CHARON: UMA MÁQUINA DE PROCESSOS EXTENSÍVEL BASEADA EM AGENTES INTELIGENTES

Leonardo Gresta Paulino Murta

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:

_____

Prof. Cláudia Maria Lima Werner, D.Sc.

_____

Prof. João Carlos Pereira da Silva, D.Sc.

_____

Prof. Itana Maria de Souza Gimenes, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2002

MURTA, LEONARDO GRESTA PAULINO

Charon: Uma Máquina De Processos Extensível Baseada Em Agentes Inteligentes [Rio de Janeiro] 2001

XI, 107 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Automação de Processos de Desenvolvimento de Software
2. Agentes Inteligentes

I. COPPE/UFRJ II. Título (série)

Aos meus pais.

Agradecimentos

A Deus, pois sem ele nada seria possível.

Aos meus pais, por sempre me apoiarem em tudo o que desejo fazer.

À professora Cláudia Werner, por ser essa orientadora tão especial, que abre mão de seus raros momentos de descanso para ajudar a me tornar, um dia, um profissional competente.

Ao amigo Márcio Barros, que me orientou extra-oficialmente em todo o processo de preparação desse trabalho e que, mais que isso, sempre serviu como um exemplo de como devo agir para poder me tornar, um dia, um bom pesquisador e professor. Sem ele, esse trabalho certamente não seria o mesmo.

À professora Itana Gimenes, por ter aceitado participar desta banca.

Ao professor João Carlos, que além de ter concordado em participar desta banca, me ensinou o que sei sobre Inteligência Artificial e serve como um exemplo de como ser professor.

Aos professores da COPPE, que me guiaram através dos caminhos da ciência, mostrando como agir e como pensar.

A todos os integrantes do Projeto Odyssey, que sempre contribuíram muito para a minha formação.

À Vanessa e aos meus amigos, que foram compreensíveis nas minhas ausências devido à preparação deste trabalho.

Ao CNPq, pelo apoio financeiro.

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

CHARON: UMA MÁQUINA DE PROCESSOS EXTENSÍVEL BASEADA EM AGENTES INTELIGENTES

Leonardo Gresta Paulino Murta

Março/2002

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

O aumento da complexidade dos sistemas, aliado à diminuição dos prazos, e conseqüente necessidade de constituir grandes equipes de desenvolvimento, torna cada vez mais importante a definição de um processo que sistematize o desenvolvimento de software. Entretanto, além da definição do processo, é necessário fazer um acompanhamento de sua execução, guiando os desenvolvedores dentro de um ambiente de desenvolvimento de software (ADS) que contenha as ferramentas necessárias para a construção de aplicações.

O objetivo deste trabalho é propor uma arquitetura extensível para máquina de processo, baseada em agentes inteligentes, que permita modelar, simular, executar e acompanhar processos de desenvolvimento de software no contexto de um ADS. O uso de agentes inteligentes possibilita que a arquitetura evolua de forma simples, através da criação de novos agentes que se comunicam com os demais através de bases de conhecimento sobre processos instanciados, segundo uma ontologia definida. As bases de conhecimento de processos possuem uma representação Prolog, com regras geradas a partir da tradução de uma representação gráfica, com notação estendida do diagrama de atividades da UML.

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

CHARON: AN EXTENSIBLE PROCESSES MACHINE
BASED ON INTELLIGENT AGENTS

Leonardo Gresta Paulino Murta

March/2002

Advisor: Cláudia Maria Lima Werner

Department: Computer and Systems Engineering

The increasing complexity of software projects, together with frequent schedule reductions, and the need for large development teams, highlights the importance of a systematic approach for software development. In such a systematic approach, a development process is planned, executed, and monitored throughout the project life cycle. In addition, it is desirable to integrate the process with a software development environment (SDE), which provides the required tools to build applications.

This work aims to develop an extensible architecture based on intelligent agents, which provides software process modeling, simulating, executing, and monitoring within a SDE. The use of intelligent agents allows the architecture to evolve in an easy way, through the creation of new agents that communicate with each other through knowledge bases representing instantiated processes. These knowledge bases attend to a predefined ontology and are created by translating a graphical representation of the process to Prolog rules. The graphical representation is based on an extended notation of the UML activity diagram.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1 – Introdução

1.1 – Motivação

O desenvolvimento de software, como toda atividade de engenharia, necessita de um processo que o sistematize, envolvendo atividades, pessoas e ferramentas necessárias para a sua execução, assim como artefatos consumidos ou produzidos pelas atividades. A partir de um processo bem definido, é possível alavancar a qualidade do produto a ser gerado, pois a qualidade do produto é fortemente dependente da qualidade do processo pelo qual ele é construído e mantido (ROCHA, 1997; GOMES et al., 2001a; GOMES et al., 2001b).

Além do estabelecimento do processo, é necessário fazer um acompanhamento de sua execução, objetivando, entre outras coisas, fornecer informações que permitam a tomada de decisões gerenciais, de acordo com situações detectadas durante sua execução, e controlar o fluxo de trabalho dos engenheiros de software através da automação do processo. O software responsável por automatizar a execução de processos é conhecido como máquina de processos (SILVA et al., 1999).

Atualmente, existem várias abordagens para a construção de ambientes de desenvolvimento de software (ADS) com ênfase na modelagem e execução de processos, chamados de ambientes centrados em processo, como exibido no Capítulo 2. Entretanto, essas abordagens têm deficiências quanto às suas formas de utilização, atuação e extensão.

Ainda não existe uma forte preocupação na forma de **utilização** das máquinas de processo. As abordagens existentes permitem que processos sejam modelados, mas restringem o seu uso a um determinado nível de abstração. Seria interessante construir processos através de blocos, ou componentes, e reutilizar esses componentes de processos em outros contextos, facilitando a utilização da máquina de processos como um todo.

