Apoio Baseado em Conhecimento à Integração de Processo em ODE

Fabiano Borges Ruy, Gleidson Bertollo, Ricardo de Almeida Falbo
Universidade Federal do Espírito Santo
Av. Fernando Ferrari, 29060-900
Vitória – ES – Brasil
+55 (27) 3335-2167
{fruy, gbertollo, falbo}@inf.ufes.br

ABSTRACT
Process integration in Software Engineering Environments (SEE) is very important to allow appropriated tool control. In this paper we present a knowledge-based approach to improve process integration in ODE, an ontology-based SEE.

Keywords
Software Engineering Environment, Software Process, Knowledge Based Support.

1. INTRODUÇÃO
Ambientes de Desenvolvimento de Software (ADSs) buscam combinar técnicas, métodos e ferramentas para apoiar o engenheiro de software na construção de produtos de software, abrangendo todas as atividades do processo de software, tais como gerência, desenvolvimento e controle da qualidade [1].

O desenvolvimento de produtos de software de qualidade, dentro do cronograma e considerando os custos planejados sempre foi um desafio para as organizações produtoras de software. Entretanto, a qualidade de um produto de software depende fortemente da qualidade do processo de software utilizado em seu desenvolvimento [2]. No contexto de ADSs, esta constatação levou à preocupação com a integração de processo, isto é, uma ligação explícita entre as ferramentas do ADS e o processo de software [3], dando origem aos ADSs Centrados em Processo.

Devido à complexidade envolvida não somente na integração de processos de software, mas em diversas tarefas apoiadas por ADSs e como forma de aperfeiçoar suas funcionalidades, tornando-os mais eficazes, algumas tecnologias têm sido adotadas na construção de tais ambientes, dentre elas o uso de técnicas baseadas em conhecimento. Assim, é importante que um ADS disponibilize uma infra-estrutura para integração de conhecimento, de modo que suas ferramentas possam oferecer suporte baseado em conhecimento aos desenvolvedores.

Este trabalho apresenta como a integração de processo é apoiada por uma infra-estrutura de suporte à integração de conhecimento em ODE (*Ontology-based software Development Environment*) [4], um ADS Centrado em Processo, construído tendo por base ontologias. A integração de processo provê formas mais efetivas de descrever e implementar processos de desenvolvimento de software, abordando uma definição explícita do processo e o controle de sua execução. A infra-estrutura de suporte à integração de conhecimento, por sua vez, permite manipular bases de conhecimento e realizar inferências usando uma máquina Prolog. O objetivo é apresentar como essa infra-estrutura suporta um apoio baseado em conhecimento à integração de processo em ODE.

O artigo encontra-se estruturado da seguinte forma: a seção 2 apresenta o ADS ODE e sua infra-estrutura de suporte à integração de conhecimento. Na seção 3, é discutida a integração de processo em ODE, considerando definição e controle de processos. Trabalhos correlatos e conclusões são discutidos nas seções 4 e 5, respectivamente.

2. SUPORTE À INTEGRAÇÃO DE CONHECIMENTO EM ODE
Construir um ADS que integre uma grande variedade de ferramentas é uma tarefa complexa. Quando isoladas, as ferramentas trabalham independentemente umas das outras, possuindo seus próprios conceitos, dados, representações e formas de atender aos propósitos para os quais foram criadas. Trabalhando em conjunto, porém, o grau de efetividade torna-se expressivamente mais elevado.

Assim, a integração passa a ser um fator de fundamental importância em ADSs. Integração demanda representação consistente das informações, interfaces padronizadas, significados homogêneos para a comunicação entre desenvolvedores e ferramentas e uma efetiva abordagem que permita aos ADSs alternar entre várias plataformas [5]. A integração de ferramentas em ADSs envolve diversas dimensões, em que destacam-se [3]:

- *Integração de Dados*: diz respeito aos meios como as ferramentas compartilham dados, incluindo serviços de repositório e de compartilhamento de dados.
- *Integração de Apresentação*: trata de aspectos de padronização das interfaces com o usuário. As interfaces de um ADS devem ser homogêneas e consistentes, permitindo que os desenvolvedores alternem entre as ferramentas que utilizam sem mudanças substanciais de estilo e, portanto, sem sofrer impacto ou ter que despender muito tempo para aprender a utilizar novas ferramentas.
- *Integração de Controle*: refere-se à habilidade de uma ferramenta notificar ou iniciar uma ação em outra, controlando os eventos ocorridos e compartilhando funcionalidades.
- *Integração de Processo*: diz respeito à ligação entre as ferramentas e o processo de software. Para integrar ferramentas, é preciso ter um foco forte no gerenciamento

do processo de software. O processo deve definir que ferramentas um desenvolvedor pode utilizar e quando ele deverá ter acesso às mesmas, em função da atividade do processo que ele está realizando.

- *Integração de Plataforma*: refere-se à independência da plataforma sobre a qual funcionará o ambiente e suas ferramentas.
- *Integração de Conhecimento:* com o aumento da complexidade dos processos de software, faz-se necessário considerar informações de natureza semântica e gerenciar o conhecimento obtido ao longo dos projetos. Desta forma, o conhecimento, assim como os dados, deve estar disponível no ambiente para ser compartilhado por diferentes ferramentas.

Dada a sua grande importância, estudos sobre integração de processo em ADSs deram origem a uma nova classe de ambientes, os Ambientes de Desenvolvimento de Software Centrados em Processo [2]. Um ADS Centrado em Processo é aquele que explora uma definição explícita do processo de software e pode ser visto, de fato, como a automatização parcial desse processo, incluindo mecanismos para: *(i)* guiar a seqüência de atividades definida; *(ii)* gerenciar os produtos que estão sendo desenvolvidos; *(iii)* executar ferramentas necessárias para a realização das atividades; *(iv)* permitir comunicação entre as pessoas; *(v)* colher dados de métricas automaticamente; (vi) reduzir erros humanos e *(vii)* prover controle do projeto à medida que este vai sendo executado [6].

2.1 O Ambiente ODE

Este trabalho enfoca o suporte, através do uso de inferências, à integração de processo no contexto do ambiente ODE (*Ontology-based software Development Environment*) [4]. ODE é um ADS Centrado em Processo, que tem sua fundamentação baseada em ontologias. A premissa do projeto de ODE é a seguinte: se as ferramentas de um ADS são construídas baseadas em ontologias, a integração dessas ferramentas pode ser facilitada, pois os conceitos envolvidos estão bem definidos na ontologia. Essa é uma das principais características que distingue ODE de outros ADSs, sua base ontológica. Especialmente em ADSs, ontologias reduzem confusões terminológicas e conceituais, facilitando o entendimento compartilhado e a comunicação entre pessoas com diferentes necessidades e pontos de vista. Além disso, a padronização de conceitos provida por uma ontologia permite que a comunicação entre as ferramentas que compõem o ambiente seja aprimorada [7]. Dentre as ontologias que compõem a base ontológica de ODE, uma delas merece destaque no contexto deste trabalho: a ontologia de processo de software [1]. Esta ontologia define os principais conceitos relacionados a processos de software e é a base para a integração de processo em ODE.

A arquitetura de ODE reflete sua base ontológica. Ela possui dois níveis: o Nível Base e o Meta-Nível. O nível base define as classes que controlam os processos definidos no ambiente (o pacote *Controle*) e suas ferramentas. O meta-nível, ou pacote *Conhecimento*, define as classes que descrevem o conhecimento sobre os objetos do nível base. A figura 1 mostra a relação entre esses dois níveis arquiteturais.

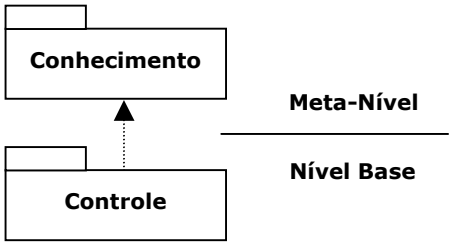

Figura 1. Arquitetura em Níveis de ODE.

