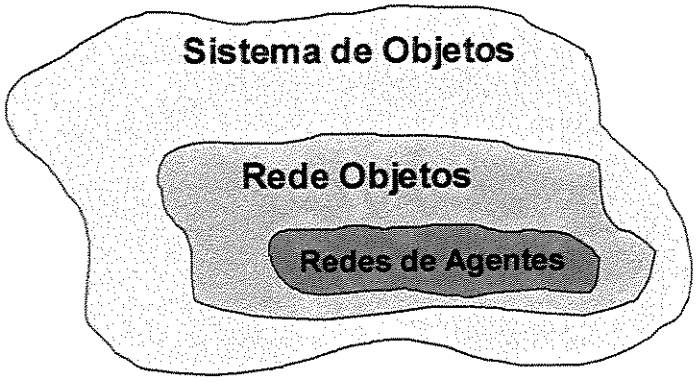


Figura 3.9 Caracterização das classes dos Sistemas de Objetos.

3.7.2. "Matching": um mecanismo de seleção para as rede de agentes.

Como observado na definição formal, é necessário colocar um conjunto de restrições para que a escolha de um determinado escopo habilitante por um objeto não entre em conflito com a escolha de outros objetos. Nesta seção, especificaremos um possível algoritmo que cria estas restrições. Este algoritmo é chamado de algoritmo de *matching*, que utiliza a função de avaliação g para determinar a função de seleção para uma rede de agentes. Para tal, alguns conceitos adicionais serão introduzidos.

3.7.2.1. Portas

O conceito de **porta** está associado ao conceito de **lugar**. Portas são os meios por onde objetos podem entrar ou sair de lugares de uma rede. Podem portanto ser consideradas como sendo interfaces de comunicação entre os lugares na rede de objetos. Estas portas são classificadas em diferentes tipos, segundo a topologia da rede (figura 3.10) e segundo o uso que se faz delas (figura 3.11).

- Segundo a topologia:

  - ***Portas de entrada:*** quando a porta é utilizada para a entrada de objetos em um lugar, seja para armazenamento, seja para processamento
  - ***Portas de saída:*** quando a porta é utilizada para a saída de objetos de um lugar, seja para um futuro processamento, seja para um futuro armazenamento.

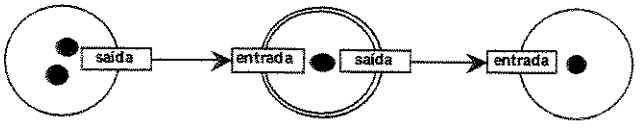


Figura 3.10 Classificação das portas segundo a topologia da rede de agentes.

Observando a figura 3.10, verificamos que as portas de entrada e saída podem ter usos bem diferenciados. No primeiro caso, um objeto armazenado sai por uma porta de saída para ser processado. Ele então entra por uma porta de entrada, em um lugar onde não permanecerá, mas onde somente será processado. Após o processamento, outro (ou o mesmo) objeto sai por uma porta de saída deste lugar onde não permaneceu, para então adentrar uma porta de entrada de um lugar onde será armazenado. Temos portanto duas condições, uma de armazenamento e outra de processamento. Estas condições demandam uma classificação adicional para as portas:

- Segundo o uso que se faz das portas:

  - ***Portas privadas:*** são portas por onde entram e saem objetos que serão processados por objetos de um determinado lugar. Quando são portas de entrada, são usadas para alimentar os objetos ativos de algum lugar ativo, de forma a permitir seus disparos. Desta forma, cada porta de entrada privada de um lugar ativo estará relacionada a

um possível parâmetro de uma função interna dos objetos que habitam este lugar. Caso sejam portas de saída, são utilizadas para enviar objetos gerados ou transformados para outros lugares. As portas privadas estão relacionadas portanto ao domínio ou contradomínio das funções de transformação, de forma que existe uma associação bi-unívoca entre as portas privadas e à classe de objetos que habitam um determinado lugar. É importante observar que portas privadas somente podem ser associadas a classes ativas e por sua vez aos lugares ativos. Um objeto que entra ou sai por uma porta privada de um determinado lugar nunca permanecerá no referido lugar.

- ***Portas públicas:*** são portas por onde entram e saem objetos de lugares onde estes são armazenados. Caso sejam portas de entrada, os objetos são introduzidos no lugar. Caso sejam portas de saída, os objetos são retirados do lugar. Como não serão processados, não existe nenhum vínculo entre as portas públicas e as classes dos objetos que habitam o lugar.

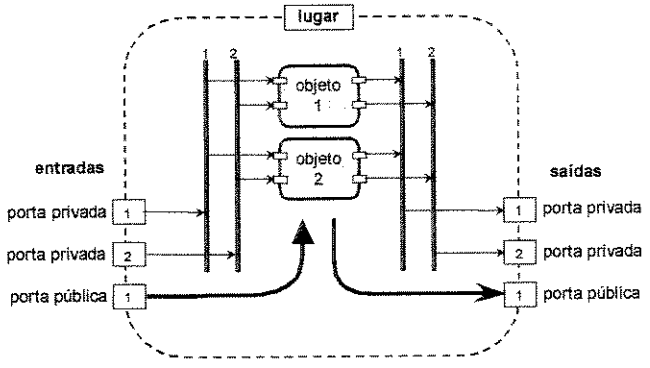


Figura 3.11 Classificação das portas segundo a semântica envolvida em elas.

O número de portas privadas num lugar vai depender da classe associada a este lugar, especificamente do número de funções de transformação e seus respectivos domínios e/ou contradomínios. O número de portas públicas num lugar pode ser qualquer.

O conceito de porta privada está associado às definições de interface de entrada (definição 2.27) e interface de saída (definição 2.29) estudadas no capítulo anterior.

3.7.2.2. Arcos

Os arcos (figura 3.12) são utilizados para criar enlaces entre os lugares, conectando portas de entradas com portas de saída e vice-versa. Em princípio, os arcos são independentes dos tipos de portas que conectam, no sentido de seu uso, mas existem algumas combinações que podem ser consideradas como inválidas:

- ***pública – pública***, (figura 3.13a) dado que os objetos que existem nestes lugares são passivos (não conténs funções de transformação a ser executadas), este tipo de enlace não tem sentido de existir. Este tipo apresentaria um comportamento de um canal sem atividade. A forma correta para fazer isto é usando um lugar ativo

intermediário que pegue o objeto do lugar de origem e coloque-o no lugar de destino (figura. 3.13b).

- ***privada – privada***, porque implica uma condição de competição entre o objeto cliente e o objeto servidor.

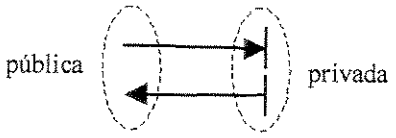


Figura 3.12 Notação gráfica usada para representar os arcos.

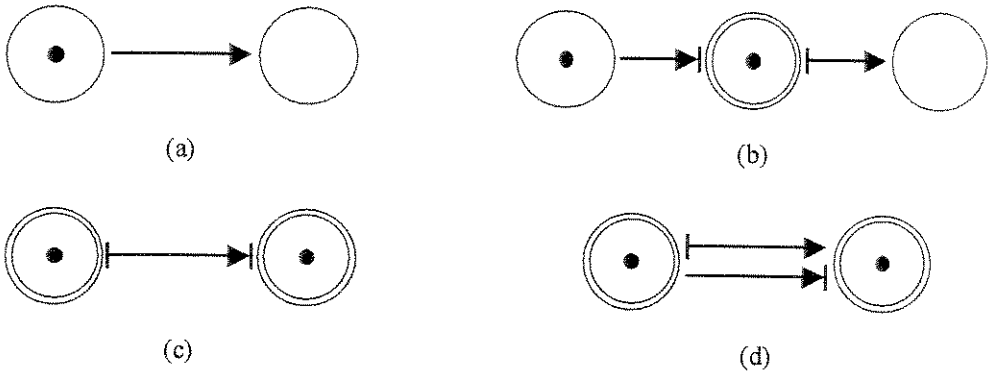


Figura 3.13 Exemplos do uso dos arcos. (a) Conexão *pública – pública*, incorreta. (b) Solução para um enlace do tipo *pública – pública*. (c) Conexão *privada – privada*, incorreto. (d) Solução para um enlace do tipo *privada – privada*.

Podemos dizer que os lugares ativos, por sua própria definição, têm pelo menos uma porta privada, podendo ou não ter portas públicas. Lugares passivos, por sua vez, devem ter pelo menos uma porta pública e nenhuma porta privada. Por este motivo podemos prescindir de utilizar os círculos duplos para representar graficamente os lugares ativos, assim os lugares passivos serão aqueles que só contém portas públicas e os lugares ativos aqueles que contém pelo menos uma porta privada. (ver figura 3.14)

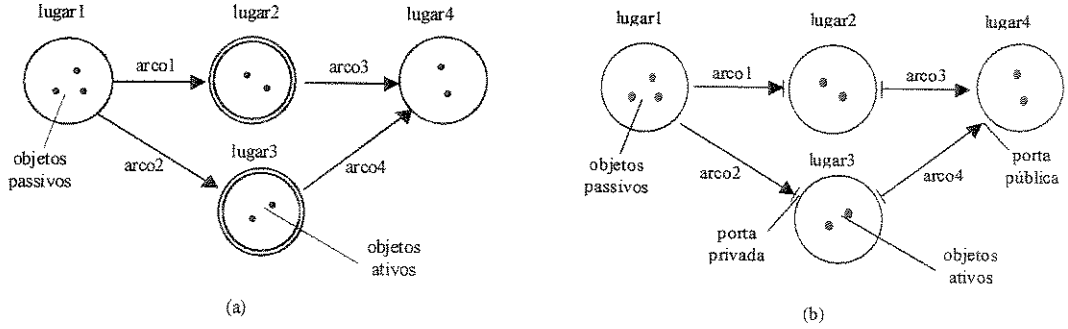


Figura 3.14 Notação gráfica utilizada. (a) Rede de Objetos. (b) Rede de Agentes.

3.7.2.3. Modo de acesso

O modo de acesso aos objetos é um conceito muito importante e necessário na dinâmica das redes de agentes. Este conceito caracteriza as regras do uso dos objetos a serem assimilados no processo de disparo dos objetos ativos, assim como as conseqüências destas quando um objeto é usado por outros objetos.

Para entender melhor a necessidade de incluir o modo de acesso nas rede de agentes, consideremos o seguinte exemplo ilustrativo (ver figura 3.15): Tem-se uma base de dados (**DataBase**) a ser consultada por duas aplicações (**Solver1** e **Solver2**), que a utilizam de forma indexada segundo suas chaves (**Key1** e **Key2** respectivamente). Estas aplicações utilizam os mesmos dispositivos para receber as consultas (**Query**) e enviar as respostas das consultas (**Answer**). Uma consulta só pode ser processada por uma aplicação (**Solver1** OU **Solver2**), que é determinada pelo campo chave pela qual as aplicações tem indexada a base de dados. Ou seja, a consulta será assimilada de forma exclusiva por uma aplicação. Por outro lado as chaves podem ser acessadas de qualquer forma pois elas não são informações compartilhadas entre as aplicações, mas neste exemplo não é interessante consumir as chaves de maneira destrutiva pois elas permitem à aplicação avaliar as consultas e ter indexada a base de dados para seu uso. Para acessar a base de dados (recurso compartilhado pelas aplicações), é interessante destacar que ela só pode ser acessada de forma compartilhada e não assimilada de forma destrutiva pelas aplicações. A partir deste exemplo podemos concluir que existe a necessidade de especificarmos diferentes modos de acesso aos objetos a serem assimilados e que estes modos devem estar relacionados a duas dimensões. A primeira delas é referente à exclusividade de uso de um determinado objeto (ou seja, se os objetos podem ou não participar do escopo habilitante de múltiplos objetos ativos). A segunda diz respeito à possível destruição dos objetos assimilados, após seu uso (ou seja, determina se os objetos devem ou não ser destruídos após a assimilação).

Os possíveis modos de acesso que os objetos podem ter são:

- ***compartilhável + não consumível***, diferentes objetos podem ler este objeto, mas não podem modificar o estado dele. O objeto permanece no mesmo lugar.
- ***compartilhável + consumível***, diferentes objetos podem ler este objeto, mas não podem modificar seu estado. Este objeto é automaticamente eliminado após sua leitura.
- ***não compartilhável + não consumível***, apenas um objeto pode ler este objeto, sem modificá-lo. Este objeto não pode ser consumido, ou seja, ele vai permanecer no lugar onde ele se encontra.
- ***não compartilhável + consumível***, apenas um objeto pode acessar este objeto, alterando ou não seu estado. Este objeto é destruído automaticamente pelo objeto que o utilizou. No entanto este pode regenerá-lo em outro lugar da rede.

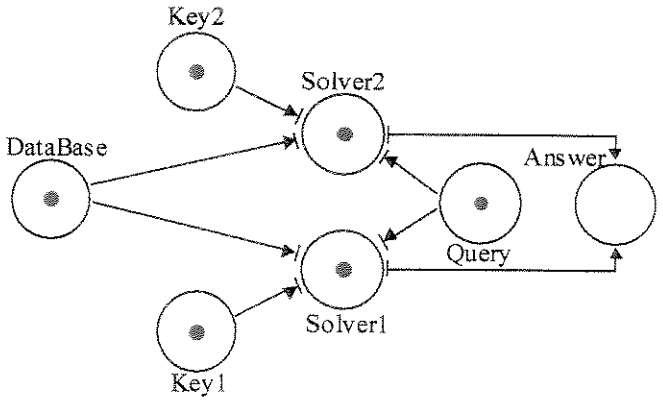


Figura 3.15 Rede de agentes exemplo para simular o acesso a uma base de dados por duas aplicações rodando de forma concorrente.

Seguindo estas idéias sobre a forma de acessar os objetos pode-se colocar a seguinte definição para o modo de acesso aos diferentes objetos a serem assimilados.

DEFINIÇÃO 3.2 – MODO DE ACESSO

O **modo de acesso** de um objeto c da classe C no instante n, $n \in \mathrm{N}$, será uma ênupla $M_c = (s,l)$, $s \in \{0,1\}$ é um valor indicando se o objeto c é $(s=1)$ ou não $(s=0)$ compartilhado e $l \in \{0,1\}$ é um valor indicando se o objeto c deve $(l=1)$ ou não $(l=0)$ ser consumido.

Exemplo: $M_c = (0,0)$, indica que o objeto c não é compartilhado e não pode ser consumido.
$M_c = (0,1)$, indica que o objeto c não é compartilhado e pode ser consumido.
$M_c = (1,0)$, indica que o objeto c é compartilhado e não pode ser consumido.
$M_c = (1,1)$, indica que o objeto c é compartilhado e pode ser consumido.

Como conseqüência do modo de acesso aos objetos, pode-se analisar como eles interagem entre si, ou seja, como um objeto ativo faz referência aos outros objetos que ele deseja utilizar. A figura 3.16 apresenta as diferentes possibilidades. No caso da figura 3.16(a) o objeto A é um objeto que é não-compartilhado e não pode ser consumido. Neste caso, apenas um dos objetos B ou C pode ler o objeto A. Assumamos que B seja o objeto que lê A, então B no campo de entrada terá uma referência ao objeto A. Lembre-se que B não pode modificar A, portanto o objeto A não passa a ser parte do objeto B. Na figura 3.16(b) temos o caso para o qual o objeto A é não-compartilhável e pode ser consumido. Neste caso assume-se também que B é o objeto que vai acessá-lo. Para este caso B acessa o objeto A, convertendo-o em parte dele, ou seja, B torna-se dono de A, sendo possível a modificação do estado de A através de alguma função de transformação de B. Os últimos casos são ilustrados na figura 3.16(c), e correspondem ao fato do objeto

A ser compartilhado para ser acessado por vários objetos. Nestas condições, ele pode ser usado por diferentes objetos concorrentemente. Desta forma ele não pode ser modificado por outro objeto. Portanto os objetos *B* e *C* nos seus respectivos campo de entrada terão uma referência ao objeto *A*. No caso em que o objeto *A* seja consumível, ele será automaticamente destruído após seu uso pelos objetos que tem interesse em sua utilização.

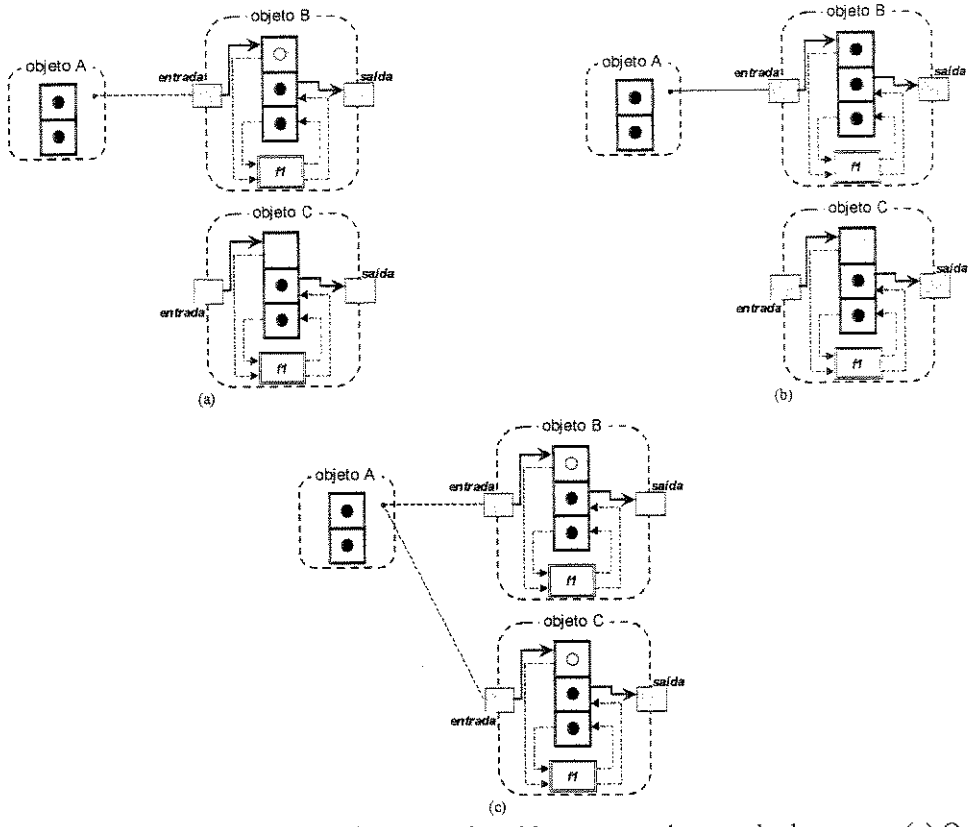


Figura 3.16 Exemplo de interação dos objetos, segundo o modo de acesso. (a) O objeto A é não-compartilhável e não-consumível. (b) O objeto A é não-compartilhável e consumível. (c) O objeto A é compartilhável, e pode ou não ser consumível.

3.7.2.4. Grau de interesse

Sempre que um objeto ativo for disparado, ele poderá ter à sua disposição mais de um objeto como possível escopo habilitante, mas só poderá utilizar um deles. Na maioria dos casos não importa qual destes objetos será assimilado, mas existem situações onde há um interesse específico em que algum(s) objeto(s) seja(m) assimilado(s) com um nível de preferência maior que os outros. Como as redes de agentes serão utilizadas no contexto da análise, *design* e modelagem de sistemas, particularmente sistemas inteligentes, é preciso que elas possuam um alto grau de generalidade. Por esse motivo, faz-se necessária a definição do **grau de interesse** que determinará o interesse (desejo, necessidade, possibilidade) de um objeto ativo em assimilar (ou consumir) um objeto ou um conjunto de objetos.

DEFINIÇÃO 3.4 – GRAU DE INTERESSE

O **grau de interesse** $\lambda : q \times \mathrm{N} \rightarrow R$ é uma função que expressa o "interesse" que uma determinada função de transformação f tem pela combinação de objetos q a serem assimilados no disparo da função f no instante n, $n \in \mathrm{N}$.

Este grau de interesse será determinado pelo projetista da rede de agentes. O mesmo pode ser expresso mediante um preço, um valor de *fitness*, ou uma solução mais elaborada. Por exemplo, o uso de medidas de necessidade e possibilidade.

3.7.2.5. Combinações

Um aspecto importante para descrever o algoritmo de *matching* são as definições dos diferentes tipos de combinações (combinação, combinação local e combinação global ou válida) que serão utilizados para especificar este algoritmo. Primeiramente será definido o conceito de combinação, que não será utilizado diretamente na descrição do algoritmo, mas é a base para a definição dos outros tipos de combinações que serão utilizados no algoritmo a ser descrito posteriormente.

DEFINIÇÃO 3.3 – COMBINAÇÃO

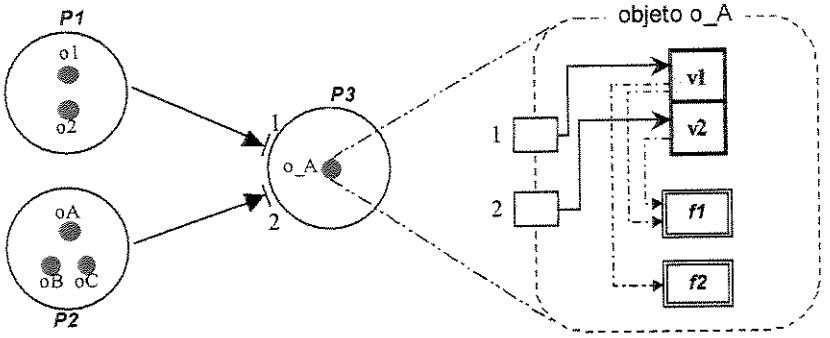
Sejam:

- c um objeto ativo da classe C.
- f uma função de transformação da classe C.
- $P = \{P_k\}$, $k \in \mathrm{N}$, o conjunto dos lugares associados aos possíveis objetos de entrada que satisfazem os requerimentos de tipo para a função de transformação f.
- $L_k = \{l_{ik}\}$, $i \in \mathrm{N}$, $k \in \mathrm{N}$, o conjunto de objetos no lugar P_k.
- $Q = \underset{j \in P}{\times} L_j$

Define-se uma **combinação** q para a função f do objeto ativo c, como sendo qualquer um dos elementos do conjunto Q.

Para cada função interna de transformação de um objeto ativo pode existir uma combinação, a não ser que alguns de seus requisitos não contenham elementos. A figura 3.17 apresenta as possíveis combinações para as funções internas de transformação ***f1*** e ***f2***. Caso o lugar **P2** não contenha nenhum elemento, então não existe nenhuma combinação para a função de transformação ***f1***.

Uma vez definido o conceito de **combinação**, pode-se definir o que será denominado **combinação local** e que será de extrema importância para o algoritmo de *matching*.



combinações para *f1*: { (o1,oA), (o1,oB), (o1,oC),
(o2,oA), (o2,oB), (o2,oC) }
combinações para *f2*: { (o1), (o2) }

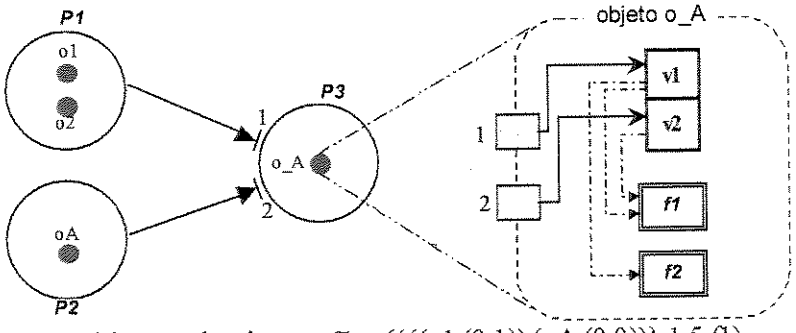
Figura 3.17 Exemplo de combinações.

DEFINIÇÃO 3.5 – COMBINAÇÃO LOCAL

Sejam:

- c um objeto ativo da classe C.
- f uma função de transformação da classe C.
- $q = (q_{i1}, q_{i2}, \ldots, q_{ik})$ a i-ésima combinação para a função de transformação f do objeto ativo c, $1 \le i \le n$, $n \in \mathrm{N}$, $n > 0$.
- λ o grau de interesse da função de transformação f pela combinação q.
- $(m_1, m_2, \ldots, m_n)$ uma ênupla, onde cada elemento m_j é o modo de acesso para cada j-ésimo objeto da combinação q, $1 \le j \le n$, $n \in \mathrm{N}$, $n > 0$.

Define-se uma **combinação local** $\overline{q}$ para a função de transformação f do objeto ativo c como uma tripla $\overline{q} = (\overline{q}_1, \overline{q}_2, \overline{q}_3)$, onde: $\overline{q}_1 = \{(q_{i1}, m_1), (q_{i2}, m_2) \ldots, (q_{in}, m_n)\}$, $\overline{q}_2 = \lambda$, $\overline{q}_3 = f$.



combinações locais para *f1*: {({(o1,(0,1)),(oA,(0,0))},1.5,*f*1),
({(o2,(1,1)),(oA,(1,0))}, 1.0,*f*1)}
combinações locais para *f2*: {({(o1, (0,0))},1.3, *f*2),
({(o2, (0,1))},1.5, *f*2) }

Figura 3.18 Exemplo de combinação local.

A figura 3.18 mostra um exemplo das possíveis combinações locais para cada uma das funções de transformação (*f1* e *f2*) definidas na classe do objeto que encontra-se no lugar **P3**. No caso da combinação local ocorre o mesmo que na definição de combinação. Já o caso em que o lugar **P2** não contenha nenhum elemento, a combinação local para a função de transformação *f1* não existirá.

Graficamente podemos representar uma combinação local como mostrado na figura 3.19. Na representação gráfica, temos o chamado "objeto proprietário" (obj_p) da combinação, o nome da função de transformação (NF) a ser disparada, o grau de interesse (GI) do objeto proprietário pelo disparo desta função com respeito ao conjunto de objetos de entrada (obj_e), colocando para cada um destes objetos de entrada o modo de acesso (e-não compartilhável (exclusivo), c-consumível) que será utilizado na assimilação dos mesmos pelo objeto proprietário, disparando a função de transformação especificada na combinação local.

