CONHECIMENTO DE TAREFA EM AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE ORIENTADOS A DOMÍNIO

Fábio Zlot

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Profa. Ana Regina Cavalcanti da Rocha, D.Sc.

Profa. Káthia Marçal de Oliveira, D.Sc.

Prof. Ricardo de Almeida Falbo, D.Sc.

Prof. Crediné Silva de Menezes, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2002

ZLOT, FÁBIO

Conhecimento de Tarefa em Ambientes de Desenvolvimento de Software Orientados a Domínio [Rio de Janeiro] 2002

VIII, 103 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Conhecimento de Tarefa
2. Ontologia
3. Métodos de Solução de Problemas

I.COPPE/UFRJ II. Título (série)

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

CONHECIMENTO DE TAREFA EM AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE ORIENTADOS A DOMÍNIO

Fábio Zlot

Junho/2002

Orientadores: Ana Regina Cavalcanti da Rocha
Káthia Marçal de Oliveira

Programa: Engenharia de Sistemas e Computação

Uma das principais razões para produtos de software não atenderem às necessidades dos clientes é a falta de entendimento de qual é o real problema a ser resolvido pelo software e, consequentemente, quais são as tarefas que ele deve realizar e como estas devem ser realizadas. Nós defendemos que o uso do conhecimento de tarefa pode auxiliar o desenvolvedor ao longo do processo de desenvolvimento de software.

Para apoiar essa idéia, definimos uma estrutura para representação do conhecimento de tarefa que apóia os engenheiros de software no entendimento de problemas a partir do entendimento das tarefas que compõem estes problemas. O conhecimento de tarefa deve se manter independente do domínio em que será aplicado, de forma que ambos os conhecimentos permaneçam genéricos e, desta forma, aplicáveis a vários sistemas.

Esta tese apresenta um modelo para a representação de conhecimento de tarefa baseado nessas características. É apresentado, também, um exemplo de como os desenvolvedores podem utilizar este modelo no auxílio ao desenvolvimento de software.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

TASK KNOWLEDGE IN DOMAIN ORIENTED SOFTWARE DEVELOPMENT ENVIRONMENTS

Fábio Zlot

June/2002

Advisors: Ana Regina Cavalcanti da Rocha
Káthia Marçal de Oliveira

Department: System and Computing Engineering

One of the main reasons why the software products does not meet the client's need is the lack of understanding of the software's real objective, and consequently, the tasks it should perform and how they should be performed. We argue that the use of task knowledge can support software engineers throughout the development process.

To reinforce and support this assumption we defined a structure to represent the task knowledge which supports software engineers in understanding problems starting from the understanding of the tasks which comprise these problems. The task knowledge should remain independent from the domain in which it will be applied, in a way that both knowledge remain generic, thus applicable to several systems.

This thesis presents a framework to description of task knowledge that follows these features. It also presents an example of how developers can use this framework to support software development.

Ao meu pai, minha mãe e
minha irmã pelo apoio e carinho
em todas as horas, sem os quais
não seria possível a realização deste trabalho.

Agradecimentos

À minha orientadora Ana Regina por acreditar em minha capacidade e pelo carinho, amizade e apoio que foram fundamentais na minha formação.

À co-orientadora e amiga Káthia pelo envolvimento e a impressionante dedicação que teve durante todo o desenvolvimento deste trabalho.

Ao Luís Filipe e Carmen por nossas conversas, por seus comentários e por sua valiosa ajuda.

Aos professores Ricardo Falbo e Crediné Menezes por contribuírem com seu conhecimento e experiência, fundamentais para que esta idéia se tornasse realidade e por aceitarem participar da banca.

Ao Gleison e Sômulo pela colaboração para a conclusão deste trabalho;

À Cátia Galotta, Augusto, Ana Mirtez e Karina, por se mostrarem sempre disponíveis para auxílios e troca de conhecimentos.

À Giselle por me acompanhar nessa caminhada e pela compreensão pelas minhas ausências.

Aos colegas de curso Leonardo, Gustavo, Raquel e Andrômeda que direta ou indiretamente me acompanharam.

À Ana Paula pela paciência com todos os alunos.

Aos professores e funcionários do Programa.

Ao CNPQ, pelo apoio financeiro.

Índice

Capítulo 1

Introdução

Neste capítulo são apresentados a motivação, o histórico da pesquisa realizada e os objetivos deste trabalho, além da organização dos próximos capítulos.

1.1 Motivação

Uma das atividades mais críticas durante o desenvolvimento de software é a correta identificação e descrição dos requisitos do produto. Para a realização desta atividade, é fundamental que haja o correto entendimento do problema que deve ser resolvido pelo sistema.

É comum desenvolver-se produtos de software que não atendem às necessidades do cliente simplesmente porque não se entendeu qual era o real objetivo do software e, consequentemente, quais eram as tarefas que ele deveria realizar. Defendemos que o entendimento do problema envolve a compreensão do domínio no qual se está trabalhando e o entendimento sobre as tarefas que fazem parte deste problema. O conhecimento sobre as tarefas permite apoiar o entendimento do problema e a atividade de desenvolvimento de forma geral. Baseados nesta idéia, propomos uma abordagem na qual o apoio ao desenvolvimento de software é realizado a partir da modelagem do conhecimento do domínio e de tarefas.

Em trabalhos anteriores, OLIVEIRA (1999a) definiu como organizar o conhecimento do domínio de forma a apoiar o desenvolvimento de software por desenvolvedores que não têm familiaridade ou experiência de realizar desenvolvimento em um determinado domínio. Oliveira (1999a) propôs não só como organizar o conhecimento do domínio mas também como utilizá-lo ao longo do desenvolvimento.

Neste contexto, Oliveira definiu Ambientes de Desenvolvimento de Software Orientados a Domínio (ADSOD) (OLIVEIRA 1999b) como uma evolução dos tradicionais Ambientes de Desenvolvimento de Software (ADS). A implementação de ADSOD se deu a partir da infra-estrutura da Estação TABA (ROCHA *et al.*, 1990), um projeto de larga

escala realizado na COPPE/UFRJ e que visa a construção de uma Estação de Trabalho configurável para apoiar o desenvolvimento de produtos de software para diferentes domínios de aplicação e abordagens de desenvolvimento. Oliveira estendeu a Estação TABA para que permitisse a inclusão do conhecimento do domínio e a instanciação, não apenas de ADS como já estava disponível, mas, também, de ADSOD.

Uma vez resolvida a questão de como apoiar o entendimento do domínio, partimos para investigar como descrever o conhecimento de tarefas de forma a apoiar no entendimento do problema pelos desenvolvedores. Acreditamos que a organização do conhecimento de tarefa, de forma bem estruturada e fácil de ser entendida, pode ser uma importante fonte de conhecimento para os desenvolvedores de software. A partir deste convencimento, iniciamos nosso trabalho buscando identificar como representar tarefas de forma a oferecer apoio aos desenvolvedores de software através de uma boa e clara definição de como esse conhecimento deve ser explicitado, estruturado e representado. Nesta modelagem do conhecimento de tarefa não existe comprometimento com qualquer aspecto que não sejam as características da própria tarefa.

Além disso, a descrição do conhecimento de tarefa deve ser realizada de forma independente do domínio em que será aplicada, permitindo que uma biblioteca de tarefas possa ser definida e utilizada ao se desenvolver produtos nos diversos domínios e que possam ser acrescentadas novas tarefas de acordo com a necessidade do sistema que está sendo desenvolvido. O conhecimento de tarefa deve ser útil para apoiar os engenheiros de software no entendimento do problema, através do entendimento das tarefas que compõem esse problema, auxiliando também, desta forma, na sua resolução.

1.2 Histórico de Pesquisa e Objetivo da Tese

O objetivo desse trabalho é, portanto, descrever os principais fundamentos para definição de uma descrição do conhecimento de tarefa em ambientes de desenvolvimento de software tendo como referencial a Estação TABA.

Inicialmente, a pesquisa se direcionou para a definição de como o conhecimento de tarefa deveria ser organizado e disponibilizado no ambiente. A resposta para esta questão veio de um estudo na área de Inteligência Artificial (IA), que utiliza ontologias como uma especificação explícita de uma conceituação (GRUBER, 1995). Em outras palavras, uma

ontologia permite que conceitos da tarefa sejam explicitamente definidos e representados, de forma que possam ser compartilhados, e que restrições sejam expressas através de axiomas que restringem os significados dos termos (USCHOLD e GRUNINGER, 1996).

Esta idéia foi originalmente proposta por FALBO (1998), que introduziu na Estação TABA a noção de Servidor de Conhecimento, como uma estrutura do conhecimento de domínio e de tarefa. Estes servidores têm como característica serem apoiados por ontologias para fixar a semântica da informação tratada e, portanto, a construção de Servidores de Conhecimento está essencialmente fundamentada no uso de ontologias.

Entretanto, a idéia de utilização de ontologias para a representação do conhecimento de tarefa em ambientes de desenvolvimento de software não foi, naquele momento, implementada na Estação TABA, pois os trabalhos tomaram outra direção: a criação de uma biblioteca de resolvedores genéricos de problemas, passíveis de serem instanciados e adaptados para aplicações particulares, baseados nos modelos de tarefa do *Common*KADS (BREUKER *et al.*, 1994), em detrimento ao uso de ontologias de tarefa.

Na nossa abordagem, defendemos o uso de ontologias como o aspecto central para a representação do conhecimento de tarefa, ou seja, optamos por aprofundar a pesquisa sobre ontologias de tarefa para permitir a sua utilização na descrição do conhecimento de tarefa, estendendo assim o auxílio do Servidor de Conhecimento no desenvolvimento de sistemas. Consideramos, entretanto, que entender uma tarefa não envolve apenas entender sobre os conceitos utilizados na mesma, mas também entender como ela é realizada. Dessa forma, associamos ontologia de tarefa e métodos de solução de problemas (MSP) em um único modelo para a representação do conhecimento de tarefa. Os MSP especificam a estratégia de inferência responsável por determinar uma solução para a tarefa (SCHREIBER, G., WIELINGA, B. e BREUKER, J., 1993).

Por fim, trabalhamos no desenvolvimento de uma nova ferramenta para a Estação TABA, que permita a descrição do conhecimento de tarefa segundo a abordagem proposta e a efetiva utilização do conhecimento ao longo do processo de desenvolvimento de software. Esta ferramenta pode ser utilizada por qualquer ADS instanciado a partir da infra-estrutura da Estação TABA.

1.3 Organização da Tese

Esta tese contém, além desta introdução, mais 5 capítulos.

No capítulo 2 são apresentadas as principais características de Ambientes de Desenvolvimento de Software, presentes na literatura. Neste capítulo é, também, apresentada uma visão geral da Estação TABA no que se refere à sua estrutura, integração de conhecimento e ambientes instanciados.

No capítulo 3 são descritas ontologias como base para a representação do conhecimento, enfocando suas principais definições, classificações e, mais detalhadamente, a ontologia de tarefa no que se refere a sua estrutura e características. São apresentados, ainda, os principais trabalhos sobre métodos de solução de problemas, assim como seus objetivos e estrutura.

No capítulo 4 é apresentada uma abordagem para a organização do conhecimento de tarefa, útil para o entendimento e resolução de problemas em ambientes de desenvolvimento de software. São apresentados os elementos que a compõe, além de sua estrutura de representação.

No capítulo 5 são apresentadas as extensões realizadas na Estação TABA de forma a incorporar a abordagem proposta, no que se refere ao modelo modificado e a implementação de uma nova ferramenta. É apresentado, ainda, um exemplo de como a organização do conhecimento de tarefa pode ser útil em ambientes de desenvolvimento de software.

Finalmente, o capítulo 6 contém as conclusões deste trabalho, bem como as perspectivas para futuras pesquisas.

Capítulo 2

Ambientes de Desenvolvimento de Software e a Estação TABA

Este capítulo apresenta as principais características de Ambientes de Desenvolvimento de Software, presentes na literatura. É apresentada, ainda, uma visão geral da Estação TABA no que se refere à sua estrutura, definições, ambientes de desenvolvimento instanciados e integração de conhecimento.

2.1 Histórico

O conceito de Ambientes de Desenvolvimento de Software (ADS) surgiu na década de 70 com o intuito de combinar técnicas, métodos e ferramentas com o objetivo de prover um meio através do qual o engenheiro de software pudesse ter um apoio ao construir produtos de software. Desde então, diferentes pesquisas foram realizadas para que esse objetivo fosse alcançado.

Penedo (PENEDO, 1993, in TRAVASSOS, 1994) trata a evolução dos ADS segundo três estágios. No primeiro estágio há a preocupação com métodos de desenvolvimento de software. As ferramentas apoiavam métodos particulares e funcionavam de forma isolada. No segundo estágio há a preocupação com a integração de ferramentas. Neste ponto começaram os primeiros estudos para se definirem estruturas (*frameworks*) de ADS. Esquemas de interface com o usuário passaram a ser padronizados, possibilitando o desenvolvimento de ferramentas consistentes em relação à interface. Atividades maiores no ciclo de vida passaram a ser consideradas. Este conjunto de características levou às primeiras definições de mecanismos de integração, e junto a isto, começou a preocupação com a distribuição do ADS. No terceiro estágio, o enfoque é dado para ADS orientados ao processo de desenvolvimento de software. Um processo de desenvolvimento é um conjunto

de atividades bem definido e ordenado, somado aos recursos utilizados e produzidos, e ao conjunto de ferramentas e técnicas para apoio à realização das atividades (PFLEEGER, 1998). O desenvolvimento de componentes é uma das principais características de um processo de desenvolvimento (ALTMAN e POMBERG, 1999) e consequentemente existem diversas pesquisas sobre o desenvolvimento de ADS para compor aplicações a partir de componentes existentes (JACOBSON *et al.*, 1992, MINOURA *et al.*, 1993, SELIC *et al.*, 1994, SHAW *et al.*, 1995). A utilização de componentes de software bem testados, possibilita a redução do tempo de desenvolvimento e o aumento da qualidade das aplicações desenvolvidas pelo ADS.

A partir dos estágios descritos anteriormente os Ambientes de Desenvolvimento de Software são classificados em três gerações:

i) **ADS de primeira geração** – correspondem aos sistemas de ferramentas isoladas, muito utilizadas dos anos 60 aos anos 80, e que não satisfazem as necessidades atuais de portabilidade e integração.

ii) **ADS de segunda geração** – ambientes baseados em estrutura, que começaram a aparecer nos anos 80 e início de 90 e que foram projetados para apoiar a integração, portabilidade e interoperabilidade de ferramentas a partir da utilização de integração de dados através de esquemas de banco de dados (BROWN, 1992, PENEDO, 1993).

iii) **ADS de terceira geração** – São caracterizados por serem ambientes centrados em processo que defendem a necessidade de integração de ferramentas incorporadas a um processo de desenvolvimento de software específico da organização (GARG e JAZAYEII, 1995).

Um ambiente de desenvolvimento de software pode ser definido, portanto, como sendo uma infra-estrutura contendo um conjunto de ferramentas integradas que se comunicam e cooperam de maneira controlada, buscando apoiar as atividades do processo de desenvolvimento de software. Para tal, o ADS contém um repositório com todas as informações relacionadas com o projeto ao longo do seu ciclo de vida e ferramentas que oferecem apoio para as várias atividades técnicas e gerenciais passíveis de automação que devem ser realizadas no projeto (MOURA, 1992). Segundo TRAVASSOS (1994), um ADS deve se preocupar com o apoio às atividades individuais e ao trabalho em grupo, o gerenciamento do projeto, o aumento da qualidade geral dos produtos e o aumento da produtividade, para permitir que o engenheiro de software possa acompanhar o projeto e

medir a evolução dos trabalhos através de informações obtidas ao longo do desenvolvimento.

Um dos fatores mais importantes no contexto de ADS é a integração das ferramentas, para buscar a melhor forma de interagir com o ambiente e estabelecer a abrangência em que deve ser feito o tratamento das informações dentro do ambiente. Para isso, a arquitetura do ambiente deve permitir que as ferramentas possam cooperar umas com as outras, que elas possam ser conectadas ao ambiente e que o ambiente forneça suporte metodológico ao desenvolvimento (TRAVASSOS, 1994). TRAVASSOS (1994) propôs, então, a inclusão de um componente para incorporar mecanismos para o armazenamento e utilização de conhecimento descrito e adquirido ao longo do processo de desenvolvimento.

O componente de conhecimento é considerado essencial para garantir a integração de ferramentas, por manter o conhecimento sobre processos, métodos e domínios de aplicação que são úteis para as ferramentas. Uma arquitetura genérica, denominada servidores de conhecimento, foi definida para permitir a inclusão de fontes diversificadas de conhecimento útil à construção de ferramentas (FALBO, 1998a, FALBO *et al.*, 1999a). Ambos os trabalhos foram realizados no contexto da arquitetura da Estação TABA (ROCHA *et al.*, 1990), que será descrita em detalhes na próxima seção. Também será visto de que forma a Estação TABA procurou definir a arquitetura robusta de um meta-ambiente que permitisse a definição de ADS específicos para a utilização de diferentes tecnologias.

2.2 A Estação TABA

Segundo Rocha (ROCHA *et al.*, 1987), um meta-ambiente é um ambiente que abriga um conjunto de programas que interage com um usuário para definir interfaces, selecionar ferramentas e definir tipos de objetos que irão compor o ambiente de desenvolvimento de software específico.

Com o objetivo de construir um meta-ambiente para desenvolvimento de software capaz de prover aos desenvolvedores de software ambientes de desenvolvimento que atendam às particularidades de domínios de aplicação e projetos específicos, vem sendo realizado na COPPE/UFRJ o projeto TABA (ROCHA *et al.*, 1987, ROCHA *et al.*, 1990, TRAVASSOS, 1994). A motivação para o desenvolvimento deste projeto está na constatação de que domínios de aplicação e projetos diferentes têm características

diferentes e que estas devem incidir nos ambientes de desenvolvimento através dos quais os engenheiros de software desenvolvem as aplicações (TRAVASSOS, 1994).

Desta forma, a Estação TABA visa permitir a especificação de ADS adequados a diferentes contextos e sua instanciação como ambiente integrado. Para atender a este objetivo, a Estação TABA possui quatro funções básicas:

i) Auxiliar o engenheiro de software na especificação e instanciação do ambiente mais adequado ao desenvolvimento de um produto específico;

ii) Auxiliar o engenheiro de software a implementar as ferramentas necessárias ao ambiente definido em (i);

iii) Permitir aos desenvolvedores do produto o uso da Estação, através do ambiente definido em (i) e gerado em (ii), e,

iv) Permitir a execução do software na própria Estação.

Estas funções determinam quatro ambientes na Estação TABA (TRAVASSOS, 1994):

- **Ambiente Especificador e Instanciador de ADS (meta-ambiente TABA)** - responsável por auxiliar o engenheiro de software na especificação e instanciação do ADS mais adequado ao desenvolvimento de um produto específico;
- **Ambiente de Construção de Ferramentas** - responsável por auxiliar o engenheiro de software a implementar as ferramentas necessárias ao ADS definido e por incorporar essas ferramentas ao meta-ambiente TABA;
- **Ambiente de Desenvolvimento** - que é o ADS que foi especificado e instanciado através do meta-ambiente, e,
- **Ambiente de Execução** - que é o local onde o software pode ser executado.

Segundo Travassos (TRAVASSOS, 1994), para atender a essas funcionalidades a Estação TABA possui os seguintes requisitos: ser configurável, possuir interface consistente, possuir mecanismo de integração, apoiar a construção de novas ferramentas, possuir conhecimento sobre processos de software e métodos de desenvolvimento, oferecer assistência inteligente ao usuário, possuir um modelo de armazenamento de dados comum, e possuir suporte a reutilização.

Além desses requisitos gerais da Estação, os ADS configurados pelo meta-ambiente TABA também possuem os seguintes requisitos: possuir apoio para o controle e

gerenciamento de versões, possuir apoio para o gerenciamento de todas as atividades realizadas ao longo do processo de desenvolvimento, possuir apoio para medição do produto, apoiar o trabalho cooperativo, possuir interfaces customizáveis, e, possuir apoio para avaliação do produto.

Inicialmente a Estação TABA considerava apenas ADS definidos com base em um determinado processo de desenvolvimento de software. Por este motivo, podemos notar que os requisitos descritos anteriormente se encaixam com as funcionalidades necessárias para a definição de um processo de desenvolvimento de software. Considerando que as particularidades de domínios de aplicação também devem ser consideradas na instanciação de um ADS para que este ofereça ferramentas mais específicas e possibilite a reutilização do conhecimento do domínio, OLIVEIRA (1999a) estendeu a definição da Estação TABA de forma que esta possa instanciar ambientes de desenvolvimento de software para domínios específicos. Nas seções 2.2.3 e 2.2.4, são apresentadas as principais definições sobre esses ambientes e suas evoluções para considerar também o conhecimento organizacional necessário aos processos de desenvolvimento. Antes, porém, detalharemos a estrutura da Estação TABA e como está sendo realizada a integração de conhecimento.

2.2.1 A Estrutura da Estação TABA

A estrutura da Estação TABA foi definida como um conjunto de componentes integrados que possuem controle sobre a sua existência, suas informações, estados e funcionalidades básicas associados. Tal conjunto de componentes tem a responsabilidade de traduzir, de acordo com a necessidade do usuário, uma representação do mundo real para o mundo computacional (TRAVASSOS, 1994). Para atender a seus requisitos, a Estação TABA possui os seguintes componentes: sistema computacional, repositório comum do ADS, controle de versões, interface com o usuário, cooperação, controle de processos, reutilização, suporte inteligente e conhecimento:

A utilização, em conjunto, destes componentes permite a integração de ferramentas ao ambiente e provê recursos para a incorporação de ferramentas externas. Segundo TRAVASSOS (1994), esses componentes definem a filosofia de integração da Estação TABA e dos seus ambientes instanciados através de quatro tipos de integração.

i) **Integração de Dados** - estabelece a forma como as ferramentas da Estação TABA realizarão o tratamento das informações, devendo prover os serviços básicos de armazenamento e gerenciamento de estruturas de informação obtidas a partir das ferramentas componentes do ambiente.

ii) **Integração de Controle** - responsável por prover, às ferramentas e ao ADS, serviços e funcionalidades básicas que permitam o seu funcionamento de forma organizada, ao longo do processo de desenvolvimento.

iii) **Integração de Apresentação** - responsável por proporcionar ao usuário a sensação de integração no ambiente. É através dela que será possível a homogeneização das formas de apresentação das informações e das técnicas de interação do usuário.

iv) **Integração do Conhecimento** - torna possível os serviços básicos de armazenamento, gerenciamento e utilização do conhecimento descrito e adquirido ao longo do processo de desenvolvimento. O conhecimento compartilhado pelas ferramentas descreve o significado das informações construídas ao longo do processo de desenvolvimento e permite a descrição da sintaxe apropriada para a apresentação dessas informações por parte das ferramentas.

2.2.2 O Servidor de Conhecimento

A integração do conhecimento tem sido constante objeto de estudo no contexto da Estação TABA, sofrendo bastantes modificações desde sua concepção até o momento atual. Inicialmente, poderia se pensar que a integração do conhecimento seria mais facilmente atingida se fossem construídos Sistemas Baseados em Conhecimento (SBC) capazes de concentrar todo o conhecimento relevante para um ADS. Porém, é impossível antever e modelar todo este conhecimento, uma vez que o conhecimento deve ser capturado e modelado à medida em que novas necessidades são detectadas. A partir da constatação de que o ponto-chave para a integração de conhecimento estava na modelagem do conhecimento, FALBO (1998a) propôs a utilização de servidores do conhecimento no qual o conhecimento é modelado para reuso e disponibilizado em um conjunto de componentes que podem ser usados por várias ferramentas a serem desenvolvidas. Esta constatação levou a uma mudança no enfoque do estudo sobre integração de conhecimento e ao invés de se buscar a integração de conhecimento através da integração de tecnologias de

representação, passou-se a explorar a construção de componentes de conhecimento, fundamentados em ontologias (FALBO *et al.*, 1998b) e em modelos de tarefas. Componentes construídos a partir de ontologias de domínio descrevem o conhecimento sobre o domínio de interesse, enquanto que componentes construídos a partir dos modelos de tarefa descrevem conhecimento genérico de tarefa, aplicável a vários domínios.

Para permitir o reuso do conhecimento do domínio, é desejável que a base de conhecimento do Servidor seja modular e baseada em ontologias. Ontologias e suas instanciações podem ser implementadas em módulos de conhecimento, de modo que a base de conhecimento de uma aplicação que se comprometa com uma ou várias ontologias venha a ser a conjunção dos módulos de conhecimento correspondentes, mais o conhecimento específico da aplicação particular.

Quanto ao reuso do conhecimento de tarefa, o Servidor de Conhecimento provê *templates* de resolvedores genéricos de problema. Ao invés de prover apenas sistemas de representação e suas máquinas genéricas de inferência, um Servidor de Conhecimento deve prover uma biblioteca de resolvedores genéricos de problemas, passíveis de serem instanciados e adaptados para aplicações particulares. Em relação à grande variedade de problemas em um certo universo de discurso, é geralmente possível identificar alguns problemas que são típicos ou recorrentes. Assim, a discussão sobre a resolução de problemas em um universo de discurso deve levar em conta os tipos de problemas freqüentes nesse universo. Tipos de problemas são, de fato, uma importante diretriz para a formulação de modelos que podem ser efetivamente aplicados na construção de SBC para o domínio em questão. No contexto de Servidores de Conhecimento, estes modelos são a base para a construção dos componentes do conhecimento de tarefa.

Apesar das ontologias poderem ser utilizadas para modelar tanto o conhecimento de domínio quanto o de tarefa, o conhecimento sobre os tipos principais de tarefas já haviam sido bastante estudados (CHANDRASEKARAN *et al.*, 1993, SCHREIBER *et al.*, 1993, BREUKER *et al.*, 1994, entre outros) e FALBO (1998a) optou por utilizar diretamente os resultados destes trabalhos, mais especificamente os modelos de tarefa do *Common*KADS, para a construção dos componentes do conhecimento de tarefa.

A biblioteca de modelos de tarefa do *Common*KADS (VALENTE et al., 1994) foi, portanto, utilizada como referência para a elaboração dos modelos de tarefa que fundamentam os componentes do conhecimento de tarefa do Servidor de Conhecimento.

Alguns dos modelos de tarefa da biblioteca *Common*KADS, tipicamente aqueles projetados para os tipos de problemas que ocorrem com maior freqüência no universo de discurso, foram selecionados, adaptados e implementados na forma de *templates* de resolvedores de problemas, formando uma camada sobre a máquina de inferência geral.

Desta forma, a arquitetura de um Servidor de Conhecimento é composta de (figura 2.1):

- uma base de conhecimento modular, onde cada módulo de conhecimento contém um corpo de conhecimento reutilizável, construído com base em uma ontologia, e,
- a máquina de inferência do sistema de representação adotado, associada a um conjunto de *templates* de resolvedores de problemas, para especializá-la para os tipos de problema mais freqüentemente encontrados no universo de discurso.

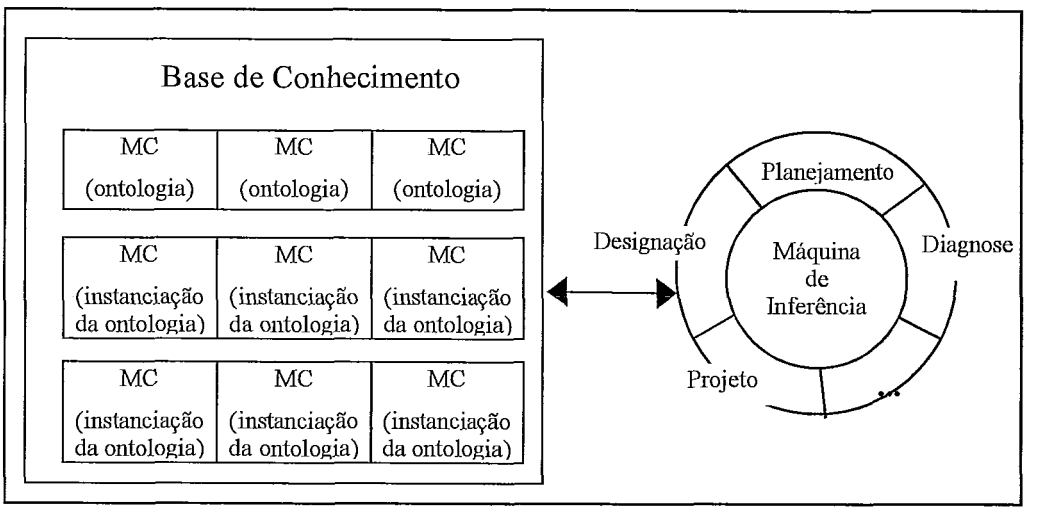

Figura 2.1: A arquitetura geral de Servidores de Conhecimento (FALBO, 1998)

Os módulos de conhecimento e os *templates* de resolvedores de tipos de problema são os componentes reutilizáveis que o Servidor de Conhecimento oferece para auxiliar o processo de construção de SBC no universo de discurso apoiado por ele. Além disso, guardam estreita relação entre si. Os papéis de conhecimento considerados em um *template* de resolvedor de problema devem ser preenchidos com o conhecimento do domínio descrito nos módulos de conhecimento.