As classes do pacote *Conhecimento* são derivadas diretamente de ontologias e seus objetos podem ser vistos como itens de instanciação de uma ontologia. Elas constituem o conhecimento do ambiente, que pode ser utilizado por todas as ferramentas que o compõem. No caso da ontologia de processo de software, os objetos da classe conhecimento descrevem o conhecimento sobre processos de software, suas atividades, artefatos produzidos e consumidos, recursos utilizados e procedimentos (métodos, técnicas e diretrizes) adotados, dentre outros.

Uma ontologia não tem o objetivo de descrever todo o conhecimento envolvido em um domínio, mas deve modelar o que é essencial para descrever esse domínio. Dessa forma, novas classes, associações, atributos e operações são criados para tratar decisões específicas na modelagem do nível base. As classes do pacote *Controle*, responsáveis pela definição e controle de processos de projetos de software em ODE, não derivam diretamente da ontologia de processo de software, mas são baseadas nelas. Algumas classes desse nível têm uma classe correspondente no meta-nível, preservando as mesmas restrições contidas no modelo de conhecimento, este sim baseado na ontologia. Dessa maneira, o pacote *Conhecimento* é usado para descrever características dos objetos do pacote *Controle*.

2.2 A Camada de Inferência de ODE

De maneira geral, sistemas convencionais que buscam representar conhecimento, dentre eles os sistemas desenvolvidos segundo o paradigma orientado a objetos, têm seu conhecimento armazenado em um repositório de dados, de onde pode ser consultado, e as regras relacionadas a esse conhecimento encontram-se embutidas no código da aplicação, não sendo possível aproveitar todo o potencial que a tecnologia baseada em conhecimento pode proporcionar.

Um primeiro passo no sentido de tirar maior proveito do conhecimento é rever a sua forma de representação. Dentre as várias formas de representar conhecimento, uma abordagem interessante é o tratamento segundo o paradigma lógico, em que fatos são armazenados e acessados a partir de um repositório de dados, e as regras são isoladas da aplicação. Separar regras da aplicação é uma maneira flexível de manter o conhecimento sem a completa reestruturação das aplicações [8]. O resultado de seu uso é um sistema mais robusto com uma base de conhecimento maleável que pode crescer e mudar na medida em que o negócio evolui. Além disso, o uso de técnicas de inteligência artificial é capaz de aperfeiçoar a manipulação dessas regras, proporcionando ao sistema um caráter mais dinâmico e permitindo a realização de inferências.

Interpretando o conhecimento sobre projetos, processos, atividades, recursos, métodos e ferramentas, um ADS pode oferecer assistência inteligente a gerentes e desenvolvedores, guiando, monitorando e auxiliando o uso de ferramentas durante o desenvolvimento de software. Assim, no contexto do ambiente ODE, foi desenvolvida uma infra-estrutura de suporte à integração de conhecimento, que tem uma camada responsável por combinar objetos e lógica, a Camada de Inferência, dotando o ambiente da capacidade de representar conhecimento através de regras e realizar inferências. Essa camada permite que o conhecimento seja obtido a partir de informações do ambiente, representado em bases de conhecimento segundo uma linguagem lógica (Prolog), e tratado através de mecanismo de inferência.

Com a introdução de uma linguagem lógica em um ambiente OO, é possível aproveitar os benefícios de um segundo paradigma. Situações como as que envolvem recursividade ou lidam com muitos relacionamentos entre objetos, exigindo casamento de padrões ou vários acessos às informações dos objetos, são mais adequadamente tratadas no paradigma lógico. Esses tipos de situações são bastante comuns, especialmente em operações envolvendo conhecimento.

2.2.1 Funcionamento da Camada de Inferência

Um dos objetivos da Camada de Inferência é criar em ODE uma forma de representação lógica do conhecimento, utilizando Bases de Conhecimento (BCs). O propósito é expressar, através de fatos e regras, informações que possam ser extraídas do ambiente, como, por exemplo, dados sobre projetos realizados.

A representação lógica das regras pode trazer vantagens, pois elas podem ser modificadas sem haver necessidade de alterar e recompilar o sistema. Taylor [9] diz que manter as regras embutidas no código funciona em pequenos sistemas, mas não é escalável e, nos grandes sistemas, requer alterações sempre que elas mudam. E acrescenta que a tarefa mais difícil da manipulação de regras é torná-la simples para que as pessoas entendam e usem. Isso pode ser alcançado com uma interface amigável. Sistemas que combinam objetos e regras com interfaces gráficas tornam-se mais robustos, inteligíveis e fáceis de manter [8].

Assim, a Camada de Inferência deve disponibilizar ao Engenheiro de Conhecimento uma maneira simples de criar e manipular BCs. Para tal, foi desenvolvido um editor de Estruturas de Bases de Conhecimento (EBCs), no qual é possível definir predicadores e regras. Essas estruturas servem como um molde quando da instanciação de uma BC, momento em que os dados são extraídos do ambiente para, junto aos predicadores, formar a base de fatos de uma BC. A partir de uma BC instanciada, é possível manipular seu conhecimento através de uma máquina de inferência Prolog [10] incorporada à camada.

As figuras 2 e 3 apresentam o esquema geral de funcionamento da Camada de Inferência e sua estrutura interna, respectivamente, que são detalhados a seguir.

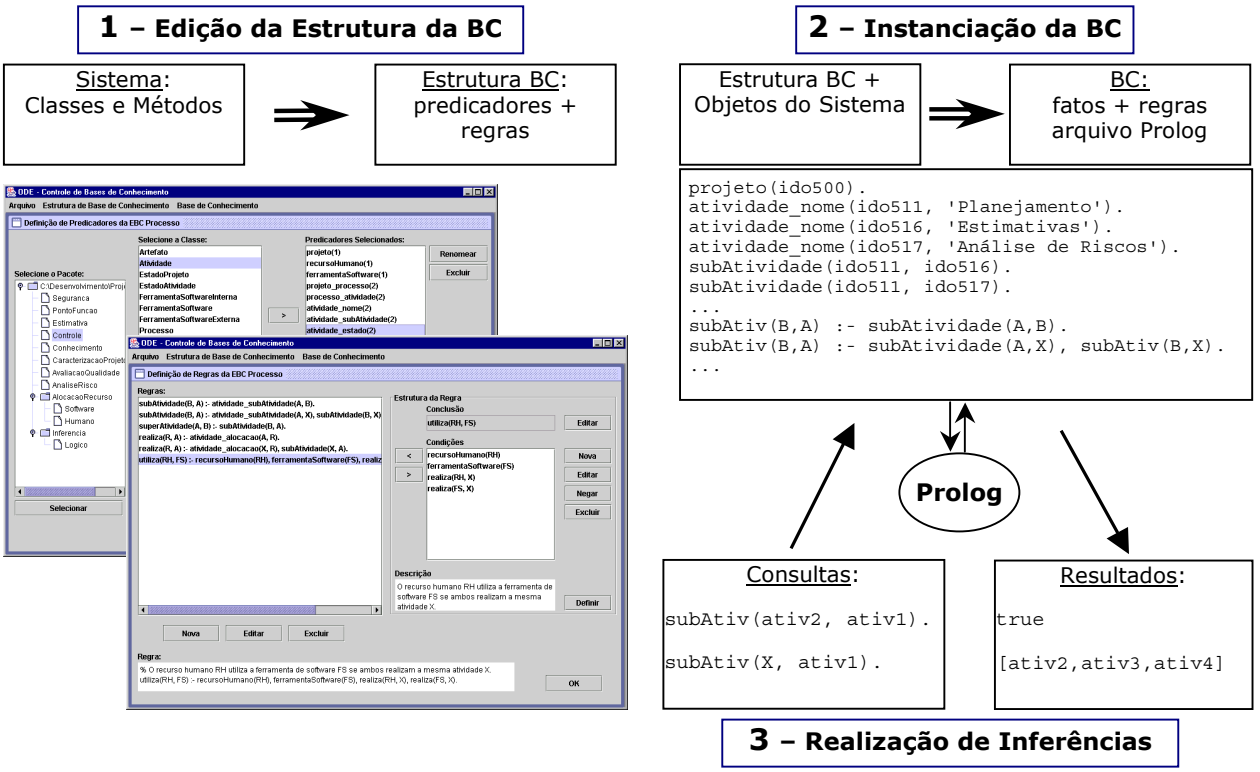

Figura 2. Esquema de utilização da Camada de Inferência.