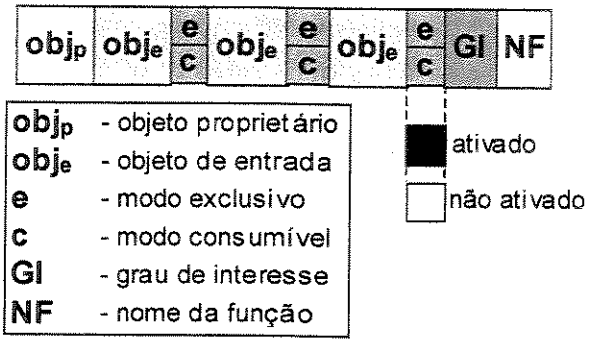


Figura 3.19 Representação gráfica de uma combinação local

A figura 3.20 apresenta um exemplo da representação gráfica das possíveis combinações locais que existem no exemplo da figura 3.18.

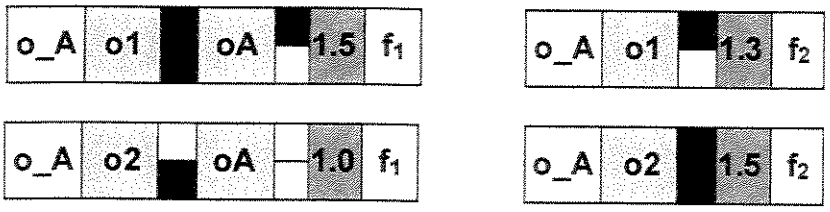


Figura 3.20 Exemplo da representação gráfica de combinações locais

DEFINIÇÃO 3.6 – COMBINAÇÃO VÁLIDA

Uma combinação válida não é nada mais que uma combinação local que tem que obedecer às seguintes restrições:

- A combinação deve ser válida para toda a rede no instante n, $n \in N$
- A combinação não pode estar em conflito com outras combinações válidas:

- se algum elemento desta combinação local é utilizado exclusivamente, então ele não pode ser parte de qualquer outra combinação válida;
- se algum elemento desta combinação local é compartilhável, então ele não pode pertencer a outra combinação que a utilize de forma exclusiva.

3.7.2.6. Algoritmo de matching

O algoritmo de ***matching*** consiste em um mecanismo que permite encontrar todas as combinações locais para cada uma das funções de transformação das classes definidas na rede de objetos. Ou seja, ele permite realizar uma avaliação prévia dos objetos disponíveis na rede (a serem assimilados pelos objetos ativos), gerando o "desejo" de um objeto ativo em consumir um dos objetos ou um conjunto de objetos que sejam parte de sua interface de entrada. Deste modo podemos dizer que este processo é uma possível definição para a função g. Além disso o algoritmo é responsável por obter as combinações válidas para a rede de agentes para um instante n, $n \in \mathrm{N}$.

Um dos possíveis algoritmos de *matching* é o algoritmo BMSA (*Best Matching Search Algorithm*), desenvolvido pelo autor e descrito a seguir:

1. Determinar o conjunto de combinações locais para cada objeto ativo. Ou seja, para cada lugar ativo, inspeciona-se os objetos nele contidos, gerando-se todas as possíveis combinações locais referentes a cada objeto.
2. Agrupar as combinações locais de cada lugar em um conjunto global.
3. Criar uma lista que contém todos os objetos que serão assimilados, destrutivamente ou não, sem repetição (em outras palavras, cria-se uma lista que é a união de todos os objetos usados nas combinações locais obtidas no passo 1).
4. Enquanto *o número de objetos a serem assimilados for* $\neq 0$ e *o número de combinações locais for* $\neq 0$ fazer:
   - Pegar da lista de combinações locais aquela que tiver maior valor no grau de interesse e mover essa combinação para a lista de combinações válidas. Caso haja duas ou mais combinações com o mesmo grau de interesse, então uma delas é escolhida aleatoriamente.
   - Eliminar todas as combinações locais onde o objeto proprietário da combinação local escolhida apareça como sendo assimilado e/ou proprietário.
   - Para cada um dos objetos sendo assimilados na combinação local escolhida fazer:
     - Se o modo de acesso deste objeto for *não-compartilhável* - então remover da lista de combinações locais todas aquelas que contenham o dito objeto e remover este objeto da lista dos objetos que serão assimilados.
     - Se o modo de acesso for *compartilhável* - então remover da lista de combinações locais todas aquelas que usam este objeto de modo *exclusivo* ou *não compartilhável*.

- Se o modo de acesso for *consumível* - então colocar uma marca especial no respectivo objeto, para indicar que o objeto deve ser destruído (esta marca será utilizada posteriormente no algoritmo de simulação da rede).

3.7.2.7. Exemplo de execução do algoritmo de matching BMSA

Apresentamos a seguir um exemplo ilustrativo simples, onde existem dois lugares (INPUT_1 e INPUT_2) que representam valores de entrada pertencentes a domínios diferentes (por exemplo, um pertencente ao conjunto dos números inteiros e outro ao conjunto dos números reais). Além dos lugares de entrada, existem dois lugares que representam operações matemáticas a serem executadas sobre estas entradas e colocam o resultado em um lugar de saída comum (OUTPUT). Um dos lugares das operações matemáticas (NEG) só atua sobre o domínio dos números reais, calculando o valor contrário ao valor de entrada e o outro (ADD) atua sobre os dois domínios (inteiros e reais), tomando um valor de cada lugar e obtendo a soma deles. Neste exemplo quando dois valores são somados, eles são eliminados de seus lugares de entrada e usados de forma exclusiva o que não ocorre quando é calculado o valor contrário para aqueles valores que pertencem ao conjunto dos números reais. A rede de agentes que modela este exemplo é mostrada na figura 3.21.

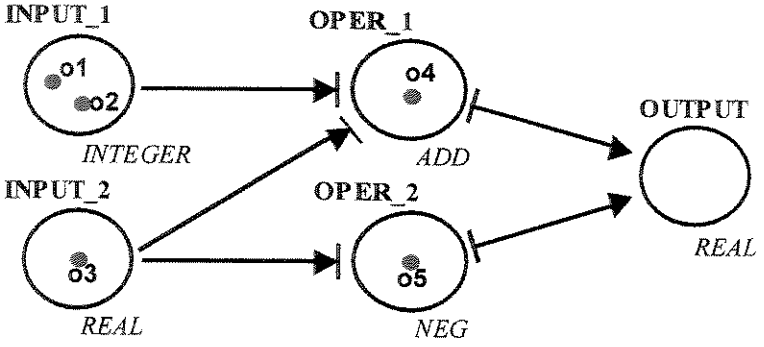
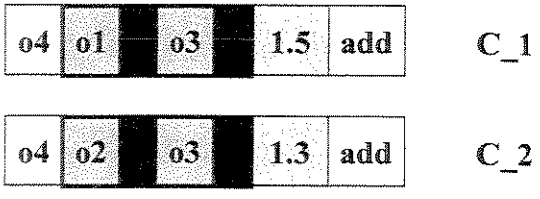


Figura 3.21 Rede de agentes exemplo para a execução do algoritmo BMSA

Execução do algoritmo BMSA:

Passo 1:

Combinações locais para o objeto ativo **o4**



Combinações locais para o objeto ativo **o5**

o5	o3		1.5	neg

C_3

Passo 2:

C_1	C_2	C_3

Passo 3:

o1	o2	o3

Passo 4:

As combinações locais **C_1** e **C_3** possuem o maior grau de interesse, de modo que teremos uma escolha aleatória. Neste exemplo, mostraremos primeiramente o que acontecerá quando a combinação **C_1** for escolhida e a seguir quando a combinação **C_3** for escolhida.

Quando a combinação **C_1** for escolhida

Combinações locais	**Combinações válidas**	**objetos**
C_1	C_1	o2
C_2		
C_3		

Neste caso somente será disparado o objeto **o4** resultando na execução da função ***add***, assimilando os objetos **o1** e **o3**. Além disto um novo objeto **o6** será gerado e posicionado no lugar OUTPUT.

Quando a combinação **C_3** é escolhida

Combinações locais	**Combinações válidas**	**objetos**
C_1	C_3	o1
C_2		o2
C_3		o3

Neste caso só será disparado o objeto **o5,** executando a função ***neg***, assimilando não destrutivamente o objeto **o3** e gerando um novo objeto **o6,** colocando-o no lugar OUTPUT.

3.7.2.8. Casos resolvidos pelo algoritmo de matching BMSA

O algoritmo de *matching BMSA* foi desenvolvido com o objetivo de resolver a maioria das possibilidades que possam aparecer na modelagem de sistemas em uma rede de agentes, sem uma preocupação mais explícita com o desempenho do mesmo. A seguir (figura 3.22) temos as diferentes possibilidades que podem existir e que são resolvidas pelo algoritmo de *matching BMSA*.

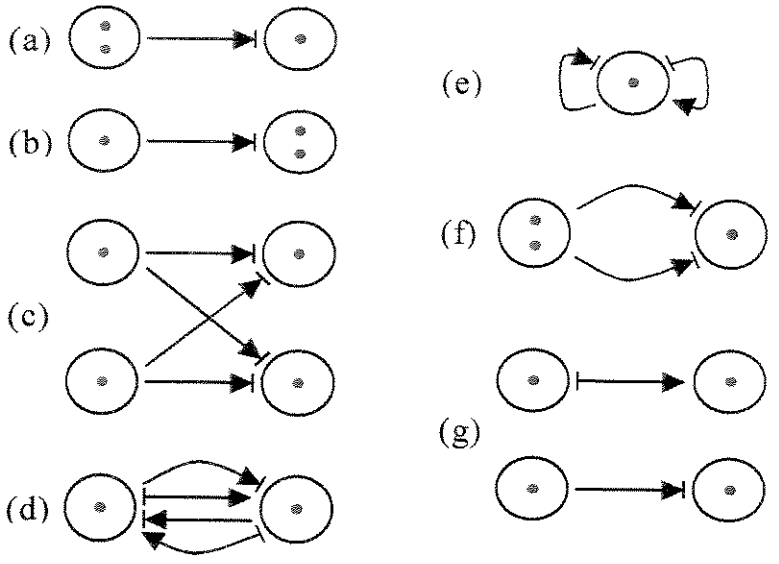


Figura 3.22 Casos resolvidos pelo algoritmo de *matching BMSA*.

No primeiro caso (figura 3.22a) temos um objeto ativo que pode assimilar (destrutivamente ou não) um objeto passivo (dos que estão disponíveis no lugar passivo - neste exemplo, dois objetos passivos), para este caso o algoritmo *BMSA*, simplesmente escolhe aquele para o qual exista maior grau de interesse (caso os dois tivessem o mesmo grau de interesse, um deles seria escolhido aleatoriamente) para ser assimilado no disparo do objeto ativo.

No exemplo representado pela figura 3.22b, temos a situação inversa ao exemplo descrito anteriormente. Neste caso, temos dois objetos ativos competindo pela assimilação de um objeto passivo. Aqui não é tão simples a escolha pelo algoritmo *BMSA*. Para o algoritmo de *matching* realizar sua escolha, ele precisará conhecer o modo de acesso de cada um dos objetos ativos para o objeto passivo. Se os objetos ativos acessam os objetos passivos de forma não exclusiva (compartilhável), ambos objetos serão disparados assimilando o objeto passivo. Caso o que tiver maior grau de interesse, acesse o objeto passivo de forma não exclusiva, então os dois objetos serão disparado. Se o objeto ativo que tiver maior grau de interesse acessar o objeto passivo de forma exclusiva, então só ele será disparado e poderá assimilar o objeto passivo. Se os objetos ativos acessam o objeto passivo de forma exclusiva, então só aquele que tiver maior grau de interesse ou for escolhido aleatoriamente (caso possuam o mesmo grau de interesse) poderá ser disparado e assimilar o objeto passivo.

Dois objetos ativos, onde cada um deles precisa de dois objetos passivos para serem disparados pelo algoritmo de *matching* são mostrados na figura 3.22c. Estes objetos ativos

estarão competindo para assimilar os objetos passivos disponíveis. Neste caso, a escolha entre o disparo de somente um dos objetos ativos ou de ambos depende do modo de acesso aos objetos passivos e do grau de interesse que eles tenham pela combinação destes objetos passivos. Para que ambos objetos possam ser disparados ao mesmo tempo, ambos devem acessar os objetos passivos de forma não exclusiva. Caso contrário, só disparará aquele objeto que tiver o maior grau de interesse, ou o que tenha sido escolhido de forma aleatória, caso os dois tenham o mesmo grau de interesse.

A figura 3.22d apresenta o caso em que um objeto ativo assimila outro objeto ativo gerando um novo objeto ativo e vice-versa. Neste caso só poderá ser disparado um objeto ativo, que será escolhido segundo seu grau de interesse (maior grau de interesse) ou de forma aleatória se tiverem o mesmo grau de interesse.

Para o caso da figura 3.22e o algoritmo de *matching* simplesmente não dispara o objeto ativo que tenta assimilar a si mesmo. Isto ocorre devido ao fato que um objeto que está se transformando (entenda-se executando alguma função de transformação) não pode estar sendo assimilado por ninguém.

No caso mostrado na figura 3.22f temos um objeto ativo que para poder ser disparado necessita pelo menos de dois objetos passivos diferentes, ou seja cada um dos objetos no lugar passivo vai ser transmitido por cada arco, já que um mesmo objeto não pode ser transmitido no mesmo tempo por ambos arcos.

No ultimo caso são mostrados o que poderíamos chamar de objetos geradores e consumidores (sensores e atuadores). No primeiro o objeto sempre é disparado gerando objetos passivos e no segundo sempre que tiver um objeto passivo disponível, este será assimilado.

3.8. Resumo

Inicialmente neste capítulo, abordou-se alguns aspectos (diferentes definições de agente, sua classificação e arquitetura) sobre a tecnologia de agentes, uma área em desenvolvimento e de muita importância na disciplina de ciência da computação e engenharia de software. Em seguida foi realizada uma pequena comparação entre agentes e objetos, permitindo mostrar que os objetos apresentados neste presente trabalho não são somente os objetos definidos pela programação orientada a objetos mas também como micro-agentes básicos os quais em cooperação e/ou colaboração quando são colocados em uma rede de objetos podem modelar ou sintetizar um agente mais complexo.

O conceito e formalização de rede de agentes como um caso especial de rede de objetos, definindo um algoritmo chamado de algoritmo de *matching* para obter as funções de seleção e possibilitar a dinâmica da rede, assim como uma possível implementação do algoritmo de *matching*, chamado BMSA (*Best Matching Search Algorithm*) foi apresentado.

CAPÍTULO 4. Implementação Computacional da Rede de Agentes

4.1. Introdução

Neste capítulo será apresentada a ferramenta denominada **ONtoolkit** (*Object Network toolkit*)[Guerrero 99b], desenvolvida com a finalidade de auxiliar no *design* e simulação de redes de agentes. Esta ferramenta computacional fornece um *engine* que implementa os mecanismos necessários para a execução de redes de agentes. Um destes mecanismos, e por sua vez o mais importante, é o algoritmo de *matching* descrito no capítulo anterior, que permite definir a dinâmica da rede no processo de simulação da mesma.

Objetivando a descrição das redes de agentes de forma independente da linguagem de programação, desenvolveu-se uma linguagem de especificação para as rede de agentes denominada de *Object Network Specification Language* (ONSL). A ONSL será descrita neste capítulo, assim como os passos a serem seguidos pelo usuário para editar e simular uma rede de agentes utilizando a ferramenta computacional desenvolvida.

4.2. Componentes do Software

O software desenvolvido (uma dentre as possíveis implementações dos conceitos definidos no capítulo anterior), é constituída por três módulos fundamentais:

- **MTON** (*Multi-Threaded Object Network*) **engine:** o conjunto de classes que implementam o formalismo das redes de agentes e sua máquina (*engine*) de simulação.
- **ON Desktop** (*Object Network Desktop*): o *front-end* para a *MTON engine.* Oferece ao usuário do software uma GUI amigável com facilidades e ferramentas para a edição e simulação de redes de agentes. (figura 4.1)
- **Plug Server**: o ponto de comunicação entre a rede de agentes executando na *MTON engine* e o mundo exterior (sensores, atuadores, outros programas, módulos de visualização, monitores, dispositivos e outros recursos disponíveis).

Cada uns dos módulos pode ou não rodar de forma independente. Alguns deles contêm um conjunto de ferramentas ou aplicativos para um melhor desempenho de suas tarefas e/ou controle. A idéia é a construção de uma ferramenta computacional que permita o *design* e simulação de sistemas, especialmente de sistemas inteligentes e que exibam as seguintes características:

- **Robustez** (*Robustness*), através da utilização de uma linguagem de especificação independente que isola o modelo da rede de agentes da linguagem de programação utilizada para sua implementação.
- **Desempenho de processamento** (*Processing performance*), através da geração automática de *stubs* para cada classe de usuário definida por meio da linguagem de especificação da rede de agentes. Estes *stubs* são compilados junto ao código do usuário. Classes externas, no caso classes *Java*, podem ser incorporadas pela máquina (*engine*) de simulação da rede e manipuladas como classes típicas da rede de agentes.
- **Escalabilidade** (Scalability), com suporte para computadores com capacidade para SMP (*Symmetric Multi Processing*) através da arquitetura *multi-thread*. Projetado para permitir qualquer outra implementação do algoritmo de *matching*, por exemplo um algoritmo de *matching* distribuído.
- **Interface amigável** (*Easy-to-use interface*), provida por uma interface de usuário gráfica (GUI, *Graphical User Interface*) e de uma linguagem de especificação para cada componente na rede de agente.
- **Portabilidade** (*Portability*), em virtude de ser totalmente implementado na linguagem *Java*, ele pode rodar independente do sistema de hardware e do sistema operacional, desde que este suporte a JVM (*Java Virtual Machine*)

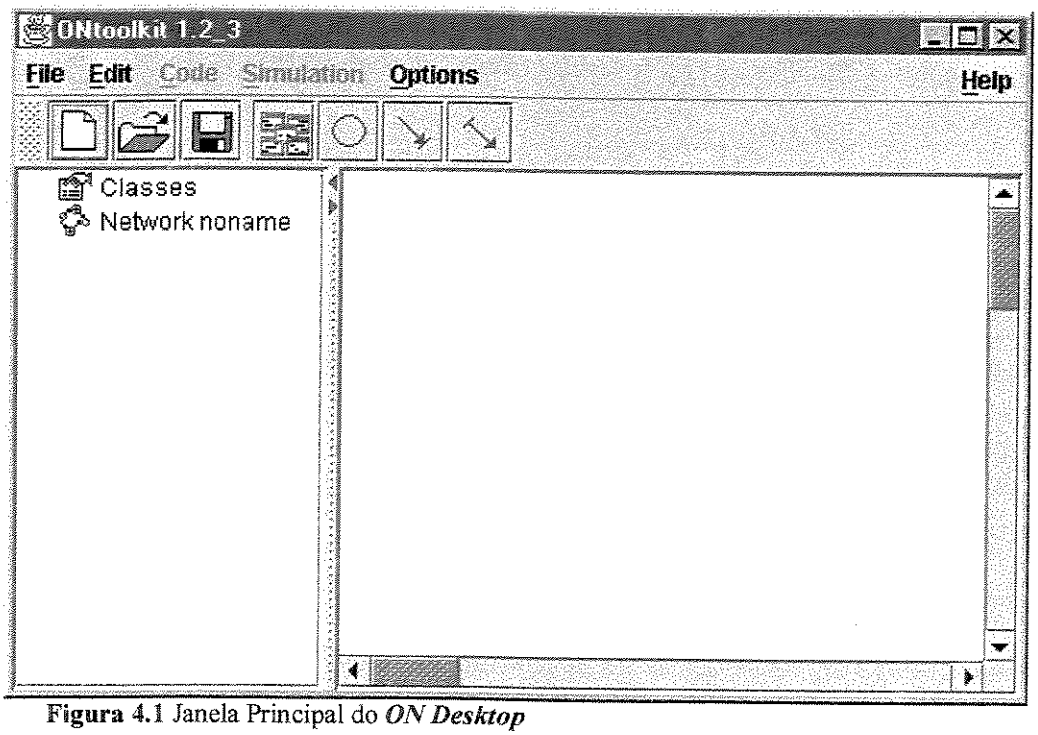


Figura 4.1 Janela Principal do ***ON Desktop***

A seguir serão explicado em detalhes os componentes da ferramenta computacional.

4.2.1. MTON (Multi-Threaded Object Network) engine

Este módulo constitui a parte essencial da ferramenta para a simulação das rede de agentes, visto que ela implementa o formalismo das redes de agentes apresentado no capítulo anterior.

Algumas características importantes deste módulo são:

- Permite instanciar, simular e fazer *debug* de qualquer rede de agente
- Foi projetado para ser executado como um processo *daemon* remoto, acessível através de um *socket* TCP, além de rodar localmente como um sistema com todas as características de um sistema *shell*
- Permite o carregamento dinâmico das classes utilizando um servidor de classes remoto
- Cada lugar da rede de agente a ser simulada, é processado em seu próprio *thread*
- As classes Java ordinárias podem ser importadas e utilizadas na *engine* como classes puras de redes de agentes. Para isso são fornecidos um conjunto de interfaces e classes abstratas Java
- Permite uma manipulação apropriadas das exceções (*exceptions*) que podem ser provocadas pelo código do usuário (*user exceptions*) e/ou pelo código interno da *engine* (*critical exceptions*)

A *MTON engine* é composta basicamente por um conjunto grande de classes abstratas, interfaces e classes de suporte. O primeiro passo para habilitar a *MTON engine* é possuir o pacote compilado Java da rede de agentes desejada. Estas classes contêm informação sobre a topologia, classes (incluindo sua implementação Java) e os objetos do *kernel* da rede de agentes em questão. Esses recursos são gerados por meio do *ON Desktop*.

Acessar diretamente a *MTON engine* é uma tarefa muito complexa, visto que diferentes protocolos de comunicação são necessários. Com a finalidade de facilitar esta comunicação desenvolveu-se um *console* interativo, com uma interface texto. Este *console* pode ser controlado localmente pelo usuário ou redirecionada para um *socket* TCP, permitindo o controle de forma remota. Quando utilizado como um servidor de *console* remoto, atua como um *daemon* do sistema, de tal forma que pode ser carregado (inicializado) na mesma JVM[1] do *ON Desktop* (por *default* roda neste modo) ou em qualquer JVM remota.

A *MTON engine* é necessária somente na simulação de rede de agentes, pois a edição e compilação da rede pode ser totalmente realizada por alguns módulos disponíveis no *ON Desktop*.

[1] Java Virtual Machine, processador de *bytecodes* gerados pelo compilador Java.

4.2.1.1. Carregando uma rede de agentes compilada no console

Sempre que o usuário desejar simular uma rede de agentes, a qual supostamente está compilada, basta executar uma opção apropriada no *ON Desktop* ou executar o comando correto no console (tabela 4.1). Isso provoca o carregamento da rede de agentes no *console*. Entretanto nem sempre as classes que definem a rede de agentes são acessíveis para o *console class loader*[2]. Nesta situações um *remote class loader* é fornecido, sendo este responsável pelo carregamento de todas as classes Java que sejam necessárias para instanciar a rede de agentes. Isto é realizado mediante sucessivas conexões (tantas quanto forem necessárias) a um servidor de arquivos disponível, o qual roda como um *daemon* na mesma JVM onde está rodando o *ON Desktop* e têm acesso direto aos arquivos que compõem o pacote desejado. Este processo é totalmente transparente para o usuário. Portanto quando o usuário utilizar o *console* na forma textual (sem a utilização do *ON Desktop*), ele deverá estar ciente das suas implicações.

A figura 4.2 apresenta todo o processo de interação entre os módulos *MTON engine* e o *ON Desktop*. O *ONSL compiler*[3] gera todas as classes Java compiladas que implementam a rede de agentes e as disponibiliza através de seu servidor de arquivos-classes (*File Server*) local. Quando o usuário solicitar o início da simulação, o *console* carregará a rede de agentes, fornecida através do servidor de arquivos e a instanciará em sua JVM local (JVM onde está rodando a *MTON engine*).

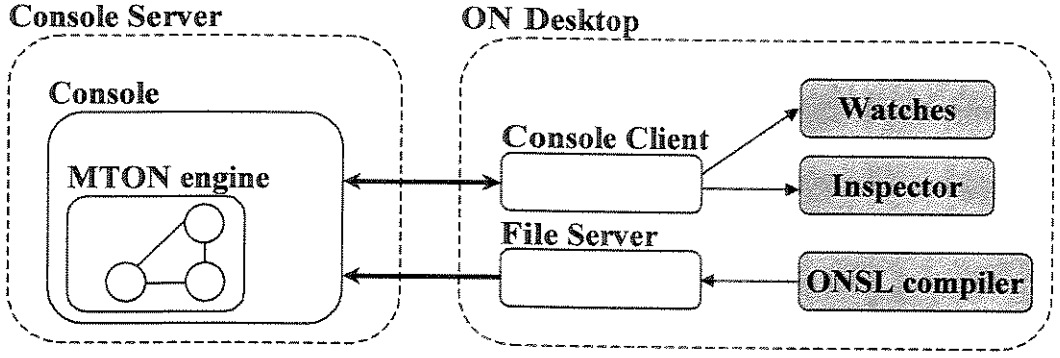


Figura 4.2 Processo de interação dos módulos *MTON engine* e o *ON Desktop*.

A seguir mostra-se uma tabela com um resumo dos principais comandos disponíveis para utilizar o *console*.

Comando	Descrição
`Server HOSTNAME:PORT`	Define a localização e a porta do atual servidor de arquivos classes (*File Server*). O valor *default* é `localhost:5000`.

[2] Módulo responsável pelo carregamento das classes. Dado o nome da classe este módulo localizará ou gerará os dados que constituem a definição da classe.

[3] *Object Network Specification Language compiler*, compilador de rede de agentes definida pela linguagem de especificação de rede de agentes.