No que se refere à tecnologia de representação utilizada para construção desses componentes, a Estação TABA provê uma classe de representação de conhecimento que permite a definição de diferentes tecnologias de representação: rede neural e sistemas

lógicos, englobando frames, sistema de programação em lógica e rede semântica. Atualmente, tem-se implementada a classe de sistema de programação em lógica usando Prolog através de uma máquina Prolog externa, o SWI-Prolog (WIELEMAKER, 1998).

A meta de um Servidor de Conhecimento no contexto de ADS é servir de base para a construção dos vários assistentes baseados em conhecimento de um ADS, garantindo uma semântica comum e permitindo o reuso e o compartilhamento do conhecimento. Um Servidor de Conhecimento provê resolvedores genéricos de problemas e bases modulares de conhecimento baseadas em ontologias, permitindo, assim, o reuso do conhecimento de tarefa e de domínio. Uma vez que o Servidor torna disponível conhecimento sobre um universo de discurso, o engenheiro de software pode se utilizar deste conhecimento para compreender este universo e para propor uma solução inicial para o problema. Desta forma, uma aquisição de conhecimento preliminar pode ser feita, sem consumir muito tempo dos especialistas. Através das ontologias do domínio, o engenheiro de software ganha um entendimento do problema e pode montar uma base de conhecimento inicial, enquanto que os modelos de tarefa podem ser usados para estabelecer uma estratégia inicial para a resolução do problema.

2.2.3 Ambientes de Desenvolvimento de Software Orientados a Domínio

O entendimento errôneo ou incompleto do problema que o software pretende resolver pode levar a produtos de software implementados corretamente, mas que resolvem os problemas errados. Segundo FISCHER (1996), problemas complexos exigem mais conhecimento do que uma só pessoa pode possuir, ou seja, a comunicação e a colaboração tornam-se fundamentais. A idéia central passou, então, a ser a de um entendimento compartilhado, onde os especialistas entendem a prática e os projetistas de sistemas conhecem a tecnologia. Tal compartilhamento precisa ocorrer durante todo o processo de desenvolvimento, uma vez que novos requisitos tendem a surgir durante o desenvolvimento, pois eles só podem ser identificados depois que parte do sistema já tiver sido projetada e implementada.

Com a intenção de facilitar a comunicação e, consequentemente, o entendimento do problema, surgiram os Ambientes de Desenvolvimento de Software Orientados a Domínio

(ADSOD), que buscam integrar o conhecimento do domínio aos Ambientes de Desenvolvimento de Software (OLIVEIRA *et al.*, 1999b).

ADSOD podem ser vistos como uma extensão dos tradicionais ambientes de desenvolvimento de software. Os ADSOD apóiam o desenvolvimento de software em domínios específicos através do uso do conhecimento deste domínio durante todo o processo de desenvolvimento para auxiliar o desenvolvedor no entendimento do problema (OLIVEIRA *et al.*, 1999c, OLIVEIRA *et al.*, 1999d). Podemos dizer que os ADSOD visam, além de oferecer ferramentas de apoio para as atividades normalmente apoiadas pelos ADS, disponibilizar o conhecimento do domínio de forma organizada e estruturada, para orientar o desenvolvimento de software. O uso do conhecimento do domínio durante o desenvolvimento de software tende a tornar o processo de entendimento do problema mais fácil e agradável, tanto para os desenvolvedores, quanto para os especialistas do domínio (FISCHER, 1994).

De forma geral, um ADSOD é caracterizado por possuir o conhecimento do domínio e permitir a investigação, o acesso e o uso desse conhecimento durante o desenvolvimento de software.

Na Estação TABA, o conhecimento do domínio é organizado em um modelo chamado Teoria do Domínio que pode ser dividido em sub-teorias. Cada sub-teoria considera os conceitos do domínio que estão semanticamente relacionados em um mesmo nível de abstração e organizados em uma ontologia do domínio, que será discutida no próximo capítulo. Faz, ainda, parte da Teoria do Domínio, a identificação das tarefas relacionadas àquele domínio e o mapeamento de quais sub-teorias são necessárias para o entendimento de cada tarefa. Esse mapeamento considera um conjunto de tarefas já previamente definidas, sendo apenas relacionados quais conceitos das ontologias de domínio são pertinentes na realização de cada tarefa.

Para que o domínio fosse considerado na instanciação de ADS, OLIVEIRA (1996) e VILLELA (1997) definiram novos requisitos para a Estação TABA. A partir do momento em que a Estação atende a estes requisitos, ela passa a poder instanciar ADS com base no processo de desenvolvimento de software, no domínio de aplicação ou em ambos (OLIVEIRA, 1999a). Os seguintes requisitos foram acrescentados aos requisitos gerais da Estação TABA:

i) **Apoiar a definição da Teoria do Domínio:** a Estação TABA deve possuir mecanismos que facilitem a definição de uma ontologia do domínio pelo engenheiro do domínio. Além disso, deve facilitar a descrição de tarefas que identifiquem as tarefas realizadas no domínio, como elas se relacionam e como se relacionam com os conceitos do domínio definidos na ontologia do domínio. Apoiar a descrição de tarefas é uma das principais facilidades que deve ser provida pela Estação TABA e, por este motivo, é um dos requisitos que procuramos atender neste trabalho. Para isso, será definida uma estrutura para a representação do conhecimento de tarefa;

ii) **Permitir a evolução de conceitos de domínio e tarefas:** a Estação TABA deve oferecer apoio ao engenheiro do domínio para que este descreva as tarefas a serem consideradas no ADSOD (caso esta descrição não exista) ou defina novos conceitos na ontologia do domínio. Este requisito, no que se refere às tarefas, também será atendido através da estrutura para a descrição de conhecimento de tarefa, que, além de apoiar a descrição, tem como objetivo permitir a sua evolução;

iii) **Permitir a construção de ferramentas específicas do domínio:** o ambiente de construção de ferramentas da Estação TABA deve poder utilizar a Teoria do Domínio na construção de ferramentas específicas do domínio;

iv) **Armazenar e indexar projetos desenvolvidos no domínio de acordo com a Teoria do Domínio:** a Estação TABA deve controlar o armazenamento e a indexação de projetos desenvolvidos no domínio, associados aos conceitos do domínio (definidos na ontologia do domínio e tarefas). Este requisito é necessário para que se possa permitir ao engenheiro de software construir ferramentas para facilitar o entendimento do domínio pelos desenvolvedores (que usam o ADSOD) através do entendimento de como conceitos do domínio foram utilizados em outros projetos, e,

v) **Permitir a especialização e instanciação de processos a partir de um processo padrão:** a Estação TABA deve permitir ao engenheiro de software definir um processo de software para o ADS a ser instanciado a partir de um processo padrão que possua atividades para investigação do domínio.

Os ADSOD instanciados pelo meta-ambiente TABA possuem, também, os seguintes requisitos:

i) **Apoiar a instanciação da Teoria do Domínio:** o ADSOD deve apoiar a instanciação de ontologias definidas na Teoria do Domínio de acordo com a aplicação a ser desenvolvida;

ii) **Facilitar o entendimento do domínio e da tarefa:** o ADSOD deve oferecer ferramentas específicas que auxiliem o desenvolvedor de software no entendimento do domínio e das tarefas realizadas no domínio para o qual será desenvolvida a aplicação, através da interação do desenvolvedor com o domínio do problema. O ADSOD deve ajudar o desenvolvedor a decompor a tarefa, direcionando-o para questões relevantes. Neste trabalho, procuramos auxiliar o entendimento da tarefa através do desenvolvimento de uma ferramenta para apoiar a descrição e evolução das tarefas a serem utilizadas no ambiente instanciado;

iii) **Oferecer apoio a diferentes tipos de usuário:** esse requisito refere-se ao fato de que, além dos desenvolvedores de software, os especialistas do domínio devem interagir com o ADSOD para introdução do conhecimento do domínio, ou seja, instanciação da ontologia do domínio;

iv) **Apoiar o acesso ao conhecimento:** esse requisito refere-se à necessidade do ADSOD oferecer mecanismos de acesso ao conhecimento, definido na Teoria do Domínio, nas diferentes atividades realizadas pelos desenvolvedores de software. Procuramos, neste trabalho, estender as definições do Servidor de Conhecimento, apresentado na seção anterior, de forma que o conhecimento de tarefa possa ser representado de acordo com a estrutura que estamos estabelecendo;

v) **Ser extensível:** isto é, permitir que novas ferramentas sejam incorporadas à medida em que se tornem necessárias, e,

vi) **Incorporar arquiteturas de referência:** o ADSOD deve permitir a definição de arquiteturas de software para os diferentes produtos desenvolvidos.

Em trabalhos anteriores, OLIVEIRA (1999a) definiu as bases para que os requisitos referentes à definição de ontologias de domínio e suas utilizações através da Teoria do Domínio fossem satisfeitos pela Estação TABA. Neste trabalho, abordaremos os pontos relevantes à descrição do conhecimento de tarefa a ser considerado nos ADSOD e no entendimento da tarefa para o domínio no qual a aplicação está sendo desenvolvida.

2.2.4 Ambientes de Desenvolvimento de Software Orientados a Organização

Os ADSOD estão sendo evoluídos para considerar não apenas conhecimento do domínio mas também o conhecimento organizacional necessário aos processos de desenvolvimento e manutenção de software e à gerência destes processos (VILLELA, ZLOT e SANTOS, 2001). A motivação para a construção de Ambientes de Desenvolvimento de Software Orientados a Organização (ADSOrg) surgiu da constatação de que duas ou mais organizações podem desenvolver software para um mesmo domínio com processos, interesses e características muito distintas, além da percepção de que o conhecimento do domínio não é o único conhecimento importante para apoiar desenvolvedores de software em suas atividades. Outros conhecimentos, tais como o conhecimento relativo às diretrizes e melhores práticas organizacionais e às lições aprendidas em experiências anteriores com o uso de processos, métodos e técnicas de software, também são extremamente importantes e úteis para os desenvolvedores. Tais conhecimentos, entretanto, variam de uma organização para outra.

Desta forma, um ADSOrg tem como objetivo permitir que desenvolvedores de software tenham acesso a todo o conhecimento acumulado pela organização e relevante para o contexto de desenvolvimento e manutenção de software, bem como promover o aprendizado organizacional neste contexto. Um ADSOrg também deve permitir a identificação de quem na organização é o detentor de determinado conhecimento ou habilidade e em que atividades dos processos da organização este conhecimento ou habilidade é necessário.

Três tipos de ADSOrg estão sendo considerados, cada um voltado para um tipo de organização: (i) para empresas que desenvolvem software para uso próprio; (ii) para empresas que desenvolvem software para terceiros, e; (iii) para empresas com desenvolvimento distribuído. Apesar de cada um dos tipos de ADSOrg possuir requisitos bem específicos, a partir dos objetivos estabelecidos foram identificados requisitos comuns: (i) possuir a representação da estrutura e dos processos organizacionais e possibilitar a fácil localização de especialistas cujo conhecimento e experiência podem ser úteis em projetos de software, (ii) armazenar conhecimento especializado sobre desenvolvimento e

manutenção de software, e fornecer este conhecimento para as equipes de projeto quando necessário, e, (iii) apoiar a contínua evolução do conhecimento armazenado no ambiente.

Para possibilitar que o requisito (i) seja satisfeito e desenvolvedores de software rapidamente encontrem, dentro da estrutura organizacional, os profissionais mais adequados para a realização de uma atividade ou para a solução de um problema, foi desenvolvida uma ferramenta que permite a realização de uma pesquisa direta para fornecer quem na organização possui determinado conhecimento ou habilidade (VILLELA, ZLOT e SANTOS, 2001). Além disso, a ferramenta permite uma pesquisa indireta que possibilita o usuário do ambiente navegar entre os conhecimentos e habilidades disponíveis na organização em busca de quem possui a habilidade ou o conhecimento mais próximo do desejado. Isto pode ser necessário quando nenhum profissional da organização possui a habilidade ou conhecimento requerido. Portanto, a ferramenta dispõe de mecanismos eficientes para busca, recuperação e visualização das informações por ela registradas e pode ser amplamente utilizada para obter informações sobre a estrutura organizacional e sobre os profissionais da organização, podendo tanto ser utilizada no planejamento de projetos, quanto no decorrer dos mesmos.

2.2.5 Ferramentas e Ambientes Instanciados

A Estação TABA foi implementada originalmente em Eiffel, na plataforma Unix da *Sun Workstation*, por ser considerada esta uma tecnologia robusta e poderosa, bastante adequada para trabalhos de pesquisa. Porém, após a introdução da filosofia de ADSOD pela Estação TABA, surgiu a necessidade de um desenvolvimento padronizado para organizações e para domínios específicos, o que se tornou um problema crítico, devido à falta de portabilidade do código para plataformas mais acessíveis e comumente utilizadas. Surgiu, então, a necessidade de se realizar uma reimplementação, onde se optou pela plataforma de microcomputadores e pelo uso da linguagem C++ (OLIVEIRA, 1999a, ZLOT e SANTOS, 1999).

Várias ferramentas já foram desenvolvidas a partir desta nova implementação, tanto para a edição de uma teoria do domínio ou processo, como para assistir as atividades de desenvolvimento de software. Segundo os seus objetivos, elas podem ser agrupadas da seguinte forma:

i) **Ferramentas para Definição do Processo de Desenvolvimento:** ferramentas desenvolvidas com o objetivo de permitir a especialização e instanciação de processos de desenvolvimento a partir de um processo padrão:

- EDIT-PRO (ZLOT e SANTOS, 1999), uma ferramenta para simples edição de um processo previamente definido pelo engenheiro de software;
- ASSIST-PRO (FALBO *et al.*, 1999b), uma ferramenta que fornece assistência inteligente na escolha do ciclo de vida e atividades mais adequadas ao sistema a ser desenvolvido, e,
- DEF-PRO (MACHADO, 2000), uma ferramenta mais completa que estabelece um modelo geral para definição, especialização e instanciação de Processos de Software na Estação TABA. Esse modelo considera a norma ISO/IEC 12207 (1998), os modelos de maturidade CMM (PAULK *et al.*, 1995) e SPICE (EMAM *et al.*, 1998), as características da organização, tipo de software, tecnologia de desenvolvimento e o tipo de ambiente que se deseja instanciar (orientados ao domínio ou não).

ii) **Ferramenta para Definição da Teoria do Domínio:** ferramentas desenvolvidas com o objetivo de apoiar a definição da teoria do domínio e permitir sua evolução:

- EDITED (OLIVEIRA *et al.*, 2000), uma ferramenta para edição da Teoria do Domínio, sendo utilizada para permitir a introdução do conhecimento do domínio no ambiente;
- REGCOM, GENESIS e NAVEGUE (GALOTTA, 2000), ferramentas para apoiar o registro de uso de conceitos da Teoria do Domínio, auxiliar o registro de instâncias de conceitos da Teoria do Domínio e assistir ao aprendizado do domínio, respectivamente.

iii) **Ferramenta para a Descrição de Tarefas:** ferramenta desenvolvidas com o objetivo de atender ao requisito de apoiar a descrição de tarefas e permitir sua evolução.

- EDITAR (OLIVEIRA, 1999a), uma ferramenta para edição de tarefas feita através de simples documentos que descrevem as principais características da tarefa e referências bibliográficas para a mesma. A ferramenta não tem por objetivo definir como a tarefa é realizada, ou mesmo auxiliar no entendimento do domínio do problema.

Já foram definidos e implementados três ambientes de desenvolvimento de software orientados a domínio na Estação TABA:

- CORDIS (OLIVERIA, 1999): é um ambiente orientado ao domínio de Cardiologia, definido em uma parceria com a Fundação Bahiana de Cardiologia / Unidade de Cardiologia e Cirurgia Cardiovascular da Universidade Federal da Bahia;
- NETUNO (GALOTTA, 2000): é um ambiente orientado ao domínio de acústica submarina, desenvolvido de acordo com as particularidades do GAS/IPqM (Grupo de Acústica Submarina do Instituto de Pesquisas da Marinha), e,
- INSECTA (FOURO, 2002): é um ambiente orientado ao domínio de entomologia, desenvolvido em parceria com o Laboratório de Entomologia da Embrapa Amazônia Oriental.

2.3 Conclusões do Capítulo

O conceito de Ambiente de Desenvolvimento de Software vem sendo objeto de estudo desde a década de 70 e apresentamos, neste capítulo, sua evolução desde ferramentas isoladas até ambientes que atendem às necessidades de atividades específicas do desenvolvimento de software, como por exemplo ambientes centrados em processo, que defendem a necessidade da integração de ferramentas incorporadas a um processo de desenvolvimento de software específico da organização.

Para possibilitar a construção de ambientes de desenvolvimento de software adequados às particularidades de processos de desenvolvimento e de projetos específicos, foi desenvolvida a Estação TABA (ROCHA *et al*, 1990). A partir da idéia de projeto orientado a domínio, OLIVEIRA (1999a) introduziu a noção de Ambientes de Desenvolvimento de Software Orientados a Domínio (ADSOD), que procuram apoiar a atividade de entendimento do problema com o uso do conhecimento do domínio, além de se preocupar com a organização e disponibilização do conhecimento para um ADS projetado especificamente para um determinado domínio. O modelo da Estação TABA foi, então, revisto e estendido, de forma a permitir a instanciação de ADSOD para qualquer domínio desejado, como também a instanciação de Ambientes de Desenvolvimento de Software Orientados a Organização (ADSOrg) (VILLELA, ZLOT e SANTOS, 2001) que consideram não apenas o conhecimento do domínio mas também o conhecimento

organizacional necessário aos processos de desenvolvimento e manutenção de software e à gerência destes processos.

Para atender às suas particularidades, o meta-ambiente TABA possui um conjunto de componentes que abrange cooperação, controle de processo, interface com usuários, conhecimento, entre outros. O componente Conhecimento incorpora mecanismos para o armazenamento e a utilização de conhecimento, que podem ser utilizados tanto pelo meta-ambiente quanto pelos ADS instanciados.

Entretanto, a infra-estrutura de conhecimento da Estação TABA não era, originalmente, suficiente para promover a integração do conhecimento. O componente de conhecimento foi, posteriormente, melhor definido numa arquitetura genérica, denominada servidores de conhecimento (FALBO, 1998a), que provê um vocabulário comum, baseado em ontologias, com interpretação definida dos termos no universo de discurso e, uma biblioteca de tarefas para os tipos de problema mais comumente encontrados no universo de discurso em questão. Falbo optou pela utilização do modelo de tarefas, em detrimento ao uso de ontologias de tarefa, devido os trabalhos sobre modelos de tarefas já estarem bastante desenvolvidos.

Nesta tese, optamos por aprofundar a pesquisa sobre ontologias de tarefa para permitir a sua utilização, junto com métodos de solução de problemas, na representação do conhecimento de tarefa, de forma a estender o auxílio do Servidor de Conhecimento no desenvolvimento de sistemas. Métodos de solução de problemas, como será visto no capítulo 3, têm por objetivo manipular os conceitos definidos na tarefa de forma a permitir que seu objetivo seja alcançado. No próximo capítulo, definiremos as principais questões relacionadas a ontologias de tarefa e métodos de solução de problemas antes de apresentarmos como podemos modelar o conhecimento de tarefa em ambientes de desenvolvimento de software.

Capítulo 3

Ontologias de Tarefa e Métodos de Solução de Problemas

Este capítulo apresenta os principais componentes de ontologia de tarefa e métodos de solução de problemas visando suas utilizações para representar o conhecimento de tarefa. Inicialmente, apresentamos as principais definições e classificações de ontologia para, a seguir, apresentarmos o papel da ontologia de tarefa na definição dos conceitos envolvidos em um problema e os principais trabalhos sobre métodos de solução de problemas, assim como seus objetivos e estrutura.

3.1 Ontologia

O termo Ontologia já é conhecido e aplicado há bastante tempo na área da Filosofia, significando um sujeito de existência ou uma explicitação sistemática da existência. Ou seja, ontologia é a parte da filosofia que trata do ser enquanto ser. Na década de 90, este conceito passou a ser utilizado na Ciência da Computação. Ontologias foram aplicadas primeiramente na área de Inteligência Artificial (IA) como uma teoria lógica que restringe os modelos de uma linguagem lógica (GRUBER 1993, GUARINO 1997). Neste sentido, dado um conjunto de predicados e funções de uma linguagem lógica, uma ontologia provê axiomas que restringem o sentido dos predicados ou permitem a realização de inferências. A partir de então, vem crescendo o interesse em ontologias como forma de aquisição e representação do conhecimento e diversas definições para o termo têm sido propostas.

GRUBER (1993, 1995) definiu ontologia como sendo uma especificação explícita da conceituação, onde conceituação é um conjunto abstrato de objetos, conceitos e outras entidades que se assume existirem em um certo domínio, assim como as relações que podem ser validadas entre eles. Uma conceituação é, portanto, uma visão simplificada e abstrata do mundo que se deseja representar (VALENTE, 1995). Segundo GRUBER

(1993), o que existe é o que pode ser representado e, portanto, a ontologia consiste de um vocabulário para a representação do conhecimento composto por uma descrição precisa dos termos do vocabulário e um conjunto formal de axiomas que restringe a interpretação e o uso desses termos.

GUARINO e GIARRETA (1995) enfraqueceram a definição dada por GRUBER ao definirem que ontologia é uma descrição parcial e explícita de uma conceituação, por considerarem que o grau de especificação de uma conceituação de uma linguagem utilizada para uma base de conhecimento depende do propósito desejado para a ontologia. Um dos propósitos é ser um sinônimo de teoria ontológica, ou seja, um conjunto de axiomas (fórmulas) que são considerados sempre verdadeiros independente dos valores particulares para os mesmos, sendo, portanto, compartilhável entre diferentes agentes. Nesse sentido, a ontologia se aproxima de uma conceituação independente e que pode ser utilizada para estabelecer o compartilhamento de uma base de conhecimento particular.

Existem outras definições sobre ontologia, como a de FIKES e FARQUHAR (1999), que definem ontologia como uma teoria sobre um domínio que especifica um vocabulário de entidades, classes, propriedades, predicados e funções e um conjunto de relações que necessariamente amarram esse vocabulário. A definição adotada no presente trabalho é a mesma adotada por OLIVEIRA (1999a), ou seja, a de USCHOLD e GRUNINGER (1996). Esta definição procura englobar todos os conceitos definidos pela ontologia, sendo, portanto, mais completa. Segundo estes autores, em uma ontologia os conceitos devem ser explicitamente definidos e representados de forma que possam ser compartilhados e possuem restrições expressas através de axiomas que explicitamente restringem os significados dos termos. Esses conceitos podem incluir estruturas conceituais para modelagem do conhecimento do domínio, protocolos para comunicação e acordos sobre a representação de teorias de um domínio particular. OLIVEIRA (1999a) considera, ainda, que esta ontologia é uma descrição parcial, projetada para ser compartilhada dentro de uma comunidade que concorda com a sua definição, e para o fim específico de desenvolvimento de software em um dado domínio.

Em uma ontologia, os axiomas são utilizados para representar as restrições entre os conceitos e satisfazem, desta forma, a competência da ontologia. Em outras palavras, a classe de questões cujas respostas podem ser derivadas a partir dos axiomas especifica o compromisso ontológico (IKEDA, SETA e MIZOGUCHI, 1997). A definição dos axiomas

é uma das atividades que apresenta maior complexidade no desenvolvimento de uma ontologia, uma vez que não é desejável escrever mais axiomas do que o necessário para caracterizar as soluções das questões de competência. FALBO (1998a) classifica os axiomas em três tipos:

i) **Axiomas epistemológicos:** derivados simplesmente da estrutura dos conceitos e não de seus significados particulares;
ii) **Axiomas de consolidação:** têm por objetivo verificar a coerência das informações existentes e não representam conseqüências lógicas, isto é, não derivam novas informações, e,
iii) **Axiomas ontológicos:** representam restrições entre os conceitos que compõem a ontologia.

HWANG (1999) listou algumas características desejáveis para uma ontologia:

i) **Ser aberta e dinâmica:** uma ontologia deve ser capaz de permitir ajustes decorrentes de mudanças na estrutura ou comportamento do domínio;
ii) **Ser escalável e interoperável:** uma ontologia deve ser facilmente escalável, considerando um domínio amplo, e adaptável a novos requisitos. Deve também ser possível integrar várias ontologias em uma nova ontologia, quando o tratamento de diferentes vocabulários conceituais é requerido;
iii) **Ser de fácil manutenção:** a manutenção de uma ontologia não deve ser uma tarefa complexa e para isso a ontologia deve ser de fácil compreensão;
iv) **Ter consistência:** uma ontologia deve ser semanticamente consistente com o domínio, e,
v) **Ser coerente com o contexto:** uma ontologia não deve ter termos muito específicos para não tornar complexa a associação com as fontes de dados e futuras integrações com outras ontologias.

Um dos fatores mais importantes de uma ontologia é o seu nível de conhecimento da conceituação. Segundo VAN HEIJST *et al.* (1997), o nível de conhecimento da conceituação de uma ontologia é dependente do domínio e da tarefa particulares para a qual é projetada, pois considera que a representação do conhecimento é fortemente afetada pela natureza do problema e a estratégia de inferência utilizada para resolvê-lo. A tarefa da aplicação determina, portanto, qual o conhecimento que deve ser codificado. Além disso, o

conhecimento deve ser codificado de forma que a estratégia de inferência utilizada possa funcionar de forma eficiente.

Essa definição é contestada por GUARINO (1997), que afirma que uma parte específica do conhecimento pode, de certa forma, ser relevante para uma tarefa, mas que isso não significa que esse conhecimento é peculiar e particular para esta tarefa. Neste trabalho seguiremos esta abordagem, uma vez que concordamos com a afirmação de que a reutilização entre várias tarefas e métodos deve sempre ser buscada e que isso implica ser cada vez mais independente da tarefa (GUARINO, 1997).

3.1.1 Aplicação de Ontologias

Atualmente, o que impede o compartilhamento de conhecimento é o fato de que diferentes sistemas usam diferentes conceitos e termos para descrever seus domínios. Essas diferenças tornam difícil a transferência do conhecimento de um sistema para outro. Para resolver este problema, ontologias são desenvolvidas para serem usadas como base para múltiplos sistemas, permitindo compartilhar uma terminologia comum, o que facilita o compartilhamento e o reuso do conhecimento. Desta forma, se forem desenvolvidas ferramentas que permitam a combinação de ontologias e a tradução entre elas, seria possível o compartilhamento mesmo entre sistemas baseados em diferentes ontologias.

Uma vez que ontologias permitem o compartilhamento e o reuso do conhecimento, é importante deixarmos claro a diferença entre esses conceitos (MIZOGUCHI, 1994). O compartilhamento de conhecimento ocorre quando múltiplos agentes se utilizam de uma parte ou de todo o conhecimento de uma base de conhecimento construída por outro agente de software. A reutilização de conhecimento ocorre quando um agente se utiliza de parte ou de toda a base de conhecimento construída por outro agente com fins diferentes do seu.

Apesar da maioria das ontologias que estão sendo desenvolvidas terem como objetivo buscar alguma forma de compartilhamento ou reuso, alguns projetos usam ontologias para estruturação da base de conhecimento, enquanto outros utilizam ontologias como parte da base de conhecimento. É necessário que seja feita uma distinção entre ontologia e base de conhecimento, uma vez que estes termos são, muitas vezes, referidos como tendo o mesmo significado. Podemos citar algumas diferenças básicas entre ontologias e base de conhecimento (GÓMEZ-PÉREZ *et al.*, 1995): (i) as definições de ontologias são mais

gerais do que o conhecimento de uma base de conhecimento, de forma que estas podem ser compartilhadas entre diferentes sistemas, devendo ser independentes do sistema que irá compartilhar ou reutilizar; (ii) as ontologias geralmente não têm métodos de raciocínio que processam sobre suas definições como é o caso das bases de conhecimento, e, (iii) a linguagem da ontologia é mais expressiva, declarativa, independente do domínio e semanticamente bem definida do que as tecnologias de representação do conhecimento. Ou seja, a ontologia fornece a estrutura básica sobre a qual a base de conhecimento pode ser construída, provendo um conjunto de conceitos e termos para descrever um determinado domínio, enquanto uma base de conhecimento usa esses termos para representar o que é verdade para um mundo real ou hipotético (SWARTOUT, 1999). A ontologia, também, pode ser vista como uma interface entre a base de conhecimento e o mundo externo.

ABU-HANNA e JANSWEIJER (1994) descrevem quatro utilizações gerais de ontologias no desenvolvimento de sistemas baseados em conhecimento:

i) **Compartilhamento**: para que as aplicações possam compartilhar bases de conhecimento é necessário que concordem na mesma definição de termos, o que é determinado pela ontologia;

ii) **Aquisição de conhecimento:** uma ontologia inclui terminologias específicas para a aplicação que podem ser utilizadas em ferramentas de aquisição que trabalham diretamente com especialistas do domínio, o que efetivamente evita erros no conhecimento adquirido e permite que o conteúdo do conhecimento possa ser aplicado;

iii) **Organização do conhecimento:** uma ontologia permite que as bases de conhecimento sejam melhor acessadas, modificadas e reutilizadas, e,

iv) **Processo de raciocínio:** uma ontologia pode ser utilizada junto com a base de conhecimento como um modelo que uma tarefa pode considerar durante sua execução.