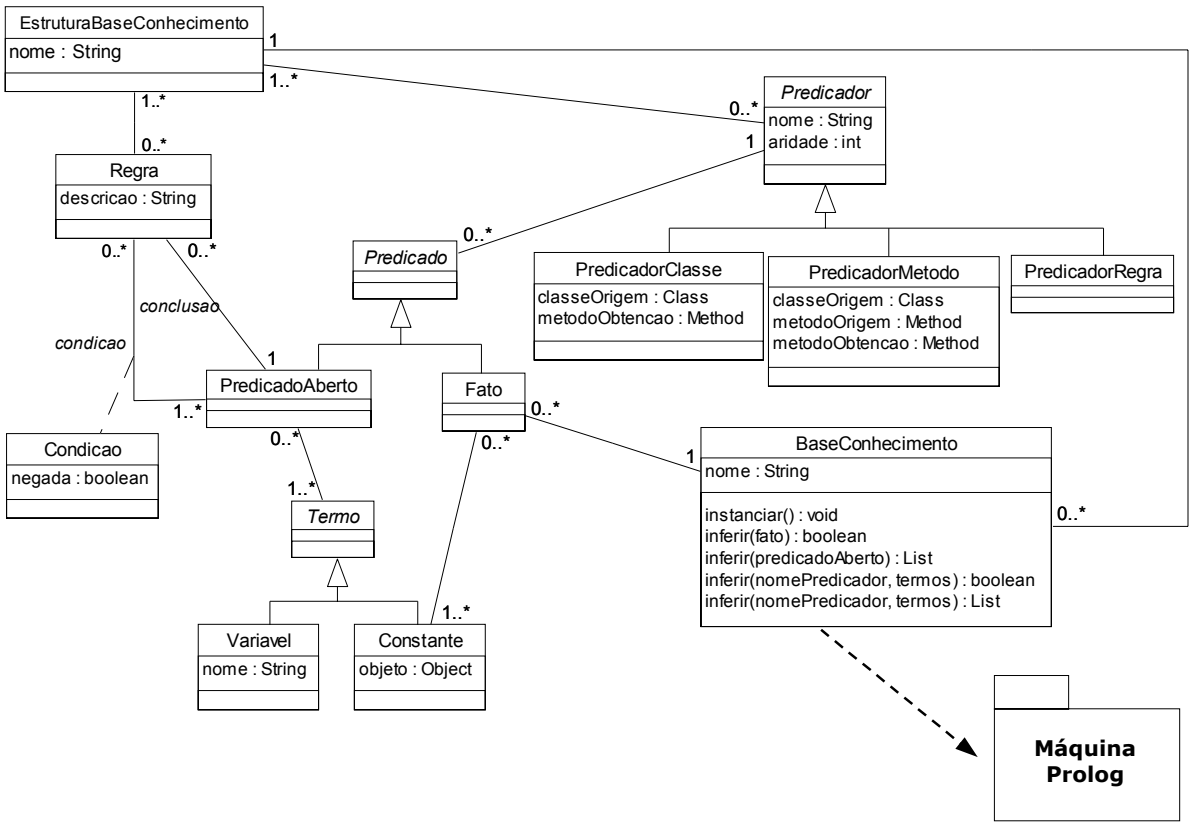

Figura 3. Estrutura Interna da Camada de Inferência - Diagrama de Classes do Pacote Modelo Lógico.

2.2.2 Edição de Estruturas de Bases de Conhecimento

Através do editor de Estruturas de Bases de Conhecimento (EBCs) provido pela Camada de Inferência é possível criar e manter as estruturas das BCs. Como uma BC é formada por um conjunto de fatos e regras, uma EBC precisa ter os elementos necessários para a criação destes. As EBCs não devem possuir fatos, mas apenas a estrutura deles, sendo compostas, assim, por ***predicadores*** e ***regras***. ***Predicadores*** são símbolos que formam enunciados a partir de *termos*. A cada predicador é associada uma *aridade*, que indica o número de termos necessários à formação do enunciado. Através dos predicadores, são formados os enunciados mais simples de uma teoria: os enunciados atômicos, chamados *predicados*. Um predicado contendo *variáveis* como termos é um *predicado aberto*. Caso todos os termos sejam *constantes*, o predicado é dito um *predicado fechado* ou *fato* [11]. ***Regras*** são modeladas como uma conjunção de *condições* e uma *conclusão*. As condições e conclusões das regras são representadas por predicados [11].

O Modelo Lógico, apresentado na figura 3, modela os conceitos da lógica de primeira ordem segundo o paradigma OO. Ele é utilizado para expressar o conhecimento, através de uma representação lógica, no ambiente.

A primeira etapa da criação de EBCs é a definição de seus predicadores. Ela é feita a partir da hierarquia de pacotes do ambiente, utilizando-se reflexão computacional para obter informações acerca de classes e métodos. É possível criar dois tipos de predicador: **`PredicadorClasse`**, para representar as classes do ambiente, e **`PredicadorMetodo`**, para representar, os atributos e as associações das classes, definidos através de métodos. Esses predicadores armazenam, além de nome e aridade, informações sobre a sua origem, que são úteis quando é feita a instanciação das BCs.

Após a definição dos predicadores, é possível montar as regras de uma EBC. Uma regra possui sempre uma conclusão e ao menos uma condição, representadas por predicados abertos. As condições são criadas a partir de predicadores existentes e de termos (constantes ou variáveis) associados a ele, formando predicados. Uma condição pode ser negada ou não e o seu número de termos é determinado pela aridade do predicador. A conclusão é criada a partir de um **`PredicadorRegra`** e de termos associados, formando um predicado aberto. Os predicadores regra não são criados quando é feita a definição de predicadores, mas somente quando são definidas as cabeças das regras. Na figura 4, pode-se observar alguns predicadores e regras definidos no editor de EBCs.

2.2.3 Instanciação de Bases de Conhecimento

Durante a instanciação de uma BC, uma EBC é utilizada para que seus predicadores originem fatos e suas regras sejam acessadas pela BC. A etapa mais importante da instanciação é a transformação dos predicadores em fatos. Uma lista de fatos é criada buscando-se, através das origens dos predicadores, os dados do ambiente que representam os termos dos fatos.