Comando	Descrição
`Load NETWORK`	Instancia a rede de agentes. A rede de agentes fornecida deve ser lida ou disponibilizada pelo atual servidor de arquivos (File Server).
`Down`	Executa o *shutdown* e remove a rede de agentes que estiver em execução atualmente na JVM.
`step [N]`	Executa `N` iterações na rede de agentes que estiver em execução. Se o valor de `N` for omitido, apenas uma iteração será realizada, ou seja, `N` = 1.
`watch PAGE COMMAND`	Adiciona uma entrada de *watch* na página de *watches* especificada.
`watch PAGE (on|off)`	Habilita (*on*) ou desabilita (*off*) a página de *watches* especificada.
`unwatch PAGE ENTRY`	Remove a entrada de *watch* especificada na pagina de *watches* especificada.
`[OBJECT].METHOD`	Invoca um método de *debugging* no objeto especificado. Se o nome do objeto for omitido, o *namespace* atual será utilizado. Dependendo do método a ser executado podem ser necessários um ou mais argumentos.
`cd (..|DIRECTORY)`	Modifica o diretório atual. A especificação do diretório não permite o uso em linha de '..' para ser agregado a outro nome como ocorre em sistema de arquivos..
`Pwd`	Mostra o diretório atual no *namespace*.
`buffer [on|off]`	Ativa(*on*)/desativa(*off*) o *buffering* ou informa o estado atual.
`echo [on|off]`	Habilita(*on*)/desabilita(*off*) o *echoing* ou informa o estado atual.
`debug [on|off]`	Ativa(*on*)/desativa(*off*) o *debugging* interno ou informa o estado atual.
`Config`	Informa todas as configurações (*debugging*, *bufferings*, *watches*, etc.) disponíveis atualmente no console.
`Help`	Apresenta uma pequena descrição de todos os comandos disponíveis no console.

Tabela 4.1 Resumo dos principais comandos do *console*.

4.2.1.2. Manipulando as exceções (Exceptions)

Durante o processo de simulação da rede de agentes, diferentes trechos do código interno do *MTON engine* e do código de usuário podem disparar exceções (*exceptions*) a qualquer instante. O *MTON engine* classifica as *exceptions* em dois grupos diferentes:

1. *User exceptions*: Sempre disparada pelo trecho de código do usuário ou por algum *stub* onde um método do código de usuário tenha sido invocado. Este tipo de *exception* é muito comum durante o processo de *debugging*. A ação executada pelo

MTON engine quando este tipo de *exception* for capturada será a eliminação de todos os *threads* em execução no *MTON engine*. O *console* (e o *console server*) descarregam todas as classes da rede de agentes e ficam aguardando por novos comandos. Esta *exception* é notificada ao usuário através da janela de simulação do *ON Desktop*.

2. *Engine exceptions*: é disparada intrinsecamente pelo código do *MTON engine* sem nenhuma relação com o código do usuário. Não obstante ela pode ser disparada devido a implementação incorreta nas classes Java externas relativas às interfaces `Mutable` e `Immutable`. A ação executada será cancelar a execução da *Java Virtual Machine*, visto que se trata de um erro muito grave no *MTON engine*.

4.2.1.3. Importando classes Java externas para o MTON engine

O *MTON engine* está habilitado a importar classes Java ordinárias (classes definidas pelo usuário na linguagem Java) dentro de seu sistema de execução, permitindo a reutilização do código já previamente disponibilizado. Contudo a importação só será possível se estas classes Java implementarem a interface `ExternalObject` (responsável pela conduta básica de um objeto externo) já disponibilizada no presente trabalho. Caso se deseje reaproveitar classes Java escritas previamente, ou da biblioteca padrão do Java, será necessário se criar classes *adapters*, que estendam a classe desejada e implementem a interface `ExternalObject` Quase todas as checagens necessárias em tempo de execução (*runtime*) devem ser realizadas pelas classes Java externas. Embora esta estratégia possa ser inadequada, ela oferece mais flexibilidade para o sistema, uma vez que o usuário estará livre para adotar qualquer recomendações relacionadas à segurança de dados (Exemplo: ocultamento de informações e encapsulamento dos dados).

A interface `ExternalObject` foi projetada para prover um mínimo de controle sobre os objetos que estão em execução nos vários *threads*. Por conveniência, uma classe *adapter* abstrata (`DefaultExternalObject`) é fornecida. Novas classes externas devem simplesmente estender esta classe, incorporando o comportamento desejado, sem que seja necessária um cuidado maior quanto aos métodos de `ExternalObject`, que são implementados de modo default em `DefaultExternalObject`. Porém uma vez que a linguagem Java não suporta herança múltipla, em muitas ocasiões, tal implementação disponibilizada não poderá ser utilizada, sendo necessário que o usuário realize sua própria implementação. Os objetos nativos da rede de agentes (aqueles definidos nas rede de agentes) não necessitam de tratamento especial pois seu controle de concorrência está embutido em seu código.

O processo de disparo na rede de agentes (ou seqüência de iteração) pode ser dividido em dois estados fundamentais. O primeiro dele é o processo de *matching* no qual todas as possíveis combinações são tratadas pelos consumidores interessados. Um aspecto importante neste ponto é que **não** pode ser realizada nenhuma modificação no estado dos objetos na rede de agentes durante esta fase, devido ao fato de ser impossível predizer a ordem na qual os consumidores interessados podem fazer seu *matching*. O segundo estado é a fase de execução do disparo da rede de agentes, quando todos os conflitos já estão

resolvidos e os objetos habilitados para executar as operações assinaladas. Neste instante deve-se ter cuidado com as modificações e sobre quais objetos estas serão realizadas, porque os objetos podem ser assimilados de diferentes formas pelos consumidores. Quando um objeto é compartilhado ele não pode ter seu estado modificado.

O passo inicial para se adaptar uma classe Java externa é implementar a interface `ExternalObject`. Para isso será necessário a implementação de três métodos. O código pode ser importado do pacote disponibilizado como:

```
import netobj.MTON.*;
```

As interfaces são:

```
public interface ExternalObject extends NetCloneable {
     public void setOwner(ExternalNetObject owner);
     public ExternalNetObject getOwner();
}
```

```
public interface NetCloneable {
     public NetCloneable createClone();
}
```

`ExternalNetObject getOwner()`
: Devolve o proprietário desta instância do objeto externo. Atualmente este valor é o mesmo valor atribuído previamente pelo método `setOwner`, ou `null` se o objeto não possuir proprietário.

`void setOwner(ExternalNetObject owner)`
: Determina o proprietário deste objeto. Se este objeto for removido o proprietário deve ser definido como `null`.

`NetCloneable createClone()`
: Cria uma nova instância que possui o mesmo valor desta primeira.

Este processo pode resultar tedioso, caso o usuário necessite realizar este procedimento várias vezes em suas classes. Uma forma simples é fazer uso de uma implementação *default* disponibilizada neste trabalho para a interface `ExternalObject`. Observe que esta implementação considera o problema de sincronização entre *threads*.

```
abstract public class DefaultExternalObject
     implements ExternalObject, Cloneable {

/** Owner of this object. */
protected ExternalNetObject __owner;

/**
* Gets the owner of this object.
*
* @return owner of this object.                    cont ...
```

```
*     If this object has no owner it will
*     return <code>null</code>.
*/
public ExternalNetObject getOwner() {
     synchronized(this) {
         return __owner;
     }
}
/**
* Sets the owner of this object.
*
* @param owner new owner of this object.
*/
public void setOwner(ExternalNetObject owner) {
     synchronized(this) {
          __owner = owner;
     }
}

/**
* Creates a clone of this object.
* @return clone of this object.
*/
public NetCloneable createClone() {
     try {
          return (NetCloneable)clone();
     } catch (CloneNotSupportedException e){
          Throwable re =
          new RuntimeException("could not" +
                               " create clone.");
          throw (RuntimeException)re.fillInStackTrace();
     }
}
}//DefaultExternalObject
```

Esta implementação fornece um *wrapper* a partir do método `netobj.MTON.createClone` para o `java.lang.Object.clone`, o qual é mais conhecido pelos programadores em Java (aqui eles possuem funcionalidade equivalente). Desta forma o usuário poderá esquecer do `createClone` e implementar somente o `clone`.

A sintaxe do método `clone` é:

`Protected Object clone() throws CloneNotSupportedException`

Este método é derivado da interface `Cloneable` e devolve uma cópia deste objeto.

Devido à limitação imposta pela linguagem Java decorrente da ausência de um mecanismo de herança múltipla, esta implementação, por *default,* não pode ser sempre utilizada. Caso uma classe seja subclasse de outra classe, ela não pode ter como classe base a classe `DefaultExternalObject`. Esta situação pode ser solucionada simplesmente implementando os métodos da interface `ExternalObject`.

O próximo passo é avaliar como os estados serão tratados, ou seja, como eles serão modificados e quem poderá disparar a mudança do estado de um objeto.

O *MTON engine* pode manipular dois tipos de instâncias relacionadas com a conduta do estado:

- **Mutable**, instâncias que podem ter seu estado modificado;
- **Immutable**, instâncias que, uma vez criadas, mantêm o estado inalterado até que eles sejam destruídos pelo *garbage collector*[4].

A segunda instância é o caso mais simples, uma vez que não existe forma dos estados serem modificados. Desta forma não existirá uma seqüência de eventos concorrentes que possam provocar problemas nestes objetos.

Por questão de conveniência é fornecida a interface `Immutable` para ser implementada por estas classes, além de ser fornecida uma implementação *default* para esta interface. Esta implementação é a mesma que `DefaultExternalObject`, ela só foi definida para aqueles programadores que são mais rigorosos com seu código.

Interface `Immutable` e sua implementação por *default* `DefaultImmutable`:

```
public interface Immutable extends ExternalObject {}
```

```
/**
 * This is a abstract helper class fo implementing
immutable objects. This
 * class exists as matter of convenience.
 *
*/

abstract public class DefaultImmutable extends
     DefaultExternalObject implements Immutable {

} //DefaultMutable
```

O caso *`Mutable`* é certamente mais difícil, devido aos fatos que agora dependem da forma como o usuário projeta a classe externa e em particular a maneira como a JVM escala os *threads* concorrentes. Não temos um padrão que garanta uma completa solução a todos estes problemas. O *MTON engine* fornece ao *designer* de classe externa informações de "runtime" visando prevenir, pelo menos, os casos patológicos. Como exemplo temos aqueles que ocorrem quando algum estado é modificado, quando a rede de agentes está em fase de *matching* ou quando um objeto está assimilando outro que por sua vez é compartilhando com outros.

A interface `Mutable` e a classe `DefaultMutable` (implementação *default* da interface) é fornecida procurando-se auxiliar o usuário no *design* de uma classe externa que será

[4] Ferramenta da JVM para eliminar todos os objetos que estiverem sem referência.

modificável. Esta classe faz uso da informação fornecida por seu *wrapper* (uma instância da classe `ExternalNetObject`) de modo a saber se o estado pode ou não ser modificado dependendo da forma de acesso que esteja sendo utilizado sobre o objeto.

Abaixo apresentamos a interface `Mutable` e sua implementação, a classe `DefaultMutable`.

```
/**
 * Interface for external objects which have mutable
 * state.
 *
 * Mutable objects may be in certain states in which
 * the MTON engine should be sure that it can pass
 * references to this object to another interested
 * consumers. This interface provides a simple
 * mechanism to do so.
 *
*/
public interface Mutable extends ExternalObject {

public String CANNOT_BE_MODIFIED =
                    "object cannot be modified.";

/**
* Determines if this object can change its internal
* state.
*
* This information is obtained from the parent,
* invoking its <code> canModifyState</code> method.
*
* @return true (free to modify state), false (denied).
*/
public boolean canModifyState();

/**
* Throws an exception if this object cannot have
* its internal state changed.
* @exception ConcurrentException if this object
*            cannot be change.
*/
public void assertModifiable()
     throws ConcurrentException;

}//Mutable
```

```
abstract public class DefaultMutable extends
          DefaultExternalObject implements Mutable {

/**
* Determines if this object can change its internal
* state. This information is obtained from the
* parent, invoking its <code>canModifyState</code>
* method. Notice that invocations to this method
                                              cont...
```

```
* should be synchronized on this object.
* @return true(free to modify state), false(denied).
*/

public boolean canModifyState() {
     if (__owner != null)
     synchronized(__owner) {
          synchronized (this) {
          return __owner.canModifyState();
          }
     }
     return true;
}
/**
* Throws an exception if this object cannot have
* its internal state changed.

* @exception ConcurrentException if this object
* cannot be change.
*/
public void assertModifiable()
          throws ConcurrentException{
     if (!canModifyState())
     throw new ConcurrentException(CANNOT_BE_MODIFIED);
}
} //DefaultMutable
```

A seguir é mostrada a listagem de uma possível classe projetada para representar um valor inteiro, permitindo algumas operações sobre ele. Neste caso ela pode ser uma subclasse de `DefaultMutable`.

```
import netobj.MTON.*;

public class xInteger extends DefaultMutable {

private int value;

public xInteger(int value){
     this.value = value;
}

synchronized public int getValue() {
     return value;
}

synchronized public void setValue(int i) {
     assertModifiable();
     value = i;
}

synchronized public void add(int value) {
     assertModifiable();
     this.value += value;
}                                                    cont...
protected Object clone() {
```

```
        return new xInteger(value);
    }
} //xInteger
```

4.2.2. Plug Server

As redes de agentes, no início, foram concebidas como um sistema fechado, no qual não existia nenhuma comunicação com o mundo externo. As novas extensões relacionadas com o desenvolvimento da ferramenta **ONtoolkit** mostraram a necessidade de se criar um dispositivo que permitisse à rede ter algum tipo de interatividade com o mundo externo. Nossa proposta é uma introdução ao conceito de *plugs*, o qual possibilita que objetos de software externos possam pegar e/ou colocar dados dentro da rede em tempo de execução.

Os *plugs* são implementados como objetos distribuídos através do Java RMI[5] *framework*. O **ONtoolkit** oferece diferentes *plugs*, prontos para serem utilizados nas redes de agentes que sejam definidas pelo mesmo.

4.2.2.1. Construindo e utilizando plugs na rede de agentes.

Para a construção de um *plug* que logo possa ser utilizado em uma rede de agentes, são fornecidas a interface `plug` e a classe abstrata `AbstractPlug` que oferecem os recursos básicos para a implementação de objetos distribuídos em Java através do RMI disponíveis para serem utilizados nas redes de agentes.

A interface `plug` fornecida é a seguinte:

```
package plug;

import java.rmi.*;

public interface Plug extends Remote {

    public void closeSession()
        throws RemoteException;

} //Plug
```

e a classe abstrata `AbstractPlug` é:

```
package plug;

import java.rmi.*;
import java.rmi.server.*;
                                                    cont...
abstract public class AbstractPlug extends
```

[5] Remote Method Invocation, serviço do Java para o trabalho com objetos distribuídos puramente Java.

```
     UnicastRemoteObject implements Plug {

     public AbstractPlug()
          throws RemoteException {
          super();
     }

} //AbstractPlug
```

Um exemplo simples de *plug* disponível no **ONtoolkit**, que mostra uma janela de *logs* é mostrado a seguir:

```
package plug.lib;

import java.awt.*;
import java.rmi.*;
import java.rmi.server.*;

import plug.*;

public interface RemoteLog extends Plug {

     public void append(String text)
          throws RemoteException;

     public void clear()
          throws RemoteException;
} //RemoteLog
```

```
package plug.lib;

import java.awt.*;
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import javax.swing.*;

import plug.*;

public class RemoteLogImpl extends AbstractPlug
     implements RemoteLog {

     private PlotFrame frame;
     private JTextArea text;

     public RemoteLogImpl()
          throws RemoteException {
          super();

//create plot panel and add it to the current frame...
          text = new JTextArea();
                                                  cont...
          frame = new PlotFrame("Remote logging",
```

```
                new Dimension(400,40));
        frame.getContentPane().add(text, "Center");
        frame.pack();
        }

        public void closeSession()
                throws RemoteException {
                frame.dispose();
        }

        public void append(String text)
                throws RemoteException {
                this.text.append(text);
        }

        public void clear()
                throws RemoteException {
                text.setText("");
        }

}//RemoteLogImpl
```

Os *plugs* podem ser incorporados na rede de agentes da seguinte maneira:

- Criar um *wrapper* para o *plug*, da mesma maneira que é criado para uma classe externa Java quando se deseja importá-la para ser utilizada na rede de agentes. Na distribuição do software são fornecidas algumas classes prontas para ser utilizadas na modelagem das rede de agentes. Entre estas classes temos uma classe `Wrapper` (`netobj.MTON.user.lang.Wrapper`) genérica que pode ser utilizada para esta função.
- Inicializar o *plug* no código da definição do *kernel* na classe onde o *plug* será utilizado.
- Incluir as chamadas desejadas ao *plug* na classe onde está sendo utilizado o *plug* (na seção *perform* das funções de transformação da classe).

A seguir será mostrado um exemplo onde o *plug* descrito na seção anterior é utilizado, ou seja, o *plug* que foi construído para ter um *log* . Neste, temos a descrição dos eventos ocorrendo em alguma classe e/ou valores dos atributos referentes a uma determinada classe.

Código de inicialização do *plug* na seção de inicialização do *kernel* da classe.

```
...
switch(kid){
        case KERNEL_ferramenta1:
        case KERNEL_ferramenta2:
                put_interno(new xInteger(-1));
                                                cont...
```

```
                RemoteLog p = (RemoteLog)createPlug(
```

```
                        "plug.lib.RemoteLogImpl");
            p.clear();
            p.append(get_interno().toString());
            put_plug(new Wrapper(p));
            break;
...
```

Código de utilização do *plug* na seção *perform* de uma função de transformação.

```
...
int x = get_interno().getValue();
x = x-1;
get_interno().setValue(x);
RemoteLog pl = (RemoteLog)get_plug().getData();
pl.clear();
pl.append(get_interno().toString());
...
```

4.2.3. ON Desktop

O *ON Desktop* foi desenvolvido como ferramenta de suporte para o usuário no *design* e simulação das redes de agentes usando a arquitetura do *MTON engine*. Esta ferramenta provê mecanismos para facilitar o trabalho com os elementos que definem uma rede de agentes (classes, lugares, arcos e os objetos do *kernel*). A janela principal do *ON Desktop* foi mostrada na figura 4.1.

O *ON Desktop* possui dois modos fundamentais de trabalho:

- **Modo de edição**, onde as classes e topologia da rede são definidas.
- **Modo de simulação**, onde o *ON Desktop* interage com o *MTON engine* e executa o processo de simulação da rede de agentes especificada, fornecendo as ferramentas para realizar as tarefas de *debuging* necessárias na rede de agente que estiver sendo simulada.

4.2.3.1. Modo de edição

No modo de edição, o **ONtoolkit** fornece todas as facilidades para o *design* da rede de agentes, oferecendo maior facilidade na criação das classes necessárias e da topologia da rede de agente. A primeira coisa que o usuário deverá fazer é definir as classes ou a maioria das classes que serão utilizadas na rede de agentes, devido a que todos os elementos da rede de agentes que definem sua topologia (lugares, arcos) estão estreitamente ligados às classes definidas e seus elementos internos (variáveis, funções de transformação e portas privadas de entrada e/ou saída). A figura 4.3 mostra a janela de edição do **ONtoolkit**.

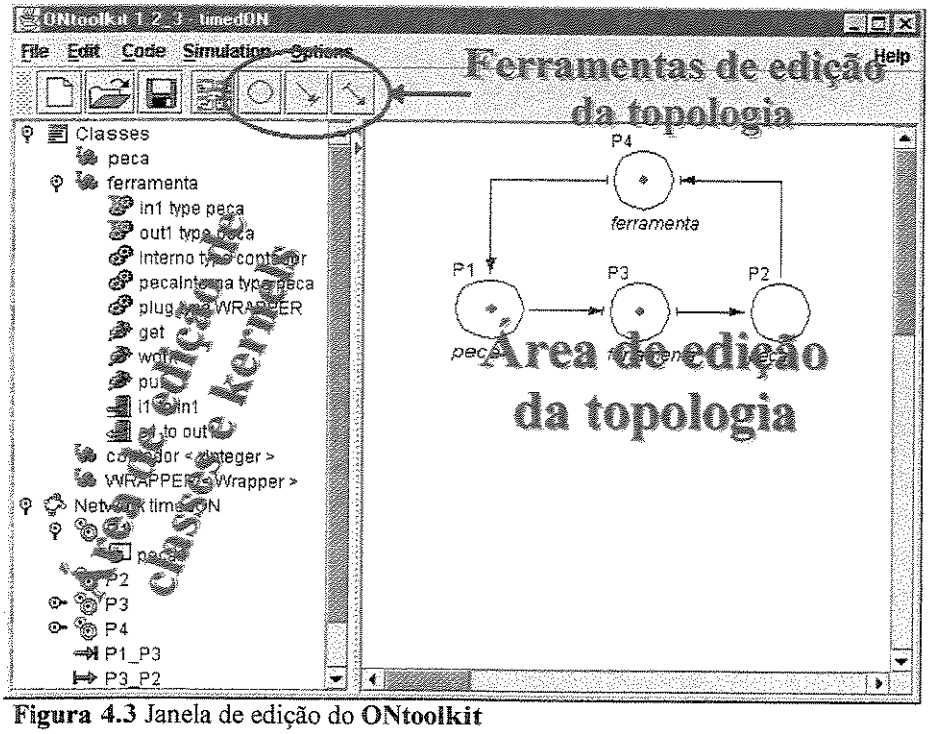


Figura 4.3 Janela de edição do **ONtoolkit**

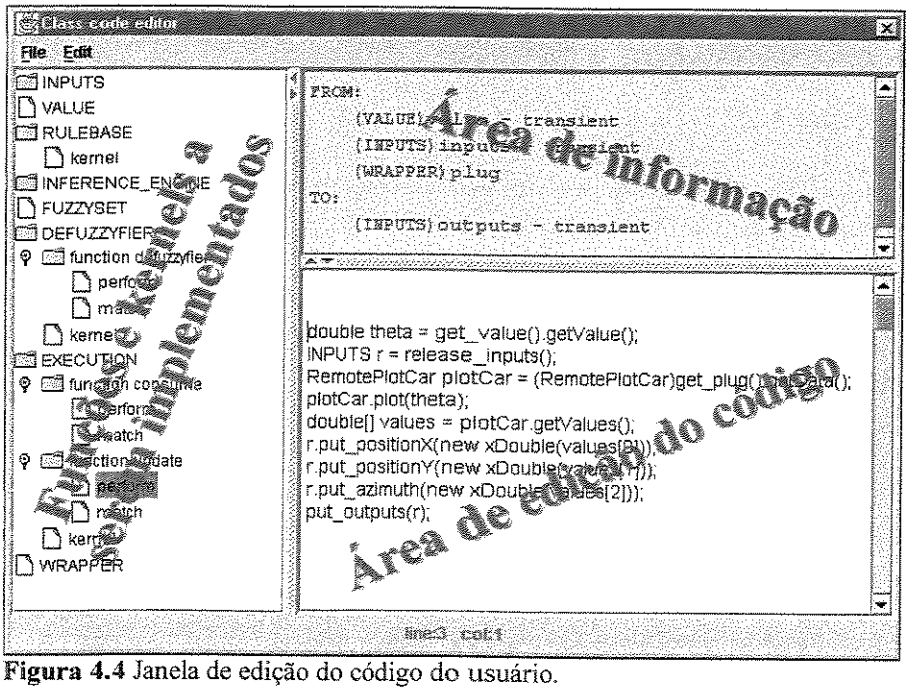


Figura 4.4 Janela de edição do código do usuário.

Para iniciar a edição da topologia da rede será necessário definir pelo menos uma classe (aquela que será utilizada na definição do lugar), uma vez que cada lugar está associado a uma classe indicando o tipo de objetos que podem ser armazenados nesse lugar. A seguir deve-se definir pelo menos um lugar ativo (aquele onde a classe associada a seu tipo é uma classe ativa, ou seja, que contém na sua definição pelo menos uma função de transformação). Somente então, pode-se definir pelo menos um arco.

Após serem especificadas as funções de transformação das classes e *kernel* da rede de agentes, o usuário pode passar a editar o código específico (em Java) das funções de transformação e de suas respectivas funções de *matching*, assim como do código de inicialização dos objetos definidos que estarão inicialmente em cada lugar, ou seja, aqueles que formam o *kernel* da rede. Com o objetivo de facilitar ao usuário a edição do código especifico, é incorporado ao **ONtoolkit** um módulo simples de edição de código. Este editor é mostrado na figura 4.4.

4.2.3.2. Modo de simulação

O modo de simulação (figura 4.5) permite ao usuário executar a rede de agentes especificada passo a passo ou através de um conjunto de iterações. Além disso fornece um conjunto de ferramentas (*watches* e *inspector*) que possibilitam o processo de *debuging* da rede de agentes. Neste modo estão disponíveis todos os comandos fornecidos pelo *console* do *MTON engine*.

Se existirem *plugs* associados à rede de agentes que estiver executando, então eles serão ativados neste modo de operação do **ONtoolkit.** As ferramentas de *debuging* podem ou não ser utilizadas, ou seja, não precisam estar ativadas durante todo o processo de simulação.

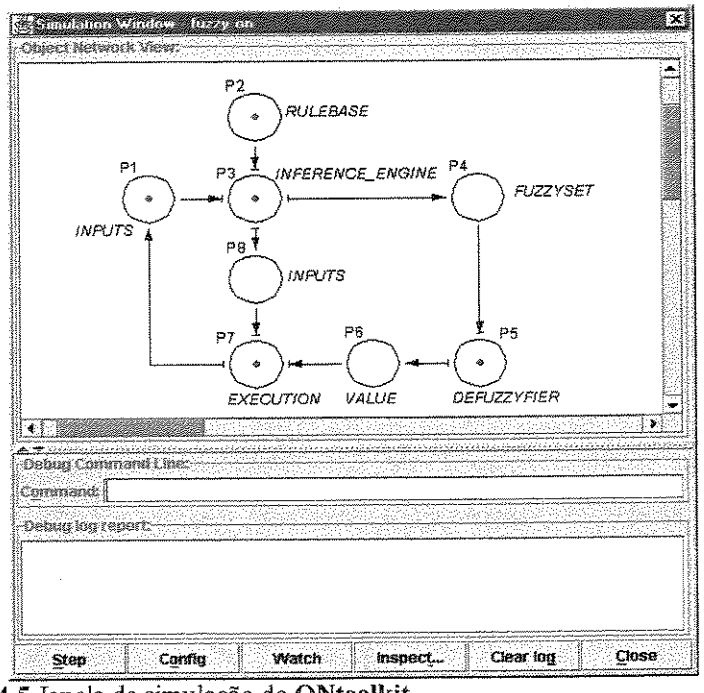


Figura 4.5 Janela de simulação do **ONtoolkit**.