No que se refere ao desenvolvimento de software, USCHOLD e GRUNINGER (1996) descrevem algumas vantagens do uso de ontologias, dentre as quais podemos citar:

- **Comunicação entre as pessoas e a organização:** as ontologias reduzem os conflitos conceituais e terminológicas dentro da organização, uma vez que provêem um *framework* unificado para a mesma. Desta forma, as ontologias permitem um entendimento compartilhado e a comunicação entre pessoas com diferentes necessidades e pontos de vista particulares em um determinado contexto;

- **Interoperabilidade entre sistemas:** refere-se à necessidade de diferentes usuários que precisam trocar dados ou utilizarem diferentes ferramentas. Ontologias podem ser usadas como uma interlíngua para apoiar a tradução entre diferentes linguagens e representações. Dessa forma, para *n* linguagens, em vez da necessidade de se criar um tradutor para cada par de linguagens (o que significa uma necessidade de n^2 tradutores), seria necessário apenas um tradutor da linguagem para a ontologia, que funciona como interlíngua, e desta para a linguagem (ou seja, *2n* tradutores), e,
- **Apoio à Engenharia de Software:** ontologias podem ser benéficas na especificação, confiabilidade do produto e reutilização durante a engenharia de software. No que diz respeito à especificação, ontologias podem ser utilizadas para promover o entendimento compartilhado de um problema ou tarefa, o que auxilia no processo de identificação dos requisitos do software. Ontologias podem prover, ainda, uma especificação declarativa de um software, o que permite aos desenvolvedores raciocinarem sobre quais são os objetivos do produto, ao invés de como é a sua funcionalidade. Em relação à confiabilidade, ontologias podem servir para a verificação manual do projeto em relação à sua especificação. Ontologias formais permitem uma verificação semi-automatizada do produto de acordo com suas especificações declarativas e podem ser utilizadas para tornar explícitos os acordos entre diferentes componentes de software, facilitando a sua integração. Finalmente, para serem efetivas, ontologias devem apoiar a reutilização, de forma que se possa importar e exportar módulos entre diferentes sistemas. Através da caracterização de classes de domínios e tarefas, ontologias fornecem uma estrutura de trabalho para determinar que aspectos de uma ontologia são reutilizáveis entre diferentes domínios e tarefas. Isso significa que ontologias podem prover uma biblioteca para reutilização de objetos para domínios e problemas de modelagem.

3.1.2 Linguagens para Definição

Para construção de uma ontologia, é necessária a representação explícita dos conceitos ontológicos em alguma linguagem formal. Segundo GUARINO (1995), uma linguagem formal só é adequada ontologicamente se, no nível sintático, possuir granularidade e capacidade suficientes para expressar os postulados de significação de suas próprias

primitivas ou se, no nível semântico, for possível dar uma interpretação lógica formal para suas primitivas básicas. Existem várias linguagens que podem ser utilizadas para a representação de uma ontologia:

- **Lógica de primeira ordem:** linguagem muito utilizada para representar ontologias, por ser uma linguagem geral, bem conhecida e expressiva. Além disso, adiciona poucos compromissos ontológicos (VALENTE, 1995);
- **KIF ("*Knowledge Interchange Format*"):** linguagem formal construída para trabalhar como uma "interlíngua", isto é, um meio para trocar conhecimento entre bases construídas em diferentes linguagens (GRUBER, 1992);
- **Ontolingua:** provê um mecanismo para escrever ontologias em um formato canônico, de forma que estas ontologias possam ser facilmente traduzidas para uma variedade de representações e sistemas de raciocínio. Ontolingua foi especificamente projetada para ser um ambiente de desenvolvimento de ontologias (GRUBER, 1992, 1993);
- **CML ("*Conceptual Modelling Language*"):** linguagem semi-formal que define uma ontologia a partir da especificação dos conceitos, atributos, expressões, estruturas e relações, utilizando, ainda, uma representação gráfica (SCHREIBER *et al.*, 1994);
- **LOOM:** linguagem que visa ser usada na construção de sistemas especialistas e outras aplicações inteligentes. É uma linguagem baseada na lógica de descrições, alcançando uma estreita integração entre os paradigmas baseados em regras e em *frames* (BRILL, 1993);
- **SHOE ("*Simple HTML Ontology Extension*"):** linguagem que possibilita a reunião de informações significativas em páginas Web e documentos, melhorando os mecanismos de busca e compartilhamento de conhecimento (LUKE e HEFLIN, 2000);
- **OIL ("*Ontology Interchange Language*"):** linguagem proposta para reunir padrões para descrever e compartilhar ontologias. Foi projetada para prover a maioria das primitivas de modelagem comumente usadas em abordagens baseadas em *frames* e em lógica de descrições (HORROCKS *et al.*, 2000);
- **TOL ("*Task Ontology representation Language*"):** linguagem que especifica o significado dos conceitos genéricos que descrevem o conhecimento sobre a resolução do problema e fornece algumas primitivas para a construção de ontologias de tarefa. (IKEDA *et al.*, 1997), e,

- **LINGO (Linguagem Gráfica para Ontologias):** linguagem gráfica com representação formal explicitamente definida em lógica de 1ª. ordem (FALBO, 1998a). LINGO foi desenvolvida a partir da idéia de que devemos utilizar uma representação gráfica específica para representar ontologias e não uma representação existente como entidade-relacionamento ou linguagem de modelagem de objetos, devido ao fato de que não se deseja incorporar na descrição da ontologia a semântica natural dessas linguagens.

3.1.3 Classificação de Ontologias

Existem diversas formas de se classificar uma ontologia. Segundo USCHOLD (1996), uma ontologia pode ser classificada em três dimensões de acordo com: (i) o propósito para o qual é definida; (ii) o grau de formalidade com que o vocabulário é criado e seu significado é definido, e, (iii) a natureza do assunto que a ontologia está caracterizando. Esta variação, implicitamente, dá origem a diferentes tipos de ontologias.

Na primeira dimensão, a ontologia pode ser classificada com relação à (i) comunicação entre as pessoas e a organização; (ii) interoperabilidade entre sistemas, e (iii) apoio à engenharia de software, conforme foi discutido na seção 3.1.1.

Na segunda dimensão, uma ontologia pode ser definida como (i) *altamente informal*: expressa imprecisamente em linguagem natural; (ii) *semi-informal*: expressa em uma forma restrita e estruturada de linguagem natural, de forma a aumentar a clareza e diminuir a ambigüidade; (iii) *semi-formal*: expressa formalmente em uma linguagem definida, e, (iv) *rigorosamente formal*: os termos são meticulosamente definidos com semântica formal, teoremas e provas de propriedades.

Finalmente, a principal classificação de uma ontologia é com relação ao propósito para o qual ela é definida. Nesta dimensão, as ontologias podem ser classificadas em três categorias:

- **Ontologias de domínio:** expressam conceituações específicas para domínios particulares;
- **Ontologias de tarefa:** especificam as conceituações necessárias para se definir uma tarefa, e,

- **Ontologias de representação:** explicam as conceituações utilizadas nos formalismos de representação do conhecimento, sendo neutras em relação às entidades do mundo (ou seja, o domínio).

De acordo com VAN HEIJST *et al.* (1997) e VAN HEIJST (1995), o propósito da ontologia classifica ontologias de domínio e de representação, definidas anteriormente, além de introduzir mais duas categorias: ontologias genéricas e ontologias de aplicação. As ontologias genéricas definem conceitos genéricos entre as várias áreas, enquanto que as ontologias de aplicação são uma combinação dos conceitos das ontologias de domínio e das ontologias genéricas e podem conter extensões para a tarefa e método específico;

GUARINO (1997), por sua vez, considera a existência de ontologias de alto-nível, ontologias de aplicação, ontologias de domínio, ontologias de tarefa e ontologias de método para representarem o propósito da ontologia. As ontologias de alto-nível correspondem às ontologias genéricas definidas por VAN HEIJST *et al.* (1997), enquanto que as ontologias de aplicação são uma especialização das ontologias de alto-nível, que incluem ontologias de domínio (ex. medicina; comércio), ontologias de tarefa (ex. parto; venda) e ontologias de método (ex. parto normal, parto cesáreo; vender no atacado, vender no varejo).

Uma ontologia de método (FALASCONI e STEFANELLI, 1994) (GENNARI, TU, ROTHENFLUSH e MUSEN, 1994) especifica um vocabulário de termos e relações necessários para representar um método de solução de problemas, que é o responsável por fornecer soluções para as tarefas. Métodos de solução de problemas, ontologias de método e ontologias de tarefa serão melhor apresentados na próxima seção, onde abordaremos os elementos utilizados para a representação do conhecimento de tarefa. Finalmente, Guarino considera as ontologias de representação com um tipo separado por serem uma ontologia de meta-nível que descreve uma classificação das primitivas sobre a linguagem de representação.

Neste trabalho, optamos pela classificação de GUARINO (1997), uma vez que ela apresenta, de maneira explícita, uma separação entre ontologia de domínio e ontologia de tarefa, sendo a combinação das duas particular para uma ontologia de aplicação.

3.2 Conhecimento de Tarefa

O primeiro passo para que possamos identificar os elementos necessários para a representação do conhecimento de tarefa é entender o que é uma tarefa. Apesar dos termos tarefa e problema serem muitas vezes referidos como tendo o mesmo significado, existem distinções básicas entre eles: (i) se por um lado dizemos que um problema pode ser resolvido, por outro lado dizemos que uma tarefa é executada. A tarefa sempre diz respeito a algo que possa ser repetido, que possa ser distribuído e que contenha um plano, isto é, métodos de solução de problemas para a sua execução; (ii) uma tarefa também difere de um problema considerando-se que na tarefa o método de solução de problemas é conhecido, enquanto no problema ele ainda tem que ser descoberto; (iii) uma tarefa é caracterizada em termos do problema a ser resolvido, e, (iv) pessoas têm problemas e não tarefas, uma vez que problemas significam desvios e complicações.

Os primeiros trabalhos que utilizaram tarefas para apoiar projetos ligados à ciência da computação surgiram a partir da necessidade da criação de uma metodologia capaz de melhorar a qualidade e a confiabilidade de sistemas ligados à Inteligência Artificial (YEN e LEE, 1993). Foi desenvolvido, então, o Modelo de Tarefas. Estes modelos são descritos em termos de estruturas de funções, onde uma função é uma descrição do objetivo de uma tarefa e de seus conceitos de entrada-saída. A entrada e a saída de uma função são representadas por papéis que o conhecimento do domínio pode desempenhar na resolução de problemas.

O Modelo de Tarefas foi, então, transportado para a área de desenvolvimento de software, com o objetivo de se criar uma análise baseada em tarefas para resolver problemas no projeto de sistemas (CHANDRASEKARAN, 1990). A maioria desses problemas são decorrência da grande quantidade de atividades no projeto e de métodos que podem ser utilizados para resolver cada uma destas atividades.

Uma abordagem diferente à do Modelo de Tarefa surgiu com o objetivo de combinar a estrutura do conhecimento e as estratégias de inferência sobre um determinado tipo de problema. Foi criado, então, o conceito de Tarefas Genéricas(CHANDRASEKARAN, 1986) com o objetivo de fornecer um vocabulário para descrever os problemas, assim como para desenvolver sistemas baseados em conhecimento para resolução de tais problemas. Desta forma, Tarefas Genéricas definem não só tarefas com características comuns, como

também o método para realizá-las. Segundo BYLANDER e CHANDRASEKARAN (1987), a seguinte sequência de passos deve ser realizada na utilização de uma tarefa genérica: (i) seleção do problema a ser resolvido; (ii) identificação da tarefa genérica que é aplicável ao problema, sendo que o problema precisa se encaixar no tipo de problema que a tarefa genérica resolve, e, (iii) aplicação da tarefa genérica para especificar o tipo de estratégia e o tipo de conhecimento para resolver o problema.

A principal diferença entre as Tarefas Genéricas e o Modelo de Tarefa reside no fato que o primeiro leva em consideração as estratégias de inferência sobre um determinado tipo de problema e não somente a estrutura de conhecimento da tarefa. A partir da definição de tarefas genéricas, diversos autores propuseram definições para o termo tarefa no que se refere à estrutura de conhecimento e à solução do problema.

BREUKER (1994) define tarefa como sendo composta por seu objetivo e pela definição do problema, enquanto que MIZOGUCHI *et al.* (1995) consideram que uma tarefa pode ser vista como uma seqüência de passos necessários para a resolução de um problema. Enquanto que, na primeira definição, a tarefa é descrita do ponto de vista estrutural, ou seja, os componentes de uma tarefa são listados de forma estática, na segunda definição é apresentada uma abordagem relativa à forma como a tarefa é utilizada para resolver um problema. Consideramos que SPEEL e ABEN (1997) unificam estas duas definições, ao afirmarem que a tarefa é composta tanto pela definição dos seus conceitos, como por métodos que manipulam esses conceitos de forma que o objetivo proposto pela tarefa possa ser alcançado. Os métodos responsáveis pela realização da tarefa são chamados de Métodos de Solução de Problemas (MSP).

No entanto, existem divergências na literatura quanto ao relacionamento existente entre a tarefa e os métodos de solução de problemas. CHANDRASEKARAN *et al.* (1998), por exemplo, afirmam que o método é composto por uma coleção de subtarefas, as informações de controle sobre a chamada das subtarefas e a especificação de qualquer informação passada entre as subtarefas. De forma semelhante MIZOGUCHI, R., SINITSA, K., IKEDA, M. (1996) definem que método é similar a tarefa, uma vez que se constitui da estrutura da tarefa mais o controle. Neste trabalho seguimos a abordagem de SCHREIBER, G., WIELINGA, B. e BREUKER, J. (1993), que definem MSP como sendo apenas a estratégia de inferência necessária para que o objetivo da tarefa possa seja alcançado, sendo desta forma o método uma parte da tarefa e não o contrário.

A partir das definições apresentadas, consideramos que uma tarefa é composta por seu objetivo, a definição dos seus conceitos e por métodos responsáveis por determinar uma solução para a tarefa. Na próxima seção, apresentaremos a ontologia de tarefa para a representação do objetivo e dos seus conceitos envolvidos na tarefa. Em seguida serão descritos os métodos de solução de problemas.

3.2.1 Ontologia de Tarefa

A ontologia de tarefa foi baseada nos modelos de tarefa e, portanto, leva em consideração somente a estrutura de conhecimento da tarefa (YEN e LEE, 1993). Ao contrário das ontologias de domínio, que são o tipo de ontologia mais comumente desenvolvido e possuem diversos trabalhos publicados na literatura, o estudo de ontologias de tarefa é uma vertente mais recente do estudo de ontologias. Sua principal motivação é facilitar a integração dos conhecimentos de tarefa e de domínio em uma abordagem uniforme e consistente, tendo por base o uso de ontologias.

A ontologia de tarefa tem o intuito de permitir o reuso e compartilhamento do conhecimento de tarefa e seu conceito foi discutido extensivamente na comunidade de pesquisa interessada em aquisição do conhecimento. MIZOGUCHI (1992), por exemplo, define uma ontologia de tarefa como sendo responsável por formalizar o conhecimento sobre a resolução de problemas independentemente do domínio. Posteriormente, foi desenvolvido um sistema para aquisição do conhecimento de tarefa denominado MULTIS (MIZOGUCHI, 1992) (TIJERINO, 1993), baseado em ontologias de tarefa. As ontologias de tarefa do sistema MULTIS foram desenvolvidas com o auxílio de avaliadores de diversos sistemas especialistas.

Seguindo a definição de GRUBER (1993), que define ontologia como sendo o conceito de reutilização de um conjunto de objetos pertencente ao universo do sistema, MOTTA e DRAHAL (1998) definiram ontologia de tarefa como sendo responsável por conceituar este conjunto de objetos abstratos para um problema específico. A partir desta idéia foi desenvolvido o projeto IBROW (MOTTA e LU, 2000), que descreve uma biblioteca de ontologias, tarefas e métodos de solução de problemas. Esta biblioteca permite o desenvolvimento de ferramentas para a *web* que auxiliam a construção de sistemas através do reuso dos componentes de conhecimento. O principal objetivo do projeto IBROW é,

portanto, desenvolver um serviço inteligente capaz de recuperar componentes de software de bibliotecas distribuídas pela *internet* e configurá-los para uma aplicação particular, de acordo com as necessidades do usuário.

Neste trabalho seguimos a definição proposta por MIZOGUCHI, VANWELKENHUYSEN e IKEDA (1995) que descrevem ontologia de tarefa como o vocabulário para descrever a estrutura de resolução de problemas de todas as tarefas existentes que sejam independentes do domínio da aplicação, o que pode ser obtido analisando as estruturas de tarefas de problemas do mundo real. Além de ser uma definição mais abrangente, estes autores também descrevem de forma explícita os detalhes de uma ontologia de tarefa. Ainda segundo estes autores, uma ontologia de tarefa é basicamente composta por duas partes: a taxonomia e os axiomas. A taxonomia é um sistema hierárquico de conceitos, enquanto que os axiomas estabelecem regras e princípios sobre estes conceitos. Os principais objetivos de uma ontologia de tarefa são:

i) Prover primitivas amigáveis de forma a permitir que os usuários possam facilmente descrever suas próprias tarefas;
ii) Permitir ao sistema simular o processo de solução do problema no nível conceitual e mostrar aos usuários o processo de execução em termos das primitivas no nível conceitual, e,
iii) Permitir ao sistema tornar a descrição da tarefa executável através da tradução para código no nível de símbolos.

Para incorporar cada um dos objetivos listados acima, foram descritos três modelos de representação da ontologia de tarefa (MIZOGUCHI, SINITSA e IKEDA, 1996):

i) **Modelo Léxico** – relaciona-se com os aspectos sintáticos da descrição da resolução do problema e é composto por:
   - **Substantivos Genéricos:** representam os objetos que refletem seus papéis no processo de solução do problema. Ex: "Problema", "Questão", "Exemplo", "Acidente", "Operação";
   - **Verbos Genéricos:** representam as atividades que aparecem no processo de solução do problema. Ex: "Fornecer", "Mostrar", "Simular";
   - **Adjetivos Genéricos**: modificam os objetos. Ex: "Resolvido", "Fácil","Correto", e,

- Outros conceitos específicos da tarefa.

ii) **Modelo Conceitual** – captura os significados no nível conceitual da descrição do problema. Neste modelo, os conceitos para representar nossa percepção da resolução do problema estão organizados num conceito genérico, como atividades, objetos, status e outros conceitos do gênero. Intuitivamente, verbos genéricos, substantivos genéricos e adjetivos genéricos no nível léxico correspondem respectivamente às atividades, objetos e status no nível conceitual, e,

iii) **Modelo Simbólico** – corresponde ao programa propriamente dito e especifica a semântica computacional do problema a ser resolvido. Para cada atividade, no nível conceitual, pelo menos um fragmento de código é realizado no nível simbólico. Podemos, então, afirmar que o nível simbólico é responsável por descrever explicita e formalmente as restrições entre as atividades e os objetos do problema.

Uma ontologia de tarefa fornece, portanto, primitivas em termos do que precisamos descrever no contexto da solução do problema, através da especificação dos objetos e dos relacionamentos entre estes objetos necessários para se executar a tarefa. Uma ontologia de tarefa também nos facilita colocar o conhecimento do domínio no âmbito do contexto da solução do problema, uma vez que expõe as funcionalidades de vários objetos que podem ser instanciados para objetos específicos do domínio da aplicação. Ou seja, a ontologia de tarefa pode nos auxiliar no desenvolvimento de ontologias de domínio, uma vez que especifica como usar conhecimentos pontuais sobre um domínio para resolver problemas.

A principal função do autor de uma ontologia de tarefa é analisar o conhecimento sobre a solução do problema e construir uma ontologia de tarefa que possa ser facilmente utilizada pelos usuários finais da ontologia, uma vez que o trabalho de descrever seus próprios processos para solucionar os problemas consome muito tempo destes usuários (IKEDA *et al.*, 1998). Para diminuir este encargo, é importante que a ontologia de tarefa reflita o entendimento conceitual da solução do problema. Por outro lado, a descrição do processo de solução do problema precisa ser rígida o suficiente para especificar sua semântica computacional.

É valido salientar que o termo "ontologia de problema" é por vezes utilizado no lugar de "ontologia de tarefa", porém, adotamos no presente trabalho a convenção utilizada pela comunidade de pesquisa em sistemas baseados em conhecimento que chama de ontologia

de tarefa a especificação dos objetos e relacionamentos entre os objetos que são necessários para a resolução de um problema.

Uma das principais vantagem da ontologia de tarefa é que ela não especifica apenas o contexto em que os conceitos do domínio serão utilizados, mas também o esqueleto do processo de solução do problema. Porém, como a ontologia de tarefa está baseada no modelo de tarefas, ela se limita apenas a permitir ao sistema simular o processo de solução do problema no nível conceitual, não demonstrando como o problema pode ser realmente resolvido. Ou seja, a ontologia de tarefa define os componentes e primitivas de unidades de inferência utilizadas durante a realização da tarefa, mas não descreve o controle sobre a mesma (MIZOGUCHI, IKEDA e SINITSA, 1997). Na próxima seção, apresentaremos os métodos de solução de problemas, responsáveis por fornecer soluções para as tarefas.

3.2.2 Métodos de Solução de Problemas

A noção de Métodos de Solução de Problemas (MSP) foi proposta por diversos trabalhos de engenharia de conhecimento, dentre os quais podemos citar Tarefas Genéricas (CHANDRASEKARAN, 1986), ROLE-LIMITING METHODS (MCDERMOTT, 1988), KADS (SCHREIBER, 1993), *Common*KADS (SCHREIBER, 1994), METHOD-TO-TASK (ERIKSSON, 1995) e Componentes de Especialidade (STEELS, 1990). Cada grupo desenvolveu suas próprias definições e seus trabalhos são brevemente descritos na seção 3.2.2.2. Uma das mais importantes afirmações sobre MSP foi proposta por SCHREIBER *et al.* (1993), que definiram os métodos de solução de problemas como sendo a estratégia de inferência responsável por determinar uma solução para a tarefa. De forma semelhante, CHANDRASEKARAN (1986) afirma que o objetivo dos métodos de solução de problemas é especificar o uso do conhecimento para solucionar problemas.

Para a resolução de uma tarefa, um MSP não faz uso de algoritmos complexos, ao invés disso utiliza heurísticas de domínio e restringe o tamanho do problema a ser resolvido através de sua decomposição em partes menores. VALENTE, BREUKER e VAN DER VELDE (1994) afirmam que um MSP possui três funções principais:

i) **Decomposição:** é a principal função de um MSP e descreve como uma certa tarefa pode ser decomposta em subtarefas com um nível de detalhe menor. A decomposição

preserva os relacionamentos da tarefa original adicionando alguns papéis intermediários.

ii) **Controle:** determina como controlar a ordem de execução das subtarefas originadas em (i). O controle determina os papéis desempenhados pelo conhecimento do domínio na resolução do problema, definindo como e quando o conhecimento do domínio é necessário para tornar mais fácil o processo de aquisição de conhecimento, ou seja, o controle especifica a capacidade de inferência do método de resolução de problemas (COELHO E., LAPALME G. e PATEL E. L., 1996), e,

iii) **Determinar o papel do conhecimento do domínio:** identifica os requisitos necessários à representação do conhecimento do domínio para permitir a realização do método. Caso o método de solução de problemas seja independente do domínio, não será necessário nenhum requisito na representação do conhecimento do domínio.

Um método pode considerar a tarefa em que é aplicado como sendo elementar ou composta, de acordo com o seu grau de complexidade. Para uma tarefa elementar, o método aplica sua função de controle para descrever as inferências necessárias para a realização da tarefa. Inferências especificam os passos necessários para satisfazer a funcionalidade do método e são descritas por suas relações de entrada e saída (DIETER FENSEL e REMCO STRAATMAN, 1998). Os objetivos de uma tarefa elementar podem, portanto, ser obtidos diretamente através da interação entre suas relações de entrada e o conhecimento do domínio. Para uma tarefa composta, o método deve primeiramente decompor a tarefa em subtarefas mais simples e posteriormente descrever, através do controle, a ordem em que as subtarefas devem ser realizadas e as interações entre elas (TAUTZ C. e ALTHOFF H., 2000). A estratégia do MSP escolhido é que define se a tarefa deve ser considerada elementar ou composta. Ou seja, para uma mesma tarefa podemos aplicar um MSP que resolva a tarefa diretamente através de inferências, ou utilizar um MSP que decomponha a tarefa em tarefas mais simples.

No processo de solução de um problema, as subtarefas resultantes da decomposição também são consideradas tarefas e, portanto, também necessitam de métodos para a sua solução. Esta recursividade ocorre até o momento em que só existam tarefas elementares, cada uma delas possuindo o seu respectivo método de solução. Quando um MSP decompõe uma tarefa, ele tem como objetivo inicial determinar subtarefas que possam ser resolvidas diretamente, ou seja, decompor a tarefa original em subtarefas elementares. Entretanto, o

método que será aplicado a cada uma dessa subtarefas é que decidirá se irá realmente resolvê-la diretamente ou aplicar uma nova decomposição.

Um bom método de solução de problemas, ao decompor uma tarefa, deve permitir o reuso do maior número possível de subtarefas (ERIKSSON *et al.*, 1995), ou seja, o desenvolvedor do método deve especificar as subtarefas com o objetivo de reutilização. Além disso, a necessidade de decompor ou não uma tarefa está ligada ao bom senso do desenvolvedor do método para tornar a tarefa clara o suficiente para ser entendida, pois um conjunto de problemas simples é mais fácil de ser compreendido e resolvido do que um problema grande e complexo. A decomposição também pode ser vista como uma forma de se modularizar o problema, o que ajuda no seu entendimento. Ou seja, quanto menor o nível de decomposição de uma tarefa, maior será o nível de abstração do entendimento do problema.

O relacionamento existente entre a tarefa, o seu MSP e o conhecimento do domínio está ilustrado na figura 3.1, que representa MSP como uma caixa preta responsável pela solução de uma determinada tarefa. Dentro desta caixa preta existem componentes responsáveis pela decomposição da tarefa em subtarefas mais simples e outros pelo controle da ordem de execução dessas subtarefas. Para permitir a resolução do problema, MSP faz uso dos conceitos existentes no conhecimento do domínio.

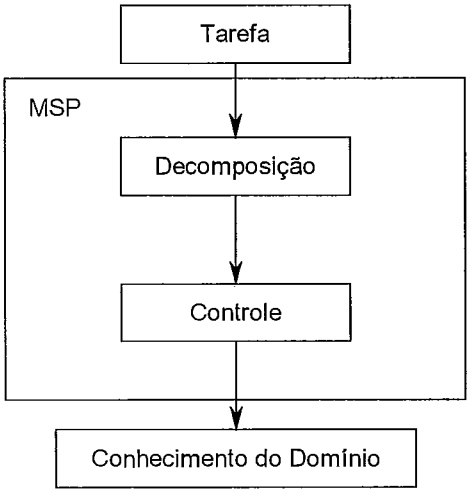

Figura 3.1 – Relacionamento entre Tarefa, MSP e Conhecimento do Domínio

3.2.2.1 Ontologia de Método

Uma tarefa pode possuir diferentes métodos para sua solução, cada um destes se utilizando de diferentes conceitos para solucionar a tarefa. A descrição desses conceitos específicos e seus relacionamentos pode ser especificada em uma ontologia de método (GENNARI, TU, ROTHENFLUSH e MUSEN, 1994). A ontologia de método define os papéis desempenhados pelo conhecimento no processo de inferência e serve como uma interface para o engenheiro de conhecimento avaliar se o domínio de uma aplicação está de acordo com a estrutura de conhecimento usada pelo método (COELHO e LAPALME, 1995). O objetivo da ontologia de método é permitir que os métodos de solução de problemas possam ser reusados e ser a interface de acesso à biblioteca de MSP, permitindo a utilização do MSP correspondente.

Uma ontologia de método permite uma definição declarativa da inferência através da organização do conhecimento usado pela inferência do MSP (COELHO e LAPALME, 1995). Desta forma, a ontologia de método se limita a descrever os tipos de entidades e relações envolvidas na inferência, não descrevendo a inferência propriamente dita. A ontologia de método também não descreve como a decomposição e o controle devem ser realizados para solucionar o problema.

COELHO e LAPALME (1995), considerando que métodos de solução de problemas são um conjunto de operações sobre uma estrutura de conhecimento, propuseram uma abordagem para reutilizar MSP que, primeiro, descreve os papéis do conhecimento e depois as operações sobre este conhecimento. Os papéis do conhecimento são definidos conforme a ontologia de método, enquanto que as operações são definidas através de inferências. As inferências são descritas de forma abstrata e sem detalhes de implementação. Para formalizar a ontologia foi utilizada Ontolingua (GRUBER, 1993). Como Ontolingua não representa o controle sobre o conhecimento, foi proposta uma extensão da linguagem para permitir a definição das inferências.

Em trabalhos anteriores, COELHO *et al.* (1996) também definiram um conjunto de operações que pode ser utilizado pelo engenheiro de software para adaptar o domínio da aplicação para um MSP existente. Caso a adaptação não seja possível, o método é rejeitado. Contudo, é fornecido um guia que permite pesquisar um outro método ou até mesmo construir um novo MSP.