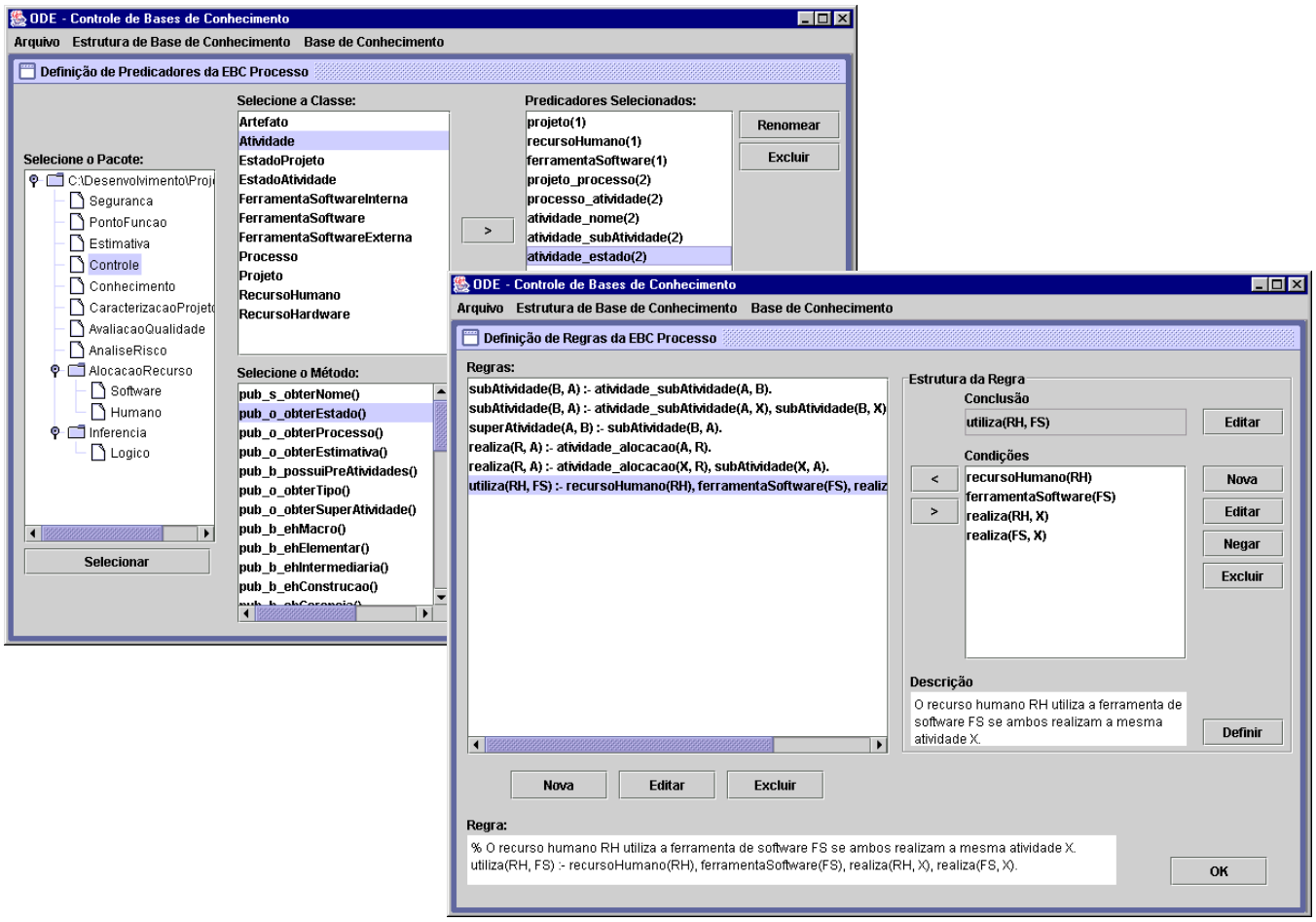

Figura 4. Definição de Predicadores e Regras.

Há duas formas de criar fatos, dependendo do tipo de predicador utilizado:

- `PredicadorClasse`: gera fatos de aridade 1 que representam logicamente as instâncias de sua classe de origem. Cada fato gerado possui como nome o nome do predicador e como único termo uma instância obtida da classe. Por exemplo, seja um predicador classe de nome `recursoHumano`, cuja classe de origem é `RecursoHumano`. Se João é uma instância dessa classe, então um fato `recursoHumano([João])` é criado.

- `PredicadorMetodo`: gera fatos de aridade 2 que são representações lógicas de atributos ou associações de instâncias da classe de origem. Os fatos são criados na forma **predicador(instância, retorno_método)**. Assim, um predicador método de nome `atividade_estado`, que representa uma associação entre objetos das classes `Atividade` e `EstadoAtividade`, origina a fatos como `atividade_estado([Análise de Requisitos],[Em Execução])`, indicando que a atividade `Análise de Requisitos` está no estado `Em Execução`.

Ao final da instanciação, as listas de fatos e de regras são utilizadas para gerar um arquivo Prolog que é a representação lógica da BC. Ele possui uma base de fatos e regras, como pode ser observado no exemplo da figura 5. Deve-se notar que as constantes criadas a partir dos objetos do domínio do problema são representadas pelos respectivos identificadores dos objetos (ido).

2.2.4 Realização de Inferências

Criar EBCs, definindo seus predicadores e regras e instanciá-las em BCs são passos que devem ocorrer anteriormente à realização de inferências. De posse de uma BC instanciada, a Camada de Inferência é capaz de atender ao seu propósito principal: realizar inferências lógicas sobre o conhecimento, utilizando uma máquina Prolog. Uma consulta, ou inferência, pode ser feita de duas maneiras distintas, dependendo do tipo de predicado utilizado:

- **`Fato`**: seus termos são constantes e o resultado da consulta indica se o fato é verdadeiro ou falso;

- **`PredicadoAberto`**: possui uma variável e constantes como termos e o resultado da inferência é uma lista com os possíveis valores para a variável.

Em ambos os casos, o que a BC faz é transformar os objetos em dados lógicos e, então, realizar a inferência sobre o arquivo Prolog gerado na instanciação da BC, utilizando uma máquina Prolog. O resultado da inferência é transformado em objetos e retornado à aplicação.

```
% Arquivo Controle.pl
% Base de Conhecimento Controle

% Base de Fatos:
projeto(ido50_0).
projeto_processo(ido50_0, ido50_10).
processo_atividade(ido50_10, ido50_11).
. . .
processo_atividade(ido50_10, ido50_19).
atividade_nome(ido50_11, 'Planejamento').
atividade_nome(ido50_12, 'Análise de Requisitos').
atividade_nome(ido50_13, 'Projeto').
atividade_nome(ido50_14, 'Implementação').
atividade_nome(ido50_15, 'Testes').
atividade_nome(ido50_16, 'Análise de Riscos').
atividade_nome(ido50_17, 'Realização de Estimativas').
atividade_nome(ido50_18, 'Gerenciar Riscos').
atividade_nome(ido50_19, 'Identificar Riscos').
subAtividade_(ido50_16, ido50_11).
subAtividade_(ido50_17, ido50_11).
subAtividade_(ido50_18, ido50_16).
subAtividade_(ido50_19, ido50_16).
. . .

% Base de Regras:

% B é sub-atividade de A se a atividade A tem B como
sua sub-atividade.
subAtividade(A, B) :- subAtividade_(A, B).

% B é sub-atividade de A se B é sub-atividade de X,
que é sub-atividade de A (transitividade).
subAtividade(A, B) :- subAtividade_(A, X),
                      subAtividade(X, B).
. . .
```

Figura 5. Representação Lógica de uma Base de Conhecimento.

As inferências podem ser realizadas de dois modos. O primeiro, via interface gráfica, é mais simples e é utilizado através de uma aplicação gráfica na qual o usuário monta as consultas desejadas e obtém os resultados das inferências visualmente. O objetivo principal desta funcionalidade é permitir testar a BC criada, realizando inferências de modo a verificar a coerência dos predicados e regras definidos. Assim, o Engenheiro de Conhecimento pode validar uma BC antes que ela seja efetivamente utilizada na construção de um sistema. Além disso, espera-se facilitar o aprendizado do usuário que não conhece a camada, para que ele possa utilizar melhor o segundo modo de inferência (via código).

Como mostra a figura 6, as consultas são realizadas sobre uma BC, bastando-se escolher os elementos necessários para montar um predicado aberto ou um fato. Inicialmente, deve-se selecionar um predicador e determinar os seus termos e, em seguida, realizar a inferência. Os resultados são apresentados no quadro à direita. Nota-se que os objetos do domínio do problema são exibidos como `Strings` entre colchetes.