Na figura 4.6 é mostrado um exemplo das ferramentas que podem ser utilizadas durante a simulação para executar o processo de *debuging* sobre a rede de agentes que está atualmente rodando no *MTON engine*.

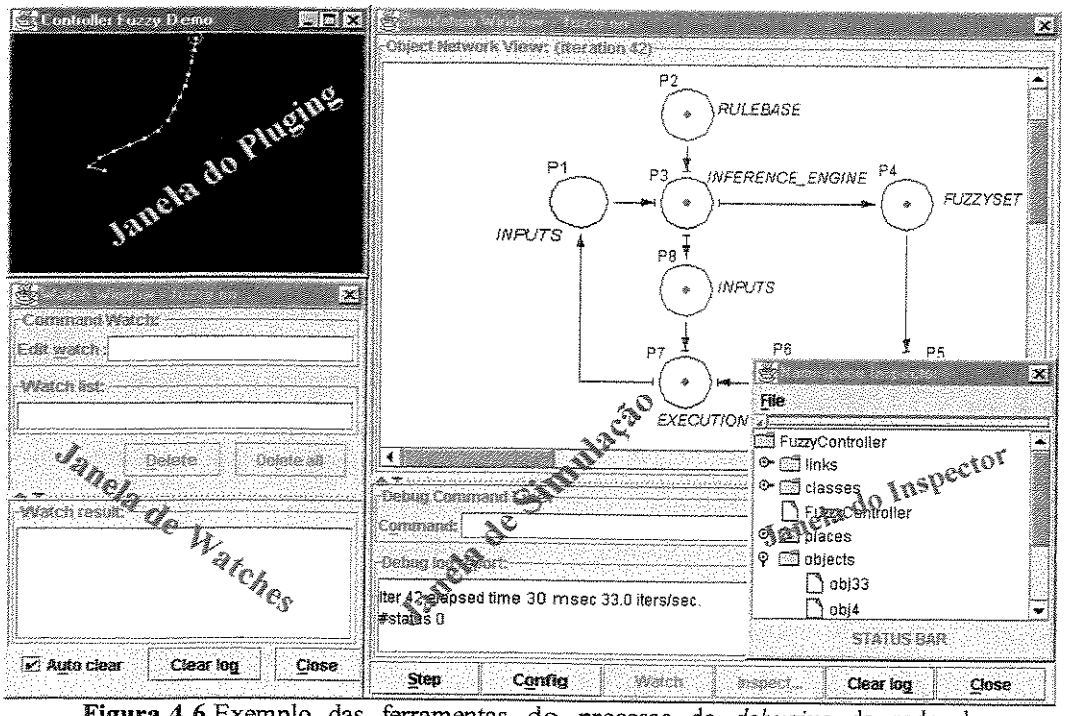


Figura 4.6 Exemplo das ferramentas do processo de *debuging* da rede de agentes.

4.3. Seqüência de uso do ONtoolkit

A seqüência típica de uso do ONtoolkit é mostrada na figura 4.7. O principal componente é o *ON Desktop*, que inclui um **Módulo de Edição da Topologia**, (fornecendo as facilidades para edição das estruturas das classes e a topologia da rede de agentes), um **Módulo de Edição do Código de Usuário**, (permitindo a edição do código específico das funções de transformação, funções de *matching* associadas e o código de inicialização dos objetos que pertencem ao *kernel* da rede de agentes), um **Módulo de Simulação** (que controla remotamente a simulação da rede de agentes) e o **ONSLC** (*Object Network Specification Language Compiler*), que utiliza os arquivos obtidos pelos módulos de edição de topologia e de código de usuário, para gerar, através da análise e interpretação dos mesmos, um conjunto de arquivos intermediários (.java), ou seja, um grupo de *stubs* e *skeletons* que implementam em código fonte a especificação do modelo de rede de agentes, de maneira automática.

O usuário deve criar as classes e a topologia da rede utilizando o Módulo de Edição da Topologia, e a seguir inserir o código específico às funções do objeto, por meio do Módulo de Edição do Código de Usuário. Na seqüência, deve-se indicar ao *ON Desktop* que inicie o processo de compilação. Inicialmente, o ONSLC é invocado, gerando os arquivos .java de *stubs* e *skeletons*. Em seguida, o *ON Desktop* invoca o compilador Java (**javac**), gerando o código executável java (.class) que é disponibilizado para simulação. Por último, temos o **MTON engine** (*Multi-Threaded Object Network*), que é a máquina de simulação das redes de agentes. Este, faz uso dos arquivos gerados pelo *ON Desktop*, em conjunto com as

classes externas devidamente compiladas, para proceder à simulação das redes de agentes. Esta simulação, pode ser controlada pelo Módulo de Simulação do *ON Desktop*, que se comunica com o MTON e gerencia o desenvolvimento da simulação, de maneira remota.

Observe que classes Java externas, ou seja, classes Java prontas, podem ser utilizadas pela rede de agentes, desde que elas implementem as interfaces que provêem os mecanismos de segurança necessários ao *MTON engine*.

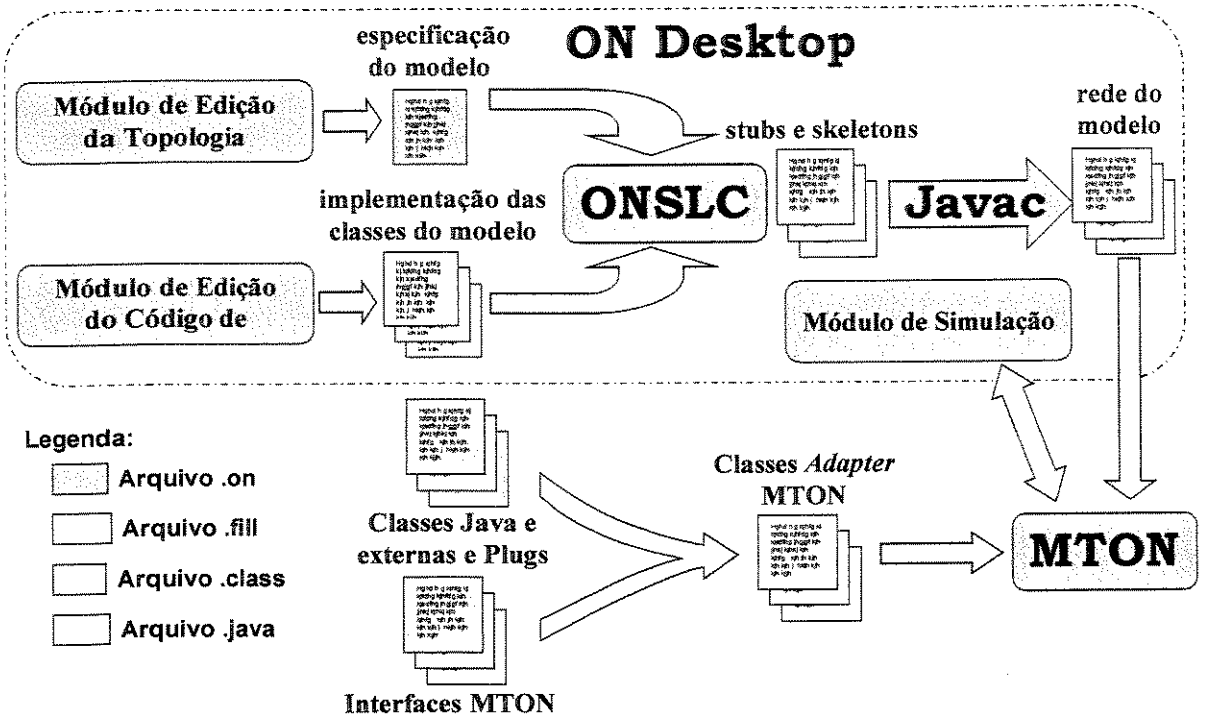


Figura 4.7 Seqüência de uso do **ONtoolkit**.

4.4. ONSL (Object Network Specification Language)

Durante o desenvolvimento da ferramenta, para satisfazer os requisitos levantados que previam uma especificação robusta e independente de plataforma, foi necessário definir uma linguagem para especificar todos os componentes de uma rede de agentes. Esta linguagem foi chamada de *Object Network Specification Language* (ONSL) e provê os *building blocks* para especificar cada uns dos elementos que compõem a rede de agentes:

- **Classes**, permitem definir as classes que serão utilizadas na rede de agentes.
  - Variáveis
  - Funções de transformação
  - Portas privadas de entrada/saída
- **Rede**, permite definir a topologia da rede de agentes.
  - Lugares
  - Arcos
  - *Kernel*

4.4.1.1. Sintaxe da linguagem

Nesta seção será apresentada a sintaxe da linguagem especificada para definir as rede de agentes.

Importando classes que estão em outro pacote (*package*) diferente daquele onde é definida a rede de agentes.

import <nome do pacote>.< * | nome da classe >;

Exemplos:

- Importar a classe `xInteger` disponível no pacote de distribuição do **ONtoolkit**.

```
import netobj.MTON.user.lang.xInteger;
```

- Importar todas as classes disponíveis no pacote anterior.

```
import netobj.MTON.user.lang.*;
```

As próximas definições (variáveis, funções de transformação e portas privadas de entrada e saídas) só podem ser feitas dentro da definição de classe que será exposta posteriormente.

Definindo uma variável

[transient] var <nome da variável> **type** <nome da classe>;

O *nome da variável* e o *nome da classe* têm que obedecer as regras de definição de um **identificador** de qualquer linguagem de programação, especialmente as do Java.

Exemplos:

```
var v1 type Class1;
var v2 type Class_2;
```

Definindo uma função de transformação

function <nome da função> <**from** <lista de variáveis> | **to** <lista de variáveis>>
[**match from** <lista de variáveis>];

O nome da função também tem que estar de acordo com as regras de definição de **identificador**. Sempre será obrigatório definir pelo menos o domínio (*from*) ou contradomínio (*to*) da função. Sempre que seja definido o domínio da função deverá ser definido o domínio da função de *matching* (*match from*) associada a esta função de

transformação. A lista de variáveis consiste em um conjunto de um ou mais nomes de variáveis que estiverem definidas no contexto da classe a qual pertence esta função.

Exemplos:

```
function add from v1, v2 to v3 match from v1, v2;
function random to v1;
function kill from v2 match from v2;
```

Definindo as portas privadas de entrada

input <número da porta> **to** <nome da variável associada>;

O *número da porta* é um número inteiro consecutivo começando sempre por 1. O *nome da variável associada* tem que seguir as regras de definição de um **identificador**.

Exemplos:

```
input 1 to v1;
input 2 to v2;
```

Definindo as portas privadas de saída

output <número da porta> **to** <nome da variável associada>;

De maneira equivalente à definição das portas privadas de entrada, o *número da porta* é um número inteiro consecutivo começando sempre por 1. O *nome da variável associada* tem que seguir as regras de definição de um **identificador**.

Exemplos:

```
output 1 to v1;
output 2 to v2;
```

Definindo uma classe externa

class <nome da classe> **is** <nome da classe externa>;

Tanto o *nome da classe* como o *nome da classe externa* têm que cumprir as restrições colocadas na definição de um **identificador**.

Exemplos:

```
class INTEGER is xInteger;
class DOUBLE is xDouble;
```

Definindo uma classe interna

```
class <nome da classe> {
    <definiçaõ do corpo da classe>
};
```

Como explicado no caso anterior o *nome da classe* tem que satisfazer as restrições colocadas na definição de um **identificador**. A *definição do corpo da classe* é um conjunto de definições de variáveis, funções de transformação e das respectivas portas privadas de entrada e/ou saída. Caso seja definida uma função de transformação então devem ser definidas as portas privadas de entradas e/ou saída referentes a seu domínio e/ou contradomínio, respectivamente.

Exemplo:

```
class Exemplo {
      var v1 type INTEGER;
      var v2 type DOUBLE;
      var v3 type DOUBLE;

      function add from v1, v2 to v3 match from v1, v2;

      input 1 to v1;
      input 2 to v2;
      output 1 to v3;
};
```

As próximas sentenças permitem a definição da topologia da rede de agentes. Começaremos pela definição dos elementos (lugares, arcos) que permitirão definir a topologia da rede a ser especificada.

```
place <nome do lugar> type <nome da classe associada>
                      ports (<número de portas públicas de entrada>,
                      <número de portas públicas de saída>)
                      [{<posição do lugar>, <diâmetro do lugar>,
                       <posição do nome do lugar>,
                       <posição do nome da classe associada ao lugar>}];
```

O *nome do lugar* e o *nome da classe associada* são identificadores. A classe associada determina o tipo do lugar, ou seja, define que tipo de objetos podem ser armazenados nesse lugar. As portas (*ports*) definem a quantidade de portas públicas de entrada e saída associadas a este lugar. As características gráficas do lugar são opcionais, mas se a rede será visualizada pelo **ONtoolkit** é obrigatória sua definição. As posições são definições de pontos no formato (x,y).

Exemplos:

```
place place_1 type C1 ports (0,2);
place place_2 type C2 ports (0,0);
      { (80,100), 20, (70,85), (70,135) };
place place_3 type C1 ports (1,1)
      { (120,100), 20, (110,85), (110,135) };
place place_4 type C1 ports (1,0)
  { (200,100), 20, (110,85), (110,135) };
```

Definindo os arcos que relacionam os lugares.

```
arc <nome do arco> from <nome do lugar de origem>
                   (<public | private> <número da porta de saída associada>)
                   to <nome do lugar de destino>
                   (<private | public> <número da porta de entrada associada>)
                   [{ <posição do nome do arco>, <true | false>,
                     <lista dos pontos intermediários do arco> }] ;
```

Como em todos os casos, o *nome do arco* e os nomes dos lugares (origem e destino) associados respondem às características de um **identificador**. Lembre-se que não são válidos aqueles arcos *public–public* e *private–private* e que o número das portas associadas começam em 1. Todas as definições de posição e/ou pontos estão no formato de definição de ponto como um par ordenado (x,y).

Exemplos:

```
arc P3_P2 from P3(private 1) to P2(public 1)
       {(209,124), false};
arc P2_P4 from P2(public 1) to P4(private 1)
       {(279,99), true, (276,38)};
```

Definindo os objetos do *kernel* da rede de agente.

```
kernel <nome do lugar> <lista de nome de objetos> ;
```

O *nome do lugar* obedece também às regras de um **identificador** e especifica o nome do lugar para o qual está se definindo os objetos que estarão inicialmente na rede e são parte da definição do *kernel* da rede de agentes. A *lista de nome de objetos*, consiste em um conjunto de um ou mais nomes de objetos que especificam a quantidade de objetos a serem colocados nesse lugar no inicio da rede de agentes e o nome de cada um deles. Cada nome de objeto também satisfaz a regras de definição de um **identificador**. Os nome de objetos são únicos para toda a rede de agentes.

Exemplos:

```
kernel P1 peca1;
kernel P3 ferramenta1;
kernel P4 ferramenta2;
```

Definindo a rede de agentes.

```
network <nome da rede de agente> {
    <definição da topologia da rede de agentes>
}
```

O *nome da rede de agente* é um **identificador** e especifica o nome da rede de agente, sendo utilizado este nome para criar o pacote que conterá todas as classes (Java) que especificam a rede de agentes. A definição da topologia da rede de agente é um conjunto de definições de lugares, arcos e dos objetos do *kernel*.

Exemplo:

```
network RedeExemplo {
   place P1 type peca ports (1,1)
      {(81,127), 20, (59,105), (59,161)};
   kernel P1 peca1;

   place P2 type peca ports (1,1)
      {(276,128), 20, (254,106), (254,162)};

   place P3 type ferramenta ports (0,0)
      {(180,128), 20, (158,106), (158,162)};
   kernel P3 ferramenta1;

   place P4 type ferramenta ports (0,0)
      {(182,38), 20, (160,16), (160,72)};
   kernel P4 ferramenta2;

   arc P1_P3 from P1(public 1) to P3(private 1)
      {(121,118), true};
   arc P3_P2 from P3(private 1) to P2(public 1)
      {(209,124), false};
   arc P2_P4 from P2(public 1) to P4(private 1)
      {(279,99), true, (276,38)};
   arc P4_P1 from P4(private 1) to P1(public 1)
      {(161,32), false, (80,38)};
}
```

4.4.1.2. Exemplo de definição de rede de agentes através da linguagem de especificação

Nesta seção é mostrada a definição de uma rede de agentes que simula uma rede de Petri simples por meio da linguagem de especificação de rede de objetos (*ONSL*). Na figura 4.8 é mostrada a visualização gráfica da rede de agentes especificada.

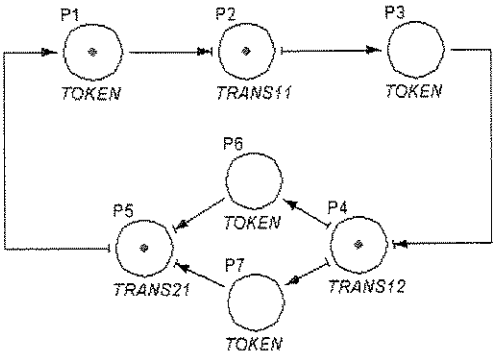


Figura 4.8 Rede de agentes que simula uma rede de petri simples.

```
class TOKEN {}

class TRANS11 {

   transient var iToken type TOKEN;
   transient var oToken type TOKEN;

   function fire from iToken to oToken match from iToken;

   input 1 to iToken;

   output 1 to oToken;
}

class TRANS12 {

   transient var iToken type TOKEN;
   transient var oTokenA type TOKEN;
   transient var oTokenB type TOKEN;

   function fire from iToken to oTokenA, oTokenB
        match from iToken;

   input 1 to iToken;

   output 1 to oTokenA;
   output 2 to oTokenB;
}

class TRANS21 {

   transient var iTokenA type TOKEN;
   transient var iTokenB type TOKEN;
   transient var oToken type TOKEN;

   function fire from iTokenA, iTokenB to oToken
        match from iTokenA, iTokenB;

   input 1 to iTokenA;
   input 2 to iTokenB;
                                                    cont...
```

```
   output 1 to oToken;
}

network petri1 {

   place P1 type TOKEN ports (1,1)
      {(88,78), 20, (66,56), (66,112)};
   kernel P1 token1;

   place P2 type TRANS11 ports (0,0)
      {(196,78), 20, (174,56), (174,112)};
   kernel P2 trans111;

   place P3 type TOKEN ports (1,1)
        {(316,78), 20, (294,56), (294,112)};

   place P4 type TRANS12 ports (0,0)
      {(275,216), 20, (253,194), (253,250)};
   kernel P4 trans121;

   place P5 type TRANS21 ports (0,0)
      {(123,217), 20, (101,195), (101,251)};
  kernel P5 trans211;

   place P6 type TOKEN ports (1,1)
        {(201,170), 20, (179,148), (179,204)};

   place P7 type TOKEN ports (1,1)
        {(202,256), 20, (180,234), (180,290)};

   arc P1_P2 from P1(public 1) to P2(private 1)
      {(93,86), false};
   arc P2_P3 from P2(private 1) to P3(public 1)
      {(198,85), false};
   arc P4_P6 from P4(private 1) to P6(public 1)
      {(258,199), false};
   arc P4_P7 from P4(private 2) to P7(public 1)
      {(259,227), false};
   arc P6_P5 from P6(public 1) to P5(private 1)
      {(184,179), false};
   arc P7_P5 from P7(public 1) to P5(private 2)
      {(190,249), false};
   arc P3_P4 from P3(public 1) to P4(private 1)
      {(345,74), false, (369,78), (369,215)};
   arc P5_P1 from P5(private 1) to P1(public 1)
      {(102,213), false, (26,217), (26,78)};
}
```

4.5. Criando uma rede de agentes com o ONtoolkit

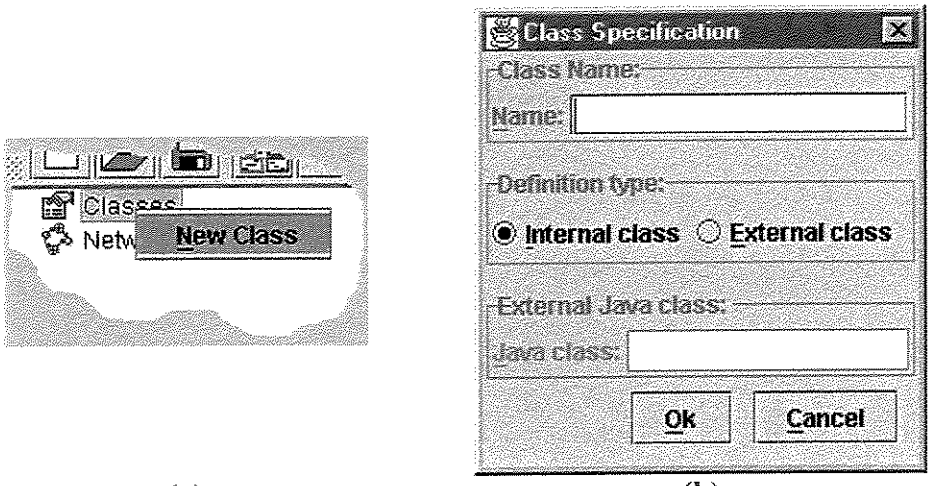
As redes de agentes são construídas através do módulo de edição do **ONtoolkit**. È aconselhável que o usuário tenha pelo menos uma noção inicial da rede a ser especificada, antes de começar a editá-la no **ONtoolkit.** Isso é recomendável, pois o processo de edição

das redes de agentes no software incorpora também funções de validação, seguindo com rigor as especificações das redes de agentes. Com isso, existem algumas regras bem extritas quanto à formação das classes e dos elementos que a compõem, bem como com a especificação dos lugares e dos arcos que os conectam.

O primeiro passo na construção da rede de agente é a definição das classes ou de alguma classe. Sem a definição de pelo menos uma classe não será possível a definição de nenhum elemento da topologia da rede, pois estes elementos estão estritamente ligados às definições das classes e os elementos que a compõem como pode ser observado na descrição da linguagem de especificação das redes de agentes ONSL.

4.5.1.1. Especificando uma classe no ONtoolkit

Para fazer a especificação de uma classe basta dar um *click* no botão direito do *mouse* sobre o elemento **Classes** na *Área de Edição das Classes e Kernel* do modo de edição do **ONtoolkit** (ver figura 4.3). Deve-se, em seguida, escolher a opção **New Class** (figura 4.9a) e preencher as especificações referentes ao nome da classe e tipo da classe (interna ou externa). Caso a classe seja externa, também será necessário especificar o nome da classe externa. (figura 4.9b).



(a) **(b)**

Figura 4.9 Construção de uma classe. (a) Comando para invocar a criação da uma classe. (b) Especificação da classe.

Após serem definidas as especificações da classe, se o tipo da classe é interna, pode-se ainda definir as variáveis, funções de transformação e as portas privadas de entrada e saída da classe.

Para invocar a definição de uma variável para uma classe, será necessário selecionar primeiramente a classe onde será definida a variável e fazer um *click* no botão direito do *mouse* na classe especificada e escolher a opção **New** - item **Var definition** (figura 4.10a) e preencher as especificações de nome e tipo da variável (figura 4.10b).

Se a classe definida é interna e vai ser uma classe ativa (com função ou funções de transformação), após serem definidas as variáveis da classe, pode-se definir suas funções de transformação e portas privadas de entrada e saída. Isto porque a definição de funções de transformação vai utilizar a definição das variáveis para poder definir seu domínio e contradomínio, assim como o domínio da função de *matching* associada à função de transformação a ser definida. O mesmo ocorre com as portas privadas de entrada e saída. Só que estas não somente estarão associadas às variáveis definidas na classe, mas estarão associadas àquelas variáveis que pertençam ao domínio (para o caso das portas privadas de entrada) ou ao contradomínio (para o caso das portas privadas de saída) de alguma função de transformação.

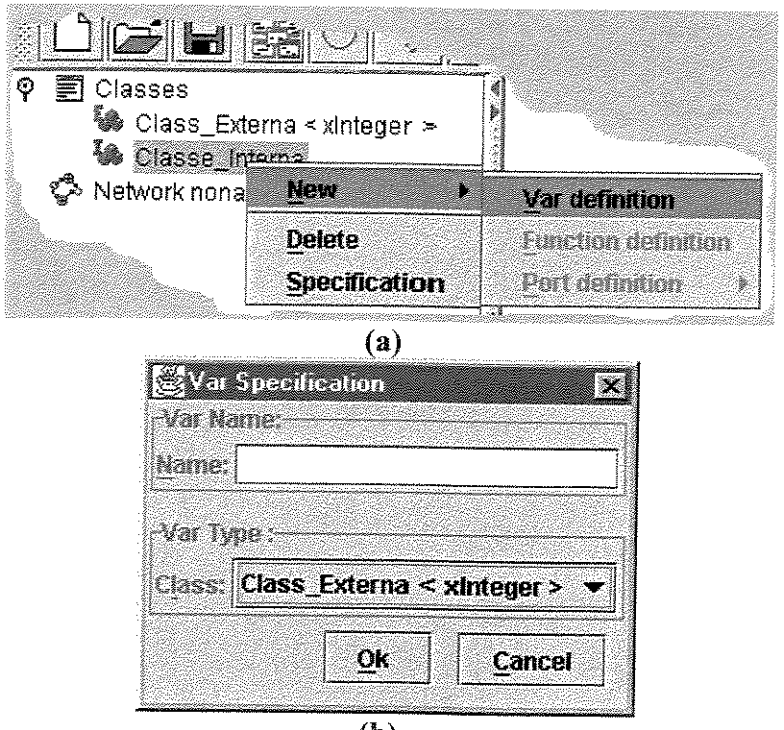
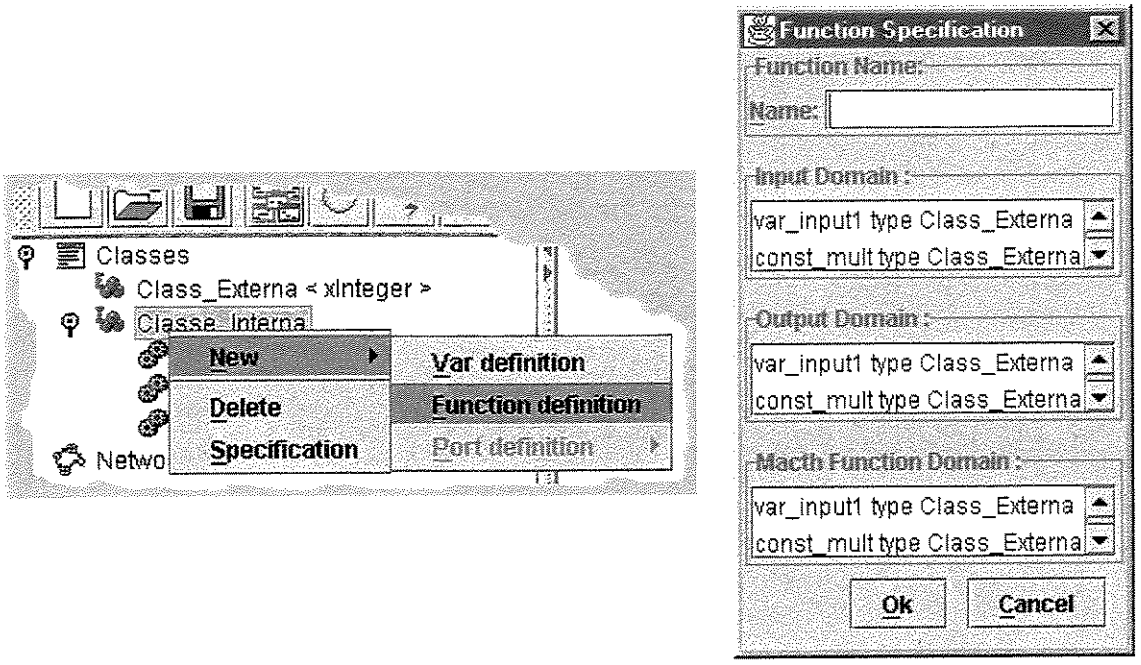


Figura 4.10 Definindo uma variável. (a) Invocando a definição de variável. (b) Especificando uma variável.