3.2.2.2 Projetos

Novas metodologias de modelagem do conhecimento para a solução de um problema surgiram a partir da idéia de Tarefas Genéricas (CHANDRASEKARAN, 1986) e Classificação Heurística (CLANCEY, 1985), cada uma delas com diferentes procedimentos para a modelagem do conhecimento. Descreveremos, a seguir, algumas delas.

Componentes de Especialidade (STEELS, 1990), assim como as Tarefas Genéricas, partem de uma análise detalhada da tarefa da aplicação. Os componentes para solução de um problema são definidos da seguinte maneira: (i) tarefas têm estrutura interna e podem ser decompostas em subtarefas; (ii) modelos de caso são construídos para cada tarefa ou subtarefa; (iii) o conhecimento do domínio requerido pelos vários tipos de modelos de caso é descrito em modelos de domínio, e, (iv) um método de solução de problema é responsável por aplicar o conhecimento do domínio a uma tarefa, podendo consultar modelos de domínio.

As abordagens de ROLE-LIMITING METHODS (MCDERMOTT, 1988) e METHOD –TO-TASK (ERIKSSON, 1995), por sua vez, visam oferecer resolvedores de problemas eficientes e esta preocupação é tão forte que não é feita uma separação entre eficiência e funcionalidade. Conforme a terminologia utilizada, um MSP é um algoritmo que determina como o conhecimento específico do domínio é utilizado na resolução de problemas. O nome ROLE-LIMITING METHODS foi dado porque estes métodos forçam a definição precisa dos papéis desempenhados pelo conhecimento na solução de problemas, definindo quando e como o conhecimento do domínio é necessário para tornar mais fácil o processo de aquisição do conhecimento.

O projeto KADS (SCHREIBER *et al.*, 1993) visa o desenvolvimento de uma metodologia ampla para a construção de sistemas baseados em conhecimento que está fundamentada em cinco princípios básicos: múltiplos modelos, modelagem de conhecimento em níveis, reutilização de componentes, refinamento de modelos e projeto preservando a estrutura. A biblioteca de tarefas genéricas *Common*KADS (BREUKER *et al.*, 1994) é uma extensão da metodologia KADS e foi desenvolvida para comportar a utilização de métodos de solução de problemas. A biblioteca está organizada segundo uma coleção de tipos de problemas, contendo oito grupos básicos de tipos de problemas: problemas de modelagem, planejamento, projeto, designação, previsão, avaliação,

monitoramento e diagnose (BREUKER, 1994). Esta coleção de tipos de problema pode ser utilizada como uma base para a modelagem do conhecimento sobre desenvolvimento de software, segundo uma perspectiva de tarefa, uma vez que todos os tipos de problema identificados na coleção do *Common*KADS ocorrem no desenvolvimento de software.

A biblioteca *Common*KADS considera a existência de três níveis de descrição de problemas: (i) verbal, que descreve o problema em linguagem natural; (ii) conceitual, que descreve o problema utilizando a linguagem de modelagem conceitual (CML) definida por SCHREIBER, 1994, e, (iii) formal, que utiliza a linguagem formal para descrever o problema. A biblioteca também classifica o conhecimento de controle do método em duas categorias: (i) o conhecimento de inferência, que especifica as inferências que constituem o método e os papéis desempenhados pelo conhecimento de domínio no processo de inferência, e, (ii) o conhecimento de tarefa, que especifica a ordem das inferências, ou seja, como as inferências podem ser combinadas para resolver uma tarefa. *Common*KADS faz uso de uma estrutura de inferência para representar a interconexão conceitual das inferências no sentido de papéis do conhecimento. Essa estrutura não especifica o fluxo de controle entre as inferências, o que é definido no nível do conhecimento de tarefa, pois apenas apresenta uma visão geral do método de solução de problemas e do conhecimento usado pelas inferências.

3.3 Conclusão do Capítulo

A aquisição e representação do conhecimento é uma atividade extremamente trabalhosa, sendo a construção de bases de conhecimento um dos maiores custos na construção de sistemas de inteligência artificial. Segundo OLIVEIRA *et al.* (1999b), surgiu, neste contexto, a necessidade de uma definição preliminar, com características de filosofia, que deveria ser feita antes mesmo da construção da base de conhecimento e que pudesse facilitar e apoiar o desenvolvimento de sistemas: a ontologia.

Pesquisas em ontologia têm buscado organizar o conhecimento considerando, separadamente, os diferentes enfoques de um problema, ou seja, os conceitos diretamente relacionados com o domínio propriamente dito, conceitos genéricos de qualquer domínio, conceitos sobre tarefas, métodos e sobre as próprias formas de representação.

Como neste trabalho estamos interessados em definir uma representação para o conhecimento de tarefa, nosso foco principal foram as ontologias de tarefa. Uma ontologia de tarefa fornece primitivas em termos do que precisamos descrever no contexto da solução do problema, ou seja, nos provê uma especificação dos objetos e dos relacionamentos entre estes objetos necessários para se executar a tarefa. A ontologia de tarefa, também, nos facilita colocar o conhecimento do domínio no âmbito do contexto da solução do problema, uma vez que expõe as funcionalidades de vários objetos que podem ser instanciados para objetos específicos do domínio da aplicação, ou seja, a ontologia de tarefa dá sentido aos conceitos especificados no conhecimento do domínio.

A ontologia de tarefa, porém, limita-se a permitir simular o processo de solução do problema conceitualmente, ou seja, não demonstra como o problema pode ser resolvido. A solução de problemas propriamente dita pode ser descrita através de métodos de solução de problemas (MSP). MSP manipulam os conceitos definidos na tarefa de forma a permitir que o problema possa ser resolvido. Ao invés de utilizar algoritmos complexos para a resolução de uma tarefa, um MSP faz uso de heurísticas de domínio e restringe o tamanho do problema a ser resolvido através de sua decomposição em partes menores.

Pode parecer estranha a diferença no nível de detalhe dado neste capítulo a MSP quando comparado a ontologia, ainda mais se for levada em consideração a importância dos MSP na descrição de tarefas como será apresentado no próximo capítulo. Porém, é necessário ressaltar que a estrutura para representação do conhecimento de tarefa será realizada, neste trabalho, sobre o esqueleto da ontologia, sendo o MSP apenas a estratégia por trás da ontologia que manipula o conhecimento de forma a permitir que o problema possa ser resolvido.

Já foram desenvolvidas diversas metodologias de modelagem do conhecimento para a solução de um problema, cada uma com seus próprios procedimentos de modelagem. No próximo capítulo, apresentaremos os motivos que nos levaram a optar por descrever uma nova proposta de representação do conhecimento de tarefa, baseada na utilização de ontologias de tarefa, para representar os conceitos envolvidos no problema, e métodos de solução de problemas, para descrever como o problema pode ser resolvido.

Capítulo 4

Descrição de Solução de Problemas: Uma Abordagem para Descrição de Conhecimento de Tarefa em ADS

Este capítulo apresenta uma abordagem para a organização do conhecimento de tarefa, útil para o entendimento e resolução de problemas em ambientes de desenvolvimento de software. Inicialmente, serão apresentados os elementos para sua composição e, em seguida, descreveremos sua estrutura de representação.

4.1 Definição

Uma das principais razões para que produtos de software não atendam às necessidades de seus usuários é a falta de entendimento de qual é o real objetivo do software e, consequentemente, quais são as tarefas que ele deve realizar e como estas devem ser realizadas. O entendimento do problema é fundamental para a correta identificação e descrição do que o software precisa realizar. Dois tipos de conhecimento estão envolvidos no entendimento do problema: (i) o conhecimento sobre os reais objetos no domínio de interesse, ou seja, o conhecimento sobre objetos, relacionamentos, eventos, estados e outros conceitos que obtemos em um domínio, e, (ii) o conhecimento sobre a resolução do problema, ou seja, o conhecimento de tarefa.

Como discutido no capítulo 2, o conhecimento do domínio já está sendo utilizado para apoiar o desenvolvimento em ambientes de desenvolvimento de software através dos ADSOD (OLIVEIRA *et al.*, 1999b). ADSOD têm por objetivo apoiar o entendimento do domínio, disponibilizando o conhecimento do domínio independente de quais sistemas já existam na empresa e dos produtos de software que se deseja construir. Esse conhecimento deve representar as características intrínsecas do domínio, suas restrições e organização. Além disso, esse conhecimento deve ser útil na construção de ferramentas de

desenvolvimento de software e deve poder ser utilizado no desenvolvimento dos próprios produtos de software.

O conhecimento de tarefa, por sua vez, tem por objetivo apoiar o entendimento do problema e a atividade de desenvolvimento de forma geral. Como descrito no capítulo 3, consideramos que uma tarefa é a sequência de passos necessários para a resolução de um problema e, desta forma, uma tarefa pode ser definida como sendo composta por seu objetivo, a definição dos seus conceitos, além do método responsável por sua resolução. A descrição de uma tarefa deve ser independente do domínio em que será aplicada, permitindo a formação de uma biblioteca de tarefas que possa ser utilizada em diversos domínios e acrescida de novas tarefas de acordo com a necessidade do sistema que está sendo desenvolvido. A partir destas definições, apresentamos uma abordagem na descrição de tarefas, com o objetivo de organizar o conhecimento que possa ser útil para apoiar os engenheiros de software no entendimento do problema, a partir do entendimento das tarefas que compõem esse problema e, desta forma, auxiliar também na sua resolução. Consideramos que entender uma tarefa, nesse contexto, envolve entender sobre os conceitos utilizados na mesma, além de entender como ela é realizada.

Para a representação dos conceitos envolvidos em uma tarefa, optamos pela utilização de ontologias de tarefa, uma vez que uma ontologia de tarefa fornece primitivas em termos do que precisamos descrever no contexto da solução do problema, através da especificação dos conceitos e das relações entre estes conceitos necessários para se executar a tarefa. Além disso, o uso de ontologias permite a definição de uma abordagem metódica e rigorosa para resolver um problema, ao contrário de uma abordagem ad-hoc, que não permite o estabelecimento de controles. A seguir, apresentaremos mais algumas características de ontologia que nos levaram à sua escolha nesta abordagem:

i) **Compartilhamento e reuso:** em qualquer software reutilizável, bases de conhecimento têm de ser projetadas para o reuso e, neste contexto, ontologias assumem um importante papel. Ontologias foram desenvolvidas para que pudessem ser usadas como base para múltiplos sistemas, permitindo compartilhar uma terminologia comum, o que facilita o compartilhamento e reuso do conhecimento;

ii) **Representação específica:** o não comprometimento da modelagem do conhecimento de tarefa com qualquer aspecto que não sejam as características da própria tarefa, além da boa e clara definição de como esse conhecimento deve ser explicitado, estruturado e

representado. Uma ontologia de tarefa permite uma descrição explícita dos conceitos e relações envolvidos no problema, além de permitir a realização de inferências.

iii) **Separação entre tarefa e domínio:** separação bem definida do que significam os conceitos de um domínio e as tarefas que são automatizadas sobre eles.

iv) **Compatibilização entre tarefa e domínio:** ontologias permitem a integração dos conhecimentos de tarefa e domínio em uma abordagem uniforme e consistente. Quando utilizamos duas estratégias diferentes para desenvolver modelos de resolução de problema, precisamos garantir que as modelagens de tarefa e de domínio tenham um acoplamento suave. Contudo, com a utilização de ontologias para representar tanto o conhecimento do domínio quanto o conhecimento de tarefa, podemos garantir que esse acoplamento pode ser realizado através do mapeamento entre os conceitos definidos no domínio e na tarefa. Uma ontologia de tarefa também permite colocar o conhecimento do domínio no âmbito do contexto da solução do problema, uma vez que expõe as funcionalidades de vários objetos que podem ser instanciados para objetos específicos do domínio da aplicação. Como já estamos considerando, nos ADSOD, o uso de ontologias de domínio para a representação do conhecimento de domínio, a utilização de ontologias de tarefa irá permitir padronizar as representações.

Uma ontologia de tarefa, entretanto, se limita a permitir simular o processo de solução do problema conceitualmente, não demonstrando como o problema pode ser resolvido. Isto ocorre pelo fato de que uma ontologia de tarefa não descreve o controle sobre a tarefa, apesar de definir componentes e primitivas de inferências utilizadas durante a realização da mesma. Portanto, para representarmos completamente o conhecimento de tarefa, precisamos ainda descrever como a tarefa é realizada.

Para descrever a solução do problema propriamente dita, consideramos a utilização de métodos de solução de problemas (MSP). Esses métodos são responsáveis por especificar as inferências necessárias para que o objetivo da tarefa seja alcançado. Ao invés de utilizar algoritmos complexos para a resolução de uma tarefa, um MSP faz uso de heurísticas de domínio e restringe o tamanho do problema a ser resolvido através de sua decomposição em partes menores.

É necessário comentar a razão pela qual utilizamos enfoques distintos para a construção dos componentes de conhecimento. A priori, podemos utilizar ontologias para modelar tanto a tarefa quanto o seu método. Contudo, uma ontologia de método (vide seção 3.2.2.1)

se limita a descrever os tipos de entidades e relações envolvidas em uma inferência, não descrevendo como a decomposição e o controle devem ser realizados para solucionar o problema. Ou seja, uma ontologia de método representa os conceitos e relações originadas pela aplicação do método, mas não descreve a inferência e o controle necessário para a resolução do problema.

É interessante, porém, que os conceitos e relações originados pela aplicação do método sejam de alguma forma explicitados, assim como é realizado pela ontologia de método. Como visto na seção 3.2.2, as subtarefas resultantes de uma decomposição são consideradas tarefas independentes no processo de solução de um problema, portanto podemos dizer que os conceitos originados pela aplicação de um método a uma tarefa são na realidade conceitos de suas subtarefas e, portanto, serão descritos em suas respectivas ontologias de tarefa. Assim sendo, uma ontologia de tarefa se torna o componente responsável por descrever todos os conceitos e relações envolvidos na realização da tarefa.

A biblioteca de tarefas genéricas *Common*KADS (BREUKER *et al.*, 1994) é referência para a descrição de conhecimento de tarefa e serviu como base para a definição da abordagem apresentada neste trabalho. A biblioteca utiliza modelos de tarefas e foi desenvolvida para comportar a utilização de métodos de solução de problemas, como visto no capítulo anterior. Além de considerarmos o uso de ontologias de tarefa em contraposição aos modelos de tarefa, descrevemos a seguir dois dos principais fatores que nos levaram a optar por descrever uma abordagem para a representação de tarefas ao invés de utilizarmos diretamente os resultados do *Common*KADS:

i) **Facilitar a descrição de novas tarefas**: tornar as descrições das subtarefas independentes da descrição da tarefa original, ou seja, permitir descrever as subtarefas da mesma forma que a tarefa que as originou, criando assim uma biblioteca composta por qualquer tarefa envolvida no entendimento do problema. Ao contrário do *Common*KADS que apresenta diversas soluções para o problema através da descrição de diferentes formas de decompor uma tarefa, estamos interessados em demonstrar apenas uma solução para o problema e formar uma biblioteca de tarefas a partir desta resolução. Ou seja, estamos interessados em decompor a tarefa de uma única maneira, porém, descrevendo suas subtarefas de forma que se tornem independentes e possam ser reutilizadas. As subtarefas descritas a partir deste processo estarão disponíveis na biblioteca de tarefas e podem, portanto, serem utilizadas para compor uma nova tarefa.

ii) **Tornar a descrição flexível:** permitir a utilização dos componentes que sejam considerados necessários para solucionar um problema sem que tenhamos que seguir uma estrutura pré-definida. Consideramos que a biblioteca *Common*KADS é uma importante ferramenta que serve como base para a descrição de tarefas, mas deve ser tomado o cuidado para só serem utilizadas as definições que sejam realmente pertinentes à resolução do problema. A biblioteca *Common*KADS, por exemplo, considera que a descrição do problema deve ser realizada primeiramente em um nível conceitual e, opcionalmente, o problema deve ser descrito verbal e formalmente. Consideramos, entretanto, que descrições textuais e formais são fundamentais na representação de um problema.

Dessa forma, optamos por combinar ontologias de tarefa e MSP em um único modelo que denominamos Descrição de Solução de Problemas (DSP). Através da inclusão da DSP em um ADSOD, obtemos uma estrutura que considera conhecimento do domínio e descrições de tarefas. Para demonstrar como essa combinação permite auxiliar no desenvolvimento de sistemas, apresentaremos, no próximo capítulo, um exemplo de como a DSP pode auxiliar os engenheiros de software na modelagem de casos de uso, amplamente utilizada na orientação a objetos. Porém, primeiramente iremos apresentar os componentes da estrutura da DSP necessários à sua descrição.

4.2 Estrutura

A estrutura de uma descrição de tarefas é a peça fundamental para permitir a realização de um mapeamento entre os papéis definidos no conhecimento de tarefa e o conhecimento do domínio correspondente. Os papéis de conhecimento são considerados o ponto-chave no controle desta interação, uma vez que, ao se mapear conceitos do domínio com os seus respectivos papéis no conhecimento de tarefa, estabelece-se claramente a relação entre os papéis de conhecimento e os conceitos das ontologias de domínio que podem preenchê-los. A partir deste mapeamento, é possível realizar um protótipo inicial do sistema e ao propor esta solução preliminar, o engenheiro de software pode identificar que tipos de conhecimento estão faltando e, assim, tem um mecanismo eficiente para guiar a aquisição do conhecimento específico de uma aplicação, junto aos especialistas.

Como mencionamos anteriormente, a estrutura da Descrição de Solução de Problemas é composta de ontologias de tarefa, que são responsáveis pela definição dos conceitos envolvidos no problema e pelos métodos responsáveis pela resolução do problema. Estamos, porém, considerando a aplicação de somente um método à tarefa, pois nosso objetivo é mostrar como, a partir de uma solução para o problema, podemos gerar uma descrição modularizada de várias tarefas, de forma que descrições futuras possam utilizar descrições já existentes. Isto é possível, uma vez que, no processo de solução do problema, as subtarefas resultantes de uma decomposição também são consideradas tarefas e precisam ser descritas na DSP, assim como a tarefa original, e disponibilizadas para poder compor uma nova tarefa.

Além disso, a estrutura da DSP precisa ser capaz de permitir que os requisitos definidos anteriormente para a representação do conhecimento de tarefa sejam satisfeitos, ou seja, a estrutura da DSP precisa ser descrita de forma a permitir o entendimento do problema pelo engenheiro de software, a correta identificação dos requisitos do produto, a definição dos papéis dos conceitos especificados no conhecimento de domínio e o apoio ao desenvolvimento de sistemas, além de tornar mais fácil a descrição de novas tarefas.

Para permitir a descrição de todos os componentes necessários para que este conjunto de requisitos seja satisfeito, optamos por modelar a DSP em três níveis de abstração, onde em cada nível há uma ênfase seletiva nos detalhes representados. Os níveis Verbal, Conceitual e Formal definem o grau de detalhamento com que as características do problema são representadas no modelo. A seguir, são apresentados cada um desses níveis, cujas descrições são compostas por componentes, tanto da ontologia de tarefa, quanto do método aplicado à tarefa. Além disso, elementos que consideramos pertinentes para a sintetização e o entendimento do problema são incorporados à DSP, permitindo que sua descrição e organização possam auxiliar no desenvolvimento de sistemas. Para ilustrar estas definições, apresentamos a DSP para a tarefa de Configuração.

4.2.1 Nível Verbal

O nível verbal pode ser considerado o mais completo dentre os três níveis que compõem a Descrição de Solução de Problemas, uma vez que descreve todas as etapas para a resolução do problema em linguagem natural. Nesta seção, são apresentadas a descrição

textual da tarefa, o método escolhido para solucioná-la, além do controle necessário para sua solução.

O primeiro passo para que seja possível a construção de uma Descrição de Solução de Problemas é descrever textualmente o problema que pretendemos solucionar. Esta descrição precisa ser feita no nível de ações, ou seja, é necessário que possamos identificar, a partir da descrição, as ações necessárias para resolver o problema. A descrição textual é uma primeira visão do que é a tarefa e é um componente importante para o entendimento do que a tarefa se propõe a fazer, uma vez que é nela que o objetivo da tarefa é descrito. A descrição textual é feita em linguagem natural, conforme pode ser visto no exemplo a seguir para a tarefa de Configuração (figura 4.1).

A tarefa de Configuração tem por objetivo determinar de que forma os parâmetros de um sistema devem ser organizados para satisfazerem as restrições que são impostas sobre eles. As restrições sobre os parâmetros têm por objetivo delimitar o espaço de soluções para a tarefa, uma vez que restringem o número de possíveis configurações válidas. A tarefa de Configuração fornece valores aos parâmetros de um sistema, verificando sempre se as restrições sobre eles estão sendo satisfeitas, sendo que, caso algum parâmetro viole uma de suas restrições, é necessário que seja estabelecido um novo valor para ele, de forma a satisfazer a restrição violada. A tarefa termina quando todos os valores de todos os parâmetros forem computados.

Figura 4.1 – Descrição Textual da DSP de Configuração

O segundo passo na descrição verbal é aplicar o MSP específico para a tarefa que está sendo descrita. Para solucionar a tarefa de Configuração, podem ser aplicados diversos MSP existentes na literatura. A estratégia do MSP escolhido é que vai determinar se a tarefa deve ser considerada elementar ou composta, ou seja, o método aplicado à tarefa é que define se vai resolver a tarefa diretamente através de inferências ou se vai utilizar uma decomposição para dividir a tarefa em tarefas mais simples. Desta forma, um problema grande e complexo pode ser dividido em um conjunto de problemas menores e independentes, mais fáceis de serem compreendidos e resolvidos.

Consideramos a tarefa de Configuração complexa o suficiente para ser dividida em subtarefas. Selecionamos, por exemplo, o método *Propor e Revisar* (P&R) que irá particioná-la em subtarefas mais simples. A decomposição poderá permitir disponibilizar um maior número de tarefas para reutilização ou, até mesmo, a composição da tarefa de Configuração a partir de tarefas já descritas previamente.

P&R é um método de solução de problemas tradicionalmente aplicado a uma tarefa de Configuração e foi definido originalmente na ferramenta de aquisição de conhecimento SALT (MARCUS, 1998). O método P&R que será apresentado é baseado na descrição realizada por COELHO e LAPALME (1996). O método decompõe, primeiramente, a tarefa de Configuração em quatro subtarefas mais simples:

i) *Seleção*: escolhe entre todos os parâmetros de um sistema, um que ainda não tenha valor definido, mas para o qual todos os parâmetros dos quais ele depende já possuam valor;

ii) *Proposição*: fornece um valor para o parâmetro escolhido a partir de uma fórmula que descreve o seu cálculo. O cálculo de um parâmetro é baseado no valor dos parâmetros dos quais ele depende;

iii) *Verificação*: analisa se alguma restrição foi violada após o valor de um parâmetro ter sido calculado. As restrições são baseadas nos valores dos parâmetros do sistema, e,

iv) *Revisão*: repara a restrição violada e calcula um novo valor para o parâmetro a partir dos valores de outros parâmetros.

Após ter definido o conjunto de subtarefas da tarefa de Configuração, o método especifica a ordem de chamada destas subtarefas, ou seja, como as subtarefas devem ser combinadas de forma a solucionar a tarefa original. A figura 4.2 descreve o controle sobre a tarefa de Configuração, segundo o método P&R.

O primeiro passo para resolver a tarefa de Configuração é fornecer um valor inicial para um determinado parâmetro do sistema. A subtarefa Seleção escolhe um parâmetro, dentre todos, para computar seu valor e, após isto, a subtarefa Proposição sugere um valor inicial para este parâmetro. Este é apenas um valor padrão para o parâmetro, podendo ser modificado, caso alguma restrição seja violada. A subtarefa Verificação é a responsável por avaliar se o valor satisfaz as restrições do sistema. Se existirem restrições violadas, estas precisam ser analisadas para que se possa gerar um novo valor para o parâmetro. Esta operação é realizada pela subtarefa Revisão, que repara o valor dos parâmetros a fim de que as restrições sejam satisfeitas. Todos esses passos são repetidos até que todos os parâmetros tenham seus valores computados e nenhuma restrição esteja sendo violada.

Figura 4.2 – Descrição do Controle da DSP de Configuração

4.2.2 Nível Conceitual

O nível conceitual da Descrição de Solução de Problemas é o responsável por permitir que as descrições do problema, que foram definidas em linguagem natural no nível verbal,

possam ser traduzidas para o nível formal. Enquanto o nível verbal permite uma série de ambiguidades por não possuir uma estrutura bem definida, as descrições no nível formal exigem um elevado nível de conhecimento do problema. O nível de descrição conceitual é um nível intermediário, possuindo uma estrutura suficiente para evitar erros de interpretação e sendo ainda informal o bastante para ser compreendido pelos envolvidos na solução do problema. A descrição conceitual é formada por: (a) uma lista dos conceitos envolvidos no problema e seus relacionamentos explicitamente identificados, e, (b) um procedimento para representação do fluxo de controle necessário para resolver a tarefa.

Os conceitos da tarefa (item (a)) são, na realidade, os papéis de conhecimento que serão preenchidos por conceitos do domínio, quando estivermos trabalhando com uma aplicação específica. Os conceitos da tarefa podem ser considerados, portanto, o ponto-chave no controle desta interação e, como tal, precisam ser descritos claramente. Além disso, a explicitação dos conceitos envolvidos na resolução da tarefa permite que estes sejam facilmente formalizados no próximo nível da DSP.

Definir estes conceitos nada mais é do que identificar os objetos especificados na descrição textual do problema. Como uma tarefa é caracterizada por seus conceitos de entrada e saída, é importante que, ao definirmos os conceitos envolvidos no problema, possamos identificar os seus respectivos papéis no processo de solução. Portanto, as entradas e saídas de uma tarefa são os papéis que o conhecimento do domínio pode desempenhar na solução do problema.

Para a Configuração, pode ser identificado imediatamente como conceito com papel de entrada o *Parâmetro,* que representa todos os parâmetros do sistema. No entanto, um parâmetro pode ser calculado a partir de outros parâmetros o que leva a duas relações: *Associação Parâmetro*, que define a de dependência entre os parâmetros e *Fórmula Parâmetro,* que determina a fórmula de cálculo do valor de um parâmetro a partir dos valores dos parâmetros dos quais ele depende. Temos, ainda, como conceito de entrada *Restrição*, que determina como avaliar os valores dos parâmetros relacionados. Para isso tem-se, ainda, a relação *Dependência Restrição,* que define os parâmetros envolvidos em uma restrição. Quando uma restrição é violada, os valores dos parâmetros são reparados para que as restrições sejam satisfeitas e com isso temos o conceito de entrada *Reparo,* que determina de que forma os valores dos parâmetros devem ser alterados para reparar a restrição, e a relação *Dependência Reparo*, que define os parâmetros necessários para

reparar uma restrição. Finalmente, temos um último conceito que representa um parâmetro do sistema com seu valor determinado e por isso chamado de *Parâmetro Valorado*. Este conceito possui tanto o papel de entrada como de saída da tarefa, uma vez que, para alguns parâmetros, o valor já é fornecido antes mesmo da tarefa ser realizada, enquanto que para outros o valor ainda precisa ser calculado.

Para permitir uma correta formalização das relações descritas anteriormente, é fundamental que os conceitos e relações envolvidos no problema sejam modelados em alguma linguagem de representação. Esta modelagem permitirá, também, que o engenheiro de software possa mapear os conceitos da tarefa para os do domínio específico, quando necessário.

A Figura 4.3 mostra a organização dos conceitos definidos para a tarefa de Configuração representados em LINGO (FALBO, 1998a). LINGO já vem sendo utilizada para a representação de ontologias de domínio em ADSOD e possui relações capazes de capturar os axiomas epistemológicos de forma implícita, ou seja, as notações descritas em LINGO refletem diretamente axiomas em lógica de 1ª. ordem. Este axiomas são apresentados na próxima seção, assim como suas respectivas formalizações. No anexo 2, encontra-se um breve resumo sobre a linguagem LINGO e seus principais componentes.