O segundo modo de uso de inferência é realizado via código implementado. Mesmo com o isolamento das regras, sistemas precisam acessá-las para funcionar. O que se pretende é fazer com que o ambiente obtenha resultados através das regras de uma BC. Trabalhando dessa forma, pode-se substituir operações antes puramente codificadas, com a vantagem de que as regras envolvidas nas operações, por serem interpretadas, podem ser alteradas, sem que haja necessidade de recompilar o sistema. Deve-se ressaltar, ainda, que o sistema continua a operar com objetos, obtidos, agora, segundo um paradigma diferente.

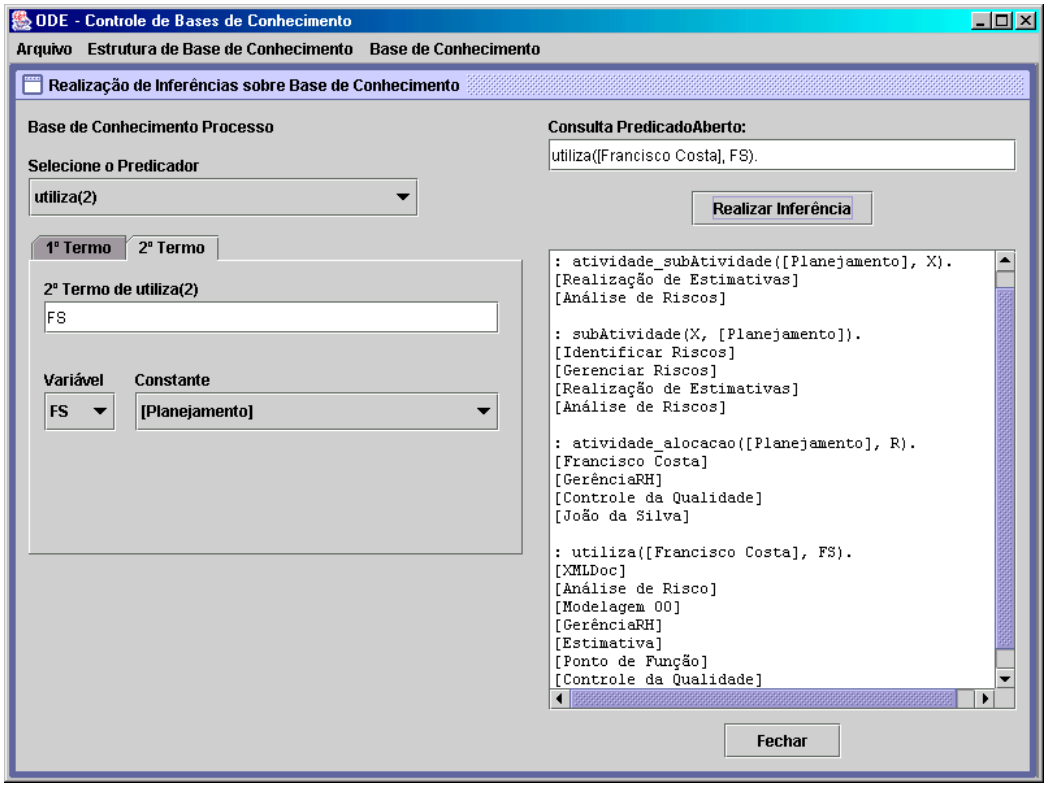

Figura 6. Realização de Inferências via Interface.

Conforme discutido anteriormente, dois métodos de consultas são oferecidos pela BC para realização da inferência codificada: um para fatos e outro para predicados abertos. Em ambos é necessário passar como parâmetro o nome do predicador da inferência e os seus termos. Os termos constantes são objetos da aplicação e os variáveis são instâncias da classe `Variavel`.

A figura 7 apresenta um trecho de código utilizado para realizar uma operação via inferência lógica, em que o resultado é uma lista de objetos. Como se pode observar, o que deve ser feito para realizar as operações é inferir sobre uma BC, passando o nome do predicador e uma lista de objetos como termos. Internamente, a BC busca o predicador e cria os termos com os objetos passados. Então, o predicado é montado para que seja realizada a consulta e o resultado é tratado e retornado.

```
//Obtém as sub-atividades da atividade passada
public List obterSubAtividade(Atividade atividade)
{
  //Realizar Inferência com PredicadoAberto
  Variavel var = new Variavel("X");
  Object[] termos = new Object[] {var, atividade};
  List lstSubAtiv =
baseConhecimento.inferir("subAtividade", termos);
  return lstSubAtiv;
}
```

Figura 7. Realização de Inferências via Código.

3. INTEGRAÇÃO DE PROCESSO EM ODE

Por ser um ADS centrado em processo, a dimensão de integração de processo é essencial em ODE. Assim, o ambiente deve suportar a definição de processos de software e se utilizar desta para estabelecer uma ligação explícita entre as ferramentas do ambiente e os processos definidos. Nas subseções que seguem, são abordadas importantes características da integração de processo em ODE, apresentando a definição e o controle de processo, e a sua ligação com as ferramentas que o automatizam.

3.1 Definição de Processo

Para se definir um processo de software, é necessário um nível de experiência e conhecimento bastante elevado por parte do gerente de projeto. Inúmeras variáveis devem ser consideradas, tais como características da equipe de desenvolvimento, características específicas do projeto e nível de conhecimento da organização em engenharia de software.

ODE utiliza uma abordagem de definição de processos em níveis [12]. A Figura 8 apresenta os diferentes níveis de definição de processos, assim como os fatores que influenciam a definição de processos nos diferentes níveis de abstração. Primeiro, define-se o processo padrão da organização. Esse processo contém os ativos de processo (atividade, artefatos, recursos e procedimentos) que deverão fazer parte dos processos de qualquer projeto da organização. A seguir, o processo padrão pode ser especializado para considerar tecnologias de desenvolvimento, paradigmas ou domínios de aplicação específicos. Durante a especialização, ativos de processo poderão ser adicionados ou modificados, de acordo com o contexto para o qual se está realizando a especialização. Finalmente, no último nível do modelo de definição de processos, está a instanciação de um processo padrão ou de um processo especializado para um projeto específico. Essa instanciação consiste na adaptação de um processo para um projeto. Nessa adaptação, devem ser consideradas as particularidades do projeto e as características da equipe de desenvolvimento. Define-se, nesse momento, o modelo de ciclo de vida a ser utilizado durante o desenvolvimento e pode-se incorporar novas atividades ao processo, bem como adicionar ou alterar seus artefatos consumidos e produzidos, recursos utilizados e procedimentos adotados [13].

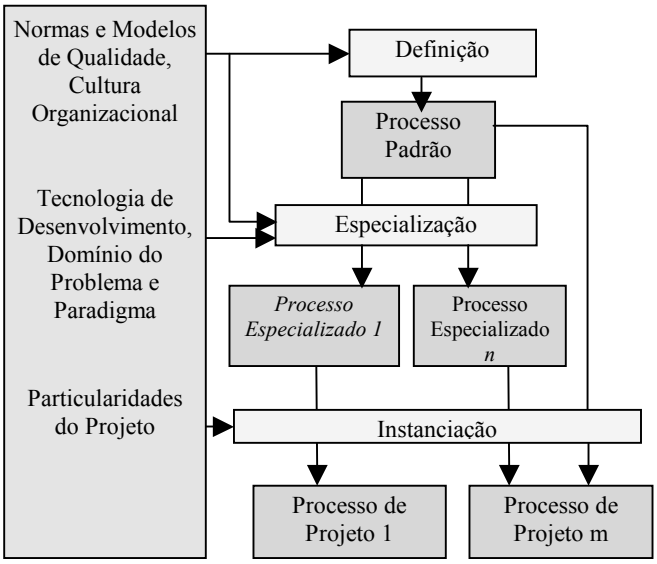

Figura 8. Modelo para Definição de Processos em Níveis [13].