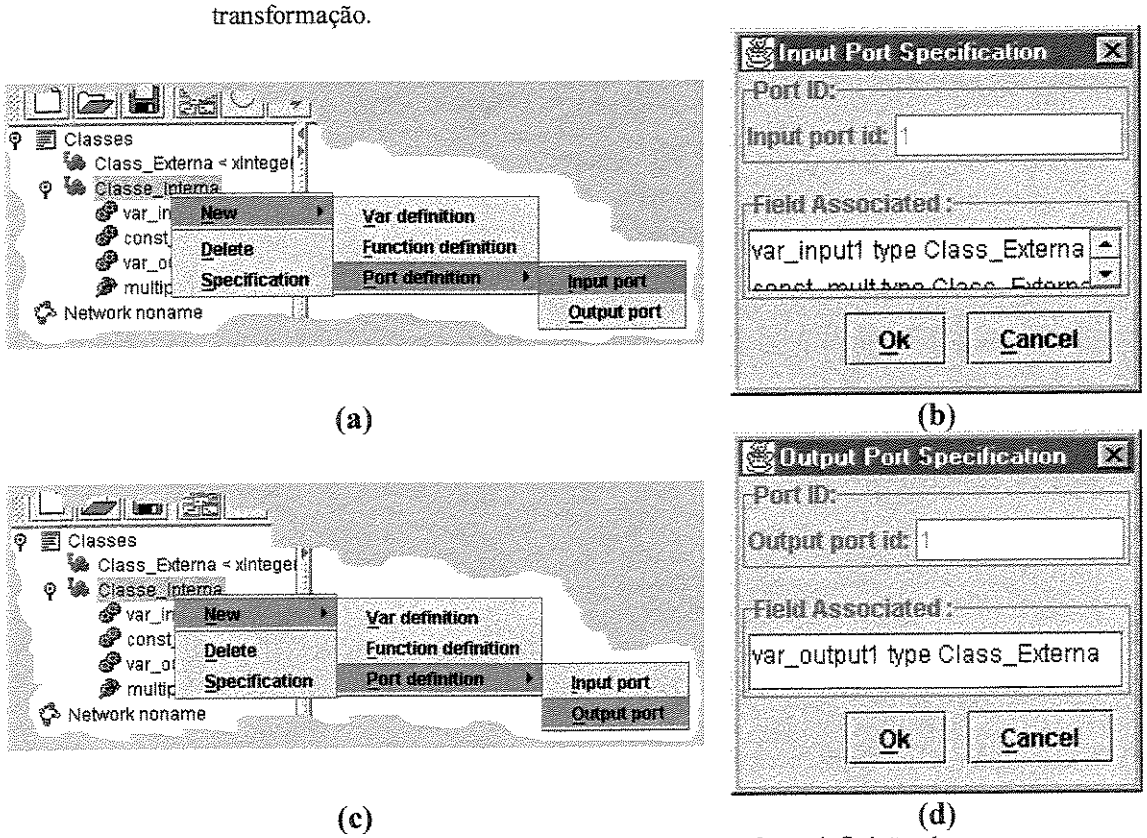
Para definir uma função de transformação segue-se o mesmo procedimento que para a definição das variáveis, só que o submenu a ser escolhido é **Function definition** (figura 4.11a). As especificações da função de transformação são colocadas na janela mostrada na figura 4.11b.

Após serem definidas as funções de transformação, deve-se habilitar as opções de definição das portas. Para definir as portas privadas de entrada e saída é escolhido o submenu **Port definition** seguindo-se o mesmo procedimento que para a definição das variáveis. Neste caso o usuário escolherá o tipo de porta privada a ser definida, **Input port** ou **Output port**, para as portas de entrada e saída respectivamente. (figura 4.12)



(a) **(b)**

Figura 4.11 Definindo uma função de transformação. (a) Invocando a definição de função de transformação. (b) Especificando uma função de transformação.



(a) **(b)**

(c) **(d)**

Figura 4.12 Definindo as portas privadas. (a) Invocando a definição da porta privada de entrada. (b) Especificando a porta privada de entrada. (c) Invocando a definição da porta privada de saída. (d) Especificando a porta privada de saída.

As classes e os elementos que a compõem (para o caso das classes internas) podem ser modificados ou removidos a qualquer instante, sempre que eles não tenham vínculo com outras definições feitas na rede. Para fazer estas operações basta escolher o elemento ou classe desejada e fazer um *click* do botão direito do *mouse* sobre o elemento escolhido e selecionar a opção desejada: **Delete** (para remover o item) ou **Specification** (para modificar as especificações do item). (figura 4.13).

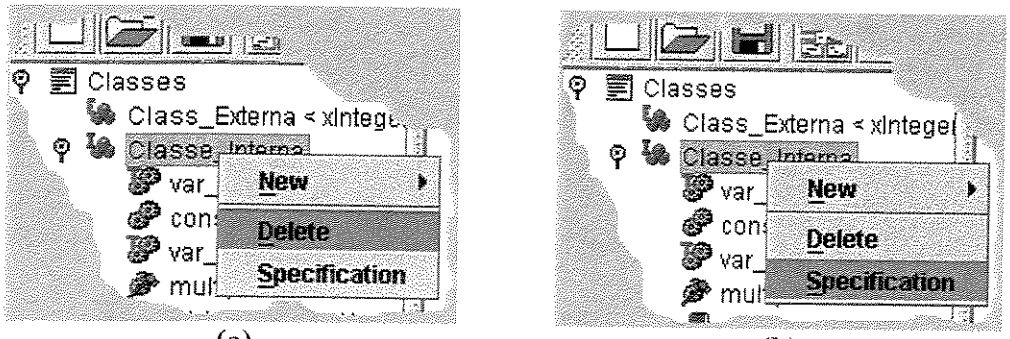


Figura 4.13 Transformando uma classe ou um elemento da classe. (a) Removendo a classe ou elemento. (b) Modificando a especificação da classe ou elemento.

4.5.1.2. Especificando a topologia da rede de agentes no ONtoolkit

O primeiro passo é especificar um nome para a rede de agentes que está sendo definida. Este nome é utilizado para identificar a rede. Por isso, é aconselhável colocar um nome que possa identificá-la de maneira representativa. Além disso, este nome é utilizado para criar o nome do pacote (*pakage*) Java que vai conter todas as classes que serão definidas para a rede de agentes. Para definir ou modificar o nome da rede de agentes (que sempre é inicializado quando se está criando uma rede nova com o nome genérico de *noname*), basta selecionar o item **Network** na *Área de edição das classes e kernel* (ver figura 4.4), fazer um *click* do botão direito do *mouse* neste item e escolher a opção **Specification** (figura 4.14).

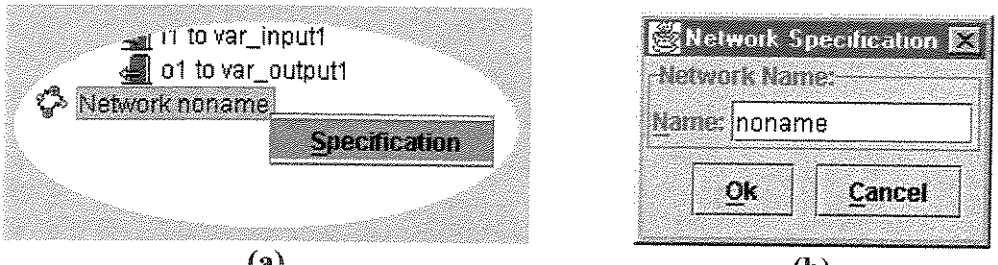


Figura 4.14 Modificando o nome da rede de agentes. (a) Invocando o comando de modificação. (b) Especificando o nome da rede de agentes.

Diferentemente da definição das classes, variáveis, funções e portas, os elementos que definem a topologia das rede de agentes (lugares e arcos) são inseridos por meio das *Ferramentas de edição* da *topologia e a Área de edição da topologia* (figura 4.3). Para

definir um lugar basta fazer um *click* do *mouse* no botão de **Insert place** (figura 4.15a) ou através do menu **Edit** item **Insert place ... .** Logo, são especificadas as características (nome e classe) deste lugar (figura 4.15b). Após definidas estas especificações é definida a posição do lugar na rede de agentes por meio de outro *click* do *mouse* na área de edição da topologia na posição desejada.

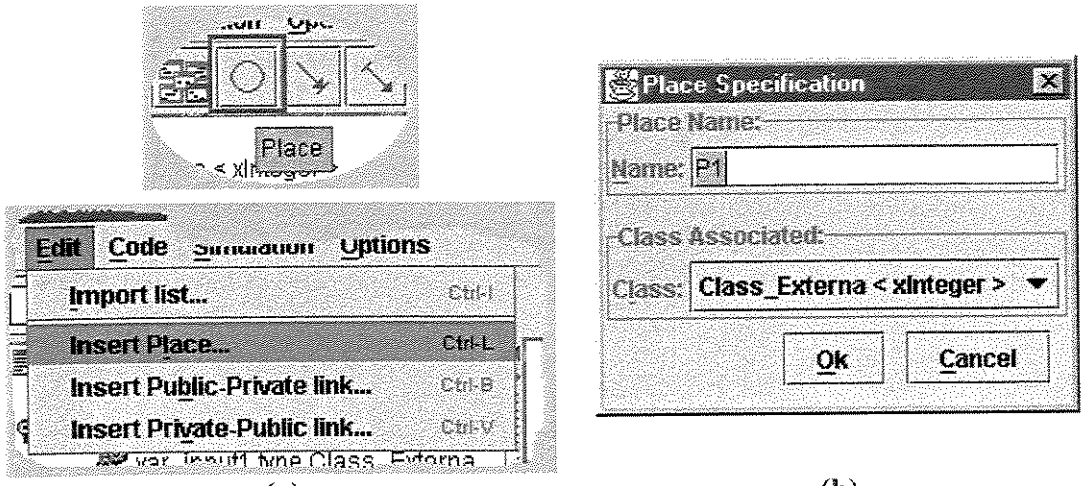


Figura 4.15 Inserindo um lugar na rede de agentes. (a) Invocando o comando de inserção de lugar. (b) Especificando um lugar.

Após definido pelo menos um lugar, podem ser definidos os arcos de tipo *public-private* e/ou *private-public*. Para definir um arco (qualquer deles), basta fazer um *click* do *mouse* nos botões correspondes à inserção de um arco *public-private* ou vice-versa.

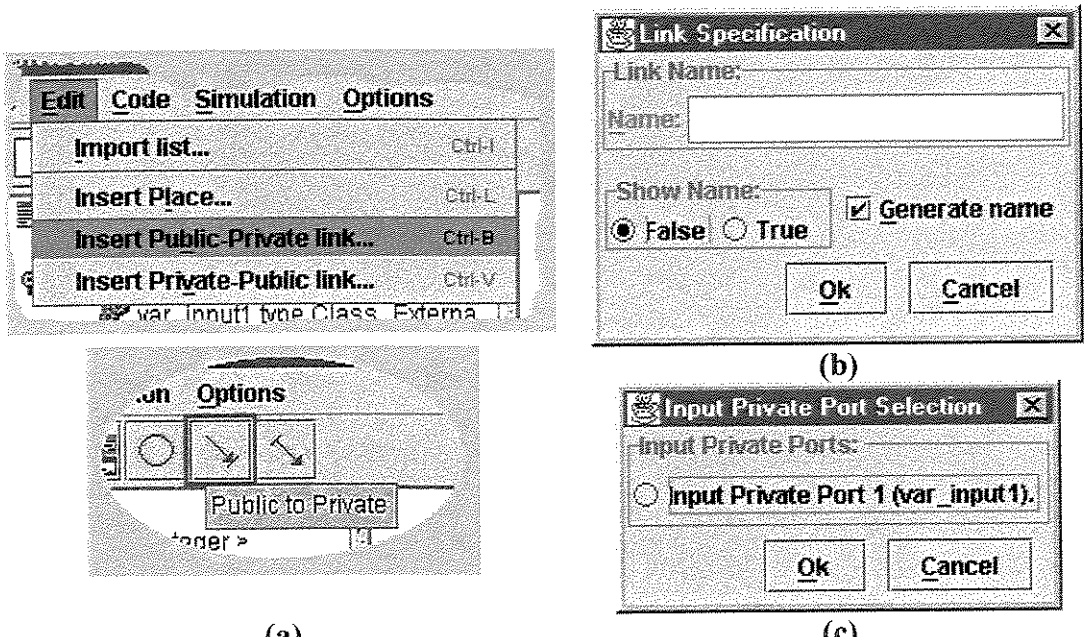


Figura 4.16 Definindo um arco *public-private*. (a) Invocando a definição do arco. (b) Especificando o arco. (c) Associando o arco a uma porta privada de entrada.

Outra maneira de fazer isso é invocar no menu **Edit** o item correspondente ao tipo de arco que se deseja criar (ver figuras 4.16a e 417a). Após escolhido o tipo de arco a ser criado, deve-se proceder à especificação do mesmo (nome - pode ser escolhido automaticamente ou pelo usuário, e se este nome será mostrado ou não na rede.).
A janela utilizada para especificar as características dos arcos é a mesma para os dois tipos de arcos e mostrada nas figuras 4.16b e 4.17b. Depois de serem colocadas as características do arco que está sendo definido, deve-se escolher o lugar de origem e o lugar de destino do arco, clicando nos respectivos lugares ou lugar, seguindo a ordem origem – destino. Estes arcos podem ter pontos intermediários entre a origem e o destino. Caso o lugar de origem e destino do arco sejam os mesmos, será necessário ter pelo menos dois pontos intermediários. Durante o processo de seleção da origem e do destino são escolhidas as portas privadas de entrada (no caso *public-private*) ou saída (no caso *private-public*) associada ao arco (figura 4.16c e 4.17c).

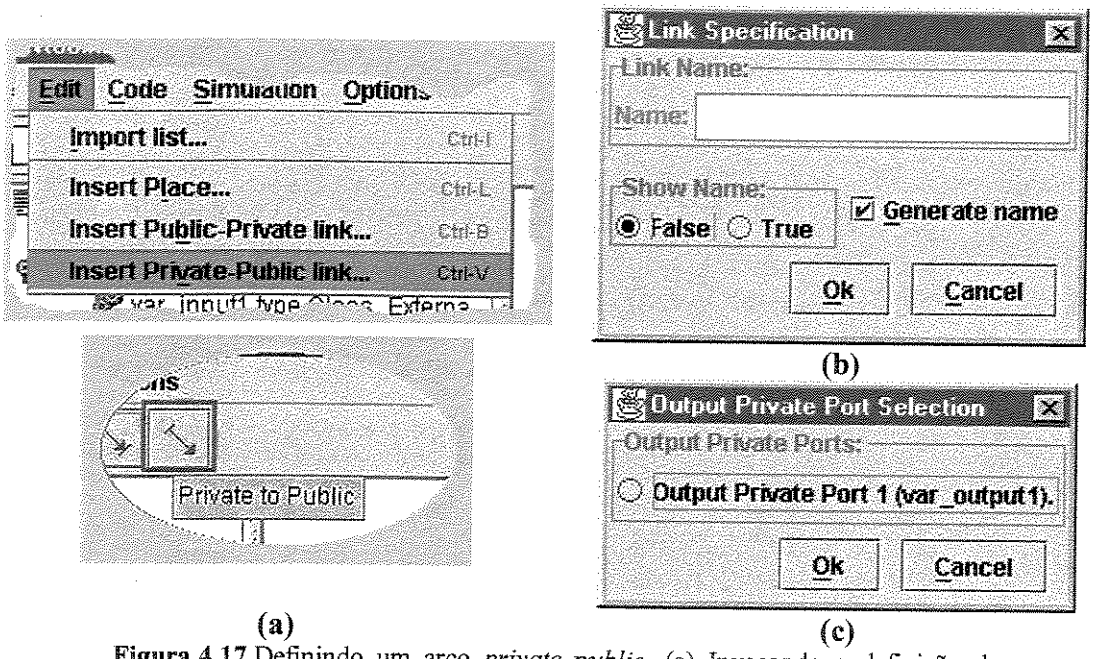


Figura 4.17 Definindo um arco *private-public*. (a) Invocando a definição do arco. (b) Especificando o arco. (c) Associando o arco a uma porta privada de saída.

O próximo passo é definir o *kernel* da rede de agentes. O *kernel* está associado aos lugares da rede, e por isso deve ser definido após a definição dos lugares. Para definir o *kernel* da rede basta definir os objetos que estarão inicialmente na rede de agentes. Para inserir um objeto em um lugar, deve-se escolher o lugar e dar um *click* do botão direito do *mouse* em cima do lugar escolhido (figura 4.18a). Logo, é especificado o nome do objeto que está sendo inserido nesse lugar e que formará parte do *kernel* da rede de agentes(figura 4.18b).

Os elementos da topologia também podem ser modificados ou removidos da mesma forma que os elementos de uma classe ou as próprias classes, tendo como peculiaridade que

eles podem também ser selecionados através de sua representação gráfica na *área de edição da topologia* da janela de edição do **ONtoolkit** (figura 4.3).

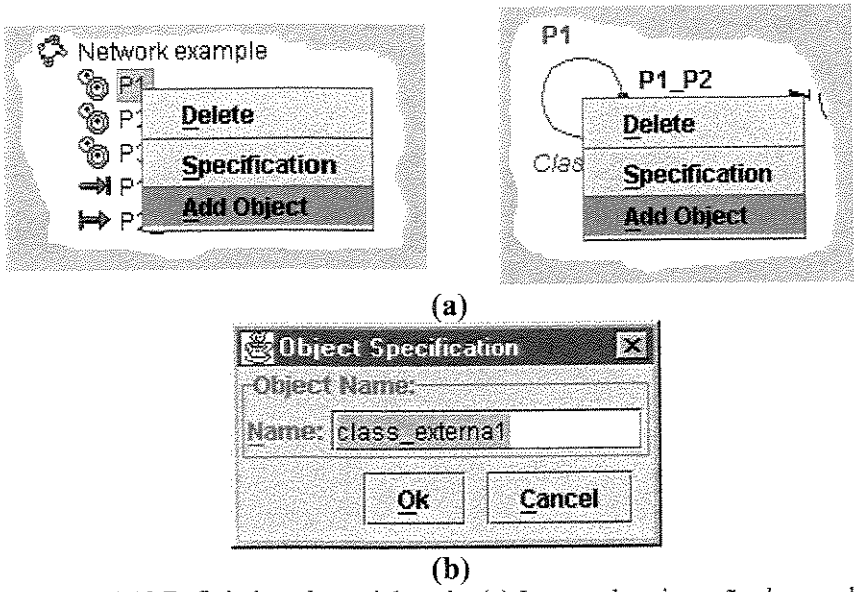


Figura 4.18 Definindo o *kernel* da rede. (a) Invocando a inserção de um objeto. (b) Especificando o objeto a ser inserido.

4.5.1.3. Inserindo o código de usuário

Os últimos passos na definição da rede de agentes, para que a mesma se encontre pronta para ser compilada, são os seguintes:

- inserir o código que implementa as funções de transformação dos objetos
- inserir o código em suas respectivas funções de *matching*
- inserir o código de inicialização dos objetos que formam o *kernel* da rede de agentes.

Para isso o usuário invocará o módulo de edição de código do *ON Desktop*, escolhendo o item **Edit code ...** no menu **Code**. Nesse instante será mostrada a janela de edição do código (figura 4.4) que apresenta as funções de transformação (função *perform*), funções de *matching* e os sub-*kernels* de cada lugar, que devem ser editadas. O usuário pode então escolher cada uma destas funções na árvore à esquerda, e editar o respectivo código na área de edição do código, à direita.

Na edição do código, para que o usuário tenha acesso às variáveis internas da classe em questão, dispomos de alguns métodos padrões que devem ser invocados pelo usuário em seu código sempre que for necessário ler ou escrever em tais variáveis. Para leitura, dispomos dos métodos `get` e `release`. Para escrita, utilizamos o método `put`.

leitura

`get` Lê o conteúdo do campo, através da obtenção de uma cópia do objeto nele disponível.

`release` Lê o conteúdo do campo, através da obtenção do próprio objeto disponível no campo. Neste caso, pode-se alterar os atributos do mesmo.

escrita

`put` Coloca um objeto no campo especificado

Estes métodos só podem ser invocados na definição do código de usuário na especificação do modelo da rede de agentes. A sintaxe destes métodos é a seguinte:

<get | release>_<nome do campo>();

put_<nome do campo>(<objeto ou valor>);

Exemplo:

```
...
xInteger x = get_var1();
String s = release_var2().toString();
put_var3(new xDouble(2.34));
put_var4(release_var1());
...
```

No caso da definição da função de *matching* é necessária ainda a utilização adicional de duas outras classes. A primeira delas é a classe `MatchResult`, que permite a definição do grau de interesse e o modo de acesso aos objetos a serem assimilados. Este último é definido utilizando-se a classe `AccessMode`.

A classe `AccessMode` é utilizada como estrutura de dados para armazenar o modo de acesso aos objetos a serem assimilados. Por este motivo, o único método que precisamos conhecer desta classe é seu construtor:

```
public AccessMode (boolean share, boolean destroy)
```

Os parâmetros deste construtor especificam se o mesmo será compartilhável (*true*) ou não (*false*) e consumível (*true*) ou não (*false*).

Os métodos fornecidos pela classe MatchResult para definir o grau de interesse da função de transformação associada à função de *matching* que está sendo definida são:

```
public MatchResult ();
```

Construtor da classe, permite a criação de um objeto desta classe.

`public void set(double value);`

Armazena o valor (*value*) do grau de interesse da função.

`public void setAccessMode(Object obj, AccessMode accessMode);`

Permite definir o modo de acesso `accessMode` ao objeto `obj`.

Toda função de *matching* necessariamente deverá retornar um objeto da classe `MatchResult` da seguinte maneira:

```
...
// cria um objeto da classe MathResult
MatchResult m = new MatchResult();
// colocando o grau de interesse da função
m.set(1.45);
// colocando o modo de aceso aos objetos a serem
// assimilados pela função de transformação.
m.setAccessMode(get_var1(),
                new AccessMode(true,false));
m.setAccessMode(get_var2(),
                new AccessMode(true,true));
return m;
```

4.6. Simulando uma rede de agentes com o ONtoolkit

Para ativar a simulação da rede de agentes, será necessário ter a rede compilada corretamente. Para isso, o usuário deve escolher o item **Compile...** no menu **Code**. Após a rede de agente ser compilada satisfatoriamente o usuário está habilitado para simular a rede de agentes. Para passar ao modo de simulação basta escolher o item **Start...** do menu **Simulation.** Em seguida, a janela de simulação (figura 4.5) será mostrada, com a rede de agentes a ser simulada, estando o módulo de simulação já em contato com o *MTON engine.* Ou seja, nesse instante a rede de agentes a ser simulada já está carregada no *MTON engine* e o console pronto a receber os comandos de simulação desejados. Todos os comandos do *console* (tabela 4.1) estão disponíveis para serem utilizados a partir do processo de simulação.

No processo de simulação, pode ser realizado ainda o *debuging* da rede de agentes em questão. Para isso o módulo de simulação oferece duas ferramentas para esse objetivo. Uma delas, a ferramenta de *watches*, pode ser ativada pressionando o botão **Watch** (figura 4.19a). Logo, é mostrada a janela dos *watches* (figuara 4.19b), onde podem ser inseridos ou removidos os comandos de *watch* desejados, que permitem visualizar os resultados obtidos durante a execução da rede.