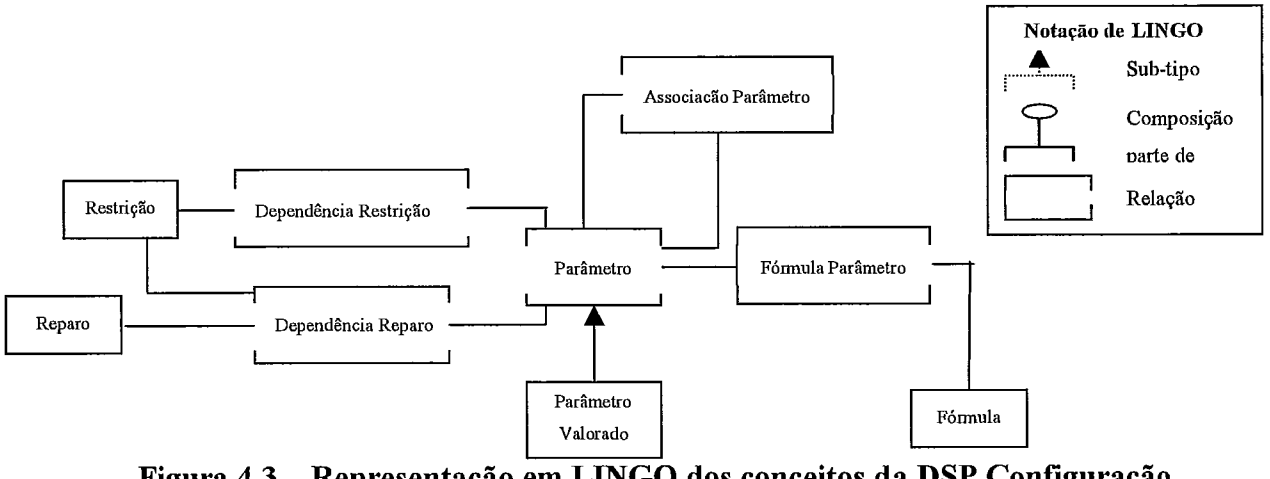

Figura 4.3 – Representação em LINGO dos conceitos da DSP Configuração

Finalmente, para permitir que o fluxo de controle sobre as subtarefas possa ser facilmente formalizado e, ao mesmo tempo, possa ser entendido pelos engenheiros de software, independentemente de seus conhecimentos na linguagem formal utilizada, optamos por utilizar um procedimento (item (b)) em linguagem estruturada para a

especificação do fluxo de controle. A figura 4.4 apresenta o fluxo de controle para a tarefa de Configuração.

Enquanto *existir algum parâmetro sem valor computado ou uma restrição que tenha sido violada* ***faça***
    Selecionar *parâmetro*
    Propor *um valor para o parâmetro*
    Para cada *restrição* ***faça***
        Verificar *restrição*
    Fim-Para-Cada
    Para cada *restrição violada* ***faça***
        Revisar *restrição*
    Fim-Para-Cada
Fim-Enquanto

Figura 4.4 – Fluxo de Controle da DSP Configuração

4.2.3 Nível Formal

O nível formal não tem por objetivo introduzir nenhum conhecimento ainda não descrito na DSP. Sua responsabilidade é, apenas, traduzir a descrição do problema e sua resolução para o nível de representação formal, ou seja, formalizar os conceitos e relações definidos e as inferências necessárias para solucionar a tarefa. Utilizamos para formalização a linguagem Prolog, que descreve o conhecimento através de fatos e regras e já está sendo utilizada para representar o conhecimento de domínio na Estação TABA.

Como estamos trabalhando com ontologia de tarefa, é necessário que os conceitos definidos no problema sejam descritos explicitamente, de forma que possam ser compartilhados, e suas restrições expressas através de um conjunto formal de axiomas que, explicitamente, restringem os significados de seus termos. A formalização destas restrições pode ser realizada a partir dos axiomas epistemológicos descritos em lógica de 1ª. ordem obtidos através da representação conceitual da tarefa em LINGO, apresentada na seção anterior. Consideremos, por exemplo, a relação *Associação Parâmetro* que representa a relação de dependência entre parâmetros. Essa dependência é descrita em lógica de primeira ordem com o seguinte axioma:

$$(\forall p1,p2)\ (associação\text{-}parâmetro\ (p1,p2) \rightarrow parâmetro(p1) \wedge parâmetro(p2)).$$

Sendo formalizada em Prolog como:

```
parâmetro(P1):- associação-parâmetro(P1,_).
parâmetro(P2):- associação-parâmetro(_,P2).
```

De forma semelhante, formalizamos todas as dependências entre os conceitos (tabela 4.1). Estes axiomas são utilizados para representar as restrições entre os conceitos da tarefa de Configuração e, desta forma, satisfazem a competência de sua ontologia.

Tabela 4.1 – Formalização em Prolog das Relações da DSP de Configuração

Axiomas em Lógica de 1ª Ordem	Axiomas em Prolog
*(∀p) (**parâmetro-valorado**(p,_) → **parâmetro**(p))*	***parâmetro** (P):- **parâmetro-valorado**(P,_).*
*(∀p,f) (**fórmula-parâmetro** (p,f) → **parâmetro**(p) ∧ **fórmula**(f))*	***parâmetro**(P):-**fórmula-parâmetro**(P,_).* ***fórmula**(F):-**fórmula-parâmetro**(_,F).*
*(∀r,p) (**dependência-restrição** (r,p) → **restrição**(r) ∧ **parâmetro**(p))*	***restrição**(R):-**dependência-restrição**(R,_).* ***parâmetro**(P):- **dependência-restrição** (_,P).*
*(∀r,p) (**dependência reparo** (rep, rst, p) → **reparo**(rep,_) ∧ **restrição**(rst) ∧ **parâmetro**(p))*	***reparo**(REP,_):-**dependência-reparo**(REP,_,_).* ***restrição**(RESTR):-**dependência-reparo**(_,RST,_).* ***parâmetro**(P):- **dependência-reparo** (_,_,P).*

As inferências necessárias para a solução do problema, também, precisam ser descritas em alguma linguagem formal. A formalização pode ser realizada a partir da tradução para Prolog do procedimento em linguagem estruturada, definido através do nível conceitual da DSP, conforme pode ser visto na figura 4.5, onde é apresentado o fluxo de controle sobre as subtarefas de Configuração. A solução em Prolog pode parecer mais simples do que no procedimento original, devido ao mecanismo de controle estar implícito na linguagem, não necessitando que os *loops* sejam descritos explicitamente.

```
Configuração (Parâmetro, Valor):- seleção (Parâmetro),
                                  proposição (Parâmetro, Valor),
                                  not (verificação (Restrição)).
Configuração (Parâmetro, Valor):- seleção (Parâmetro),
                                  proposição (Parâmetro,_),
                                  verificação (Restrição),
                                  revisão (Restrição, Parâmetro, Valor).
```

Figura 4.5 – Formalização em Prolog do Controle da DSP de Configuração

4.3 Descrição das Subtarefas

A principal característica da DSP é a sua capacidade de tornar as descrições das subtarefas independentes da descrição da tarefa original. Consideramos que as subtarefas são tarefas com um nível de detalhe menor e, portanto, precisam ser descritas na DSP, assim como a tarefa original, e disponibilizadas para poder compor uma nova tarefa. Acrescentado o fato de que um método de solução de problemas, ao decompor uma tarefa, deve permitir o reuso do maior número possível de subtarefas, é possível, então, a formação de uma biblioteca de tarefas. Uma vez que, no processo de solução de um problema, as subtarefas resultantes de uma decomposição são também consideradas tarefas, elas necessitam de métodos para sua solução. Esta recursividade ocorre até o momento em que só existam tarefas elementares, cada uma delas possuindo o seu respectivo MSP. Quando um método decompõe uma tarefa, ele tem como objetivo inicial determinar subtarefas que possam ser resolvidas diretamente, ou seja, decompor a tarefa original em subtarefas elementares. Entretanto, o método que será aplicado a cada uma dessas subtarefas é que define se a subtarefa será realmente resolvida diretamente ou se será aplicada uma nova decomposição.

Caso o método considere uma tarefa como elementar, ele simplesmente descreverá a sua solução. Caso contrário, o método irá decompor a tarefa em subtarefas mais simples e demonstrará como o controle sobre a chamada destas subtarefas deverá ser realizado, como foi mostrado para a tarefa de Configuração. Deste modo, a única diferença entre a representação na DSP de uma tarefa composta e de uma tarefa elementar é que para a tarefa elementar não é preciso descrever as regras de controle que definem a ordem em que as subtarefas são realizadas, uma vez que uma tarefa elementar não precisa mais ser decomposta. Ao invés disso, são descritas as regras de inferência que definem como podemos solucionar a tarefa, a partir da interação entre seus conceitos.

Ao descrevermos as subtarefas de Configuração, optamos por aplicar métodos que considerem as tarefas Seleção, Proposição e Revisão como tarefas elementares, enquanto que para a tarefa Verificação foi aplicado um método de decomposição, com o objetivo de tornar mais fácil a sua resolução. Entendemos que a escolha por decompor ou não uma tarefa está relacionada ao bom senso do desenvolvedor do método, para torná-la clara o suficiente para ser entendida, pois um problema grande e complexo deve ser dividido em

problemas menores e independentes mais fáceis de serem compreendidos e resolvidos. Neste caso, porém, iremos aplicar um método de decomposição à tarefa de Verificação para exemplificar a recursividade na solução de um problema, ou seja, para mostrar que uma subtarefa originada de uma decomposição, por ser uma tarefa independente, também pode possuir um método que a decomponha em menores partes.

A seguir, são apresentadas as DSP das tarefas Seleção, Proposição e Revisão (Tabela 4.2, Tabela 4.3 e Tabela 4.4, respectivamente) para, em seguida, ser descrito como a tarefa de Verificação pode ser realizada. A maioria dos conceitos que são apresentados já foram descritos na tarefa de Configuração, uma vez que a decomposição preserva os relacionamentos da tarefa original, adicionando alguns papéis intermediários. Os conceitos introduzidos nas subtarefas são apenas conceitos utilizados para permitir a comunicação entre duas ou mais tarefas, ou seja, estes conceitos possuem o papel de saída em uma tarefa, enquanto ocupam o papel de entrada em outras.

Tabela 4.2 – DSP da tarefa Seleção

Nível Verbal:	
Descrição Textual: A tarefa de *Seleção* tem por objetivo determinar os parâmetros de um sistema que satisfazem a uma determinada condição. A condição para que um parâmetro seja selecionado é que ele não possua ainda valor, mas que todos os parâmetros dos quais ele depende já tenham seus valores computados.	
Nível Conceitual:	
Entrada:	*Parâmetro* – representa todos os parâmetros do sistema. *Associação Parâmetro* – define a relação de dependência entre os parâmetros. *Parâmetro Valorado* – representa os parâmetros do sistema com seus respectivos valores.
Saída:	*Parâmetro Selecionado* – representa os parâmetros que satisfazem à condição estabelecida.
Nível Formal:	
Axiomas (1ª. Ordem):	1) $(\forall p1,p2)$ (associação-parâmetro (p1,p2) $\rightarrow$ parâmetro(p1) $\wedge$ parâmetro(p2)) 2) $(\forall p)$ (parâmetro-valorado(p,_) $\rightarrow$ parâmetro(p)) 3) $(\forall p)$ (parâmetro-selecionado(p) $\rightarrow$ parâmetro(p))
Axiomas (Prolog):	1) *parâmetro(P):-associação-parâmetro(P,_).* *parâmetro(P):-associação-parâmetro(_,P).* 2) *parâmetro (P):- parâmetro-valorado(P,_).* 3) *parâmetro(P):-parâmetro-selecionado(P).*

Inferência:

% verifica se os parâmetros associados possuem valor
associação-parâmetro-valor(_,[]).
associação-parâmetro-valor (Parâmetro,[Um-Parâmetro|Lista-Parâmetros]):-
    parâmetro-valorado(Um-Parâmetro,_), associação-parâmetro-valor (Parâmetro, Lista-Parâmetros).
% fim
seleção(Parâmetro):- *parâmetro*(Parâmetro), *associação-parâmetro*(Parâmetro,Lista-Parâmetros),
    associação-parâmetro-valor (Parâmetro,Lista-Parâmetros).
parâmetro-selecionado(Parâmetro):- *seleção*(Parâmetro).

Tabela 4.3 – DSP da tarefa Proposição

Nível Verbal:
Descrição Textual: A *Proposição* é um tarefa que tem por objetivo fornecer um valor para um determinado parâmetro baseada nos valores de outros parâmetros do sistema. É assumido que os parâmetros envolvidos no cálculo possuem valor.
Nível Conceitual:
Entrada: *Parâmetro-Selecionado* – representa os parâmetros do sistema que terão seu valor calculado. *Fórmula-Parâmetro* – determina de que forma o valor de um parâmetro pode ser calculado a partir dos valores dos parâmetros dos quais ele depende. **Saída:** *Parâmetro Valorado* – representa os parâmetros do sistema com seus respectivos valores.
Nível Formal:
Inferência: *Proposição*(Parâmetro,Valor):-*parâmetro-selecionado*(Parâmetro), *fórmula-parâmetro* (Parâmetro,Valor). *parâmetro-valorado*(Parâmetro,Valor):-*proposição*(Parâmetro, Valor).

Tabela 4.4 – DSP da tarefa Revisão

Nível Verbal:
Descrição Textual: A tarefa de *Revisão* tem por objetivo reparar uma restrição que tenha sido violada por algum parâmetro do sistema. A tarefa calcula um novo valor para o parâmetro, baseada nos valores de outros parâmetros do sistema, de forma a satisfazer à restrição.
Nível Conceitual:
Entrada: *Restrição violada* – representa restrições que não foram satisfeitas. *Dependência Reparo* – define os parâmetros envolvidos no reparo de uma restrição. *Reparo* – determina de que forma os valores dos parâmetros devem ser alterados para reparar uma restrição. **Saída:** *Parâmetro Valorado* – representa os parâmetros do sistema com seus respectivos valores.
Nível Formal:
Inferência: % verifica se parâmetros envolvidos no reparo possuem valor dependência-reparo-valor(_,[]). Dependência-reparo-valor(Reparo,[Um-Parâmetro|Lista-Parâmetros]):- *Parâmetro-valorado*(Um-Parâmetro,_), dependência-reparo-valor (Reparo, Lista-Parâmetros). % fim *revisão*(Restrição,Parâmetro,Valor):- *restrição-violada*(Restrição), *dependência-reparo*(Reparo,Restrição,[Parâmetro|Lista-Parâmetros]), dependência-reparo-valor(Reparo,Lista-Parâmetros), *reparo*(Reparo,Valor). *parâmetro-valorado*(Parâmetro,Valor):-*revisão*(_,Parâmetro,Valor).

Como mencionamos anteriormente, optamos por aplicar um método para decompor a tarefa de Verificação em subtarefas mais simples. Este método aplicará à tarefa apenas uma

decomposição básica para dividir o problema em duas partes. A primeira, denominada de Exame, tem como objetivo identificar se a restrição satisfaz às condições necessárias para ser avaliada, enquanto a segunda, denominada de validação, é responsável por avaliar se esta restrição está sendo violada. Nas tabelas 4.5, 4.6 e 4.7 são apresentadas, respectivamente, as DSP da tarefa Verificação e suas subtarefas, às quais foram aplicados métodos que as consideram como tarefas elementares.

Tabela 4.5 – DSP da tarefa Verificação

Nível Verbal:
Descrição Textual: A tarefa *Verificação* tem por objetivo avaliar se uma determinada restrição está sendo violada pelos valores dos parâmetros do sistema. Para a restrição poder ser avaliada, é necessário que os parâmetros envolvidos na mesma possuam valor.
Subtarefas: *Exame* – examina se a restrição pode ser avaliada, verificando se os parâmetros envolvidos na restrição possuem valor. *Validação* – determina se a restrição está sendo satisfeita.
Descrição do Controle: Para resolver a tarefa precisamos primeiramente determinar se a restrição pode ser avaliada. A subtarefa *Exame* verifica se os parâmetros envolvidos na restrição possuem valor. Caso a restrição tenha sido examinada com sucesso, a subtarefa *Validação* pode determinar se a restrição está sendo violada.
Nível Conceitual:
Entrada: *Dependência Restrição* – define os parâmetros envolvidos em uma restrição. *Restrição* – determina de que forma as restrições podem ser avaliadas a partir dos valores dos Parâmetros envolvidos na mesma. *Parâmetro Valorado* – representa os parâmetros do sistema com seus respectivos valores. **Saída:** *Restrição violada* – representa as restrições que não foram satisfeitas.
Descrição Controle (Conceitual): *Examinar* restrição Se sucesso então *Validar* restrição Fim-Se
Nível Formal:
Axiomas (1ª. Ordem): 1) *(∀r) (restrição-violada(r) → restrição(r))*
Axiomas (Prolog): 1) *restrição(R):-restrição-violada(R)*
Descrição Controle (Formal): *Verificação*(Restrição) :- *exame*(Restrição), *validação*(Restrição).

Tabela 4.6 – DSP da tarefa Exame

Nível Verbal:
Descrição Textual: A tarefa de *Exame* tem por objetivo verificar se uma determinada restrição do sistema satisfaz à condição de que todos os parâmetros envolvidos na mesma possuam valor.
Nível Conceitual:
Entrada: *Dependência Restrição* – define os parâmetros envolvidos em uma restrição. *Parâmetro Valorado* – representa os parâmetros do sistema com seus respectivos valores. **Saída:** *Restrição Selecionada* – representa as restrições que satisfazem a condição estabelecida.
Nível Formal:
Inferência: % verifica se os parâmetros envolvidos na restrição possuem valor dependência-restrição-valor(_,[]). dependência-restrição-valor (Restrição, [Um-Parâmetro|Lista-Parâmetros]):- *Parâmetro-valorado*(Um-Parâmetro,_),dependência-restrição-valor(Restrição, Lista-Parâmetros). % fim *exame*(Restrição):- *dependência-restrição* (Restrição, Lista-Parâmetros), dependência-restrição-valor (Restrição, Lista-Parâmetros). *Restrição-selecionada*(Restrição):- *exame*(Restrição). % fim

Tabela 4.7 – DSP da tarefa Validação

Nível Verbal:
Descrição Textual: *Validação* é uma tarefa que tem por objetivo avaliar se uma determinada restrição do sistema está sendo violada pelos valores dos parâmetros envolvidos na mesma.
Nível Conceitual:
Entrada: *Restrição Selecionada* – representa as restrições do sistema a serem validadas. *Restrição* – determina de que forma as restrições podem ser validadas a partir dos valores dos parâmetros envolvidos na mesma. **Saída:** *Restrição violada* – representa as restrições que não foram satisfeitas.
Nível Formal:
Axiomas (1ª. Ordem): 1) $(\forall r)$ *(restrição-selecionada(r)* $\rightarrow$ *restrição(r))* 2) $(\forall r)$ *(restrição-violada(r)* $\rightarrow$ *restrição(r))* **Axiomas (Prolog):** 1) *restrição(R):-restrição-selecionada(R).* 2) *restrição(R):-restrição-violada(R).* **Inferência:** *validação*(Restrição):- *restrição-selecionada* (Restrição), not (*restrição*(Restrição)). *restrição-violada*(Restrição) :- *validação*(Restrição).

A seguir, são apresentadas duas estruturas que resumem a resolução da tarefa de Configuração apresentada nesta seção. A primeira estrutura apresentada é denominada

Estrutura Tarefa-Método (figura 4.6) e tem por objetivo permitir a visualização de como as subtarefas e os métodos devem ser combinados para a realização da tarefa original. O modelo ilustra como as tarefas podem ser, em alguns casos, decompostas em subtarefas mais simples pelo método de solução de problemas, enquanto que, em outros, são resolvidas diretamente. As tarefas compostas e elementares são identificadas, no diagrama, respectivamente, através de elipses hachuradas e elipses simples.

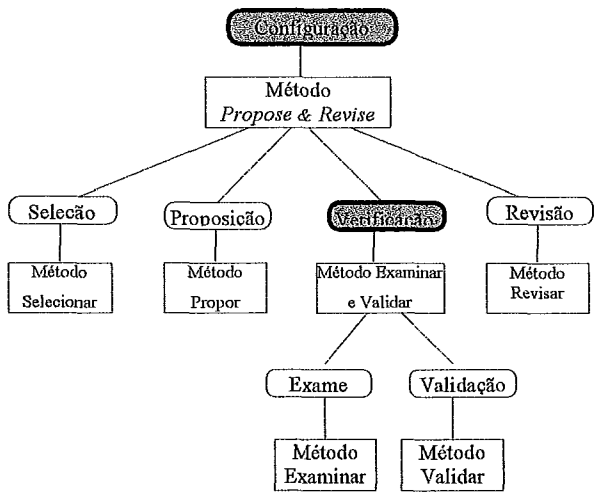

Figura 4.6 – Estrutura Tarefa-Método da DSP da tarefa de Configuração

O segundo modelo apresentado é chamado de Estrutura de Inferência (SCHREIBER *et al.*, 1993) e descreve os relacionamentos entre as subtarefas, representadas por elipses, e os seus respectivos conceitos, representados por retângulos. Através da Estrutura de Inferência (figura 4.7), é possível ter-se uma visão geral do método de solução de problemas e do conhecimento usado pelas subtarefas, uma vez que a estrutura descreve o fluxo de conhecimento entre as subtarefas (indicado por setas) necessário para a resolução do problema.

À esquerda da figura é apresentada a Estrutura de Inferência gerada pela aplicação do método P&R à tarefa de Configuração. Primeiramente, a subtarefa Seleção escolhe um parâmetro do sistema para que este tenha seu valor computado pela subtarefa Proposição. A subtarefa Verificação avalia, então, se o valor do parâmetro satisfaz às restrições do sistema. Caso existiam restrições violadas, estas serão reparadas pela subtarefa Revisão, de forma a gerar um novo valor para o parâmetro.

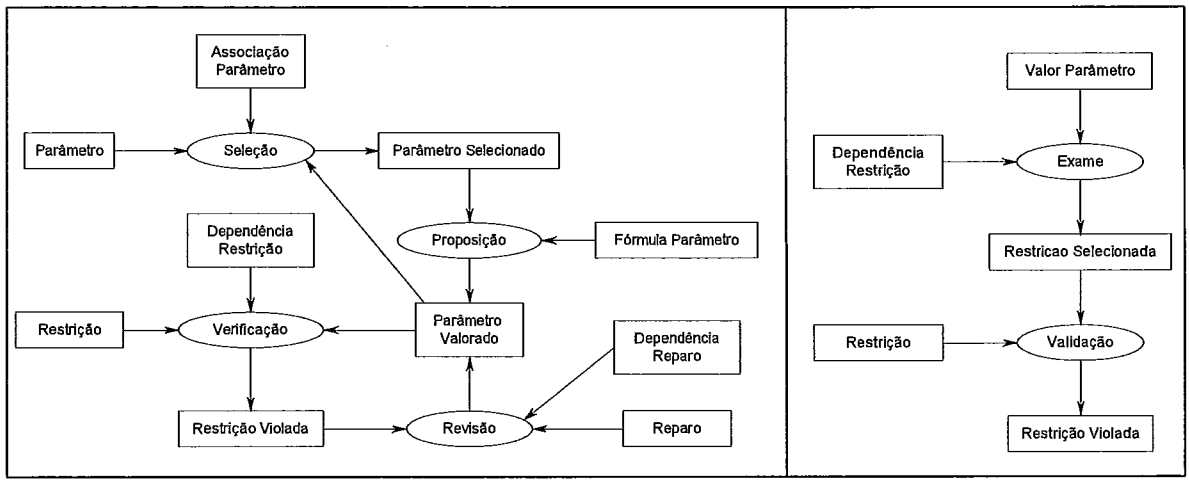

Figura 4.7 – Estrutura de Inferência das DSP de Configuração e Verificação

O fluxo de conhecimento necessário para solucionar a tarefa de Verificação é apresentado à direita. Inicialmente a subtarefa *Exame* determina se a restrição pode ser avaliada, verificando se os parâmetros envolvidos na restrição possuem valor. Caso a restrição tenha sido examinada com sucesso, a subtarefa *Validação* determina se a restrição está sendo violada.

4.4 Conclusão do Capítulo

Neste capítulo apresentamos uma estrutura para organizar o conhecimento de tarefas, útil para o entendimento e resolução de problemas em ambientes de desenvolvimento de software. Esta estrutura, chamada de Descrição de Solução de Problemas, permite não só armazenar documentos descrevendo os objetivos da tarefa, como também os conceitos envolvidos na tarefa e as inferências necessárias para sua resolução. A estrutura apresentada tem como elemento-chave a unificação de ontologias de tarefa, para a representação dos conceitos envolvidos, e métodos de solução de problemas, que especificam as inferências necessárias para que o objetivo da tarefa possa ser alcançado. Através da DSP, podemos, ainda, identificar que conceitos são importantes na realização das tarefas e, com isso, buscar conhecimento do domínio correspondente.

Foram apresentadas, ainda neste capítulo, as motivações pela escolha de ontologias de tarefa e métodos de solução de problemas para representação do conhecimento de tarefa, de forma a permitir o entendimento do problema pelo engenheiro de software e o apoio ao desenvolvimento de sistemas. Para permitir a descrição de todos os componentes

necessários para sua representação, organizamos a DSP em três níveis de abstração, onde em cada nível é dada uma ênfase seletiva nos detalhes representados, permitindo, desta forma, que sua descrição e organização possam auxiliar no desenvolvimento de sistemas, como será apresentado no próximo capítulo.

Capítulo 5

Descrição de Solução de Problemas na Estação TABA

Este capítulo apresenta como a Descrição de Solução de Problemas foi incorporada na infra-estrutura da Estação TABA. Inicialmente, apresentamos as extensões realizadas na Estação para, em seguida, apresentarmos a implementação de uma ferramenta para a descrição de tarefas. Finalmente, mostra-se um exemplo da assistência da DSP em ambientes de desenvolvimento de software.

5.1 Introdução

A utilização de uma estrutura para a representação do conhecimento de tarefa é um dos requisitos da Estação TABA (ROCHA *et al*, 1990), devido a este conhecimento permitir tornar o processo de entendimento do problema mais fácil para os desenvolvedores. Para permitir que este requisito seja satisfeito pela Estação, são utilizados Servidores de Conhecimento (FALBO, 1998) que, conforme visto na seção 2.2.2, têm como objetivo prover uma biblioteca de tarefas para os tipos de problema mais comumente encontrados no universo de discurso em questão. Esta biblioteca é baseada nos modelos de tarefa do *Common*KADS.

Contudo, nesta tese, estamos propondo uma representação do conhecimento de tarefa baseada em ontologias de tarefa e métodos de solução de problemas, em detrimento aos modelos de tarefa e, portanto, nos deparamos com a necessidade de estender a atual estrutura do Servidor de Conhecimento. Na próxima seção, são apresentadas as modificações necessárias no Servidor de Conhecimento para permitir que este comporte a estrutura da DSP, que é uma abordagem metódica e rigorosa para resolver problemas e permite disponibilizar conhecimento de tarefa de forma organizada e estruturada para

orientar o desenvolvimento de software. A DSP permite apoiar o entendimento da tarefa, disponibilizando conhecimento de tarefa independente de quais sistemas já existam na empresa e dos produtos de software que se deseja construir. Além disso, esse conhecimento é útil na construção de ferramentas de desenvolvimento de software e pode ser utilizado no desenvolvimento dos próprios produtos de software.

Já foi desenvolvida na Estação TABA uma ferramenta para descrição de tarefas, denominada EDITAR (OLIVEIRA, 1999a), com o simples objetivo de descrever as principais características de uma tarefa e referências para a literatura específica. Esta ferramenta foi desenvolvida independente do Servidor de Conhecimento, somente para permitir que fosse possível identificar as tarefas relacionadas a um determinado domínio e mapear o conhecimento do domínio necessário para o entendimento de cada tarefa, não tendo por objetivo definir como a tarefa é realizada ou mesmo auxiliar no entendimento do domínio do problema. Para permitir que os elementos que compõem uma DSP possam ser cadastrados e, portanto, seja representado o conhecimento necessário para o entendimento do problema, assim como sua resolução, apresentamos na seção 5.3 uma nova implementação para a ferramenta de edição de tarefas.

5.2 A Extensão ao Servidor de Conhecimento

A atual arquitetura dos Servidores de Conhecimento é descrita na figura 5.1 e foi definida por FALBO (1998a) para permitir a inclusão de fontes diversificadas de conhecimento, devido o componente de conhecimento ser considerado essencial para garantir a integração de ferramentas por manter o conhecimento sobre processos, métodos e domínios de aplicação.

Um dos pontos principais da arquitetura do Servidor de Conhecimento é a separação entre o conhecimento do domínio do problema e o conhecimento geral da solução do problema. Para permitir o reuso de conhecimento do domínio, é utilizada uma base de conhecimento modular e baseada em ontologias. Quanto ao reuso do conhecimento de tarefa, o Servidor de Conhecimento provê uma biblioteca de resolvedores genéricos de problemas, passíveis de serem instanciados e adaptados para aplicações particulares.