Os diferentes níveis de definição de processo ocorrem em diferentes níveis da arquitetura de ODE (apresentada na figura 1). Os processos padrão e especializado descrevem as atividades que devem pertencer aos processos de quaisquer projetos de uma organização, ou seja, esses processos descrevem o conhecimento a respeito dos processos dessa organização. Dessa forma, de acordo com a arquitetura de ODE, processos padrão e especializado são definidos no Meta-Nível e sua modelagem encontra-se no *Pacote Conhecimento*, representado na figura 9.

As classes *ConhecimentoProcessoPadrao* e *ConhecimentoProcessoEspecializado* representam os processos padrão e especializados definidos em ODE, respectivamente. Para cada processo padrão ou especializado definido, devem ser estabelecidas as atividades que o compõem. As atividades de um processo padrão ou especializado são representadas pela classe *ConhecimentoAtividadeProcesso*. Para cada uma dessas atividades, devem ser definidas suas subatividades, sua ordem de precedência, seus artefatos consumidos ou produzidos (classe *ConhecimentoArtefato*), seus recursos demandados (classe *ConhecimentoRecurso*) e os procedimentos utilizados (classe *ConhecimentoProcedimento*).

Após encerrar a definição de um processo padrão ou especializado, é preciso definir os modelos de ciclo de vida adequados para esse processo. Isso é necessário para permitir que os modelos de ciclo de vida estejam em conformidade com a precedência entre atividades definida no processo padrão ou especializado. Os modelos de ciclo de vida são representados pela classe *ConhecimentoModeloCicloVida*.

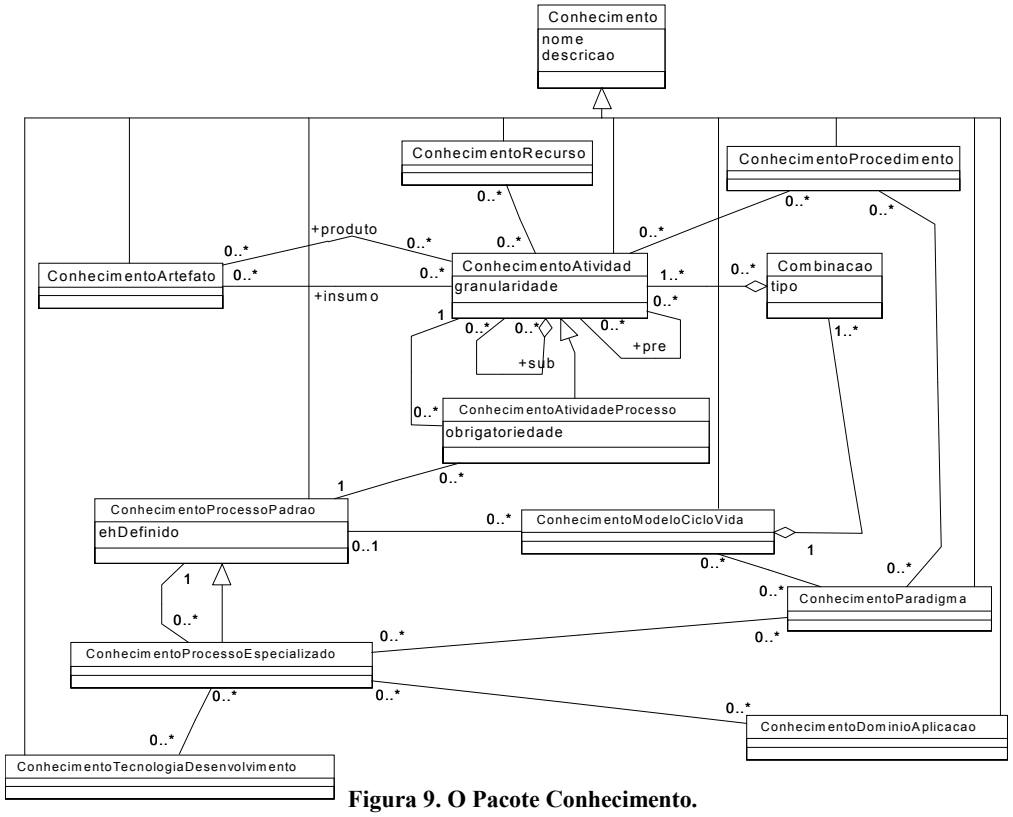

Figura 9. O Pacote Conhecimento.

Definidos os modelos de ciclo de vida para um processo padrão ou especializado, estes podem ser utilizados na definição de processos de projeto. Deve-se ressaltar que, ao contrário da definição de processos padrão e especializados, que ocorre no meta-nível da arquitetura de ODE, a definição de processos de projeto ocorre no nível base, mais especificamente no pacote Controle, mostrado parcialmente na figura 10. A classe *Processo* representa um processo de projeto, definido para um projeto de software específico (representado pela classe *Projeto*). Um processo de projeto pode ser definido tendo por base um processo padrão ou especializado, ou, então, ser definido a partir do repositório de conhecimento do ambiente. Por isso, um *Processo* pode ter, ou não, uma associação com *ConhecimentoProcessoPadrao*. Quando se define um processo de projeto a partir de um processo padrão ou especializado, todos os seus ativos são replicados para o novo processo, de acordo com o processo padrão/especializado e o modelo de ciclo de vida escolhidos. Nesse momento são criadas as atividades do processo e definidos os artefatos, recursos e procedimentos utilizados.

A janela de definição de processos padrão, especializado ou de projeto é similar e está apresentada na figura 11. A árvore localizada na parte esquerda da janela contém os ativos de processo já definidos para o processo em definição (atividades, artefatos, recursos e procedimentos). Na parte direita, encontra-se um painel que possibilita a escolha dos ativos de processo para o item selecionado na árvore.

A lista mais à direita contém os elementos já definidos para o processo. A lista mais à esquerda contém os ativos sugeridos a partir da realização de inferência nas bases de conhecimento

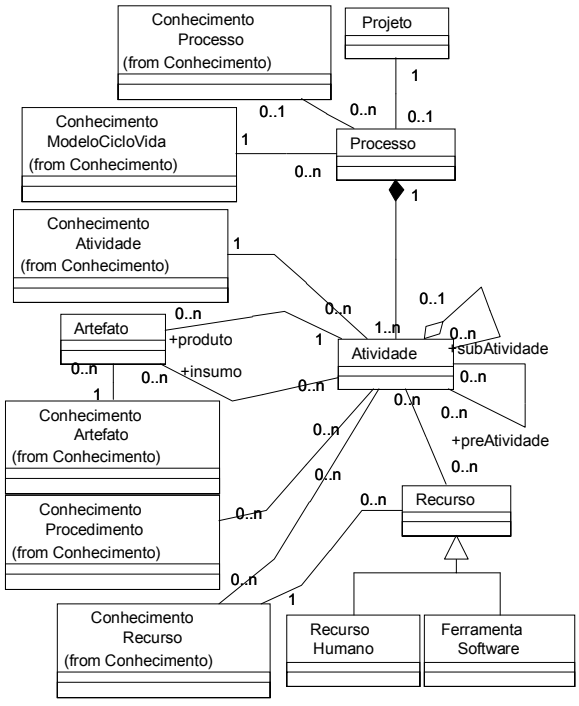

Figura 10. O Pacote Controle (Modelo Parcial).