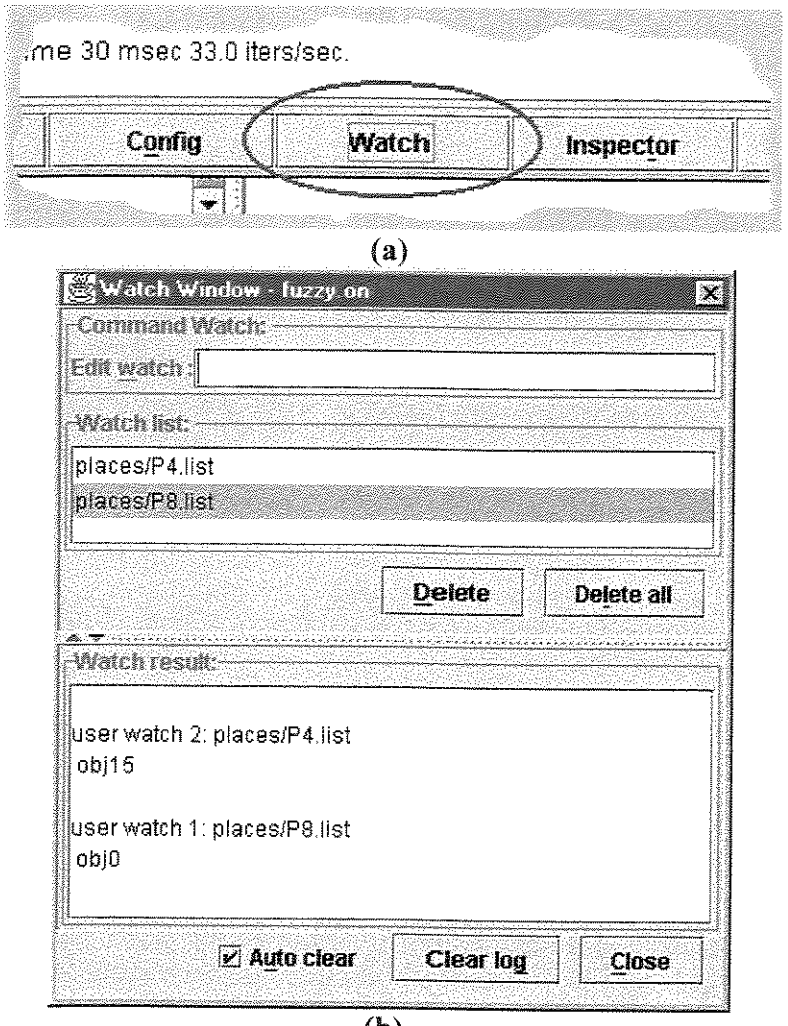
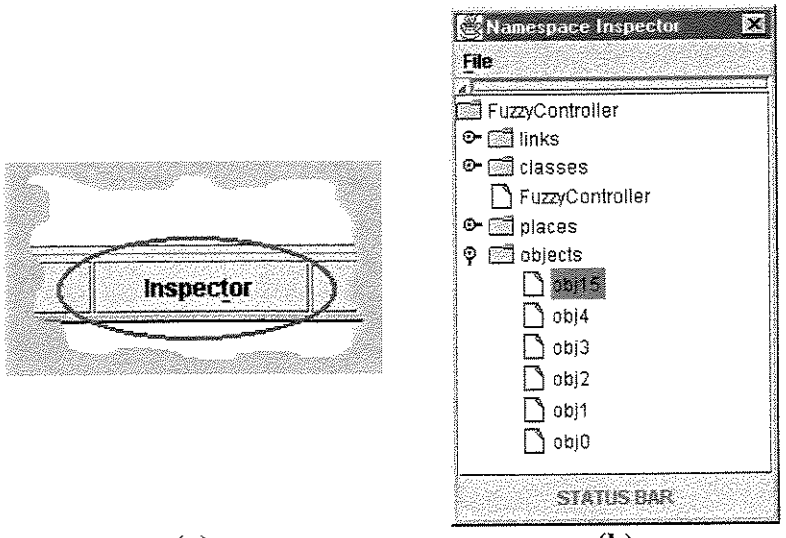


Figura 4.19 Utilizando a ferramenta *Watch*. (a) Invocando o *watch*. (b) Janela de edição e visualização dos *watches*.

A outra ferramenta de *debuging* oferecida pelo **ONtoolkit** é chamada de *inspectors*. Nesta, o usuário pode, a qualquer instante da simulação, inspecionar qualquer elemento da topologia e objetos contidos na rede de agentes. Para ativar o *inspector* basta somente selecionar o botão **Inspector** na janela de simulação (figura 4.20a). De imediato, será visualizada a janela de inspeção, (figura 4.20b) onde pode-se escolher o elemento a ser inspecionado.

Para encerrar a seção de simulação basta selecionar o botão **Close** da janela de simulação, o sistema descarrega a rede do *MTON engine*, fecha todas as ferramentas de *debugings* (*watches*, *inspectors* e *plugs*) que estiverem sendo utilizadas na simulação, assim como a conexão com o *console* e retorna à janela de edição da topologia da rede de agentes.



(a) (b)

Figura 4.20 Utilizando a ferramenta *Inspector*. (a) Invocando o *inspector*. (b) Janela de seleção dos elementos a serem inspecionados.

4.7. Resumo

Neste capítulo foi apresentado uma ferramenta computacional para o *design* e simulação de redes de agentes que permitem a elaboração de modelos em redes de agentes para as mais diferentes finalidades. Dentre outras, esta ferramenta pode ser utilizada por pesquisadores da área de Inteligência Artificial para a modelagem de sistemas inteligentes. Esta ferramenta implementa a teoria desenvolvida sobre as rede de agentes apresentada nos capítulos anteriores.

Descreveu-se a sintaxes da linguagem desenvolvida para a especificação das rede de agentes e os mecanismo utilizados no **ONtoolkit** para edição de uma rede de agentes, tanto das classes utilizadas nelas como da topologia que ela vai apresentar. Além disso, apresentou-se as ferramentas disponíveis no software para a simulação das rede de agentes, possibilitando fazer os ajuste necessários para que o sistema modelado pela rede de agentes tenha um desempenho melhorado.

Ofereceu-se o mínimo que o usuário deve conhecer do software para que ele este pronto para usá-lo de maneira correta e eficiente.

CAPÍTULO 5. Exemplos de Redes de Agentes

5.1. Introdução

Neste capítulo serão apresentados dois exemplo da aplicação das redes de agentes como ferramentas para o *design* e simulação de sistemas inteligentes. Estes exemplos foram desenvolvidos utilizando a ferramenta computacional descrita neste trabalho e apresentada no capítulo 4. Os exemplos escolhidos para ilustrar a utilização do software e das redes foram a solução ao problema do caixeiro viajante utilizando técnicas da computação evolutiva e o controle inteligente para a navegação de um veículo autônomo, os quais serão descritos a seguir.

5.2. Solução do problema do caixeiro viajante utilizando algoritmos genéticos

O problema do caixeiro viajante (*Traveler Salesman Problem, TSP*) é um problema tradicional da pesquisa operacional. O mesmo consiste em, dado um conjunto de N cidades (com localização conhecida), encontrar uma trajetória que percorra todas as cidades, em seqüência e sem repetição, e que apresente a menor distância percorrida (figura 5.1). O fato do problema ser muito simples de anunciar, não implica que seja simples também sua solução. Muitas técnicas já foram aplicadas para obter soluções rápidas e ótimas a este problema. Uma destas técnicas corresponde à aplicação dos algoritmos genéticos (GA). Diversos operadores genéticos (*crossovers* e mutações basicamente) foram desenvolvidos especialmente para serem aplicados em soluções para o TSP envolvendo algoritmos genéticos.

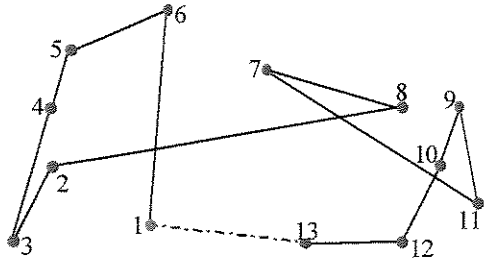


Figura 5.1 Problema do caixeiro viajante para 13 cidades.

Neste exemplo mostraremos duas versões de projetos de redes de agentes que simulam a aplicação de um algoritmo genético para obter as soluções ao problema do caixeiro viajante para 30, 100 e 101 cidades. Os dados referentes às cidades foram armazenados em arquivos texto (ncit30.dat, ncit100.dat e ncit101.dat) para cada problema. Os arquivos ncit*.dat são fornecidos com a seguinte configuração: na primeira linha é apresentado o número de cidades, e nas linhas seguintes, as coordenadas (x,y) de cada cidade.

A primeira versão envolve um algoritmo genético cuja rede de agentes representa separadamente cada indivíduo da população como um objeto. Na segunda versão, utiliza-se uma rede de agentes onde populações inteiras são encapsuladas dentro de um único objeto.

5.2.1. Algoritmo genético para resolver o TSP - Representação por Indivíduos

Nesta solução, trabalha-se com cada um dos indivíduos da população de forma isolada, exceto para o operador de *crossover*, que precisa de dois indivíduos.

A figura 5.2 mostra a rede de agentes que simula um algoritmo genético neste caso. Para todos os problemas (diferentes cidades) é utilizado o mesmo algoritmo genético.

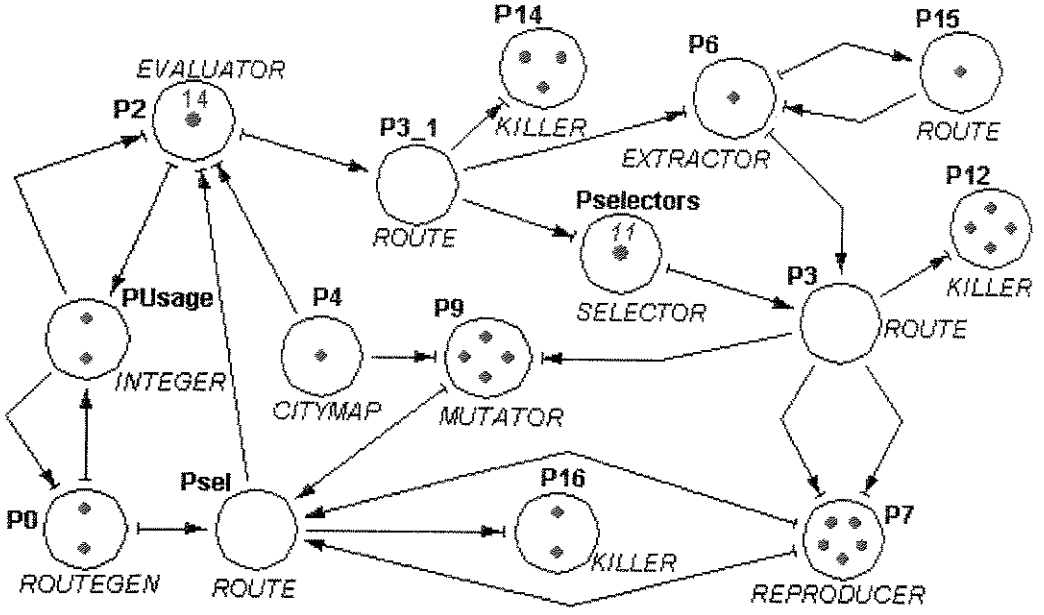


Figura 5.2 Rede de agentes para o TSP - Representação por Indivíduos.

No lugar **P0,** que tem como classe associada *ROUTEGEN,* temos dois objetos geradores de caminhos (ou percursos) de forma aleatória. O lugar **P4,** com classe associada *CITYMAP,* representa o mapa das cidades, ou seja, uma lista das cidades com suas coordenadas (x,y). Nos lugares **Psel**, **P3**, **P3_1** e **P15** que têm como classe associada *ROUTE*, representam-se as rotas a serem percorridas. Assim, cada indivíduo em um destes lugares, representa uma possível solução ao problema do caixeiro viajante, pois uma rota contém a ordem em que deverão ser percorridas as cidades. Os lugares **P12**, **P14** e **P16** contém os objetos que são responsáveis por manter a taxa de crescimento da população de forma adequada. Estes lugares estão associados à classe *KILLER*. O lugar **P7** vai conter objetos que atuarão como os operadores de *crossover*, este lugar está associado à classe *REPRODUCER*. Outro operador genético está representado no lugar **P9**, associado à classe *MUTATOR,* que representa os objetos que agirão como operadores de mutação. No lugar **Pselectors** que está associado à classe *SELECTOR*, representa os últimos operadores

genéticos utilizados: os operadores de seleção dos indivíduos. O lugar **PUsage** contém objetos da classe *INTEGER,* que representam contadores. Por último temos o objeto responsável pela extração da solução do problema (caminho de menor distância). Este objeto encontra-se localizado no lugar **P6** que tem a classe **EXTRACTOR** como classe associada.

A tabela 5.1 mostra as classes utilizadas na rede de agente para simular a solução para TSP através da utilização de um GA com representação para indivíduos.

Classe	**Descrição**	**Definição**
DOUBLE	Classe *wrapper* para importar a classe externa xDouble (representa um valor real)	DOUBLE < xDouble >
INTEGER	Classe *wrapper* para importar a classe externa xInteger (representa um valor inteiro)	INTEGER < xInteger >
ROUTE	Classe *wrapper* para importar a classe externa Route, que representa uma rota ou percurso a ser percorrido e por sua vez é a codificação utilizada para cada indivíduo do GA	ROUTE < Route >
CITYMAP	Classe *wrapper* para importar a classe externa CityMap, que armazena as especificações das cidades (posição x, posição y)	CITYMAP < CityMap >
KILLER	Classe projetada para manter o tamanho da população de forma limitada.	KILLER iRoute type ROUTE kill i1 to iRoute
EVALUATOR	Classe utilizada para avaliar as rotas geradas pelo gerador de percursos (ROUTEGEN). Calcula a distância resultante do percurso de toda as cidades seguindo a rota definida e determinando a rota que será regenerada. Se a rota a ser avaliada é muito antiga, ela será removida da rede.	EVALUATOR iroute type ROUTE icitymap type CITYMAP oroute type ROUTE iUsage type INTEGER oUsage type INTEGER eval kill i1 to iroute i2 to icitymap i3 to iUsage o1 to oroute o2 to oUsage

Classe	Descrição	Definição
ROUTEGEN	Classe responsável pela geração aleatória de novas rotas.	ROUTEGEN oroute type ROUTE iUsage type INTEGER oUsage type INTEGER generate i1 to iUsage o1 to oroute o2 to oUsage
MUTATOR	Classe responsável pelos operadores de mutação a serem utilizados sobre os indivíduos da população.	MUTATOR iroute type ROUTE oroute type ROUTE iCityMap type CITYMAP mutate_I mutate_CROSS i1 to iroute i2 to iCityMap o1 to oroute
SELECTOR	Classe utilizada para escolher o indivíduo (percurso) de maior *fitness* (menor distância).	SELECTOR iRoute type ROUTE oRoute type ROUTE select i1 to iRoute o1 to oRoute
EXTRACTOR	Classe encarregada de extrair a melhor solução ou melhor indivíduo da população.	EXTRACTOR iRoute type ROUTE oRoute type ROUTE oBestRoute type ROUTE iBestRoute type ROUTE extract i1 to iRoute i2 to iBestRoute o1 to oRoute o2 to oBestRoute
REPRODUCER	Classe responsável pelo operador de *crossover* a ser utilizado sobre os indivíduos da população.	REPRODUCER irouteA type ROUTE irouteB type ROUTE orouteA type ROUTE orouteB type ROUTE reproduce i1 to irouteA i2 to irouteB o1 to orouteA o2 to orouteB

Tabela 5.1 Classes utilizadas na definição da rede de agentes para solucionar o TSP utilizando um GA a nível de indivíduos.

O *kernel* desta rede de agentes está formado por 50 objetos, deles somente 2 precisam de código de inicialização do usuário. Um deles é uma rota gerada aleatoriamente e colocada no lugar **P15**, que será utilizado como o padrão a ser comparado para a obtenção da melhor solução ao problema. O código de inicialização para este objeto que pertence a classe *ROUTE* é simples e mostrado a seguir.

```
put(new Route(30));
```

Outro objeto do *kernel* da rede de agentes que precisa ser inicializado é o objeto que conterá a informação referente às cidades. O código dele também é simples e será mostrado em seguida.

```
put(new CityMap(new Resource("/data/ncit30.dat")));
```

Para utilizar a mesma rede para resolver o problema do TSP para o caso da 100 cidades e 101 cidades basta só modificar as linhas de código mostradas anteriormente. Para o caso do objeto referente ao percurso, basta modificarmos o `30` pelo `100` ou `101` respectivamente. Para o caso dos dados da cidades é só modificar o nome dos arquivos de dados `"/data/ncit30.dat"` por `"/data/ncit100.dat"` ou `"/data/ncit101.dat"` respectivamente. Além disso será preciso modificar a linha de código do usuário referente à rota no módulo *perform* da função de transformação `generate` da classe *ROUTEGEN* mostrada a seguir, colocando `100` e `101` onde está o `30` respetivamente.

```
//create the new route...
Route newRoute = new Route(30);
newRoute.setLastUpdate(getTime());
put_oroute(newRoute);

//update the usage counter...
xInteger newUsage = clone_iUsage();
newUsage.add(-1);
put_oUsage(newUsage);
```

A seguir é mostrado o código das funções de transformação e de *matching* associada.

Classe `SELECTOR`

Função de transformação: `select`

perform

```
put_oRoute(release_iRoute());
```

match

```
MatchResult m = new MatchResult();
m.set(get_iRoute().getFitness());
m.setAccessMode(get_iRoute(), new
AccessMode(false, true));
return m;
```

Classe EVALUATOR

Função de transformação: eval

Perform

```
Route route = release_iroute();
route.evalFitness(get_icitymap());
route.setLastUpdate(getTime());
put_oroute(route);
```

Match

```
MatchResult m = new MatchResult();
m.set(getTime() -
get_iroute().getLastUpdate());
m.setAccessMode(get_iroute(), new
AccessMode(false, true));
m.setAccessMode(get_icitymap(), new
AccessMode(true, false));
return m;
```

Função de transformação: kill

Perform

```
//update the usage counter...
xInteger newUsage = clone_iUsage();
newUsage.add(1);
put_oUsage(newUsage);
```

Match

```
MatchResult m = new MatchResult();
int wTime = getTime() -
get_iroute().getLastUpdate();
if (wTime > 20) {
     m.set(wTime+1);
     m.setAccessMode(get_iroute(), new
AccessMode(false, true));
     m.setAccessMode(get_iUsage(), new
AccessMode(false, true));
}
return m;
```

Classe KILLER

Função de transformação: kill

perform

match

```
MatchResult m = new MatchResult();
m.set(-1);
m.setAccessMode(get_iRoute(), new
AccessMode(false, true));
return m;
```

Classe EXTRACTOR

Função de transformação: extract

perform

```
System.err.println("-> best="+get_iRoute());
put_oRoute(clone_iRoute());
put_oBestRoute(release_iRoute());
```

match

```
MatchResult m = new MatchResult();
if (get_iRoute().getFitness() >
get_iBestRoute().getFitness()) {
     m.set(get_iRoute().getFitness()+1);
     m.setAccessMode(get_iRoute(), new
AccessMode(false, true));
     m.setAccessMode(get_iBestRoute(), new
AccessMode(false, true));
}
return m;
```

Classe MUTATOR

Função de transformação: mutate_I

perform

```
Route route = clone_iroute();
route.mutation_I();
route.setLastUpdate(getTime());
put_oroute(route);
```

match

```
MatchResult m = new MatchResult();
if (Math.random()<.1) {
     m.set(get_iroute().getFitness()*Math.r
andom());
     m.setAccessMode(get_iroute(), new
AccessMode(true, true));
}
return m;
```

Função de transformação: mutate_CROSS

perform

```
Route route = clone_iroute();
route.mutation_CROSS_P(get_iCityMap());
route.setLastUpdate(getTime());
put_oroute(route);
```

match

```
MatchResult m = new MatchResult();
if (Math.random()<.5) {
     m.set(get_iroute().getFitness()*Math.r
andom());
     m.setAccessMode(get_iroute(), new
AccessMode(true, true));
     m.setAccessMode(get_iCityMap(), new
AccessMode(true, false));
}
return m;
```

Classe ROUTEGEN

Função de transformação: generate

Perform

```
//create the new route...
Route newRoute = new Route(30);
newRoute.setLastUpdate(getTime());
put_oroute(newRoute);

//update the usage counter...
xInteger newUsage = clone_iUsage();
newUsage.add(-1);
put_oUsage(newUsage);
```

Match

```
MatchResult m = new MatchResult();
if (get_iUsage().getValue()>0) {
     m.set(1.0);
     m.setAccessMode(get_iUsage(), new
AccessMode(false, true));
}
return m;
```

Classe REPRODUCER

Função de transformação: select

perform

```
Route r1 =
Route.crossover_OX(get_irouteA(),
get_irouteB());
r1.setLastUpdate(getTime());
put_orouteA(r1);

Route r2 =
Route.crossover_OX(get_irouteB(),
get_irouteA());
r2.setLastUpdate(getTime());
put_orouteB(r2);
```

match

```
MatchResult m = new MatchResult();
if (Math.random()<.6) {
     m.set(get_irouteA().getFitness()*Math.
random());
     //m.set(1.0);
     m.setAccessMode(get_irouteA(), new
AccessMode(true, true));
     m.setAccessMode(get_irouteB(), new
AccessMode(false, true));
}
return m;
```

5.2.2. Algoritmo genético para resolver o TSP - Representação por Populações

Outra solução utilizada para resolver o problema do TSP é utilizar um GA onde representa-se populações inteiras como objetos. Neste caso, os operadores genéticos utilizados são aplicados a todos os indivíduos da população e não a alguns dos indivíduos como acontecia no caso exposto na seção anterior. Na figura 5.3 é mostrada a rede de agentes projetada para simular esta nova solução ao problema do TSP. Observe-se que é muito mais simples no sentido da topologia que a rede de agentes utilizada na solução descrita anteriormente. Mas, em termos de código, é bem mais sofisticada.

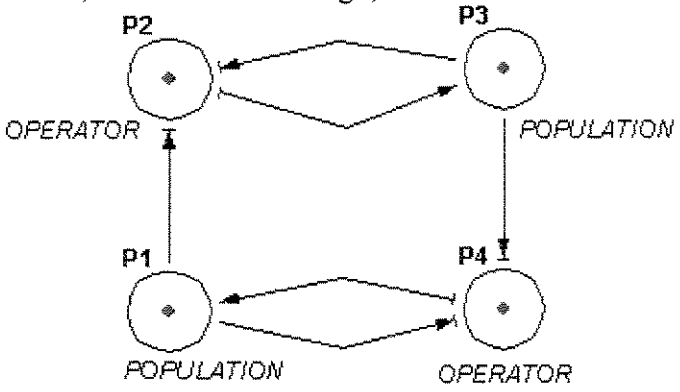


Figura 5.3 Rede de agentes para o TSP - Representação por Populações.

Neste caso temos somente 4 lugares. Dois deles, estão representando as populações a serem utilizadas e os outros dois os operadores genéticos a serem utilizados. Os lugares **P1** e **P3,** que estão associados à classe *POPULATION,* contém os objetos que representam as populações a serem usadas no GA. Nos lugares **P2** e **P4** temos os objetos que atuarão como os operadores genéticos que serão utilizados sobre as populações. Estes lugares estão associados à classe *OPERATOR.*

A seguir, na tabela 5.2, estão representadas as classes utilizadas para este exemplo.

Classe	Descrição	Definição
POPULATION	Classe *wrapper* para importar a classe externa MTONPopulation que representa uma população de indivíduos independente de codificação para ser utilizada por um GA.	POPULATION < MTONPopulation >
POP_OPERATOR	Classe *wrapper* para importar a classe externa MTONPopOperator, que representa um operador genético para trabalhar sobre uma população de indivíduos.	POP_OPERATOR < MTONPopOperator >

Classe	Descrição	Definição
ROUTE	Classe *wrapper* para importar a classe externa Route, que representa uma rota ou percurso a ser percorrido e por sua vez é a codificação utilizada para cada indivíduo do GA	ROUTE < Route >
CITYMAP	Classe *wrapper* para importar a classe externa CityMap, que armazena as especificações das cidades (posição x, posição y)	CITYMAP < CityMap >
WRAPPER	Classe *wrapper* para importar a classe externa Wrapper (classe genérica de *wrappers* para classes externas, utilizada para importar o *plug* a ser utilizado)	WRAPPER < Wrapper >
OPERATOR	Classe que representa os operadores genéticos a serem utilizados na solução do TSP utilizando um GA, onde estes operadores trabalham sobre populações de indivíduos.	OPERATOR iPopN type POPULATION oPopN type POPULATION operator type POP_OPERATOR iPopP type POPULATION plug type WRAPPER plug2 type WRAPPER plug3 type WRAPPER perform i1 to iPopN i2 to iPopP o1 to oPopN

Tabela 5.2 Classes utilizadas na definição da rede de agentes para solucionar o TSP utilizando um GA com representação baseada em populações.

O *kernel* para esta rede de agentes é inicializado com 4 objetos, um em cada lugar. Assim, temos dois objetos formando as populações e dois representando os operadores a serem utilizados neste exemplo. Todos estes objetos precisam de inicialização. Os códigos referentes à inicialização de cada um deles será mostrado a seguir. No primeiro caso é mostrado o código onde são inicializados os objetos das populações `population1` para o objeto no lugar **P1** e `population2` para o objeto no lugar **P3**. As populações são inicializadas com tamanho 100, o objeto `population1` vai conter 100 indivíduos, gerados aleatoriamente. O objeto `population2` inicialmente não contém indivíduos.

```
CityMap cityMap = new CityMap(new
                        Resource("/data/ncit30.dat"));

switch (kid) {
case KERNEL_population1:
   int P1SIZE=100;
   MTONPopulation population1 = new MTONPopulation(
                                                    P1SIZE);
     // Generates a initial random population
   for (int i=0; i<P1SIZE; i++)
      population1.addIndividual(new Route(
                                        cityMap.getSize()));
   population1.setParameter(cityMap);
   population1.setLastUpdate(1); // startup!!!
   put(population1);
   break;
case KERNEL_population2:
   int P2SIZE=100;
   MTONPopulation population2 = new MTONPopulation(
                                                    P2SIZE);
   population2.setParameter(cityMap);
   put(population2);
   break;
}
```

A seguir é apresentado o código de inicialização dos objetos que representam os operadores genéticos.

```
switch(kid) {
case KERNEL_operator1:
   RemotePlot2D p = (RemotePlot2D)createPlug(
                        "plug.lib.RemotePlot2DImpl");
   p.setXRange(0,100);
   p.setYRange(0,100);
   put_plug(new Wrapper(p));
   put_operator(new MTONPopOperator(
                        new IndSelection[]{
                              new IndSelBest(),
                              new IndSelRandom()},
                        new OperatorOX(.6)));

   RemotePlot p2 = (RemotePlot)createPlug(
                        "plug.lib.RemotePlotImpl");
   p2.listPlot(200,0,10);
   p2.plot(0);
   put_plug2(new Wrapper(p2));
   RemoteLog p3 = (RemoteLog)createPlug(
                        "plug.lib.RemoteLogImpl");
   put_plug3(new Wrapper(p3));

   break;
case KERNEL_operator2:
   put_plug(new Wrapper(null));
   put_plug2(new Wrapper(null));
                                                    cont...
```

```
   put_plug3(new Wrapper(null));
   put_operator(new MTONPopOperator(
                        new IndSelection[]{
                              new IndSelRandom()},
                        new OperatorCROSS(.6)));
break;
}
```

Neste caso são inicializados os operadores genéticos: operador OX para o caso do operador no lugar **P2** e o operador CROSS para o caso do operador no lugar **P4**. Além disso são inicializados os *plugs* que serão utilizados neste exemplo para mostrar os resultados obtidos pelo GA. Neste caso os *plugs* que estão sendo inicializados só serão utilizados pelo operador OX, ou seja, pelo objeto `operator1`.