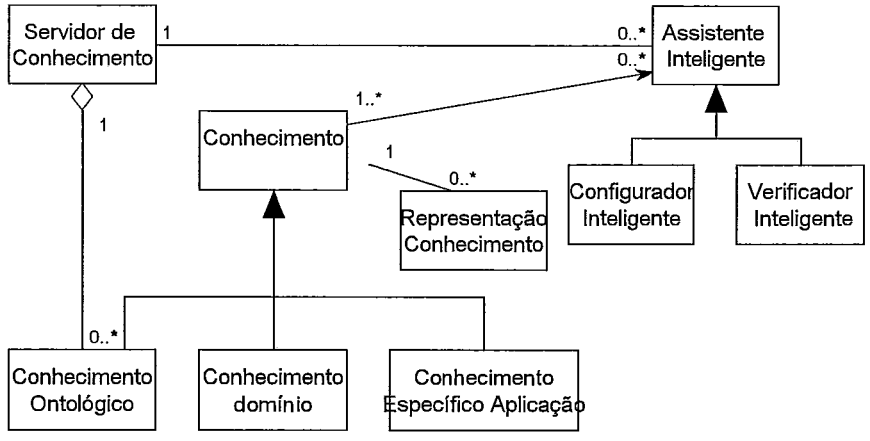

Figura 5.1: Arquitetura atual do Servidor de Conhecimento

A classe *Servidor de Conhecimento* agrega a arquitetura dos Servidores de Conhecimento, descrita na seção 2.2.2. Os módulos de conhecimento, propostos nesta arquitetura, foram modelados como objetos da classe *Conhecimento,* enquanto que os *templates* de resolvedores de problemas foram modelados como subclasses da classe *Assistente Inteligente.* Os objetos da classe *Conhecimento Ontológico* representam o conhecimento capturado nas ontologias propriamente ditas, em contraste com os objetos da classe *Conhecimento Domínio*, que representam instanciações das ontologias e, portanto, conhecimento factual. Já o conhecimento que é específico para uma aplicação particular é capturado pelos objetos da classe *Conhecimento Especifico Aplicação.* A classe *Representação Conhecimento* é responsável por permitir a definição de diferentes tecnologias de representação de conhecimento: rede neural e sistemas lógicos, englobando frames, sistema de programação em lógica e rede semântica. A figura 5.1 apresenta, ainda, a hierarquia das classes *Assistente Inteligente,* que contempla dois resolvedores de problemas, *Configurador Inteligente* e *Verificador Inteligente,* que devem ser construídos com base nos respectivos modelos de tarefa. Qualquer *template* de resolvedor de problemas deve ser modelado como uma classe abstrata nesta hierarquia.

Consideramos, porém, que o conhecimento sobre uma tarefa não se limita simplesmente à descrição de sua resolução, devendo ser composto também pela representação do conhecimento necessário para o entendimento do problema. Ou seja, ao invés de descrevermos a resolução do problema de forma genérica que possibilite entendermos como um problema pode ser resolvido, a DSP realiza uma descrição em que as tarefas são descritas genericamente para permitir o seu entendimento, enquanto que a resolução é apresentada de forma explícita, ou seja, formalmente.

Além disso, uma vez que a separação entre o conhecimento do domínio do problema e o conhecimento da solução do problema é uma das principais características da arquitetura do Servidor de Conhecimento, as ontologias passam a ter um papel fundamental, pois permitem uma separação bem definida do que significam os conceitos de um domínio e as tarefas que são automatizadas sobre eles. Por isto, ao serem utilizadas, na DSP, ontologias tanto para representar o conhecimento de domínio quanto o conhecimento de tarefa, estamos garantindo a integração destes conhecimentos em uma abordagem uniforme e consistente. Para incorporar a estrutura da DSP na Estação TABA, fez-se necessário alterar a arquitetura do Servidor de Conhecimento, apresentada anteriormente. A nova arquitetura é apresentada na figura 5.2.

Esta nova estrutura do Servidor de Conhecimento permite a representação das tarefas descritas conforme a DSP, na Estação TABA. Os módulos de conhecimento praticamente não sofreram alterações, uma vez que a DSP não trata do conhecimento de domínio e de aplicação. O *Conhecimento Ontológico* passou a representar, além do conhecimento capturado nas ontologias de domínio, conhecimento obtido nas ontologia de tarefa, ou seja, axiomas (restrições) provenientes das relações entre os conceitos de uma tarefa. O *Conhecimento Ontológico* deve ser fornecido pelo engenheiro de conhecimento na forma de um arquivo previamente preparado, uma vez que a versão corrente do Servidor de Conhecimento não dispõe de uma ferramenta para edição de ontologias.

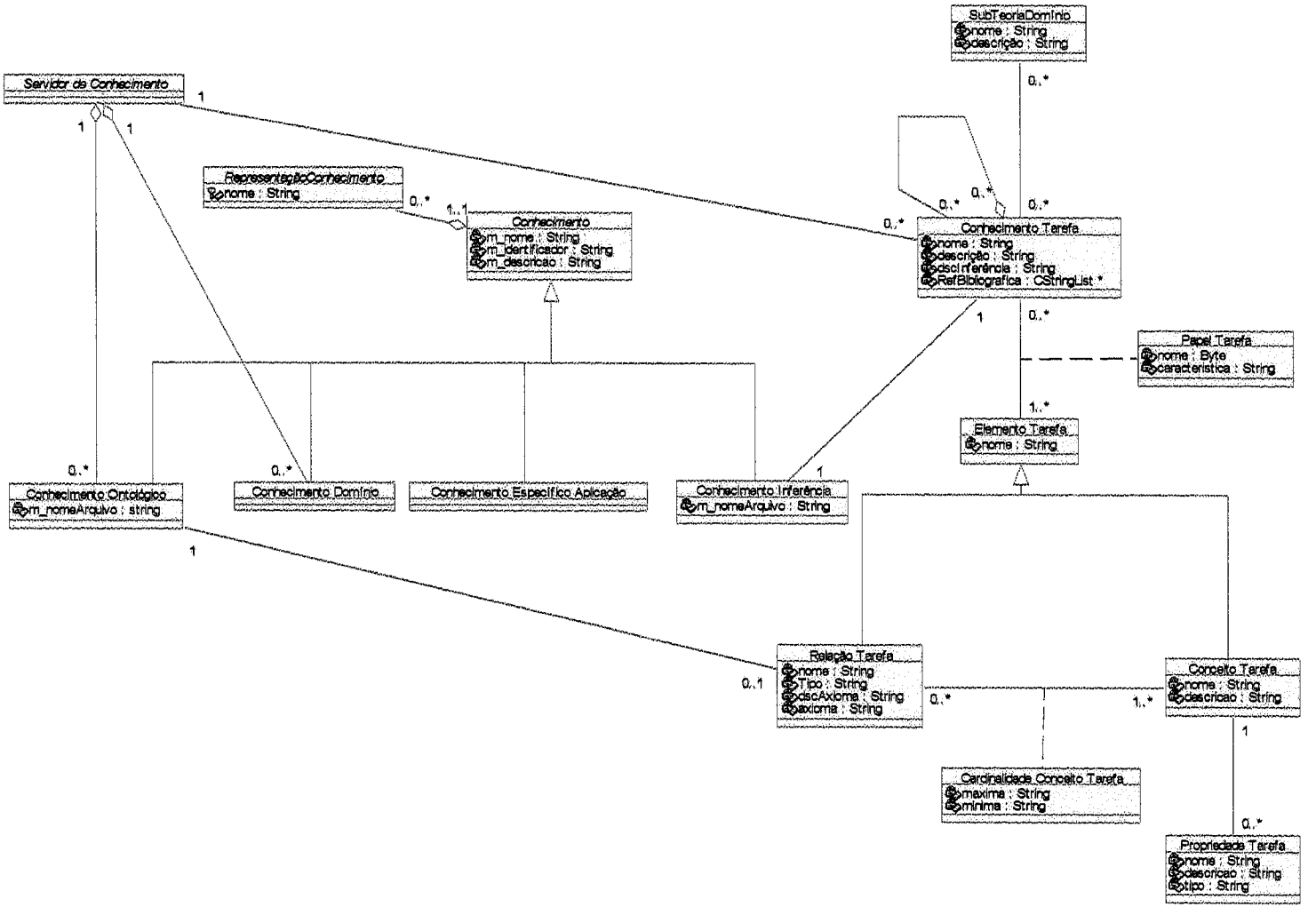

Figura 4.9: Nova Arquitetura do Servidor de Conhecimento da Estação TABA

O *Conhecimento Inferência* utiliza esta mesma técnica para representar as inferências necessárias para a realização de uma determinada tarefa. Finalmente, a classe *SubTeoriaDomínio* é descrita para permitir a identificação das tarefas relacionadas a um determinado domínio e o mapeamento de quais sub-teorias são necessárias para o entendimento de cada tarefa. Na tabela 5.1, são apresentadas as novas classes necessárias para a representação da DSP.

Tabela 5.1: Representação das Classes da Descrição de Solução de Problemas.

Classe	Descrição
Conhecimento Tarefa	Repositório de tarefas definidas conforme a DSP. Uma tarefa pode ser definida como uma agregação de subtarefas.
Elemento Tarefa	São os elementos manipulados por uma tarefa. Estes elementos podem ser tanto conceitos da tarefa como relações entre estes conceitos.
Papel Tarefa	Representa os papéis dos elementos na resolução da tarefa. Ex: entrada; saída.
Conceito Tarefa	São os conceitos envolvidos em uma tarefa
Propriedade Tarefa	Representa as propriedade de um conceito
Relação Tarefa	Representa as relações entre os conceitos de uma tarefa, definindo axiomas.

Estas classes vieram substituir a hierarquia do assistente inteligente, que era utilizado apenas para disponibilizar os resolvedores de problema para o Servidor de Conhecimento. O assistente inteligente foi substituído, uma vez que é mais vantajoso representar a resolução através de objetos de classes ao invés de subclasses. Desta forma, é possível solucionar algumas restrições características do assistente inteligente, como: (i) associar conhecimento de domínio aos papéis do conhecimento de uma tarefa; (ii) navegar pelas tarefas do servidor de conhecimento para selecionar as tarefas que irão compor uma aplicação; (iii) compor uma tarefa para a aplicação, a partir de tarefas já cadastradas, e, (iv) modelar qualquer tarefa que possa auxiliar no entendimento do problema sem a necessidade de se representar uma nova classe na arquitetura do Servidor.

5.3 Ferramenta para Descrição de Tarefas

Para apoiar a descrição do conhecimento de tarefa na Estação TABA, já havia sido desenvolvida por OLIVEIRA (1999a) uma primeira versão de um editor de tarefas. A descrição de tarefas neste editor era feita de forma simplificada, pois tinha somente o

objetivo de permitir a identificação das tarefas relacionadas a um determinado domínio, durante a definição da Teoria do Domínio. Esse editor, denominado EDITAR, consistia da entrada de um texto descrevendo a tarefa e de uma lista de referências que podiam ser pesquisadas para melhor esclarecimento da tarefa. A ferramenta não tinha por objetivo definir como a tarefa era realizada ou auxiliar no entendimento do domínio do problema.

Conforme discutido anteriormente, decidimos estender a ferramenta EDITAR para permitir que os elementos que compõem uma DSP possam ser cadastrados e, portanto, o conhecimento necessário para o entendimento do problema, assim como para sua resolução, possa ser representado. Os requisitos de entrada do EDITAR englobam, portanto, os elementos utilizados para definir uma DSP, de forma a ser útil em um ADS. A ferramenta permite a edição da tarefa, definida pelo engenheiro de software, e a visualização de todas as tarefas já descritas organizadas em uma biblioteca, de forma a facilitar a busca de tarefas e sua reutilização na descrição de problemas complexos. A edição de tarefas pode ser feita através da opção *EDITAR* no menu Ferramentas da Estação TABA (ver Figura 5.3).

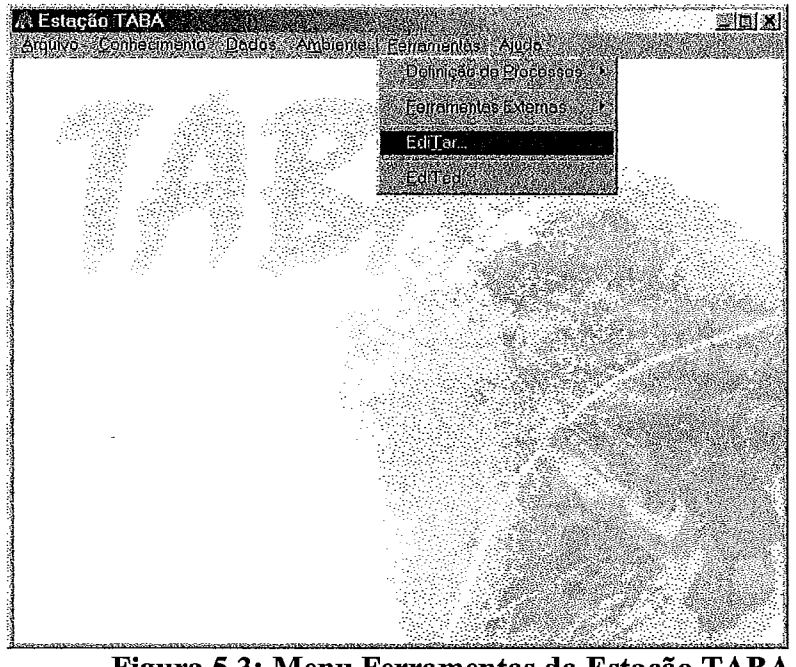

Figura 5.3: Menu Ferramentas da Estação TABA.

Inicialmente, o engenheiro do ambiente deve entrar com o nome e a descrição da tarefa que deseja editar (Figura 5.4). O engenheiro deve informar, também, se a tarefa é elementar ou composta, uma vez que as informações definidas na DSP são distintas para este dois tipos.

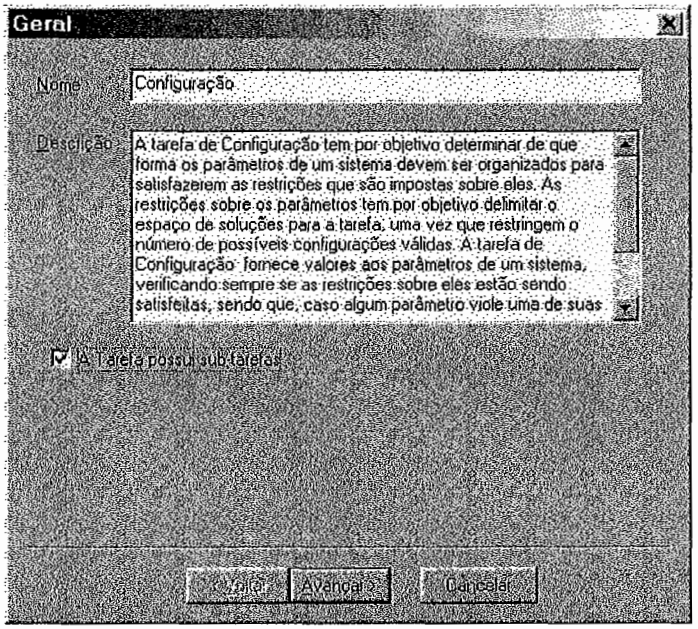

Figura 5.4: Descrição Geral da Tarefa.

No caso de uma tarefa composta, é necessário que sejam especificadas as subtarefas que a compõem. As tarefas são cadastradas de forma *bottom-up*, ou seja, as subtarefas devem ser definidas antes da tarefa principal. Uma estrutura de árvore é utilizada para organizar as tarefas já cadastradas na biblioteca, de forma que estas possam ser selecionadas para compor a nova tarefa. Ao selecionar uma tarefa na árvore, o usuário tem acesso às suas respectivas subtarefas e às informações sobre sua descrição. Além disso, são apresentados os conceitos e relações envolvidos na tarefa, conforme mostra a Figura 5.5.

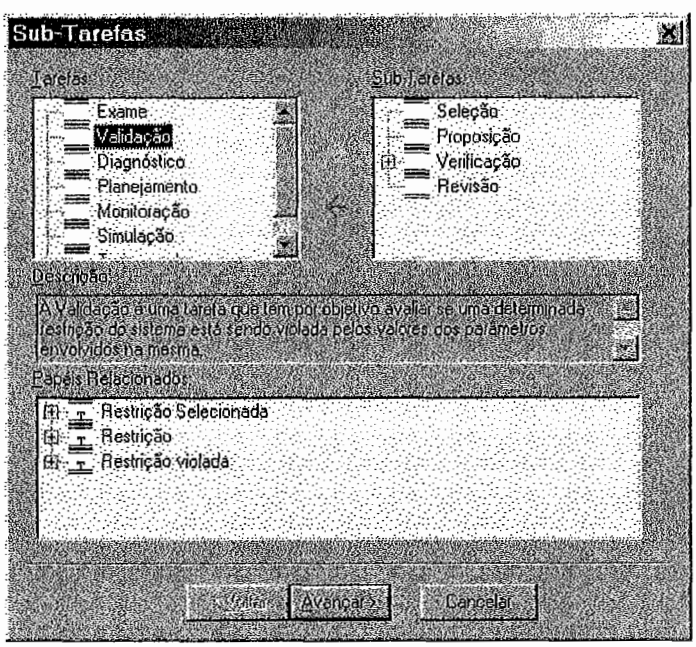

Figura 5.5: Seleção das Subtarefas.

Em seguida, deve ser definido o conjunto de conceitos e relações envolvidos na tarefa, assim como os seus papéis na resolução da mesma (Figura 5.6). No caso de uma tarefa composta, uma vez que a decomposição preserva os relacionamentos da tarefa original, os conceitos e relações envolvidos na tarefa podem ser selecionados a partir dos conceitos e relações definidos para suas subtarefas. Para o caso de uma tarefa elementar, todos os conceitos e relações já cadastrados na biblioteca são apresentados para que seja possível a reutilização na nova tarefa. Quando o usuário seleciona um conceito qualquer apresentado, a ferramenta disponibiliza sua descrição na parte inferior da tela. Para o caso de uma relação, são apresentados também os conceitos envolvidos.

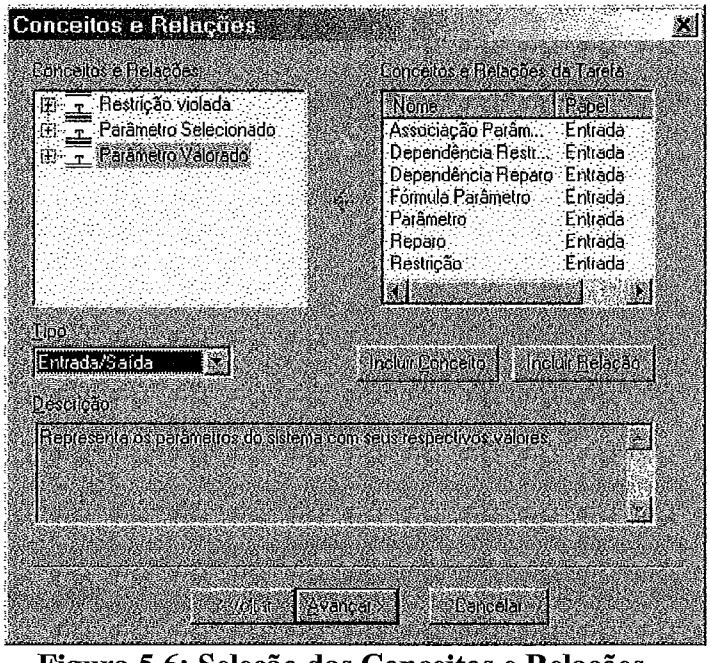

Figura 5.6: Seleção das Conceitos e Relações.

Para uma tarefa elementar, pode ser necessária a definição de um novo conceito ou relação, que deverá ser feita através dos botões *Incluir Conceito* e *Incluir Relação*, respectivamente. Ao acionar o botão *Incluir Conceito*, a ferramenta permite que sejam cadastrados a descrição do conceito e suas propriedades. A inserção de uma relação entre conceitos é realizada a partir do botão *Incluir Relação*, que permite ao engenheiro do conhecimento cadastrar a descrição e o tipo da relação, os conceitos envolvidos na mesma com suas respectivas cardinalidades, além do conhecimento ontológico (axiomas) proveniente da relação, como mostra a Figura 5.7. As relações do tipo *subtipo* e *partede*

entre dois conceitos são geradas automaticamente pela ferramenta EDITAR, enquanto que as demais restrições devem ser entradas pelo usuário em Prolog, para formarem os módulos de conhecimento do Servidor de Conhecimento, a serem utilizados por futuras ferramentas. Na Estação TABA, a forma de representação está sendo implementada através de sistema de programação em lógica utilizando uma máquina Prolog externa, o SWI-Prolog (WIELEMAKER, 1998).

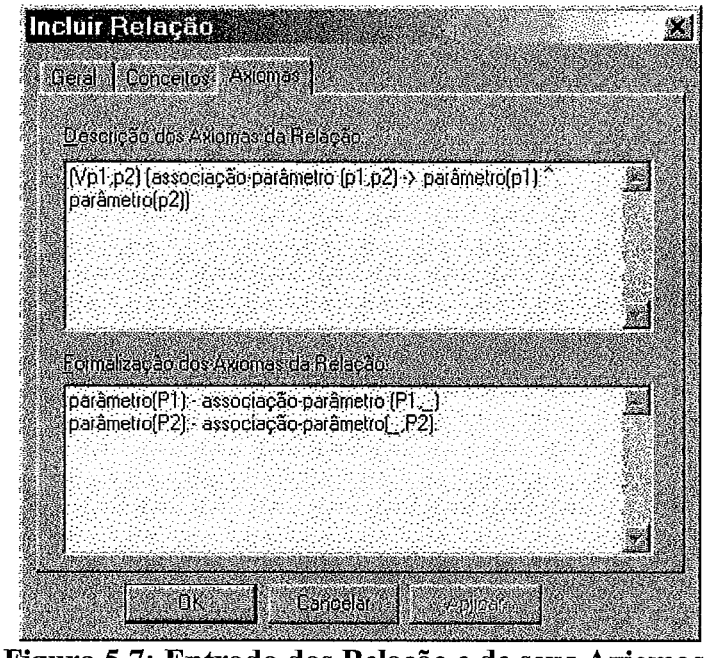

Figura 5.7: Entrada das Relação e de seus Axiomas.

Por fim, o engenheiro de software cadastra as inferências necessárias para que a tarefa possa ser resolvida, conforme é apresentado na Figura 5.8. A formalização (axiomas) destas inferências é mapeada para a classe *Conhecimento Inferência*, e deve ser descrita em Prolog, assim como foi realizado para as relações entre os conceitos.

Para permitir que o engenheiro de software tenha um entendimento completo do problema, a ferramenta EDITAR possui, ainda, um modo navegação. Neste modo, é possível visualizar todas as tarefas e suas respectivas subtarefas, organizadas em uma estrutura de árvore, permitindo uma maior compreensão da tarefa e dos componentes disponíveis (figura 5.9). É apresentada, inclusive, uma lista de referências para melhor esclarecimento da tarefa. Desta forma é possível apoiar a descrição e evolução das tarefas a

serem utilizadas no ADS e a efetiva utilização do conhecimento de tarefa ao longo do processo de desenvolvimento de software.

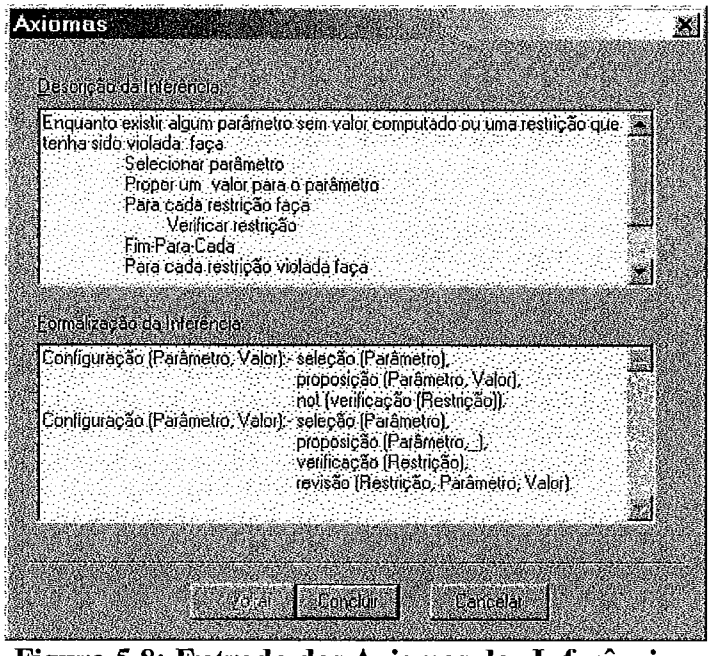

Figura 5.8: Entrada dos Axiomas das Inferências.

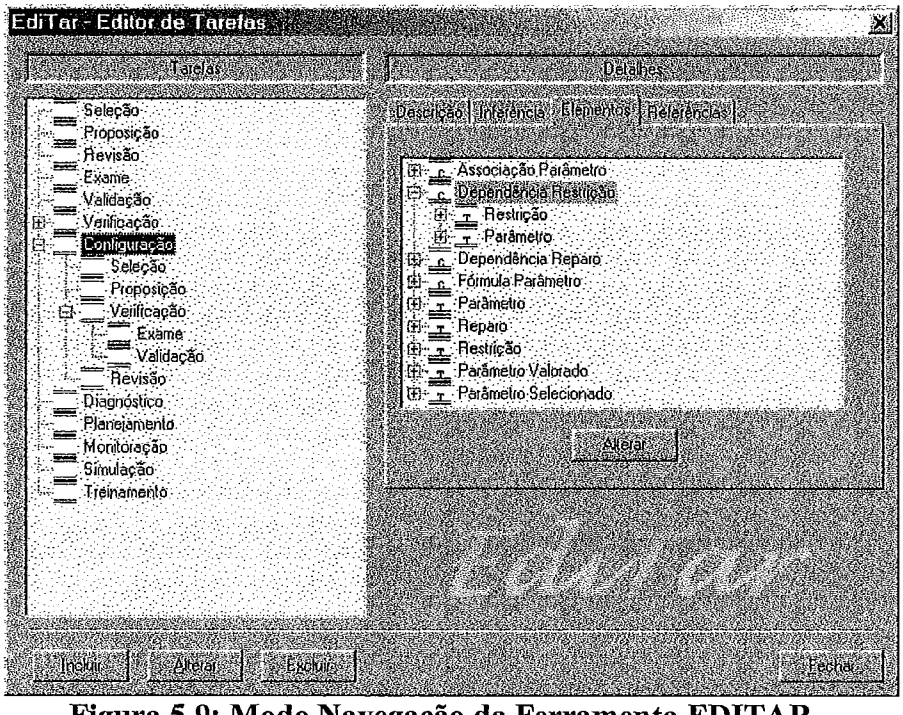

Figura 5.9: Modo Navegação da Ferramenta EDITAR.

5.4 Um Exemplo de Assistência em ADS

Através do desenvolvimento da Descrição de Solução de Problemas e de sua inclusão na Estação TABA, obtemos uma estrutura que considera tanto o conhecimento do domínio, descrito nos ADSOD, quanto o conhecimento de tarefa. O conhecimento de tarefa se mantém independente do domínio em que será aplicado, de forma que ambos os conhecimentos permaneçam genéricos e, desta forma, aplicáveis a vários sistemas. Nesta seção, será mostrado como esses conhecimentos podem auxiliar no desenvolvimento de sistemas em um ADS.

Consideremos, por exemplo, um situação de uso no ambiente NETUNO, desenvolvido para o domínio de acústica submarina (GALOTTA, 2000). O conhecimento do domínio foi descrito através de ontologias e considera conceitos sobre sonares, sons, embarcações e outros conceitos envolvidos na propagação de som no meio marítimo. O objetivo de um sonar na acústica submarina é detectar alvos a distância e, para isso, é necessário realizar a configuração de seus atributos. Dentre os atributos do sonar (Ex: sensibilidade, eixo acústico, índice de diretividade, voltagem no terminal, intensidade acústica, freqüência), alguns já possuem seu valor predefinido (Ex: índice de diretividade = 30dB, freqüência = 10Hz), enquanto outros precisam ser calculados a partir dos valores de outros atributos (Ex: Sensibilidade = 20 log (voltagem nos terminais) – 10 log (intensidade acústica)). Após serem calculados, os atributos do sonar podem vir a infringir alguma de suas restrições, sejam estas em relação a um outro atributo ou em relação ao seu próprio limite máximo/mínimo (Ex: 0dB <= sensibilidade <= 200dB). Caso isto aconteça, são reajustados os valores dos atributos necessários para que a restrição seja reparada.

Um dos problemas a ser automatizado nesse domínio é, portanto, a configuração de sonares. Consultando a DSP descrita sobre configuração, o desenvolvedor pode entender sobre essa tarefa (ver figura 4.1 e 4.2) e, a partir das informações obtidas, identificar o que é necessário para a realização da tarefa de configuração (ou seja, conceitos de entrada e saída como *Parâmetro, Restrição,* entre outros – ver seção 4.2.2). Esses conceitos devem ser buscados na própria ontologia do domínio (podendo ser conceitos ou suas propriedade) ou identificados no momento de elicitação de requisitos. Os conceitos principais da tarefa de configuração são os próprios conceitos/propriedades do sonar, além de relações que se

referem a outros conceitos do domínio, como podemos ver pela discussão acima sobre valores de configuração de sonar.

Os conceitos definidos em uma DSP, porém, podem não encontrar correspondentes diretos no conhecimento do domínio que está sendo aplicado, pelo fato de que a descrição de um domínio pode não estar completa, por ser o conhecimento do domínio de natureza evolutiva. Não é objetivo de uma ontologia de domínio descrever todo o conhecimento a ser codificado em uma base de conhecimento, afinal algum conhecimento empírico, compilado ou prático, que é dependente da tarefa ou aplicação particular, pode encontrar lugar apenas em uma ontologia de aplicação (FALBO, 1998a). O conhecimento desempenha papéis na resolução de problemas e, uma vez que muitos destes papéis são dinamicamente atribuídos, não podem fazer parte de uma ontologia de domínio. Portanto, não cabe à ontologia de domínio estabelecer interações entre conhecimento de domínio e conhecimento geral de resolução de problema. Até mesmo uma ontologia de aplicação não necessariamente contempla todo o conhecimento a ser codificado em uma base de conhecimento (GUARINO, 1998).