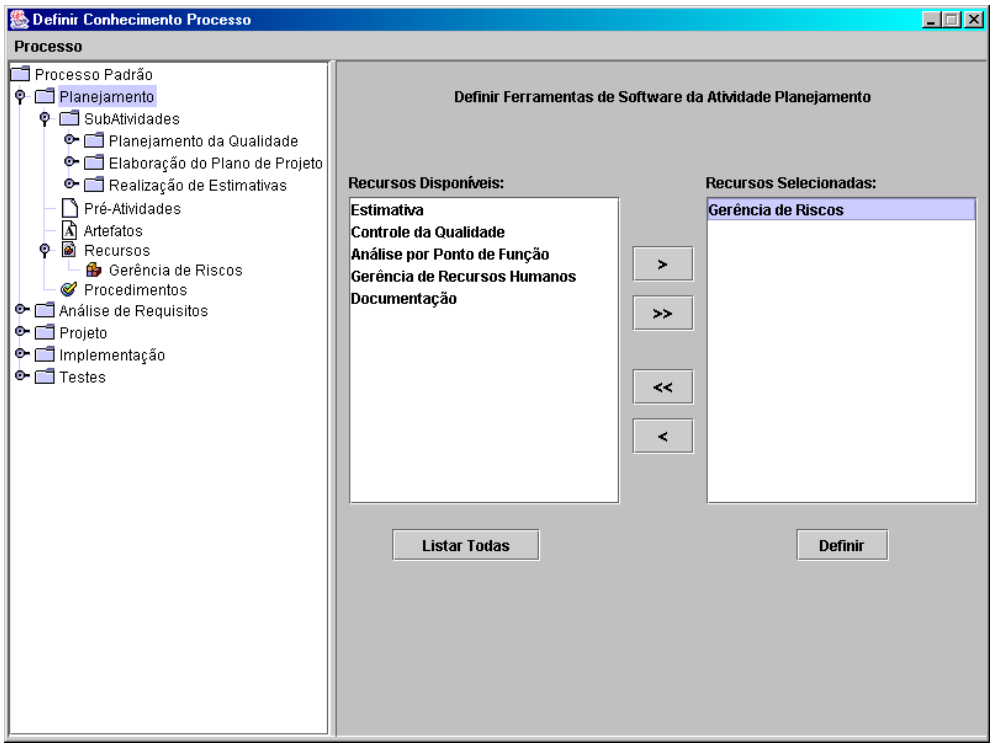

Figura 11. Definição de Processos Padrão, Especializado e de Projeto.

definidas no ambiente, por meio da camada de inferência. As bases de conhecimento, previamente definidas pelo Gerente de Conhecimento com base no pacote *Conhecimento*, são instanciadas na inicialização da ferramenta e utilizadas para que sejam realizadas inferências no momento em que são feitas as sugestões, oferecendo, assim, apoio baseado em conhecimento a essa tarefa.

Para ilustrar o uso de inferências na ferramenta, toma-se o caso da definição de subatividades de uma determinada atividade. A ferramenta de definição de processo poderia sugerir, em um primeiro momento, somente as subatividades associadas a uma atividade, conforme definido no modelo do pacote *Conhecimento*. Porém, ela fornece também uma forma mais elaborada de sugestão, considerando a regra de subatividades mostrada na figura 12. Assim, são exibidas, também, as subatividades das subatividades, de acordo com a transitividade imposta pela regra, tornando o apoio à definição de processos mais efetivo.

```
% Regra de Subatividade (A é subatividade de B)
subAtividade(A, B) :- subAtividade_(A, B).
subAtividade(A, B) :- subAtividade_(A, X),
                      subAtividade(X, B).

% Regra de Recurso(Atividade A usa Recurso R)
usa(A, R):- subAtividade(X, A), usa(X, R).
```

Figura 12. Algumas Regras Utilizadas

De forma semelhante, a sugestão de recursos para uma atividade é mais efetiva se considerar a regra de recursos, também mostrada na figura 12. O uso dessa regra, conforme representado no trecho de código da figura 13, permite que uma atividade aloque também recursos que estejam definidos para suas subatividades. Como pode ser visto na figura 11, é possível usar uma ferramenta de gerência de riscos na atividade de Planejamento, sem que necessariamente, no processo em definição, a atividade Análise de Riscos, à qual a ferramenta está associada no modelo de Conhecimento, tenha sido definida como uma subatividade de Planejamento. Isso decorre da inferência realizada usando a regra da figura 12 e do fato do modelo de Conhecimento definir que Análise de Riscos é uma potencial subatividade de Planejamento

```
//Obtém a sugestão de recursos para a atividade
passada
  public List obterRecursosSugeridos(Atividade
atividade) {
    Variavel var = new Variavel("R");
    Object[] termos = new Object[]
        {var, atividade.obterConhecimento()};
    List lstConhRecursos =
        baseConhecimento.inferir("usa",termos);
    return lstConhRecursos;
  }
```

Figura 13. Trecho de código usando a Camada de Inferência

3.2 Controle de Processo

Uma vez definido o processo do projeto, é necessário que este possa ser controlado e acompanhado pelo ambiente. Este controle engloba desde a gerência dos estados das atividades até a disponibilização de ferramentas de software pelo ambiente, permitindo a execução das atividades do processo.

Durante a definição de um processo de projeto, são definidos os tipos de recurso necessários para uma realizar uma atividade. Por exemplo, define-se que será necessário um recurso do tipo *Analista* (instância da classe *ConhecimentoRecursoHumano*) para executar a atividade de *Análise de Requisitos*. Somente em um segundo momento, durante a alocação de recursos, é que se aloca uma pessoa, por exemplo, João (uma instância da classe *RecursoHumano* do pacote *Controle*), que naturalmente deve

exercer a função de Analista. De maneira análoga, ocorre a alocação de ferramentas de software para uma determinada atividade. Assim, depois de definida a necessidade de uma *Ferramenta de Modelagem* para a execução da atividade de *Análise de Requisitos*, deve-se informar qual a ferramenta de modelagem disponível na organização que será efetivamente utilizada nesta atividade.

Com base nas atividades que compõem um processo e nos recursos humanos e ferramentas de software alocados a essas atividades, o ambiente é capaz de se configurar de acordo com o usuário corrente, ou seja, a partir da identificação do recurso humano que efetuou *login* no ambiente, as funcionalidades são disponibilizadas de maneira personalizada, considerando o seu perfil e sua alocação a projetos.

Figura 14. O Ambiente Configurado.

O acesso às ferramentas do ambiente é determinado pela alocação de recursos e pelos estados das atividades. Em um projeto, um recurso humano deve acessar as ferramentas necessárias à execução de suas atividades. Assim, somente são exibidas no menu, as ferramentas alocadas às atividades às quais o usuário corrente também está alocado, desde que as atividades estejam em execução. Esse caráter configurável pode ser visto na figura 14, que mostra a janela principal do ambiente ODE, em uma situação em que um Gerente de Projeto está ativo no ambiente. No menu ferramentas, são oferecidas as ferramentas de apoio à realização de estimativas usando pontos de função e de apoio ao planejamento da qualidade (ControlQ), que, de fato, constituem todo o conjunto de ferramentas que esse usuário pode ter acesso, considerando as atividades ativas do processo (*Planejamento*) para as quais ele está alocado.

```
% O recurso humano RH pode executar a ferramenta de
software FS se ambos estão alocados à mesma
atividade X e o estado de X é em execução.
utiliza(RH,FS) :- recursoHumano(RH),
                  ferramentaSoftware(FS),
                  aloca(X,RH), aloca(X,FS),
                  atividade_estado(X, ido1_131).
```

Figura 15. Regra da Base de Conhecimento de Configuração.

Essa seleção das ferramentas é realizada através da base de conhecimento de configuração do ambiente, que possui a regra **`utiliza`**, mostrada na figura 15, onde `ido1_131` representa o estado "Em Execução".

Dessa forma, quando um usuário entra no ambiente, identificando-se, o menu é dinamicamente montado, obtendo-se as ferramentas de software através de inferências com o método mostrado na figura 16. A utilização da Camada de Inferência nesse caso apóia a integração de processo em ODE, permitindo a configuração do menu do ambiente através de inferências, o que além de ser mais simples e eficiente, é mais flexível, permitindo que a regra utilizada seja facilmente alterada.