A seguir é mostrado o código das funções de transformação e de *matching* associada.

Classe `OPERATOR`

Função de transformação: `eval`

perform

```
System.err.println("\nperforming...");

int lu = get_iPopP().getLastUpdate();
MTONPopulation pop = release_iPopN();
pop.clear();
get_operator().perform(get_iPopP(), pop);
pop.update();

//update time...
pop.setLastUpdate(pop.getLastUpdate()+1);
get_operator().setLastUpdate(lu);
put_oPopN(pop);

System.err.println("#time "+getTime()+
      "  best="+(1/pop.getMaxFitness())+
      "  avg="+(1/pop.getAvgFitness())+
      "  worst="+(1/pop.getMinFitness()));

if (get_plug().getData() == null)
     return;
//------------ plotting routine ------------
RemotePlot2D p =
(RemotePlot2D)get_plug().getData();
p.setEnabled(false);

CityMap map = (CityMap)get_iPopP().getParameter();

NodeArray nodes =
(NodeArray)pop.getBest().getChromossome();

                                        cont ...
```

Classe OPERATOR

Função de transformação: `eval`

perform

```
p.clear();
for (int i=0; i<map.getSize(); i++) {
  int c = nodes.getNode(i);
  p.plot(map.getX(c), map.getY(c));
}
p.autoScale();
p.setEnabled(true);
RemotePlot p2 = (RemotePlot)get_plug2().getData();
p2.plot(1/pop.getMaxFitness());
//----part #3
RemoteLog p3 = (RemoteLog)get_plug3().getData();
p3.clear();
p3.append("#time "+getTime()+
     "  best="+(1/pop.getMaxFitness())+
     "  avg="+(1/pop.getAvgFitness())+
     "  worst="+(1/pop.getMinFitness()));
//--------- end of the plotting -------
```

match

```
MatchResult m = new MatchResult();

System.err.println("get_iPopP().getLastUpdate()
="+
     get_iPopP().getLastUpdate());
System.err.println("get_operator().getLastUpdat
e()="+
     get_operator().getLastUpdate());

if (get_iPopP().getLastUpdate() ==
(get_operator().getLastUpdate()+1)) {
     m.set(1.0);
     m.setAccessMode(get_iPopN(), new
AccessMode(false, true));
     m.setAccessMode(get_iPopP(), new
AccessMode(true, false));
}
return m;
```

Esta rede de agentes também pode ser utilizada para resolver os problemas das 100 e 101 cidades. Para isso, basta modificar o arquivo de dados das cidades a serem utilizadas. Em outras palavras, basta modificar a primeira linha do código de inicialização dos objetos que representam as populações, trocando o nome do arquivo `ncit30.dat` por `ncit100.dat` ou `ncit101.dat` para que o problema seja resolvido.

5.2.3. Resultado das simulações

Os resultados que serão mostrados nesta seção foram obtidos de simulações feitas sobre a rede de agentes que resolve o problema do caixeiro viajante utilizando um algoritmo genético, onde seus operadores genéticos operam sobre populações de indivíduos, em

outras palavras utilizando o algoritmo genético com representação por populações. Os resultados para a outra rede são análogos a estes.

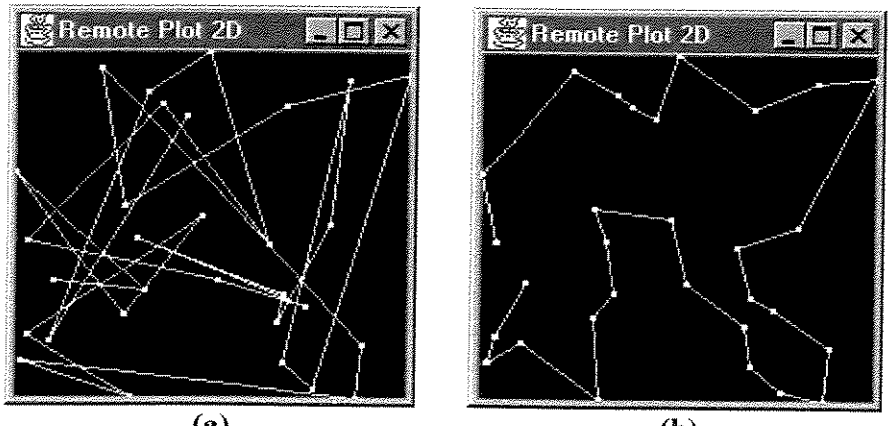
Os resultados aqui apresentados somente ilustram que as redes de agentes podem ser utilizadas para simular e projetar sistemas a qualquer nível de resolução. No entanto para a obtenção de melhores resultados, necessitar-se-á de maior número de iterações da rede.

Na tabela 5.3 são mostrados os resultados obtidos nas simulações feitas com a rede de agentes para o problema do TSP. Observe que somente para o caso do problema das 100 cidades é que a rede de agentes projetada não conseguiu atingir o valor ótimo conhecido para este problema. Para este caso, pode-se obter o valor ótimo utilizando a rede modificando alguns dos parâmetros do algoritmo genético utilizado, por exemplo, aumentando o tamanho da população (aspecto importante no trabalho com algoritmos genéticos), as taxas dos operadores genéticos utilizados e trocar ou incrementar o número dos operadores genéticos específicos para este tipo de problema. Estas recomendações podem servir ao leitor como exercício na utilização, compreensão e aprendizado das rede de agentes.

Nro. Cidades	Solução (conhecida)	Solução (rede de agentes)
30 cidades	48872.4026	**48872.40262872694**
100 cidades	21285.4432	21298.97774792846
101 cidades	65455.8441	**65455.84412271572**

Tabela 5.3 Resultados obtidos pela rede de agentes na solução ao problema do caixeiro viajante.

A figura 5.4 mostra o percurso ótimo para o problema das 30 cidades, resultado obtido a partir da simulação da rede de agentes e visualizado por meio de um *plug*. Este resultado foi obtido na oitava iteração da rede de agentes, quando se obteve o valor ótimo para este problema.



(a) **(b)**

Figura 5.4 Problema do caixeiro viajante para 30 cidades. (a) Percurso inicial. (b) Percurso de menor distância.

A seguir é mostrada a figura 5.5 que representa a solução obtida pela rede de agentes para o TSP no caso de 100 cidades. Esta solução, que não é a ótima, foi obtida após 90 iterações. Neste caso o GA atingiu um mínimo local e não conseguiu sair dele.

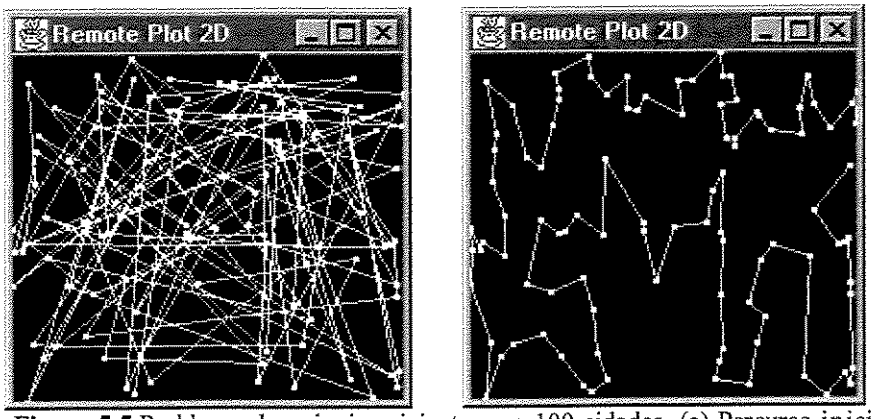


Figura 5.5 Problema do caixeiro viajante para 100 cidades. (a) Percurso inicial. (b) Percurso de menor distância.

O resultado para o problema do caixeiro viajante para 101 cidades, que convergiu após 30 iterações, corresponde à solução ótima conhecida para este problema, sendo mostrado na figura 5.6

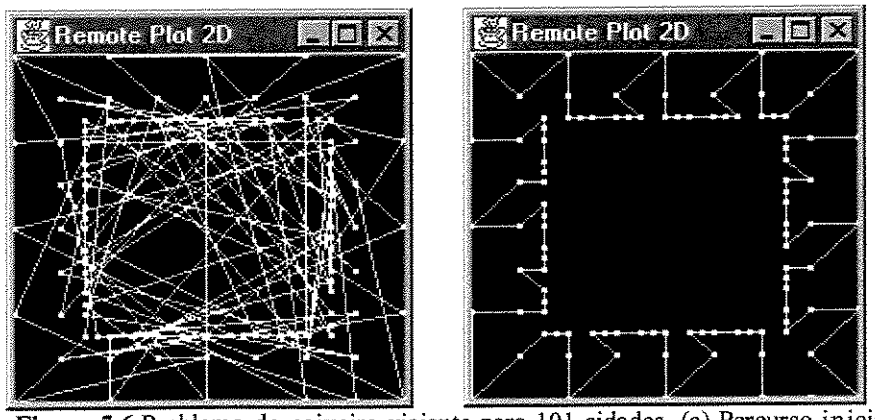


Figura 5.6 Problema do caixeiro viajante para 101 cidades. (a) Percurso inicial. (b) Percurso de menor distância.

5.3. Navegação simples de um veículo autônomo utilizando um controlador fuzzy

Neste exemplo é modelada a navegação simples de um veículo autônomo. Esta, é realizada por meio de um controlador fuzzy, responsável pelo controle dos movimentos do veículo para chegar até a meta. Neste exemplo o veículo não terá que lidar com obstáculos em seu ambiente. Por isso dizemos que é um problema de navegação simples. O ambiente do veículo é um quadrado de 100 unidades. A meta é chegar até o topo de seu ambiente (posição no eixo da y = 100 unidades) com o veículo alinhado perpendicularmente ao limite superior do ambiente. Os valores de entrada para o controlador são a posição e o ângulo do

veículo em relação ao eixo x (ou seja, x, y e azimute). O azimute é o grau de inclinação do veículo (ângulo da frente do veículo referente ao eixo de coordenadas), que está definido na faixa [-90°, 270°] (figura 5.7). As rodas do veículo só podem ser deslocada em um ângulo de 60 graus, ou seja, [-30°,30°]. O controlador será o responsável pela determinação do ângulo que deverá ser direcionado às rodas do veículo para movimentar-se até a próxima posição, deslocando-se a uma velocidade constante. A posição e ângulo inicial do veículo são geradas aleatoriamente.

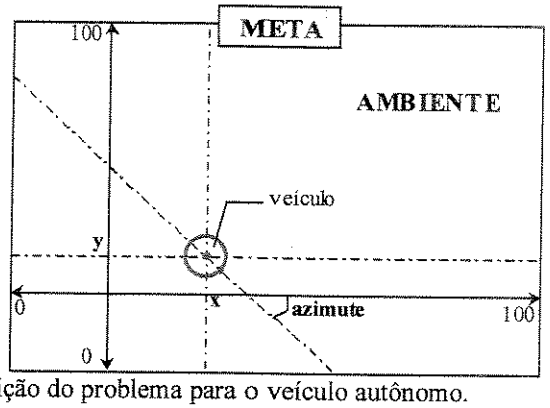


Figura 5.7 Descrição do problema para o veículo autônomo.

A figura 5.8 apresenta a rede de agentes projetada para simular este problema. No lugar **P1** temos a entrada do controlador formada pela posição (x,y) e o azimute do veículo. Para este lugar, está associada a classe *INPUT,* que contém estas informações do veículo. Em **P2** é definida a base de regras (classe associada *RULEBASE*), que é utilizada pela máquina de inferência para obter o conjunto *fuzzy* resultante. O lugar **P3** contém a máquina de inferência fuzzy (*Inference Engine*) do controlador. Este lugar está associado à classe *INFERENCE_ENGINE.* Ela tem como dados de entrada, um objeto da classe *INPUT* contendo as informações atuais do veículo e outro objeto da classe *RULEBASE,* representando o conjunto de regras *fuzzy* que serão utilizadas pela máquina de inferência para inferir o resultado a ser obtido. Como dados de saída ela também terá dois objetos. Um deles é um objeto da classe *INPUT,* que é colocado no lugar **P8.** Este, conterá uma copia exata da entrada, com o objetivo de alimentar o objeto em **P7** com os dados da entrada. O outro objeto de saída é da classe *FUZZYSET* e é colocado no lugar **P4.** Este, contém o conjunto *fuzzy* resultante da inferência realizada pela máquina de inferência. Em **P5** é definido o objeto responsável pela defuzzyficação do conjunto *fuzzy* obtido no lugar **P4**. A classe associada a este lugar é *DEFUZZYFIER* que terá como objetivo obter o valor do ângulo das rodas do veículo, colocando-o no lugar **P6**, a partir do conjunto *fuzzy* gerado em **P4** pela máquina de inferência. Por último no lugar **P7**, que está associado à classe EXECUTION, encontra-s o objeto que é responsável por executar o deslocamento e atualizações dos dados do veículo.

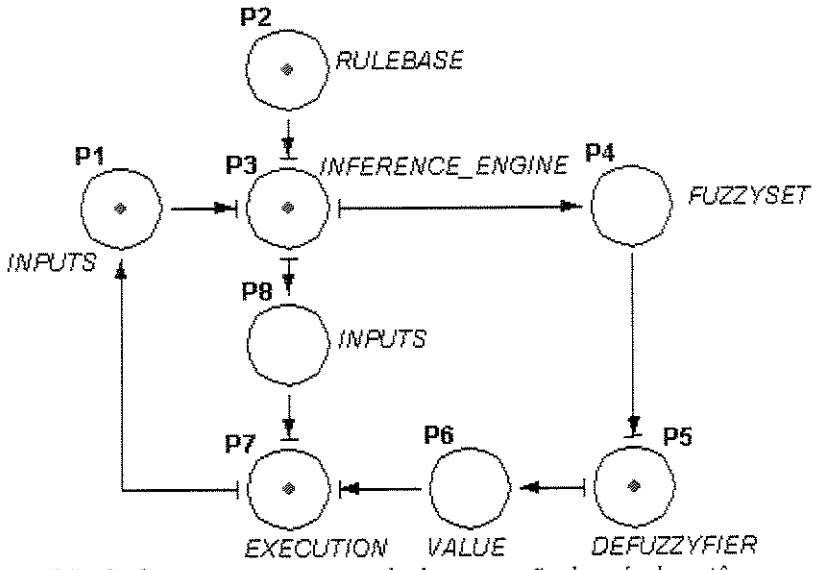


Figura 5.8 Rede de agente para o controle da navegação do veículo autônomo.

A seguir é mostrada a tabela 5.4, onde são descritas as classes utilizadas na rede de agentes para simular a navegação do veículo autônomo.

Classe	Descrição	Definição
VALUE	Classe *wrapper* para importar a classe externa xDouble (representa um valor *double*)	VALUE < xDouble >
RULEBASE	Classe *wrapper* para importar a classe externa RuleBase (representa o conjunto de regras *fuzzy*)	RULEBASE < RuleBase >
FUZZYSET	Classe *wrapper* para importar a classe externa FuzzySet (representa um conjunto *fuzzy*)	FUZZYSET < FuzzySet >
WRAPPER	Classe *wrapper* para importar a classe externa Wrapper (classe genérica de *wrappers* para classes externas, utilizada para importar o *plug* a ser utilizado)	WRAPPER < Wrapper >
INPUTS	Classe que especifica os dados do veículo (x, y, azimute)	INPUTS positionX type VALUE positionY type VALUE azimuth type VALUE

Classe	Descrição	Definição
INFERENCE_ENGINE	Classe que representa a máquina de inferência *fuzzy*. Tem como entrada a informação do veículo e o conjunto de regras *fuzzy*. Como saída tem uma cópia dos dados do veículo e o conjunto *fuzzy* resultante da inferência.	INFERENCE_ENGINE inputs type INPUTS ruleBase type RULEBASE inf_output type FUZZYSET info type INPUTS runInference i1 to inputs i2 to ruleBase o1 to inf_output o2 to info
DEFUZZYFIER	Classe responsável pela defuzzyficação de um conjunto *fuzzy*. Neste caso utilizada para obter o ângulo das rodas do veículo.	DEFUZZYFIER inf_input type FUZZYSET inf_output type VALUE defuzzyfier i1 to inf_input o1 to inf_output
EXECUTION	Classe encarregada pela execução da movimentação do veículo até a próxima posição através da informação do ângulo das rodas e a velocidade do veículo.	EXECUTION value type VALUE inputs type INPUTS outputs type INPUTS plug type WRAPPER consume update i1 to inputs i2 to value o1 to outputs

Tabela 5.4 Classes utilizadas na definição da rede de agentes para a navegação do veículo autônomo utilizando um controlador *fuzzy*.

O *kernel* da rede de agente para o problema da navegação do veículo autônomo é definido por 5 objetos: 3 objetos ativos (uma máquina de inferência, um defuzzyficador e um executor) e 2 passivos (um representando os dados inicias do veículo e outro contendo a base de regras *fuzzy*). Destes objetos, 3 deles precisam de código de inicialização os quais são mostrados a seguir.

Código de inicialização do objeto de entrada do controlador, ou seja, dos dados iniciais do veículo, que são gerados de forma aleatória.

```
put_positionX(new xDouble(Math.random()*90+5));
put_positionY(new xDouble(Math.random()*90+5));
put_azimuth(new xDouble(Math.random()*360-90));
```

No caso do objeto executor é necessária a inicialização do *plug* utilizado para visualizar o percurso do veículo autônomo até chegar a meta.

```
// Plug inicialization

RemotePlotCar plotCar =
(RemotePlotCar)createPlug("plug.lib.RemotePlotCarImpl");
plotCar.setXRange(0,100);
plotCar.setYRange(0,100);
put_plug(new Wrapper(plotCar));
```

Código de inicialização do objeto que representa a base de regras utilizada pela máquina de inferência.

```
java.util.List rules = new Vector();

// Declarations of Universe of Discourse
 DefaultUniverse position =
        new DefaultUniverse("position X",0,100,50);
 DefaultUniverse azimuth =
        new DefaultUniverse("azimut angle",-90,270,50);
 DefaultUniverse theta =
        new DefaultUniverse("angle of truck",-30,30,50);

// Declaration of linguistic terms and membership
// functions used in this demo
 FuzzySet xZero = new DefaultFuzzySet("X Zero",
                                position);
 ((DefaultFuzzySet)xZero).triang(20d,50d,80d);
 FuzzySet xNeg = new DefaultFuzzySet("X Negative",
                                position);
 ((DefaultFuzzySet)xNeg).trapeze(0d,0d,40d,65d);
 FuzzySet xPos = new DefaultFuzzySet("X Positive",
                                position);
 ((DefaultFuzzySet)xPos).trapeze(35d,60d,100d,100d);
 FuzzySet phiZero = new DefaultFuzzySet("Azimuth Zero",
                                azimuth);
 ((DefaultFuzzySet)phiZero).triang(-10d,95d,200d);
 FuzzySet phiNeg = new DefaultFuzzySet("Azimuth
                                Negative",azimuth);
 ((DefaultFuzzySet)phiNeg).trapeze(-90d,-90d,0d,130d);
 FuzzySet phiPos = new DefaultFuzzySet("Azimuth
                                     Positive",azimuth);
 ((DefaultFuzzySet)phiPos).trapeze(50d,150d,270d,270d);
 FuzzySet thetaZero = new DefaultFuzzySet("Theta Zero",
                                          theta);
 ((DefaultFuzzySet)thetaZero).triang(-2.5d,0d,2.5d);
 FuzzySet thetaSmallNeg = new DefaultFuzzySet("Theta
                                Small Negative",theta);
 ((DefaultFuzzySet)thetaSmallNeg).trapeze(-12.5d,-6d,
                                          -4d,2.5d);
 FuzzySet thetaMediumNeg = new DefaultFuzzySet("Theta
                                Medium Negative",theta);
 ((DefaultFuzzySet)thetaMediumNeg).trapeze(-22.5d,-16d,
                                          -14d,-7.5d);
                                               cont...
```

```
FuzzySet thetaBigNeg = new DefaultFuzzySet("Theta Big
                                Negative",theta);
 ((DefaultFuzzySet)thetaBigNeg).trapeze(-30d,-30d,
                                        -22.5d,-12.5d);
 FuzzySet thetaSmallPos = new DefaultFuzzySet("Theta
                                Small Positive",theta);
 ((DefaultFuzzySet)thetaSmallPos).trapeze(-2.5d,4d,
                                        6d,12.5d);
 FuzzySet thetaMediumPos = new DefaultFuzzySet("Theta
                                Medium Positive",theta);
 ((DefaultFuzzySet)thetaMediumPos).trapeze(7.5d,14d,
                                        16d,22.5d);
 FuzzySet thetaBigPos = new DefaultFuzzySet("Theta Big
                                Positive",theta);
 ((DefaultFuzzySet)thetaBigPos).trapeze(12.5d,22.5d,
                                        30d,30d);

 //Create list of rules
 rules.add(new Rule(new
FuzzySet[]{xNeg,phiNeg},thetaSmallNeg));
 rules.add(new Rule(new
FuzzySet[]{xZero,phiNeg},thetaMediumNeg));
 rules.add(new Rule(new
FuzzySet[]{xPos,phiNeg},thetaBigNeg));
 rules.add(new Rule(new
FuzzySet[]{xNeg,phiZero},thetaMediumPos));
 rules.add(new Rule(new
FuzzySet[]{xZero,phiZero},thetaZero));
 rules.add(new Rule(new
FuzzySet[]{xPos,phiZero},thetaMediumNeg));
 rules.add(new Rule(new
FuzzySet[]{xNeg,phiPos},thetaBigPos));
 rules.add(new Rule(new
FuzzySet[]{xZero,phiPos},thetaMediumPos));
 rules.add(new Rule(new
FuzzySet[]{xPos,phiPos},thetaSmallPos));

 put(new RuleBase(rules));
```

A seguir é mostrado o código das funções de transformação e de *matching* associada.

Classe DEFUZZYFIER

Função de transformação: `select`

perform

```
double result = get_inf_input().defuzzyfier();
put_inf_output(new xDouble(result));
```

match

```
MatchResult m = new MatchResult();
m.setAccessMode(get_inf_input(), new
AccessMode(false,true));
m.set(1.0);
return m;
```

Classe INFERENCE_ENGINE

Função de transformação: runInference

perform

```
double inputs_data[] = new double[2];
inputs_data[0] =
get_inputs().get_positionX().getValue();
inputs_data[1] =
get_inputs().get_azimuth().getValue();

DefaultInferenceEngine inferenceEngine =
     new DefaultInferenceEngine(get_ruleBase());

put_inf_output(inferenceEngine.runInference(inp
uts_data));
put_info(release_inputs());
```

match

```
MatchResult m = new MatchResult();
m.setAccessMode(get_inputs(), new
AccessMode(false,true));
m.setAccessMode(get_ruleBase(), new
AccessMode(false,false));
m.set(1.0);
return m;
```

Classe EXECUTION

Função de transformação: consume

Perform

Match

```
MatchResult m = new MatchResult();
RemotePlotCar p =
(RemotePlotCar)get_plug().getData();
double[] r = p.getValues();
if (r==null){
     p.plot(get_inputs().get_positionX().getValu
e(),

     get_inputs().get_positionY().getValue(),

     get_inputs().get_azimuth().getValue());
}
if(get_inputs().get_positionY().getValue() <
98){
     m.setAccessMode(get_inputs(), new
AccessMode(false,false));
     m.set(0.0);
}
else {
     m.setAccessMode(get_inputs(), new
AccessMode(false,true));
     m.set(2.0);
}
return m;
```

Classe EXECUTION

Função de transformação: update

Perform

```
double theta = get_value().getValue();
INPUTS r = release_inputs();
RemotePlotCar plotCar =
(RemotePlotCar)get_plug().getData();
plotCar.plot(theta);
double[] values = plotCar.getValues();
r.put_positionX(new xDouble(values[0]));
r.put_positionY(new xDouble(values[1]));
r.put_azimuth(new xDouble(values[2]));
put_outputs(r);
```

Match

```
MatchResult m = new MatchResult();
m.setAccessMode(get_inputs(), new
AccessMode(false,true));
m.setAccessMode(get_value(), new
AccessMode(false,true));
m.set(3.0);
return m;
```

A figura 5.9 mostra o valor do objeto **obj5** no lugar **P4** obtida durante a fase de simulação, após a primeira iteração, utilizando a ferramenta de *debuging* **Inspector**. Neste exemplo, o objeto pertence à classe FUZZYSET e é o conjunto *fuzzy* obtido da inferência feita pela máquina de inferência *fuzzy* utilizando o método de inferência de MAMDANI. O universo de discurso é o ângulo das rodas do veículo. Este universo de discurso está definido no intervalo [-30,30]. Os valores que definem a função de pertinência para este conjunto *fuzzy* é o seguinte. O número de discretizações do intervalo é 50 e a unidade de divisão do intervalo referente a quantidade de partições é 1.2. Na figura 5.10 é apresentado o valor do objeto **obj6** obtido após a segunda iteração da rede da mesma forma que no exemplo anterior. Neste caso o objeto pertence à classe VALUE, que representa um valor real. Particularmente, este é o valor obtido na defuzzyficação do conjunto *fuzzy* mostrado na figura 5.10 utilizando como método de defuzzyficação o centro de massa. Este valor corresponde ao ângulo (em graus) de torção das rodas do veículo para o a próxima iteração.