Além disso, nem todos os conceitos definidos na DSP precisam ser mapeados para algum conceito do domínio. Este é o caso de conceitos que são utilizados apenas para permitir a comunicação entre duas ou mais tarefas e, portanto, suas instâncias são definidas durante as inferências que realizam a tarefa. Para os outros conceitos, caso não sejam encontrados correspondentes na ontologia de domínio, dois procedimentos podem ser seguidos. Caso o engenheiro de software considere que este conceito é essencial ao domínio em que se está trabalhando, ele deverá introduzi-lo na ontologia de domínio respectiva de forma a estendê-la. Caso contrário, ele deverá acrescentar este novo conceito ao conhecimento de aplicação, que é o responsável por armazenar as informações relativas a uma aplicação específica que está sendo desenvolvida, tais como heurísticas, regras de negócio e conhecimento compilado.

Para demonstrar como a DSP, além de apoiar no entendimento do problema e do que deve ser considerado, pode ainda auxiliar na especificação da solução, apresentaremos nesta seção um exemplo de como a DSP é utilizada na especificação de casos de uso (JACOBSON *et al.*, 1992). Resumidamente, um caso de uso descreve as funcionalidades de um sistema para cada usuário. Um caso de uso é composto de: (i) um nome e descrição, que

determinam uma funcionalidade do sistema, (ii) atores, que são usuários, outros sistemas ou qualquer agente externo ao sistema que interagem com o mesmo, (iii) um fluxo de eventos, que determina a sequência de passos que o caso de uso precisa realizar, e, (iv) um diagrama, que é uma representação gráfica da interação entre atores e casos de uso. Além disso, quando um caso de uso utiliza, dentro de seu fluxo de eventos, um outro caso de uso, dizemos que este é um caso de uso incluído do primeiro. A seguir, apresentamos como a DSP pode auxiliar na descrição destes componentes:

i) **Nome e Descrição do Caso de Uso:** A descrição do caso do uso pode ser definida utilizando a descrição do problema em linguagem natural, presente no nível verbal da DSP. O desenvolvedor de software sempre pode modificar a descrição proposta para melhor adaptá-la à função do sistema. O nome do caso de uso deve estar relacionado com a função do sistema que está sendo especificada, devendo ser baseada no nome da própria tarefa. A tabela 5.2 apresenta, primeiramente, o trecho da descrição textual da DSP Configuração que foi utilizada como base para a descrição do caso de uso, descrito em seguida.

Tabela 5.2: Descrição Textual da DSP Configuração / Descrição do Caso de Uso Configurar Sonar.

***DSP Configuração**: A tarefa de Configuração fornece valores aos parâmetros de um sistema, verificando sempre se as restrições sobre eles estão sendo satisfeitas, sendo que, caso algum parâmetro viole uma de suas restrições, é necessário que seja estabelecido um novo valor para ele, de forma a satisfazer a restrição violada. A tarefa termina quando todos os valores de todos os parâmetros forem computados.*	***Caso de Uso Configurar Sonar:** função realizada para fornecer valores aos atributos de um sonar, verificando sempre se as restrições sobre eles estão sendo satisfeitas. Caso algum atributo do sonar viole uma de suas restrições, é necessário que seja estabelecido um novo valor para ele, de forma a satisfazer a restrição violada. A função termina quando todos os valores de todos os atributos do sonar forem computados.*

ii) **Atores:** O ator de um caso de uso pode ser identificado através da elicitação de requisitos ou, no que se refere à Estação TABA, consultando o conhecimento organizacional definido nos ADSOrg (VILLELA, ZLOT e SANTOS, 2001), que permite a identificação, dentro de uma estrutura organizacional, dos profissionais mais

adequados para a realização de uma atividade ou para a solução de um problema. Um ADSOrg (vide seção 2.2.4) possui ferramentas que permitem que os usuários do ambiente possam navegar entre os conhecimentos e habilidades disponíveis na organização em busca de quem possui a habilidade ou o conhecimento mais próximo do desejado. Assim sendo, após identificar a tarefa da DSP que será utilizada para auxiliar a modelagem do caso de uso, o desenvolvedor deve também definir o ator que deve realizá-la, ou seja, dentre as pessoas que trabalham na organização, quais são aquelas que têm o perfil para executar a tarefa. Para fazer essa escolha, é fundamental que o engenheiro de software entenda como a tarefa funciona e, portanto, faça uso das descrições da DSP. Para o caso da configuração do sonar, iremos considerar que o ***técnico sonar*** foi identificado como o ator mais apropriado para interagir com o sistema.

iii) **Fluxo de eventos principal:** O fluxo de eventos de um caso de uso pode ser obtido a partir da ordem em que as subtarefas são chamadas durante o fluxo de controle da tarefa. Dentre os três níveis de descrição do controle da tarefa definidos na DSP, o mais propício para auxiliar a definição do fluxo de eventos é o presente no nível conceitual, uma vez que este apresenta uma estrutura bem definida e é, ao mesmo tempo, independente de uma linguagem formal. É preciso enfatizar que o engenheiro de software pode alterar qualquer informação obtida através do fluxo de controle, adicionar eventos que não foram fornecidos pela DSP ou rejeitar outros que tenham sido apresentados. A DSP tem por objetivo auxiliar o engenheiro de software no desenvolvimento de sistemas e não substituí-lo. Como o objetivo do fluxo de eventos é mostrar a interação entre o sistema e os atores, mas esta interação só está presente na interface gráfica do sistema, iremos aqui simular uma interface em que o usuário é o responsável por solicitar o início da configuração, enquanto que o sistema fica responsável por realizar as atividades envolvidas na tarefa, além de informar ao usuário os valores finais para os atributos do sonar. Os atores que interagem com o sistema durante o fluxo de eventos são os mesmos que foram definidos na etapa anterior, conforme pode ser visto na tabela 5.3, que especifica o fluxo de eventos principal do caso de uso *Configurar Sonar*, no qual o *técnico sonar* é o responsável pela realização

da tarefa. Na tabela, é apresentada primeiramente a descrição do controle da DSP, para em seguida ser descrito o fluxo de eventos do caso de uso.

Tabela 5.3: Descrição do Controle em Linguagem Estruturada da DSP Configuração / Fluxo de eventos principal do Caso de Uso Configurar Sonar.

Controle da DSP:	***Fluxo de Eventos:***
	1) O técnico sonar solicita o início da configuração do sonar
Enquanto** existir algum parâmetro sem valor computado ou uma restrição que tenha sido violada **faça	*2) **Enquanto** existir algum atributo do sonar sem valor computado ou uma restrição que tenha sido violada:*
***Selecionar** parâmetro*	*2.1)O sistema **seleciona** um atributo do sonar*
***Propor** um valor para o parâmetro*	*2.2)O sistema **propõe** um valor para o atributo*
Para cada** restrição **faça	*2.3)**Para cada** restrição do sonar:*
***Verificar** restrição*	*2.3.1) O sistema **verifica** se a restrição foi violada*
Fim-Para-Cada	***Fim-Para-Cada***
Para cada** restrição violada **faça	*2.4)**Para cada** restrição do sonar violada:*
***Revisar** restrição*	*2.4.1) O sistema **revisa** a restrição*
Fim-Para-Cada	***Fim-Para-Cada***
Fim-Enquanto	***Fim-Enquanto***
	3) O sistema informa os valores dos atributos e o caso de uso termina

iv) **Casos de uso incluídos:** Para que o engenheiro de software identifique um caso de uso incluído, ele pode fazer uso da descrição das subtarefas definidas na DSP de configuração. Ou seja, o engenheiro identifica, dentre as subtarefas especificadas, aquelas para as quais ele precisou modelar um caso de uso. Para entendermos como isto funciona, iremos considerar que é interessante no sistema que o *técnico sonar* possa chamar, a qualquer instante, a funcionalidade que verifica se os valores dos atributos do sonar estão satisfazendo todas as restrições existentes. Para isto, seria preciso modelar um caso de uso somente para esta funcionalidade. A definição deste caso de uso, que chamaremos de *Verificar Sonar*, pode ser auxiliada pela DSP de *Verificação*, descrita na seção 4.3. Para que o engenheiro de software identifique, portanto, que *Verificar Sonar* é um caso de uso incluído do caso *Configurar Sonar,* ele pode fazer uso da descrição das subtarefas que foram originadas pela aplicação do método de solução do problema, que é especificada no nível verbal da DSP.

v) **Diagrama de casos de uso:** Como, agora, trata-se apenas de produzir a representação gráfica (figura 5.10), o desenvolvedor de software pode efetuar a modelagem, considerando os elementos anteriormente descritos. Como o diagrama é modelado a partir dos atores e não a partir das funções, para representá-lo é preciso apenas identificar quais os casos de uso que um ator específico tem a responsabilidade de realizar. No sistema que estamos representando, o ator *técnico sonar* realiza os casos *Configurar Sonar* e *Verificar Sonar*, sendo que o segundo é um caso de uso incluído do primeiro. Essas funcionalidades, no entanto, fazem parte de um conjunto de outras funções nos sistemas de propagação de som.

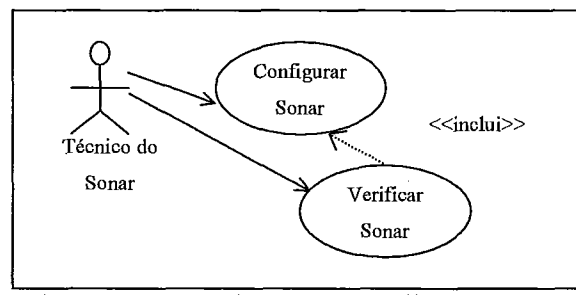

Figura 5.10 – Diagrama de Casos de uso

5.5 Conclusão do Capítulo

A DSP permite apoiar o entendimento da tarefa, disponibilizando conhecimento de tarefa independentemente do domínio em que está sendo aplicado, o que torna este conhecimento genérico e aplicável a diferentes sistemas. Com a descrição de diferentes DSP para problemas distintos, podemos gerar a descrição modularizada de várias tarefas, de forma que descrições futuras possam utilizar descrições já existentes. A integração da Descrição de Solução de Problemas em ADSOD, na Estação TABA, permite que tanto o conhecimento do domínio quanto o conhecimento de tarefa possam ser representados, permitindo ao desenvolvedor de software utilizar esses conhecimentos para melhor entender sobre o domínio em que estiver trabalhando e sobre o problema em questão que deve resolver.

As principais extensões ao modelo da Estação TABA necessárias para a representação do conhecimento de tarefa foram, no que se refere ao meta-ambiente, para permitir a definição das especificidades da tarefa, ou seja, dos elementos da Descrição de Solução de Problemas.

Neste capítulo apresentamos as extensões realizadas na arquitetura do Servidor de Conhecimento para contemplar esse objetivo. Foi ainda definido e construído um editor de tarefas para permitir que sejam cadastrados os elementos que compõem uma DSP, possibilitando, desta forma, a representação do conhecimento necessário para o entendimento do problema e de sua resolução. Finalmente, apresentamos um exemplo de como desenvolvedores de software podem utilizar a DSP para apoiar a modelagem de casos de uso. Esse exemplo, apesar de simples, serve para ilustrar como o conhecimento está sendo utilizado em Ambientes de Desenvolvimento de Software da Estação TABA e a capacidade da DSP em apoiar no entendimento do problema e na especificação da solução.

Capítulo 6

Conclusões e Perspectivas Futuras

No desenvolvimento de software, o primeiro passo é o entendimento do problema que deve ser resolvido pelo sistema. A principal causa para o desenvolvimento de produtos de software que não atendem às necessidades do cliente é a falta de entendimento, por parte dos desenvolvedores, de qual é o real objetivo do software e, consequentemente, quais são as tarefas que ele deve realizar e como estas devem ser realizadas.

Com o objetivo de solucionar esta questão, optamos por definir uma estrutura para a representação de conhecimento de tarefa, que apóia os engenheiros de software no entendimento do problema em ambientes de desenvolvimento de software a partir do entendimento das tarefas que compõem esse problema. A esta estrutura demos o nome de Descrição de Solução de Problemas (DSP). A DSP é capaz de disponibilizar, para o desenvolvedor, o conhecimento de tarefa necessário para orientá-lo ao longo de todo o processo de desenvolvimento. Este conhecimento é organizado de forma bem estruturada e fácil de ser entendida.

De acordo com a proposta deste trabalho, a DSP utiliza ontologias de tarefa para representação do conhecimento de tarefa. Ontologias de tarefa fornecem primitivas em termos do que precisamos descrever no contexto da solução do problema, através da especificação dos conceitos e das relações entre estes conceitos necessários para se executar a tarefa. Porém, se limita a permitir simular o processo de solução do problema conceitualmente, não demonstrando como o problema pode ser resolvido. Uma vez que consideramos que entender uma tarefa não envolve simplesmente entender sobre os conceitos utilizados na mesma, mas também entender como ela é realizada, combinamos a ontologia de tarefa e métodos de solução de problemas (MSP) em um único modelo para a representação do conhecimento de tarefa. Os MSP especificam a estratégia de inferência

responsável por determinar uma solução para a tarefa (SCHREIBER, G., WIELINGA, B. e BREUKER, J. (1993))

A DSP permite apoiar o entendimento do problema, disponibilizando conhecimento de tarefa que é independente do domínio em que está sendo aplicado, o que torna este conhecimento genérico e aplicável a diferentes sistemas em diferentes domínios. Com a descrição de diferentes DSP para problemas distintos, é possível formar uma biblioteca de tarefas que pode ser utilizada em diversos domínios e que é passível de evolução, acrescentando-se novas tarefas de acordo com a necessidade do sistema que está sendo desenvolvido. Além disso, a DSP facilita o entendimento do conhecimento de domínio, especificando como os conceitos do domínio podem ser utilizados na resolução das tarefas.

São, portanto, contribuições dessa tese:

i) A proposta de representação do conhecimento de tarefa, concretizada na forma de Descrição de Solução de Problemas;

ii) A utilização de ontologias de tarefa para organizar a descrição de tarefas, de forma que ontologias passam a ser utilizadas não apenas para auxiliar o entendimento do domínio mas, também, das tarefas que são utilizadas durante todo o desenvolvimento de software, permitindo a integração dos conhecimentos de tarefa e do domínio em uma abordagem uniforme e consistente.

iii) A especificação das tarefas de Configuração e Avaliação e suas respectivas subtarefas, segundo a DSP, e,

iv) A descrição de exemplos do auxílio provido pela DSP no desenvolvimento de sistemas, através da modelagem de casos de uso para as tarefas de Configuração e Avaliação.

Além disso, a definição da DSP e sua descrição no contexto da Estação TABA, implicou na:

i) Extensão do modelo TABA, de forma a permitir a definição da DSP;

ii) Definição e construção de uma ferramenta específica para a descrição do conhecimento de tarefa segundo a abordagem proposta, permitindo a efetiva utilização do conhecimento ao longo do processo de desenvolvimento de software. A ferramenta EDITAR possui dois modos de operação: modo construção e modo navegação. No modo construção, é possível descrever o conhecimento de tarefa, enquanto que no

modo navegação é permitido ao engenheiro de software navegar pelo conhecimento para ter uma compreensão da tarefa e dos componentes disponíveis.

iii) Extensão do auxílio do Servidor de Conhecimento, permitindo a navegação pelas tarefas do servidor, a composição de uma tarefa a partir de tarefas já cadastradas e a modelagem de qualquer tarefa que possa auxiliar no entendimento do problema sem a necessidade de se representar uma nova classe na arquitetura do servidor.

Muito trabalho nesta área ainda será necessário até que seja possível a construção de ambientes que apóiem completamente as atividades presentes no desenvolvimento de software considerando o conhecimento de tarefa. Várias outras pesquisas, ainda, serão necessárias, como por exemplo:

i) Disponibilização de outras linguagens de representação do conhecimento, além do Prolog, na Estação TABA que possam ser utilizadas na edição da Descrição de Solução de Problemas, permitindo, desta forma, que o engenheiro de software selecione aquela que se mostrar mais adequada para o problema em mãos.

ii) Extensão da ferramenta EDITAR para permitir a geração automática de axiomas epistemológicos e o desenho gráfico do modelo em LINGO. Esta ferramenta também deve ser capaz de gerar, automaticamente, a formalização das inferências a partir de sua descrição em linguagem estruturada;

iii) Especificação e construção de uma ferramenta que permita o mapeamento entre os papéis definidos no conhecimento de tarefa e o conhecimento do domínio correspondente. Com este mapeamento, tem-se um modelo completo do esqueleto da aplicação e ao propor esta solução preliminar, o engenheiro de software pode identificar que tipos de conhecimento estão faltando e, assim, tem um mecanismo eficiente para guiar a aquisição do conhecimento específico de uma aplicação, junto aos especialistas.

iv) Extensão da ferramenta EDITAR para permitir o apoio a definição do fluxo de controle da tarefa, no nível conceitual. Este apoio pode ser realizado através da disponibilização das subtarefas e conceitos que irão compor o fluxo, permitindo, desta forma, que o engenheiro de software defina, apenas, a estrutura do controle. Além disso, estaríamos garantindo a consistência das informações tratadas.

v) Definição e construção de uma ferramenta de auxílio à modelagem de casos de uso a partir do conhecimento de tarefa, representado conforme a DSP.

vi) Extensão da ferramenta EDITAR para permitir a descrição gráfica do fluxo de conhecimento na tarefa, de forma a auxiliar o engenheiro de software no entendimento da resolução do problema.

Consideramos ainda, neste trabalho, a aplicação de somente um método à tarefa, pois nosso objetivo era mostrar como, a partir de uma solução para o problema, podemos formar uma biblioteca de tarefas. Um projeto futuro é considerar a associação de mais de um MSP à uma tarefa e consequentemente a utilização de uma biblioteca de MSP, baseada em ontologias de método. Essa pesquisa implica em um estudo detalhado sobre o mapeamento entre os conceitos do domínio e seus respectivos papéis no conhecimento de tarefa, uma vez que a ontologia de método deve servir como uma interface para o engenheiro de conhecimento avaliar se o domínio de uma aplicação está de acordo com a estrutura de conhecimento usada pelo método.

Por fim, seria interessante a avaliação de experiências de uso da DSP em situações reais de desenvolvimento de software. O entendimento do um processo organizacional de uma empresa, por exemplo, pode ser mais facilmente realizado por intermédio do entendimento das tarefas que a compõem, ao passo que uma habilidade requerida ou possuída na organização pode ser definida como uma habilidade para se executar uma determinada tarefa.

Referências Bibliográficas

ABU-HANNA, A., JANSWEIJER, W., 1994, "Modeling Domain Knowledge Using Explicit Conceptualization". *IEEE Expert* (Oct), pp 53-63.

ALTMANN, J., POMBERGER, G., 1999, Cooperative Software Development: Concepts, Model and Tools, *Proceedings of the TOOLS-30 Conference*, Santa Barbara, USA, IEEE Society Press, pp 194 - 277.

BREUKER, J., 1994, "A Suite of Problem Types". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp.57-88.

BREUKER, J., VAN DE VELDE, W., 1994, *CommonKADS Library for Expertise Modelling*, Amsterdan, Holanda, IOS Press.

BRILL, D., 1993, "Loom Reference Manual, Version 2.0". University of Southern California.

BROWN, A., EARL, A., MCDERMID, J., 1992, *Software Engineering Environments: Automated Support for Software Engineering*, England, McGraw-Hill.

BYLANDER, T., CHANDRASEKARAN, B., 1987, Generic Tasks for Knowledge-Based Reasoning: the "Right" Level of Abstraction for Knowledge Acquisition. International Journal of Man-Machine Studies, 26, 231-243.

CHANDRASEKARAN, B., 1986, Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design, *IEEE Expert*, 1(3), pp. 23-30.

CHANDRASEKARAN, B., 1990, Design Problem Soving: A Task Analysis. *Research in Engineering Design.*

CHANDRASEKARAN, B., JONHSON, T.R., 1993, Generic Tasks and Task Structures: History, Critique and New Directions, *in Second Generation Expert Systems, David J.M., Krivine J.P., and Simmons R. (Eds.)*, Springer-Verlag, London, pp. 233-272.

CHANDRASEKARAN, B., JOSEPHSON, J. R., BENJAMINS, V., 1998, The Ontology of Tasks and Methods, In: *Proceedings of KAW'98, Eleventh Workshop on Knowledge Acquisition, Modeling and Management*, Alberta, Canada, Abril.

CLANCEY, W., J., 1985, Heuristic Classification. *Artificial Intelligence Journal* 27, 289-350.

COELHO, E., LAPALME, G., 1995, Extending Ontolingua for Representing Control Knowledge. *In IJCAI Workshop on Basic Ontological Issues in Knowledge Sharing.*

COELHO, E., LAPALME, G., 1996, Describing Reusable Problem Solving Methods with a Method Ontology. In Gaines, B. R. & Musen, M. A., editors, Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, pp. 3.1, Alberta, Canada. SRDG Publications, University of Calgary.

COELHO, E., LAPALME, G., PATEL, V., 1996, From KADS Models to Operational Problem Solving Methods: Perspectives in the Domain View. *In 6th Workshop on Knowledge Engineering: Methods & Languages.*

EMAM, K. E., DROUIN, J., MELO W., 1998, "SPICE – The Theory and Practice of Software Process Improvement and Capability Determination", *IEEE Computer Society Press*.

ERIKSSON, H. *et al.*, 1995, Task Modeling with Reusable Problem-Solving Methods. *Artificial Intelligence*, 79: 293-326.

FALASCONI, S., STEFANELLI, M., 1994, A Library of Medical Ontologies. *In Proceedings of ECAI94 Workshop on Comparison of Implemented Ontologies.* Amsterdam, The Nederlands, European Coordinating Committee for Artificial Intelligence (ECCAI): 81-92.

FALBO, R., 1998a, Integração de Conhecimento em um Ambiente de Desenvolvimento de Software, Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

FALBO, R.A., MENEZES, C.S., ROCHA, A.R.C., 1998b, "Integração de Conhecimento sobre Processos de Software em um Ambiente de Desenvolvimento". In: *Anais da IX Conferência Internacional de Tecnologia de Software (IX CITS)*, Curitiba, Paraná, Brasil, Junho.

FALBO, R., MENEZES, C., ROCHA, C., 1999a, "Using Knowledge Servers to Promote Knowledge Integration in Software Engineering Environments". In: *Proceedings of the 11th Software Engineering an Knowledge Engineering Conference*, pp. 170-175, Kaiserslautern, Alemanha, Jun.

FALBO, R., MENEZES, C., ROCHA, A. R., 1999b, "Assist-Pro: Um Assistente Inteligente para Apoiar a Definição de Processos de Software", In: *Anais do XIII Simpósio Brasileiro de Engenharia de Software*, pp. 147-162, Florianópolis, Brasil, Out.

FENSEL, D., STRAATMAN, R., 1998, The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency. *International Journal Human-Computer Studies*, V.48, pp. 181-215.

FIKES, R., FARQUHAR, A., 1999, "Distributed Repositories of Highly Expressive Reusable Ontologies", *IEEE Intelligent Systems & their applications*, v. 14, n. 2 (Mar/Apr), pp. 73-79.

FISCHER, G., MCCALL, R., OSTWALD, J. *et al.*, 1994, "Seeding, Evolutionary and Reseeding: Supporting the Incremental Development of Design Environments", *Proceedings of Conference in Human Factors in Computer Systems – CHI'94*, pp. 292-298, Boston, USA, Apr.

FISCHER, G., 1996, "Seeding, Evolutionary and Reseeding: capturing and evolving knowledge in domain-oriented design environments", In: Sutcliffe, A., Benyon, B. van Assche (eds) - IFIP 8. 1/13. *Joint Working Conference – Domain-Knowledge for Interactive System Design*, pp 1-16, Geneva, Switzerland, Maio.

FOURO, A. M., 2002, "Apoio à Construção de Bases de Dados de Pesquisa em Ambientes de Desenvolvimento de Software Orientados a Domínio", Tese de M. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

GALOTTA, C. G. M., 2000, "Netuno: Um Ambiente de Desenvolvimento de Software Orientado ao Domínio de Acústica Submarina", Tese de MSc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

GARG, P., JAZAYERI, M., 1995, "Introduction". In: Garg, P., Jazayeri, M. (eds), *Process-Centered Software Engineering Environments*, capítulo 1, IEEE Computer Society Press.

GENNARI, J. H., TU, S. W., ROTHENFLUH, T. E., MUSEN, M. A., 1994, Mapping Domains to Methods in Support of Reuse. *International Journal of Human and Computer Studies*, 41: 399-424.

GÓMEZ-PÉREZ, A, 1995, "Some Ideas and Examples to Evaluate Ontologies"; In: *Proceedings of 11th Conference in Artificial Intelligence for Applications*, pp 299-305, Los Angeles, EUA, Feb.

GRUBER, T. R., 1992, Ontolingua: A mechanism to support portable ontologies, version 3. 0. *Technical Report, Knowledge Systems Laboratory*, Stanford University, USA

GRUBER, T., 1993, "A Translation Approach to Portable Ontology Specifications"; *Knowledge Acquisition,* 5(2), pp 199-220.

GRUBER, T. R., 1995; "Toward Principles for the Design of Ontologies used for Knowledge Sharing", *International Journal Human-Computer Studies*, No 43, pp 907-928.

GUARINO, N., 1995, "Formal ontology, conceptual analysis, and Knowledge representation", *International Journal of Human-Computer Studies*; vol. 43, n. 5/6, pp 625-639.

GUARINO, N., GIARRETA, P., 1995, "Ontologies and Knowledge Bases - Towards a Terminological Clarification", In: Mars, N. J., (ed.), *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, IOS Press, Amsterdan, pp 25-32.

GUARINO, N., 1997, "Understanding, Building, and using Ontologies: A comentary to 'Using Explicit Ontologies in KBS Development', by Heijst, Screiber e Wielenga", *International Journal Human-Computer Studies*, vol 46, No 2/3, pp 293-310.

GUARINO, N., 1998, "Formal Ontology abd Information System", In: Guarino, N. ed) *Formal Ontology in Information System,* pp. 3-15, IOS Press.

HORROCKS, I., FENSEL, D., HARMELEN, F., DECKER, S., ERDMANN, M., KLEIN, M., 2000, OIL in a Nutshell. *Proceedings of the ECAI'00 Workshop on Application of Ontologies and PSMs*, Berlin, Germany.

HWANG, C. H., 1999 Incompletely and Imprecisely Speaking: Using Dynamic Ontologies for Representing and Retrieving Information. *Austin: Microeletronics and Computer Technology Corporation*, Texas, 1999. (Technical Report MCC-INSL-043-99)

IKEDA, M., SETA, K., MIZOGUCHI, R., 1997, *Task Ontology Makes It Easier To Use Authoring Tools*, IJCAI-97, Nagoya Japan, pp.342-347.

IKEDA, M., SETA, K., KAKUSHO, O., MIZOGUCHI, R., 1998, An Ontology for Building a Conceptual Problem Solving Model. *Proceedings of 13th European Conference on Artificial Intelligence*, ECAI'98, Brighton, England, Agosto.

JACOBSON, I., CHRISTERSON, M., JOHNSON, P., OVERGAARD. G., 1992, "Object-Oriented Software Engineering: A Use Case Driven Approach", capítulo 11, p 291. Addison-Wesley.

LUKE S., HEFLIN J., 2000, "SHOE 1.01. Proposed Specification", SHOE Project, Fevereiro.

MACHADO, L. F. C., 2000, "Modelo para Definição, Especialização e Instanciação de Processos de Software", Tese de M. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

MARCUS, S., 1988. SALT: A Knowledge-Acquisition Tool for Propose-and-Revise Systems. *In Marcus, S., editor, Automating Knowledge Acquisition for Expert Systems,* pp. 81--123. Boston: Kluwer Academic Publishers.

MCDERMOTT, J., 1988, Toward a taxonomoy of problem solving methods. *In Marcus, S. (Ed.)., Automating Knowledge Acquisition for Expert Systems. Kluwer Academic Publishers*.

MINOURA, T., PARGAONKAR, S., REHFUSS, K., 1993, Structural Active Object Systems for Simulation. *In Proceedings of OOPSLA'93*, pp. 338-355, ACM, Outubro.

MOTTA, E., ZDRAHAL, Z., 1998, A principled approach to the construction of a task-specific library of problem solving components. *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, Canada.

MOTTA, E., LU, W., 2000, A Library of Components for Classification Problem Solving. *Pacific Rim Knowledge Acquisition Workshop*, Sydney, Australia, Dezembro 11-13.

MOURA, L. M. V., ROCHA, A. R. C., 1992, *Ambientes de Desenvolvimento de Software,* Publicações Técnicas COPPE/UFRJ, ES-271/92, Rio de Janeiro, Brasil.

MIZOGUCHI, R., *et al.*, 1992, Task ontology and its use in a task analysis interview system -- Two-level mediating representation in MULTIS --, *Proceedings of the JKAW'92*, pp.185-198.

MIZOGUCHI, R., 1994, Knowledge Reuse and Ontology, In: Fuchi, K., Tokoi, T. (eds) *Knowledge Building and Knowledge Sharing*, Ohmsha, ltd e IOS Press pp. 165-174.