Supondo-se que, para permitir um maior contato entre desenvolvedores e ferramentas, decida-se que a melhor política é exibir as ferramentas alocadas às atividades às quais o usuário está alocado, independente do estado da atividade. Assim, o usuário poderia acessar ferramentas de atividades futuras e rever o seu trabalho em ferramentas alocadas a atividades já finalizadas. Essa modificação no funcionamento do ambiente é plenamente suportada pela Camada de Inferência, bastando ao Gerente de Conhecimento redefinir a regra **`utiliza`**, desconsiderando a sua última condição. Posteriormente, seria preciso somente reinstanciar a base de conhecimento, não sendo necessário alterar ou recompilar o código do ambiente.

4. TRABALHOS CORRELATOS

A Estação TABA [14] é um meta-ambiente que gera ambientes de desenvolvimento de software adequados a projetos específicos a partir de um processo de software definido. A Estação TABA possui ferramentas de apoio à definição e acompanhamento de processos. A integração de processo na estação é considerada quando, a partir de um processo definido, um ADS é instanciado e configurado de acordo com as características desse processo, possuindo uma ligação com as ferramentas necessárias à sua execução. Tal como TABA, ODE suporta a definição e acompanhamento de processos, porém o ambiente se configura dinamicamente a partir das alocações de recursos humanos e ferramentas de software às atividades do processo do projeto corrente.

Ainda no contexto da estação TABA, utiliza-se a linguagem Prolog para representar módulos de conhecimento em ADSs [1, 15]. A linguagem lógica é usada para formalizar a descrição de conhecimento, transcrevendo relações e axiomas de ontologias para uma representação lógica, que é utilizada posteriormente no entendimento de problemas e na construção de aplicações. A abordagem é praticamente a mesma à apresentada neste trabalho, no que se refere ao encapsulamento e comunicação com uma máquina Prolog. Entretanto, não há o conceito de Estruturas de Bases de Conhecimento, nem apoio à edição de regras, cabendo ao engenheiro de conhecimento realizar a sua entrada diretamente em Prolog. Além disso, não há nenhuma funcionalidade de apoio à verificação das bases de conhecimento editadas.

```
//Obtém as ferramentas de software disponíveis ao recurso humano passado
  public List obterFerramentasUtilizadas(RecursoHumano recursoHumano) {
    Variavel var = new Variavel("FS");
    Object[] termos = new Object[] {recursoHumano, var};
    List lstFerramentas = baseConhecimento.inferir("utiliza", termos);
    return lstFerramentas;
  }
```

Figura 16. Trecho de Código da Configuração do Ambiente

A ferramenta Charon [16] fornece suporte para a modelagem, simulação, execução e acompanhamento de processos de software. Essa ferramenta representa informações sobre o processo através de fatos em bases de conhecimento e mantém regras incorporadas a agentes inteligentes. Uma máquina de inferência é utilizada para que os agentes façam consultas e modificações nas bases de conhecimento. Assim, os fatos são determinados pela aplicação e a alteração das regras consiste na modificação ou inclusão de agentes, não havendo uma forma flexível de manutenção de fatos ou regras por parte do usuário.

Outros ambientes, tais como EPOS [17] e Oz [18], também centrados em processo, utilizam regras para a modelagem de processos. Entretanto, a maior parte das abordagens encontradas utiliza a lógica para propósitos específicos do desenvolvimento de software, como a modelagem de processos, e não se preocupa em oferecer uma infra-estrutura para representação e manipulação de bases de conhecimento em lógica. Ou seja, tais abordagens não oferecem suporte automatizado à edição de predicados e regras e, geralmente, trabalham com as regras embutidas no código da aplicação.

5. CONCLUSÕES

Este artigo apresentou uma abordagem fundamentada em conhecimento para a integração de processo no ambiente ODE. Foi apresentada uma ferramenta que provê apoio automatizado à definição de processos de software, além de outras facilidades também baseadas em conhecimento para o controle e configuração do ambiente.

Existe uma forte ligação entre a ferramenta e a ontologia de processos na qual o ambiente se fundamenta. O uso de inferências possibilitou à ferramenta operar com maior fidelidade em relação à ontologia, respeitando suas restrições e axiomas. As situações demonstradas fazem parte de um conjunto de operações que aproximam a ferramenta de seu modelo conceitual. Realizar esse tipo de operação, muitas vezes, é extremamente custoso se feito da maneira convencional. Porém, o uso de deduções lógicas possibilitou que as operações fossem implementadas de maneira mais fácil e obtendo melhor desempenho em sua execução. A abordagem utilizada permite maior consistência na utilização de conhecimento e privilegia a manutenção e extensão do ambiente.

A definição de processos de software em níveis abre espaço para a melhoria contínua de processos de software, baseada na captura de experiências e informações de projetos que utilizam o processo padrão como base. Neste contexto, o uso de inferências pode apoiar a busca e disseminação de dados históricos, obtidos de projetos anteriores, fornecendo subsídios para um contínuo aprimoramento de processos.

6. AGRADECIMENTOS

O presente trabalho foi realizado com apoio da CAPES e do CNPq, entidades do governo brasileiro voltadas ao desenvolvimento científico e tecnológico.

7. REFERÊNCIAS

[1] Falbo, R.A. *Integração de Conhecimento em um Ambiente de Desenvolvimento de Software*. Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, Dezembro/1998.

[2] Fuggetta, A. *Software Process: A Roadmap*, In: Proc. of The Future of Software Engineering, ICSE'2000, Limerick, Ireland, 2000.

[3] Pfleeger, S.L., *Software Engineering: Theory and Practice*, 2nd Edition, New Jersey: Prentice Hall, 2001.

[4] Bertollo, G.; Ruy, F.B.; Mian, P.G.; Pezzin, J.; Schwambach, M.; Natali, A.C.C.; Falbo, R.A. *ODE – Um Ambiente de Desenvolvimento de Software Baseado em Ontologias*. Anais do XVI Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas, Gramado, Outubro 2002.

[5] Pressman, R.S. *Engenharia de Software*, 5 ed. Rio de Janeiro: McGraw-Hill, 2002.

[6] Christie, A.M., *Software process automation: the technology and its adoption*. Pittsburgh, 1995.

[7] Falbo, R.A., Guizzardi, G., Natali, A.C.C., Bertollo, G., Ruy, F.B., Mian, P.G. *Towards Semantic Software Engineering Environments*. In: Proc. of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE'02, Ischia, Italy, 2002.

[8] Rasmus, D.W. *Ruling Classes: The heart of knowledge-based systems*. Object Magazine, July 1995.

[9] Taylor, D. *Building intelligent objects*. Object Magazine, July/1995.

[10] DEIS (Dipartimento di Elettronica, Informatica e Sistemistica), Università di Bologna a Cesena, Italy. *tuProlog Developer's Guide*. Disponível em <http://lia.deis.unibo.it/research/tuprolog>. Acesso em: Março/2003.

[11] Araribóia, G. *Lógica para Informática - Um Estudo Introdutório*. 1a. edição. Niterói: ILTC, 1989.

[12] Bertollo, G.; Falbo, R.A. *Apoio Automatizado à Definição de Processos de Software em Níveis*. Anais do II Simpósio Brasileiro de Qualidade de Software, Fortaleza, Setembro, 2003.

[13] Rocha, A. R. C., Maldonado, J. C., Weber, K. C., *Qualidade de Software: Teoria e Prática*. São Paulo: Prentice Hall, 2001.

[14] Travassos, G.H. *O Modelo de Integração de Ferramentas da Estação TABA*. Tese de Doutorado. COPPE/UFRJ, Rio de Janeiro, Março/1994.

[15] Zlot, F. *Conhecimento de Tarefa em Ambientes de Desenvolvimento de Software Orientados a Domínio*. Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, Junho/2002.

[16] Murta, L.G.P. *Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes*. Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, Março/2002.

[17] Jaccheri, M., Conradi, R., *Techniques for Process Model Evolution in EPOS*. IEEE Transactions on Software Engineering, v. 19, n. 12, December 1993.

[18] Ben-Shaul, I.Z., Kaiser, G.E., *Federating Process-Centered Environments: The Oz Experience. Automated Software* Engineering, vol. 5, n. 1, January 1998.