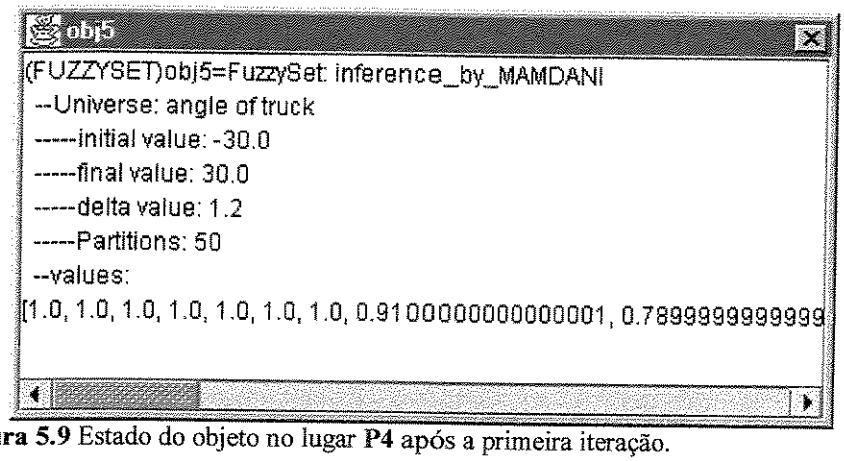


Figura 5.9 Estado do objeto no lugar **P4** após a primeira iteração.

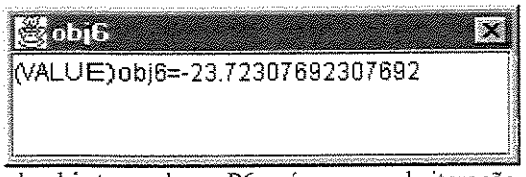


Figura 5.10 Estado do objeto no lugar P6 após a segunda iteração.

A figura 5.11 apresenta dois resultados obtidos da simulação do veículo autônomo.

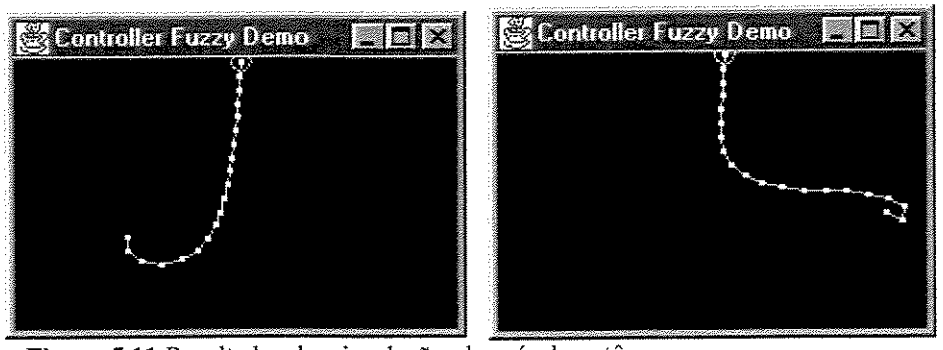


Figura 5.11 Resultados de simulações do veículo autônomo.

5.4. Resumo

Nesse capítulo desenvolveu-se alguns exemplos de aplicação das redes de agentes para o *design* e simulação de sistemas inteligentes. Foi apresentado um exemplo onde esta ferramenta foi utilizada para projetar um algoritmo genético com o objetivo de solucionar o problema do caixeiro viajante para 30, 100 e 101 cidades. Outro exemplo envolve o projeto do controle inteligente, utilizando as ferramentas da lógica *fuzzy*, para solucionar o problema da navegação simples de um veículo autônomo. Estes exemplos são distribuídos junto à ferramenta **ONtoolkit** descrita no capítulo anterior.

No próximo capítulo, têm-se a conclusão deste trabalho, e os trabalhos futuros a serem desenvolvidos.

CAPÍTULO 6. Conclusões e Trabalhos Futuros

Neste trabalho, apresentou-se os fundamentos e a implementação de uma ferramenta para a modelagem e simulação de sistemas a eventos discretos de alta complexidade chamada de "Rede de Agentes". Partiu-se das considerações formais envolvendo os conceitos de objetos, sistemas de objetos e redes de objetos, desenvolvidos em [Gudwin 96], procedeu-se a uma reformulação de alguns destes conceitos, de modo a propiciar os desenvolvimentos seguintes e seguiu-se para uma especialização destes modelos que acabou por resultar na definição formal das redes de agentes. A principal contribuição neste sentido foi a determinação de um procedimento geral e automático para a determinação de funções de seleção, consolidado no chamado "algoritmo de *matching*", e sua especificação formal. A esta contribuição formal, que tem um aspecto mais conceitual, sucedeu-se uma contribuição prática, na concepção e desenvolvimento de uma ferramenta computacional que validasse as contribuições formais e ao mesmo tempo pudesse ser utilizada no futuro pelo nosso grupo de pesquisa.

Com o desenvolvimento e efetiva implementação do software **ONtoolkit**, pode-se dizer que obteve-se um efetivo salto qualitativo e quantitativo na aplicabilidade do conceito de redes de agentes. A princípio apenas uma ferramenta puramente conceitual, passou-se a uma ferramenta de aplicação útil, simples de ser usada e apropriada para investigações científicas na área de sistemas inteligentes. Com esta ferramenta torna-se mais fácil o *design* e simulação de sistemas inteligentes, com uma infra-estrutura formal consolidada. Ao mesmo tempo, esta implementação serve de protótipo para o desenvolvimento de novos tipos de sistemas inteligentes, pois as redes de agentes permitem a integração de diferentes tipos de metodologias em um ambiente comum. Com a possibilidade de se implementar rapidamente novos modelos, é possível analisar a demanda por novas facilidades que possam ser incorporadas, realimentando o potencial de desenvolvimento das próprias redes de agentes. Mais do que isso, ela é um banco de teste para a semiótica computacional, onde seus modelos e estruturas podem ser instanciadas, avaliadas e testadas. Além disso, ela atualmente constitui um objeto de pesquisa, onde os passos para o futuro estão sendo planejados.

Duas categorias de aperfeiçoamentos devem ser resolvidas no futuro para as redes de agentes. A primeira delas é referente ao software desenvolvido, considerando novas soluções tecnológicas, para atingir maior eficiência e confiabilidade. Entre elas podemos citar:

- Extensão do modelo computacional do *MTON engine*, de um sistema multiprocessador para um modelo distribuído. Isto pode ser feito usando a tecnologia de objetos distribuída CORBA (*Common Object Request Broker Architecture*) e/ou RMI (*Remote Method Invocation*).

- Otimização da implementação do algoritmo de *matching* utilizado hoje nas rede de agentes.
- Introdução dos conceitos de análise e *design* utilizados na engenharia de software para oferecer uma ferramenta bem conhecida para o desenvolvimento de aplicações de software orientado a objetos de alta complexidade.
- Criação de um biblioteca específica para o trabalho com redes neurais como é feito hoje com outras técnicas, como os algoritmos genéticos e a lógica *fuzzy*.

A segunda categoria é representada pela extensões que podem ser feitas no modelo formal das redes de agentes, as quais podem incluir as seguintes:

- Expansão do modelo existente, para um modelo de redes hierárquicas, onde um objeto da rede pode ser modelado por uma rede de agentes mesmo.
- A utilização de objetos com campos e/ou objetos *fuzzy* na sua estrutura.

Destas extensões, podemos dizer que esta implementação é justamente um passo no ambicioso objetivo da criação de uma teoria geral para os sistemas inteligentes.

Referências

[Agre 87] Agre, P.; Chapman, D. – PENGI: An implementation of a theory of activity – Proceedings of the Sixth National Conference on Artificial Intelligence – Seattle – pp. 268-272 – 1987.

[Brenner 98] Brenner, W.; Zarnekow, R.; Wittig, H. – Intelligent Software Agents: Foundations and Applications – Springer-Verlag Berlin Heidelberg New York – 1998.

[Brooks 86] Brooks, R.A. – A robust layered control system for a mobile robot – IEEE Journal of Robotics and Automation – Vol. 2 – No. 1 – pp. 14-23 – 1986.

[Brooks 90] Brooks, R.A. – Elephants don't play chess – Maes, P. (eds.): Designing Autonomous Agents – The MIT Press – Cambridge – MA – pp. 3-15 – 1990.

[Brooks 91a] Brooks, R.A. – Intelligence without reason – Proceedings of the Twelfth International Joint Conference on Artificial Intelligence – Sydney – Australia – pp. 569-595 – 1991.

[Brooks 91b] Brooks, R.A. – Intelligence without representation – Artificial Intelligence – No. 47 – pp. 139-159 – 1991.

[Cassandras 93] Cassandras, C.G. – Discrete Event Systems: Modeling and Performance Analysis – Aksen Associates Incorporated Publishers & IRWIN – USA – 1993.

[Ceška 96] Ceška M.; Janoušek V. – Object Orientation in Petri Nets. Object Oriented Modeling and Simulation – Proceeding of the 22nd Conference of the ASU, University Blaise Pascal – Clermont-Ferrand – France – pp. 69-80 – 1996.

[Ceška 97a] Ceška M.; Janoušek V. –A Formal Model for Object Oriented Petri Nets Modeling – Advances in Systems Science and Applications – Vol. Special Issue – No. 1-6 – International Institute for System Studies – SW Texas State University – Texas – USA – pp. 119-124 – 1997.

[Ceška 97b] Ceška, M.; Janoušek, V.; Vojnar, T. – PNtalk - A Computerized Tool for Object Oriented Petri Nets Modelling – Lecture Notes in Computer Science – No. 1333 – Springer Verlag Heidelberg-Berlin – Berlin – pp. 591-610 – 1997.

[Chistensen 92] Chistensen, S.; Petrucci, L. – Towards a Modular Analysis of Coloured Petri Nets – 13th International Conference on Application and Theory of Petri Nets – Lecture Notes in Computer Science – Vol. 616 – pp. 113-133 – 1992.

[Fengler 96] Fengler W.; Wendt A.; Bogoljubow J.; Däne B. – The Use of Fuzzy Coloured Petri Nets for Modelling and Simulation in Manufacturing – 17th International Conference on Application and Theorie of Petri Nets: Workshop for Manufacturing and Petri Nets – Osaka – 1996.

[FIPA 99] FIPA – FIPA Content Language – FIPA Draft 18-1999 – 1999.

[Franklin 96] Franklin S., Graesser A. – Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents – Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages – Springer-Verlag, 1996.

[Genrich 81] Genrich, H.J.; Lautenbach, K. – System Modelling with High Level Petri Nets – Theoretical Computer Science – Vol. 14 – 1981 – pp. 109-136.

[Gudwin 96] Gudwin, R.R. – Contribuições ao Estudo Matemático de Sistemas Inteligentes – PhD Thesis – DCA-FEEC-UNICAMP – 1996.

[Gudwin 97a] Gudwin, R.R.; Gomide, F.A.C. – Computational Semiotic: An Approach for the Study of Intelligent Systems – Part I: Foundations – (Technical Report RT-DCA 09 – DCA-FEEC-UNICAMP, 1997).

[Gudwin 97b] Gudwin, R.R.; Gomide, F.A.C. – Computational Semiotic: An Approach for the Study of Intelligent Systems – Part II: Theory and Application – (Technical Report RT-DCA 09 – DCA-FEEC-UNICAMP, 1997).

[Gudwin 97c] Gudwin, R.R.; Gomide, F.A.C. – An Approach to Computational Semiotics – Proceedings of the ISAS'97 – Intelligent Systems and Semiotics: A Learning Perspective – International Conference – Gaithersburg – USA – pp. 467-470 – 1997.

[Gudwin 97d] Gudwin, R.R.; Gomide, F.A.C. – A Computational Semiotic Approach for Soft Computing – Proceedings of the IEEE SMC'97 – Vol. 4 – IEEE International Conference on Systems, Man and Cybernetics – Orlando – USA – pp. 3981-3986 – 1997.

[Gudwin 98a] Gudwin, R.R.; Gomide, F.A.C. – Object Network: A Formal Model to Develop Intelligent Systems in Computational Intelligence and Software Engineering – J. Peters and W. Pedrycz (eds.) – World Scientific Pub Co – 1998.

[Gudwin 98b] Gudwin, R.R.; Gomide, F.A.C. – Object Networks: A Modeling Tool – Proceedings of FUZZY-IEEE98 – WCCI'98 – IEEE World Congress on Computational Intelligence – Anchorage, Alaska – USA – pp. 77-82 – 1998.

[Gudwin 99a] Gudwin, R.R.; Gomide, F.A.C. – Object Network: A Computational Framework to Compute with Words in Computing with words in Information/Intelligent Systems, L.A. Zadeh and J. Kacprzyk (Eds.) – Springer-Verlag – Berlin – 1999.

[Gudwin 99b] Gudwin, R.R. – From Semiotics to Computational Semiotics – Proceedings of 7th International Congress of the International Association for Semiotic Studies – IASS/AIS – Dresden – 1999.

[Guerrero 98] Guerrero, J.A.S.; Suárez, L.L; Gudwin, R.R. – Comparación de la Aplicación de Métodos Inductivos y Evolutivos para el Aprendizaje de Redes Neuronales – Resúmenes del 6to. Congreso Internacional de Nuevas Tecnologías y Aplicaciones Informáticas – VI Convención y Feria Internacional Informática'98 – C. Habana – 1998.

[Guerrero 99a] Guerrero, J.A.S.; Suárez, L.L; Gudwin, R.R. – Análise da Importância de Parâmetros em um Algoritmo Genético por Meio de sua Aplicação no Aprendizado de uma Rede Neural – Anais do XIX Congresso da Sociedade Brsileira de Computação SBC'99 – Rio de Janeiro – pp. 423-435 – 1999.

[Guerrero 99b] Guerrero, J.A.S.; Gomes, A.S.R.; Gudwin, R.R. – A Computational Tool to Model Intelligent Systems – Anais do IV Simpósio Brsileiro de Automação Inteligente SBAI'99 – São Paulo – pp. 227-232 – 1999.

[Janoušek 95a] Janoušek V. – PNtalk: Object Orientation in Petri Nets – Proceeding of European Simulation Multiconference ESM'95 – Prague Technical University – Prague – pp. 196-200 – 1995.

[Janoušek 95b] Janoušek V. – Functional and Object Oriented Structuring of Petri Nets – Proceeding of Computer Science – VSB Ostrava – Ostrava – pp.44-47. – 1995.

[Janoušek 96] Janoušek V. – The PNtalk language and System – Proceeding of MOSIS'96 – MARQ Ostrava – Ostrava – pp.233-238 – 1996.

[Janoušek 97] Janoušek V. – Reflective Approach to Petri Net Simulation – Proceeding of MOSIS'97 – MARQ Ostrava – Ostrava – pp. 209-304 – 1997.

[Jennings 98] Jennings, N.R.; Wooldridge, M.J. – Applications of Intelligent Agents - Jennings, N.R.; Wooldridge, M.J. (eds.): Agent Technology: Foundations, Applications, and Markets – Springer-Verlag – 1998 – pp.3-27.

[Jensen 90] Jensen, K. - Coloured Petri Nets: A High Level Language for System Design and Analysis – Lecture Notes in Computer Science – Vol. 483 – Advances in Petri Nets, pp. 342-416 – 1990.

[Jensen 94] Jensen, K. – An Introduction to the Theoretical Aspects of Coloured Petri Nets. – Bakker, J.W.; Roever, W.P.; Rozenberg, G. (eds.): A Decade of Concurrency – Lecture Notes in Computer Science – Vol. 803 – Springer-Verlag – 1994 – pp. 230-272.

[Jensen 97] Jensen, K. – A Brief Introduction to Coloured Petri Nets. – Brinksma, E. (ed.): Tools and Algorithms for the Construction and Analysis of Systems. – Proceeding of the TACAS'97 Workshop – Enschede – Netherlands – Lecture Notes in Computer Science – Vol. 1217 – Springer-Verlag – 1997 – pp. 203-208.

[Jensen 98] Jensen, K. – An Introduction to the Practical Use of Coloured Petri Nets. – Reisig, W.; Rozenberg, G. (eds.): Lectures on Petri Nets II: Applications – Lecture Notes in Computer Science – Vol. 1492 – Springer-Verlag – 1998 – pp. 237-292.

[Koda 96] Koda, T.; Maes, P. – Agents with Faces: The Effects of Personification of Agents – Proceedings of HCI'96 – London – pp. 98-103 – The British HCI Group – 1996.

[Labrou 99] Labrou, Y.; Finin, T.; Peng, Y. – Agent Communication Language: The Current Landscape – IEEE Intelligent Systems & their applications – Vol. 14 – No. 2 – March/April 1999 – pp. 45-52 – 1999.

[Lashkari 94] Lashkari, Y.; Metral, M.; Maes, P. – Collaborative Interface Agents – Proceedings of the Twelfth National Conference on Artificial Intelligence – Vol. 1 – AAAI Press – Seattle – 1994.

[Maes 91] Maes, P. – The agent network architecture (ANA) – SIGART Bulletin – Vol. 2 – No. 4 – pp. 115-120 – 1991.

[Maes 94] Maes, P. – Agents that Reduce Work and Information Overload. – Communications of the ACM – Vol. 37 – No.7 – ACM Press – 1994 – pp. 31-40.

[Murata 89] Murata, T. – Petri Nets: Properties, Analysis and Applications – Proceedings of the IEEE, Vol. 77, No. 4. – 1989.

[Ndumu 97] Ndumu D.T.; Nwana H.S. – Research and Development Challenges for Agent-Based Systems – IEE Proceedings on Software Engineering – Vol. 144 – No. 1 – 1997.

[Newman 98] Newman A.; Shatz, S.M.; Xie, X. – An Approach to Object System Modeling by State-Based Object Petri Nets – International Journal of Circuits, Systems and Computer. 1998.

[Nwana 96a] Nwana, H.S. – Software Agents: An Overview – Knowledge Engineering Review – Vol. 11 – No. 3 – 1996 – pp. 1-40.

[Nwana 96b] Nwana, H.S.; Ndumu, D.T. – An Introduction to Agent Tecnology – BT Technology Jornal – Vol. 14 – No. 4 – 1996 – pp. 55-67.

[Nwana 96c] Nwana, H.S.; Lee,L.; Jennings, N.R. – Coordination in Software Agent Systems – BT Technology Jornal – Vol. 14 – No. 4 – 1996 – pp. 79-88.

[Nwana 96d] Nwana, H.S.; Wooldridge M.J. – Software Agent Technologies - BT Technology Jornal – Vol. 14 – No. 4 – 1996 – pp. 68-78.

[Nwana 98] Nwana, H.S.; Ndumu, D.T. – A Brief Introduction to Software Agent Technology – Jennings, N.R.; Wooldridge, M.J. (eds.): Agent Technology: Foundations, Applications, and Markets – Springer-Verlag – 1998 – pp. 29-49.

[Pedrycs 94] Pedrycs, W.; Gomide F. – A Generalized Fuzzy Petri Net Model – IEEE Transactions on Fuzzy Systems – Vol. 2 – No. 4 – 1994 – pp. 295-301.

[Petrie 96] Petrie, C.J. – Agent-Based Engineering , the WEB, and Intelligence – IEEE Expert Intelligent Systems & their Applications – Vol. 11 – No. 6 – 1996 – pp. 24-29.

[Russell 95] Russell S. J.; Norvig P. – Artificial Intelligence: A Modern Approach – Englewood Cliffs – NJ – Prentice Hall – 1995.

[Singh 98] Singh, M.P. – Agent Communication Language: Rethinking the principles – Computer – Vol. 31 – No. 12 — December 1999 – pp. 40-47 – 1999.

[Ullman 79] Ullman, J.; Hopcroft, J. – Introduction to Automata Theory, Languages, and Computation – MA: Addison-Wesley – 1979.

[Valk 78] Valk, R. – Self-Modifying Nets - A Natural Extension of Petri Nets – Lecture Notes in Computer Science — Vol. 62 - Automata, Languages and Programming – 1978 – pp. 464-477.

[Valk 81] Valk, R. – Generalizations of Petri Nets – Lecture Notes in Computer Science – Mathematical Foundations of Computer Science – Vol. 118 – pp. 140-156.

[Vojnar 99a] Vojnar, T. – Specifying Properties of Systems to be Checked Using Their Object-Oriented Petri Net-Based Models – Proceedings of 21th International Workshop on Advanced Simulation of Systems ASIS'99 – No. Acta MOSIS No. 78 – MARQ Ostrava – Krnov – Czech Republic – pp.219-224 – 1999.

[Vojnar 99b] Vojnar, T. – Towards Using State Spaces of Object-Oriented Petri Nets – Proceedings of 33rd Spring International Conference on Modelling and Simulation of Systems MOSIS'99 – MARQ Ostrava – Rožnov pod Radhoštěm – Czech Republic – pp. 141-148 – 1999.

[Wand 89] Wand, Y. – A Proposal for a Formal Model of Objects in Object_Oriented Concepts, Databases and Applications – W. Kim and F. Lochovsky, eds. –, ACM Press and Addison-Wesley – New York – pp. 537-559 – 1989.

[Wooldridge 95] Wooldridge, M.J.; Jennings, N.R. – Intelligent Agents: Theory and Practice – Knowledge Engineering Review – Vol. 10 – No. 2 – Cambridge University Press – 1995 – pp. 115-152.

[Zhou 89] Zhou, M.C.; Dicesare, F. – Adaptive Design of Petri Net Controllers for Error Recovery in Automated Manufacturing Systems – IEEE Transactions on System, Man and Cybernetics – Vol. 19 – No. 5 – 1989 – pp. 963-973.

Índice Remissivo das Referências

Referência	Título	Referenciada em
[Agre 87]	PENGI: An implementation of a theory of activity	p62
[Brenner 98]	Intelligent Software Agents: Foundations and Applications	p63
[Brooks 86]	A robust layered control system for a mobile robot	p61
[Brooks 90]	Elephants don't play chess	p61
[Brooks 91a]	Intelligence without reason	p61
[Brooks 91b]	Intelligence without representation	p61
[Cassandras 93]	Discrete Event Systems: Modeling and Performance Analysis	p1
[Ceška 96]	Object Orientation in Petri Nets. Object Oriented Modeling and Simulation	p5
[Ceška 97a]	A Formal Model for Object Oriented Petri Nets Modeling	p5
[Ceška 97b]	PNtalk - A Computerized Tool for Object Oriented Petri Nets Modelling	p5
[Chistensen 92]	Towards a Modular Analysis of Coloured Petri Nets	p5
[Fengler 96]	The Use of Fuzzy Coloured Petri Nets for Modelling and Simulation in Manufacturing	p6
[FIPA 99]	FIPA Content Language	p68
[Franklin 96]	Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents	p59
[Genrich 81]	System Modelling with High Level Petri Nets	p5
[Gudwin 96]	Contribuições ao Estudo Matemático de Sistemas Inteligentes	p1, p2, p6, p12, p34, p37, p40, p56, p149
[Gudwin 97a]	Computational Semiotic: An Approach for the Study of Intelligent Systems – Part I: Foundations	p6
[Gudwin 97b]	Computational Semiotic: An Approach for the Study of Intelligent Systems Part II: Theory and Application	p6
[Gudwin 97c]	An Approach to Computational Semiotics	p6

Referência	Título	Referenciada em
[Gudwin 97d]	A Computational Semiotic Approach for Soft Computing	p6
[Gudwin 98a]	Object Network: A Formal Model to Develop Intelligent Systems	p6
[Gudwin 98b]	Object Networks: A Modeling Tool	p6
[Gudwin 99a]	Object Network: A Computational Framework to Compute with Words	p6
[Gudwin 99b]	From Semiotics to Computational Semiotics	p1, p6
[Guerrero 98]	Comparación de la Aplicación de Métodos Inductivos y Evolutivos para el Aprendizaje de Redes Neuronales	p48
[Guerrero 99a]	Análise da Importância de Parâmetros em um Algoritmo Genético por Meio de sua Aplicação no Aprendizado de uma Rede Neural	p48
[Guerrero 99b]	A Computational Tool to Model Intelligent Systems	p87
[Janoušek 95a]	PNtalk: Object Orientation in Petri Nets	p5
[Janoušek 95b]	Functional and Object Oriented Structuring of Petri Nets	p5
[Janoušek 96]	The PNtalk language and System	p5
[Janoušek 97]	Reflective Approach to Petri Net Simulation	p5
[Jennings 98]	Applications of Intelligent Agents	p57
[Jensen 90]	Coloured Petri Nets: A High Level Language for System Design and Analysis	p5
[Jensen 94]	An Introduction to the Theoretical Aspects of Coloured Petri Nets	p5
[Jensen 97]	A Brief Introduction to Coloured Petri Nets	p5
[Jensen 98]	An Introduction to the Practical Use of Coloured Petri Nets	p5
[Koda 96]	Agents with Faces: The Effects of Personification of Agents	p64
[Labrou 99]	Agent Communication Language: The Current Landscape	p68
[Lashkari 94]	Collaborative Interface Agents	p64
[Maes 91]	The agent network architecture (ANA)	p62
[Maes 94]	Agents that Reduce Work and Information Overload	p64
[Murata 89]	Petri Nets: Properties, Analysis and Applications	p1, p5
[Ndumu 97]	Research and Development Challenges for Agent-Based Systems	p63

Referência	Título	Referenciada em
[Newman 98]	An Approach to Object System Modeling by State-Based Object Petri Nets	p5
[Nwana 96a]	Software Agents: An Overview	p63
[Nwana 96b]	An Introduction to Agent Tecnology	p63
[Nwana 96c]	Coordination in Software Agent Systems	p64
[Nwana 96d]	Software Agent Technologies	p68
[Nwana 98]	A Brief Introduction to Software Agent Technology	p63
[Pedrycs 94]	A Generalized Fuzzy Petri Net Model	p6
[Petrie 96]	Agent-Based Engineering , the WEB, and Intelligence	p64
[Russell 95]	Artificial Intelligence: A Modern Approach	p58
[Singh 98]	Agent Communication Language: Rethinking the principles	p68
[Ullman 79]	Introduction to Automata Theory, Languages, and Computation	p1
[Valk 78]	Self-Modifying Nets - A Natural Extension of Petri Nets	p5
[Valk 81]	Generalizations of Petri Nets	p5
[Vojnar 99a]	Specifying Properties of Systems to be Checked Using Their Object-Oriented Petri Net-Based Models	p5
[Vojnar 99b]	Towards Using State Spaces of Object-Oriented Petri Nets	p5
[Wand 89]	A Proposal for a Formal Model of Objects	p7
[Wooldridge 95]	Intelligent Agents: Theory and Practice	p57, p59, p61, p63, p67
[Zhou 89]	Adaptive Design of Petri Net Controllers for Error Recovery in Automated Manufacturing Systems	p5