MIZOGUCHI, R., VANWELKENHUYSEN, J., IKEDA, M., 1995, Task Ontology for Reuse of Problem Solving Knowledge. N.J.I Mars, Ed. IOS Press 1995.

MIZOGUCHI, R., SINITSA, K., IKEDA, M., 1996. Task Ontology Design for Intelligent Educational/Training Systems. *ITS'96 Workshop on Architectures and Methods for Designing Cost-Effective and Reusable ITSs*, Montreal.

MIZOGUCHI, R., IKEDA, M., SINITSA, K., 1997. Roles of Shared Ontology in AI-ED Research. In de Boulay, B. and Mizoguchi, R., editors, *Artificial Intelligence in Education AI-ED 97*, pp. 537-544, Kobe, Japan. IOS Press.

NBR ISO/IEC 12207, 1998, Tecnologia de Informação – Processos de Ciclo de Vida de Software, Associação Brasileira de Normas Técnicas, Rio De Janeiro, Brasil.

OLIVEIRA, K. M., 1996, "Um Ambiente de Desenvolvimento de Software Orientado ao Domínio", Monografia de Tópicos Especiais em Engenharia de Software – Análise de Domínio, COPPE/UFRJ, Rio de Janeiro, Brasil.

OLIVEIRA, K. M., 1999a, "Modelo para Construção de Ambientes de Desenvolvimento de Software orientados a Domínio", Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

OLIVEIRA, K. M., ROCHA, A. R., TRAVASSOS, G. H., 1999b, "A Domain-Oriented Software Development Environment for Cardiology", In: *Proceedings of America Medical Informatics Association Conference – AMIA*, Washington, D.C., Nov. (In press)

OLIVEIRA, K. M., ROCHA, A. R., TRAVASSOS, G. H. *et al.*, 1999c, "Using Domain-Knowledge in Software Development Environments", In: *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*, pp. 180-187, Kaiserlauther, Alemanha, Jun.

OLIVEIRA, K. M., ROCHA, A. R., TRAVASSOS, G. H. *et al.*, 1999d, "O uso da Teoria do Domínio no Processo de Desenvolvimento de Software", In: *Anais da X Conferencia Internacional de Tecnologia de Software*, pp 223-235, Curitiba, Brasil, Mai.

OLIVEIRA, K. M., GALOTTA, C., ROCHA, A. R., *et al.*, 2000, "Construção de Ambientes de Desenvolvimento de Software Orientados a Domínio", In: *Jornadas*

Iberoamericanas de Ingenier a de Requisitos y Ambientes de Software IDEAS 2000 – Cancún, México, Abr.

PAULK, M. C., WEBER, C. V., CURTIS, B. *et al.*, 1995, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Carnegie Mellon University, Software Engineering Institute, Addison-Wesley Longman Inc.

PENEDO, M. H., 1993, *Towards understanding Software Engineering Environment,* Techinical Report IMPSEE-TRW-93-003, Aug.

PFLEEGER, S. L., 2001, *Software Engineering – Theory and Practice*, 2nd ed. New-Jersey, Prentice-Hall Inc.

ROCHA, A.R.C., AGUIAR, T.C., BLASCHEK, J.R.S, 1987, Ambientes para Desenvolvimento de Software: Definição de Termos. Relatório Técnico do Programa de Engenharia de Sistemas e Computação ES-137/87, COPPE/UFRJ.

ROCHA, A. R. C., AGUIAR, T. C., SOUZA, J. M., 1990, "Taba: A Heuristic Workstation for Software development", In: *Proceedings of COMPEURO 90*, Tel Aviv, Israel, Maio.

SCHREIBER, G., WIELINGA, B., BREUKER, J., 1993, *KADS: A Principled Approach to Knowledge-Based System Development*, Academic Press, London.

SCHREIBER, G., WIELINGA, B., AKKERMANS, H. *et al*, 1994, "CML: The CommonKADS Conceptual Modelling Language", In: *Proceedings of 8th European Knowledge Acquisition Workshop/EKAW94 - Lecture Notes in Artificial Intelligence 867*, L. Steels, G. Schreiber e W. Van de Velde (eds.), pp 1-19.

SELIC, B., GULLEKSON, G., WARD, P., 1994, "Real-Time Object-Oriented Modeling". John Wiley and Sons.

SHAW, M., DELINE, R., KLIEN, D., ROSS, T., YOUNG, D., ZELESNIK, G., 1995, Abstractions of Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* 21, 4, pp.14-335, Abril.

SPEEL, P., ABEN, M., 1997, Applying a Library of Problem-Solving Methods on a Real-Life Task. *International Journal of Human and Computer Studies*, 46: 627-652.

STEELS, L., 1990, "Components of Expertise", *AI Magazine*, Summer.

SWARTOUT, W., TATE, A., 1999, "Ontologies – Guest Editors Introduction". *IEEE Intelligent Systems & their applications*, vol. 14, n. 1 (Jan/Fev), pp. 18-19.

TAUTZ, C., ALTHOFF, H., 2000, A Case Study on Engineering Ontologies and Related Processes for Sharing Software Engineering Experience. *SEKE 2000*, pp.318-327.

TIJERINO, A., Y., *et al.*, 1993, MULTIS II : Enabling End-Users to Design Problem-Solving Engines Via Two-Level Task Ontologies, *Proceedings of EKAW '93*, pp. 340-359.

TRAVASSOS, G. H., 1994, O Modelo de Integração de Ferramentas da Estação TABA, Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

USCHOLD, T., 1996, "Building Ontologies: Towards a Unified Methodology"; In: *Proceedings of 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems; Cambridge*, UK, Dec. VAN HEIJST, G., 1995, *The Role of Ontologies in Knowlege Engineering*, PhD. Thesis, Universidade de Amsterdan, Holanda.

USCHOLD, M., GRUNINGER, M., 1996, "Ontologies: principles, methods and applications", *The Knowledge Engineering Review,* Vol 11:2, pp 93-136.

VALENTE, A., BREUKER, J., VAN DE VELDE, W., 1994, "The CommonKADS Expertise Modeling Library". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp. 31-56.

VALENTE, A., 1995, *Legal Knowledge Engineering - A modelling Approach*, Amsterdan, Holanda, IOS Press.

VAN HEIJST, G. VAN, SCHREIBER, A TH., WIELENGA, B. J., 1997, "Using Explicit Ontologies in KBS Development", *International Journal of Human-Computer Studies*, vol 45, No 2/3; pp 183-292.

VILLELA, K., TRAVASSOS, G. H., 1997, *Requisitos de um Ambiente de Desenvolvimento de Software para o Domínio de Cardiologia*, Relatório Técnico, Projeto CNPq, Rio de Janeiro, Brasil.

VILLELA, K., ZLOT, F., SANTOS, G., 2001, "Knowledge Management in Software Development Environments", 14th International Conference on Software & Systems Engineering and their Applications – ICSSEA 01; Paris, France, Dezembro.

WERNECK, V. M., 1995, "Ambiente para Desenvolvimento de Software Baseado em Conhecimento", Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

WERNER, C. M. L., TRAVASSOS, G. H., DA ROCHA, A. C. *et al.,* 1997, "Memphis: A Reuse Based O. O. Software Development Environment", In: *Proceedings of TOOLS,* Beijing, China, Sep.

WIELEMAKER, J., 1998, *SWI-Prolog 3.2.8 - Reference Manual.* (ftp://swi.psy.uva.nl/pub/SWI-Prolog).

YEN, J., LEE, J., 1993, A Task-Based Methodology for Specifying Expert Systems. *Proceedings of the IEEE Iintenational Conference.*

ZLOT, F., SANTOS, G., 1999, "Definição e Instanciação de Ambientes na Estação TABA". Projeto Final de Curso, Universidade Federal do Rio de Janeiro, Brasil.

Anexo 1

Descrição de Solução de Problemas da tarefa Avaliação

Descreveremos, neste anexo, a Descrição de Solução de Problemas para a tarefa de Avaliação, por ser esta um problema pertencente a diversos domínios e que freqüentemente aparece como sub-parte de outros problemas. Será apresentado, ainda, um exemplo de seu auxílio na modelagem de um caso de uso.

A1.1 DSP de Avaliação

A tarefa de Avaliação tem por objetivo definir uma decisão sobre um determinado caso. Para isso, são comparadas as informações provenientes do caso em questão com as de um problema específico. As informações sobre o problema são fundamentais para permitir determinar uma decisão sobre o caso. A tarefa pode, portanto, ser resolvida simplesmente obtendo-se a decisão a partir da comparação entre a descrição do caso e o problema específico. Porém, a descrição do caso e do problema estão, provavelmente, em níveis de abstração distintos. Portanto, consideramos necessária a aplicação de um método para normalizar esses níveis de abstração antes de realizarmos a comparação descrita anteriormente. Supomos a aplicação do método definido pela biblioteca CommonKads (BREUKER e VAN DER VELDE, 1994), que propõe a decomposição da tarefa de Avaliação em 3 subtarefas mais simples. Esta decomposição permitirá normalizar os níveis de abstração da descrição do caso e do problema específico, de forma a possibilitar a comparação entre eles. A seguir, é apresentada a descrição da DSP para a tarefa de Avaliação (tabela A1.1).

Tabela A1.1 – DSP da tarefa Avaliação

Nível Verbal:

Descrição Textual: A tarefa de *Avaliação* tem por objetivo analisar os fatos sobre um determinado caso. Para isso, são comparadas as informações provenientes do caso em questão com as de um problema específico. As regras de negócio descritas no problema permitem a definição de uma decisão sobre o caso avaliado.

Subtarefas: *Especificação* – define regras à partir dos fatos descritos para um determinado problema. A tarefa também especifica o item referenciado nas regras.

Abstração – identifica os fatos sobre um caso que são relevantes à um determinado problema, representado aqui apenas por uma de suas instâncias.

Comparação – compara dois fatos semelhantes e determina uma decisão à partir das regras descritas no problema.

Descrição do Controle: O primeiro passo para resolver a tarefa de avaliação é diminuir o nível de abstração do problema, ou seja, do modelo do sistema. Isto é realizado pela subtarefa *Especificação*, que define regras de negócio a partir do modelo do sistema, chamada de norma. É estabelecido, ainda, o termo utilizado nas regras especificadas. Posteriormente, são identificados os fatos sobre a descrição do caso que são relevantes a avaliação. Para isto, a subtarefa *Abstração* define uma descrição abstrata do caso levando em consideração o termo que foi especificado, aumentando, assim, o nível de abstração da descrição do caso. Uma vez trabalhando no mesmo nível de abstração, a subtarefa *Comparação* confronta a descrição abstrata do caso com as normas específicas e determina uma decisão sobre o caso em questão. Caso as normas estabelecidas sejam insuficientes para que seja definida uma conclusão sobre o caso, todos os passos são repetidos até que uma decisão possa ser determinada.

Nível Conceitual:

Entrada: *Descrição do Caso* - representa todos os fatos que são conhecidos sobre o objeto do domínio que está sendo avaliado.

Modelo do Sistema – define toda a informação sobre o alvo da avaliação caso o alvo atendesse a todas as expectativas para um objeto do domínio em particular.

Saída: *Decisão* – representa a conclusão sobre o caso avaliado.

Descrição Controle (Conceitual):

Enquanto uma decisão não for determinada faça
    Especificar as regras do Modelo do Sistema
    Abstrair um fato da Descrição do Caso
    Comparar as informações para obter uma Decisão
Fim-Enquanto

Nível Formal:

Descrição Controle (Formal):

avaliação(Decisão) :- e*specificação*(Fato,Valor,Decisão), abstração(Fato,Valor), *comparação*(Fato,Decisão).

As subtarefas geradas a partir da aplicação do método à tarefa de Avaliação, são tarefas independentes e, portanto, também possuem um método associado a elas. Desta forma, as subtarefas também precisam ser descritas na DSP. Consideramos que as subtarefas Especificação, Abstração e Comparação são tarefas elementares, uma vez que

podem ser simplesmente compreendidas pelo engenheiro de software. Apresentaremos a seguir as DSP destas tarefas (tabela A1.2, tabela A1.3, tabela A1.4).

Tabela A1.2 – DSP da tarefa Especificação

Nível Verbal:
Descrição Textual: A tarefa de *Especificação* tem por objetivo selecionar regras à partir dos fatos descritos para um determinado problema. A tarefa estabelece, ainda, o item utilizado nas regras.
Nível Conceitual:
Entrada: *Modelo do Sistema* – representa toda a informação sobre o alvo do problema. **Saída:** *Norma* – representa as regras de negócio obtidas a partir do *Modelo do Sistema.* *Termo* – define o item referenciado na norma.
Nível Formal:
Axiomas (1ª. Ordem): *//Toda norma é parte do modelo-do-sistema* 1) *(∀f)(∀v) (∀d) (norma(f,v,d) →modelo-do-sistema(f,v,d))* *//Todo termo é parte do modelo-do-sistema* 2) *(∀f) (termo(f) →modelo-do-sistema(f,_,_))* **Axiomas (Prolog):** 1) *modelo-do-sistema (F,V,D):- norma(F,V,D).* 2) *modelo-do-sistema (F,_,_):- termo(F).* **Inferência:** *especificação*(Fato, Valor, Decisão):- *modelo-do-sistema*(Fato, Valor, Decisão). *norma*(Fato, Valor, Decisão):- *especificação*(Fato, Valor, Decisão). *termo*(Fato):- *especificação*(Fato, _, _).

Tabela A1.3 – DSP da tarefa Abstração

Nível Verbal:
Descrição Textual: A tarefa de *Abstração* tem por objetivo descrever os fatos sobre um objeto do domínio que são relevantes à um determinado problema. O problema é representado aqui apenas por uma de suas instâncias (item).
Nível Conceitual:
Entrada: *Descrição do Caso* – representa todos os fatos que são conhecidos sobre um determinado objeto do domínio. *Termo* – define uma instância do problema em questão **Saída:** *Descrição Abstrata do Caso* – representa apenas os fatos sobre o objeto do domínio que são relevantes ao problema.
Nível Formal:
Axiomas (1ª. Ordem): *// Toda descrição-abstrata-do-caso é uma descrição-do-caso* 1) *(∀f)(∀v) (descrição-abstrata-do-caso(f,v) →descrição-do-caso(f,v))* **Axiomas (Prolog):** *(descrição-do-caso (F,V):- descrição-abstrata-do-caso (F,V))* **Inferência:** *abstração*(Fato,Valor):- *descrição-do-caso*(Fato, Valor), *termo*(Fato). *descrição-abstrata-caso*(Fato, Valor):- *abstração*(Fato, Valor).

Tabela A1.4 – DSP da tarefa Comparação

Nível Verbal:
Descrição Textual: A tarefa de *Comparação* tem por objetivo confrontar dois fatos semelhantes e estabelecer uma decisão à partir de regras definidas.
Nível Conceitual:
Entrada: *Norma* – representa as regras de negócio obtidas a partir das informação sobre um problema. *Descrição Abstrata do Caso* – representa todos os fatos sobre um objeto do domínio que são relevantes ao problema **Saída:** *Decisão* – representa a decisão tomada sobre o caso.
Nível Formal:
Inferência: *comparação*(Fato, Decisão):-*descrição-abstrata-do-caso*(Fato, Valor), *norma*(Fato, Valor, Decisão), *decisão*(Decisão):- *comparação*(_, Decisão).

Para que seja possível visualizar a solução completa da tarefa de Avaliação, apresentamos, na figura A1.1, a Estrutura de Inferência (SCHREIBER *et al.*, 1993) da tarefa, que permite uma visão geral do método de solução de problemas e do conhecimento usado pelas subtarefas. Através da estrutura, podemos verificar que a subtarefa *Comparação* somente define uma decisão sobre o caso, após as subtarefas *Abstração* e *Especificação* normalizarem os níveis de abstração do caso e do modelo do sistema, respectivamente.

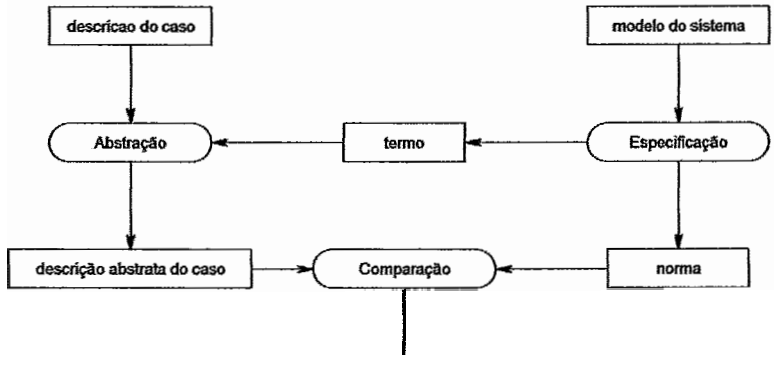

Figura A1.1 – Estrutura de Inferência da DSP de Avaliação

A1.2 Auxílio na Modelagem de Caso de Uso

Para ilustrar como a DSP de Avaliação pode auxiliar na modelagem de um caso de uso, consideremos, por exemplo, um situação de uso no ambiente CORDIS, desenvolvido para o domínio de cardiologia (OLIVEIRA,1999). Um sistema em cardiologia possui, dentre as suas diversas funcionalidades, a tarefa de avaliar os resultados de um eletrocardiograma (ECG). O eletrocardiograma de um paciente possui diversas características (Ex: Duração da Onda P, Amplitude Onda T, Segmento ST), que precisam ser comparadas com alterações cardíacas típicas, como um evento coronariano agudo, para que seja possível se determinar uma conclusão sobre o paciente (Ex: Supra desnivelamento do segmento ST, Inversão da Onda T).

Um dos problemas a ser automatizado nesse domínio é, portanto, a avaliação de ECG. O desenvolvedor pode entender sobre esta tarefa consultando a DSP descrita sobre avaliação e, a partir das informações obtidas, identificar que conceitos são necessários para a realização da tarefa. O ECG é um sinal baseado em sinal elétrico pertencente como instância da ontologia de domínio. Suas características podem ser obtidas através das alterações no ritmo ou particularizadas como propriedades do ECG. As alterações cardíacas as serem analisadas devem ser elicitadas para a aplicação específica a ser desenvolvida. A seguir, descrevemos o caso de uso para Avaliar o ECG à partir da DSP de Avaliação e do domínio de cardiologia, descrito no ADSOD CORDIS.

i) **Nome e Descrição do Caso de Uso:** Na tabela A1.6 é apresentada a descrição do caso de uso Avaliar ECG. No lado esquerdo da tabela, é apresentada a descrição textual da DSP Avaliação, utilizada como base para a descrição do caso de uso.

Tabela A1.6: Descrição Textual da DSP Avaliação / Descrição do Caso Avaliar ECG.

***DSP Avaliação**: A tarefa de Avaliação tem por objetivo analisar os fatos sobre um determinado caso. Para isso, são comparadas as informações provenientes do caso em questão com as de um problema específico. As regras de negócio descritas no problema permitem a determinação de uma decisão sobre o caso avaliado.*	***Avaliar ECG:** função realizada para analisar as características de um eletrocardiagrama, comparando-os com os de uma alteração cardíaca típica, para determinar uma conclusão sobre o ECG avaliado.*

ii) **Fluxo de eventos principal:** a tabela A1.7 especifica o fluxo de eventos principal do caso de uso Avaliar ECG, onde consideramos que o *eletrocardiografista* é o ator

responsável por iniciar a função do sistema. A definição do ator foi baseada no conhecimento socio-organizacional definido nos ADSOrg (vide seção 2.2.4) e nas descrições da DSP Avaliação. Na tabela, é apresentada primeiramente a descrição do controle da DSP, para em seguida ser descrito o fluxo de eventos do caso de uso.

Tabela A1.7: Descrição do Controle da DSP Avaliação / Fluxo de eventos principal do Caso de Uso Avaliar ECG.

Controle da DSP:	***Fluxo de Eventos:***
	1) O eletrocardiografista solicita o início da avaliação do ECG
Enquanto *uma decisão não for determinada* ***faça***	*2)* ***Enquanto*** *uma conclusão não for determinada* ***faça***
Especificar as regras do Modelo do Sistema	*2.1) O sistema* ***especifica*** *as* alterações cardíacas típicas.
Abstrair um fato da Descrição do Caso	*2.2)O sistema* ***abstrai*** *uma característica do ECG.*
Comparar as informações para obter uma Decisão	*2.3) O sistema* ***compara*** *as informações para obter uma conclusão .*
Fim-Enquanto	***Fim-Enquanto***
	3) O sistema informa a conclusão determinada

iii) **Casos de uso incluídos:** para auxiliar na definição dos casos de uso incluídos, o engenheiro de software pode fazer uso da descrição das subtarefas de Avaliação (*Especificação*, *Abstração* e *Comparação*) definida em sua DSP, identificando aquelas para as quais ele precisou modelar um caso de uso.

i) **Diagrama de casos de uso:** representação gráfica que será formada, no sistema que estamos representando, pelo ator *eletrocardiografista*, pelo caso de uso *Avaliar ECG* e por seus possíveis casos de uso incluídos. Todos esses elementos foram determinados à partir da DSP Avaliação.

Anexo 2

LINGO: LINguagem Gráfica para descrever Ontologias

O presente anexo tem por objetivo apresentar as bases da linguagem gráfica para descrever ontologias LINGO, definida por FALBO (1998a). LINGO é uma linguagem gráfica capaz de representar a conceituação de um domínio. Neste sentido, em sua forma mais simples, LINGO possui primitivas para representar apenas conceitos e relações, cujas notações estão apresentadas na figura A2.1.

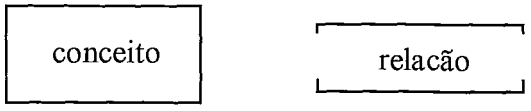

Figura A2.1 - Notações utilizadas para conceitos e relações (FALBO, 1998a)

A figura A2.2 mostra um exemplo de uma relação binária entre os conceitos de atividade e produto, em um contexto de manufatura, indicando que atividades geram produtos. Cardinalidades são usadas para mostrar quantas instâncias de um conceito podem participar da relação. Uma vez que a cardinalidade (0..n) não impõe nenhum axioma, ela não é representada.

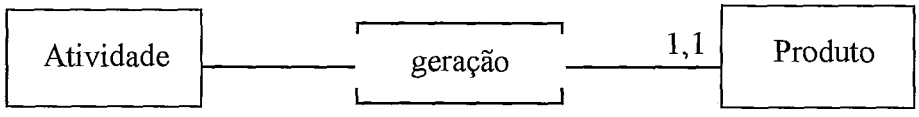

Figura A2.2 - Exemplo de uma rela binária entre conceitos (FALBO, 1998a)

O uso da cardinalidade (1,1) na figura A2.2, por exemplo, induz os seguintes axiomas:

$$(p)\ (produto(p)\ (a)\ (gera\ (p,a))$$
$$(p,a_1,a_2)\ (gera\ (p,a_1)\ gera\ (p,a_2)\ a_1 = a_2)$$

Axiomas da forma do primeiro axioma são um reflexo da cardinalidade mínima 1. Axiomas da forma do segundo axioma, por outro lado, são reflexo da cardinalidade máxima 1.

As relações não estão restritas a relações binárias. Relações de ordem superior, tais como relações ternárias, são igualmente válidas. Além disso, relações entre instâncias de um mesmo conceito também são válidas. Neste caso, sugere-se o uso de papéis, como mostra a figura A2.3.

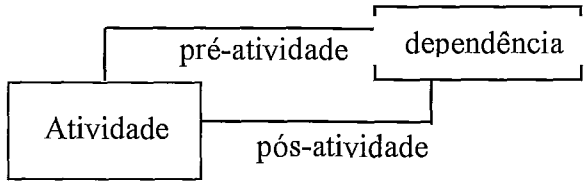

Figura A2.3 - Uso de papéis (FALBO, 1998a)

Neste caso, o seguinte axioma pode ser derivado:

$$(a_1,a_2)\ (preatividade(a_1,a_2)\ posatividade(a_2,a_1))$$

Alguns tipos de associações têm uma semântica forte e, na verdade, escondem por detrás uma ontologia genérica, tal como a relação de composição. Para estes tipos de associações é proposta uma notação especializada. De fato, esta é a principal característica de LINGO: qualquer notação proposta além das notações básicas para conceitos e relações visa incorporar uma teoria. Uma vez que a teoria é incorporada, axiomas podem ser automaticamente gerados. Estes axiomas dizem respeito simplesmente à estruturação dos conceitos e são ditos *axiomas epistemológicos*. Assim, ainda que LINGO seja uma representação de nível epistemológico, ela incorpora um mecanismo de inclusão de teorias no nível ontológico.

Na versão corrente de LINGO, apenas dois tipos especiais de associações foram consideradas: a relação todo-parte e a associação sub-tipo-de. Quaisquer que sejam os

elementos envolvidos em uma relação todo-parte, há um conjunto de propriedades que são sempre válidas, entre elas:

(x,y) (partede(x,y) partede(y,x))

(x,y,z) (partede(x,y) partede(y,z) partede(x,z))

(x,y) (disjunto(x,y) (z) (partede(z,x) partede(z,y)))

(x) (at mico(x) (y) (partede(y,x))

Ao utilizar uma relação todo-parte, estamos importando e aplicando uma teoria abstrata de composição de elementos ao conteúdo da ontologia em desenvolvimento. A figura A2.4 mostra a notação empregada para representar relações de composição.

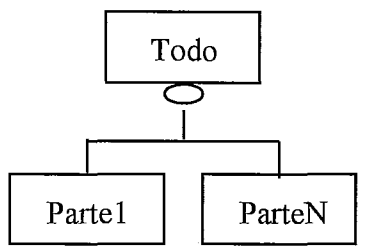

Figura A2.4 - Nota para composição (FALBO, 1998a)

Uma vez que a relação de composição é uma relação entre instâncias de conceitos, cardinalidades devem ser representadas. Para representar este tipo de relação, utilizamos uma linha cheia com uma pequena elipse junto ao todo.

Um tipo de associação que também merece atenção especial é a associação "sub-tipo-de". Ao construirmos uma taxonomia de conceitos, ocorre o compromisso implícito com uma ontologia de sub-tipos, que diz, entre outras coisas, que:

(x, y) (subtipo(x, y) supertipo(y, x))

Para representar este tipo de associação, é usada a notação mostrada na figura A2.5.

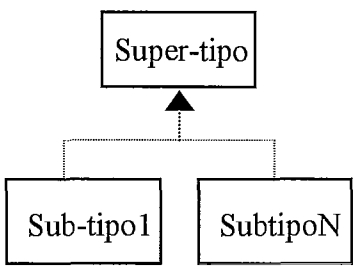

Figura A2.5 - Nota para hierarquia de conceitos (FALBO, 1998a)

Uma vez que a associação sub-tipo-de se dá entre conceitos e não entre suas instâncias, utiliza-se uma linha pontilhada para representá-la, com uma seta apontando para o super-tipo. LINGO também pode ser usada para expressar condicionantes entre relações.

Por exemplo: dados três conceitos *A*, *B* e *C* e duas relações *r1* e *r2*, entre instâncias dos conceitos *A* e instâncias dos conceitos *B* e *C*, respectivamente, queremos expressar uma condicionante entre as relações, dizendo que se uma instância de *A* está relacionada com uma instância de *B*, então ela não pode estar relacionada com uma instância de *C*. Para tal, a seguinte notação foi proposta:

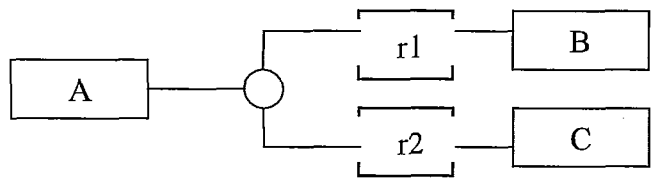

Figura A2.6 - Condicionante ou-exclusivo entre rela es (FALBO, 1998a)

Ao utilizarmos a notação da figura A2.6, assume-se que:

(a) ((a A) ((b) (b B) r1(a,b)) ((c) (c C) r2(a,c)))

(a) ((a A) ((c) (c C) r2(a,c)) ((b) (b B) r1(a,b)))

Analogamente, a notação da figura A2.7 estabelece uma condicionante de obrigatoriedade entre relações: se uma instância de *A* está relacionada com uma instância de *B*, então ela obrigatoriamente tem de estar relacionada com uma instância de *C*:

(a) ((a A) ((b) (b B) r1(a,b)) ((c) (c C) r2(a,c)))

(a) ((a A) ((c) (c C) r2(a,c)) ((b) (b B) r1(a,b)))

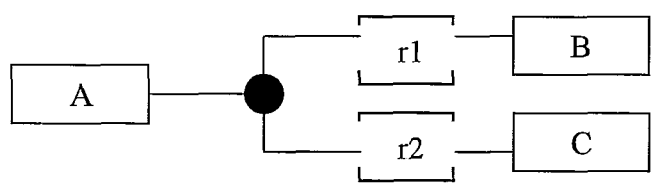

Figura A2.7 - Condicionante de obrigatoriedade entre relações (FALBO, 1998a)

Notações para outros tipos de condicionantes podem ser incorporadas a LINGO, que deve ser considerada uma linguagem aberta, suscetível a extensões para capturar outras necessidades da modelagem no nível ontológico.