A **atuação** da maioria das abordagens é reativa, consistindo no tratamento de eventos provenientes dos próprios desenvolvedores. Todavia, a máquina de processos pode agir de forma pró-ativa, prevendo certas situações e atuando sem a interferência do desenvolvedor, fazendo com que a execução siga adiante ou notificando um determinado desenvolvedor que algo deve ser feito num dado momento.

Os requisitos de um ambiente centrado em processo podem variar com o tempo, motivando a **extensão** do ambiente. A facilidade de extensão do ambiente está diretamente relacionada com a forma em que a sua arquitetura foi projetada, facilitando o atendimento de novos requisitos. Arquiteturas seguindo padrões como o MVC (GAMMA et al., 1994), que privilegiam a separação entre os dados que representam as entidades envolvidas no problema (modelo), a lógica de execução (controle) e a comunicação com o usuário ou outros sistemas (visão), tornam mais simples as manutenções, inclusive no tocante da extensão do sistema, facilitando o atendimento de novos requisitos, como, por exemplo, a coleta de uma determinada métrica.

A motivação deste trabalho, portanto, está na possibilidade de melhorar os produtos de software gerados através de um ambiente que facilite a modelagem e acompanhamento de processos e diminuir os custos financeiros e o tempo necessário para a inclusão de novos requisitos nesse ambiente, através do apoio às necessidades de atuação e extensão detectadas nas ferramentas existente.

1.2 – Objetivo

Dentre as tecnologias atualmente disponíveis para o desenvolvimento de sistemas, a tecnologia de agentes inteligentes apresenta a estruturação necessária para a construção de sistemas pró-ativos e de fácil extensão, devido às suas características de decomposição, abstração e organização (JENNINGS, 2000).

A integração das tecnologias de processos e agentes pode ocorrer segundo duas perspectivas (JOERIS et al., 1997a; JOERIS et al., 1997b):

- Na primeira perspectiva, os agentes inteligentes funcionam como auxiliares ou substitutos dos agentes humanos no desenvolvimento de software. Os agentes não estariam implementando a máquina de processos em si, mas sim, seriam usuários dela durante a execução de um processo instanciado. Desta forma, o domínio de processos poderia ser visto como uma aplicação de agentes inteligentes.
- Na segunda perspectiva, os agentes são utilizados para a construção da própria máquina de processos. Em assim sendo, os agentes humanos não seriam substituídos por agentes inteligentes, mas estariam interagindo com agentes inteligentes durante a execução do processo. Nessa perspectiva, o agente não sabe o que o agente humano tem que saber para

cumprir seu papel na execução do processo, mas sabe como executar um processo ou fornecer outros serviços associados a essa execução.

Os conhecimentos necessários para os agentes inteligentes na primeira perspectiva estão relacionados com o domínio para o qual o processo foi modelado, que pode ser, por exemplo, a construção de um sistema para medicina. Já os conhecimentos necessários para os agentes na segunda perspectiva estão relacionados com o próprio domínio de automação de processos, independentemente do tipo de processo que esteja modelado. Para atender a motivação descrita anteriormente, a integração entre as tecnologias de processo e agentes visa a construção de uma máquina de processos extensível, voltada para o desenvolvimento de software, de acordo com a segunda perspectiva.

O objetivo deste trabalho é fornecer um ambiente para a modelagem, instanciação, simulação, execução, monitoramento e evolução de processos de software reutilizáveis que seja pró-ativo e de fácil extensão, utilizando uma arquitetura para a construção de agentes inteligentes e uma ontologia para a comunicação entre os agentes que define o vocabulário necessário para a representação dos processos modelados.

Para que esse objetivo possa ser atingido, objetivos intermediários devem ser alcançados, como a adoção de uma notação para a representação gráfica de processos, o mapeamento da notação gráfica para uma representação textual, a verificação de erros no processo modelado, a criação de regras para a execução do processo e acompanhamento dessa execução pelos desenvolvedores envolvidos, o suporte a visualização do estado atual da execução e a utilização de uma estrutura que permita a modificação do processo durante a sua própria execução.

O ambiente de desenvolvimento de software adotado para ilustrar a implementação da máquina de processos Charon [1]será o Ambiente Odyssey (WERNER et al., 2000), que visa prover uma infra-estrutura de suporte à reutilização baseada em modelos de domínio. A reutilização de software no Ambiente Odyssey ocorre através dos processos de Engenharia de Domínio, que tem o objetivo de construir componentes

[1] Charon (Caronte) é o barqueiro da mitologia grega encarregado em atravessar o rio Styx (Estige) guiando as almas que foram enterradas com uma moeda debaixo da língua para pagar a viagem (ROCHA, 2002). Esse nome foi escolhido pois a abordagem se propõe a guiar os desenvolvedores através de um processo para a construção de software.

reutilizáveis que solucionem problemas de domínios de conhecimento específicos, e de Engenharia da Aplicação, que tem o objetivo de construir aplicações em um determinado domínio, reutilizando os componentes genéricos já existentes. O Ambiente Odyssey utiliza extensões de diagramas da UML para representar o conhecimento de um domínio, e permite que esses diagramas sejam reutilizados para facilitar a construção de aplicações.

1.3 – Organização

Este trabalho está organizado em 5 capítulos. Neste primeiro capítulo, de introdução, são exibidos a motivação, o objetivo e a organização do trabalho.

No segundo capítulo são apresentadas algumas abordagens sobre automação de processos de software existentes na literatura. Essas abordagens são comparadas segundo critérios propostos dentro do próprio capítulo.

No terceiro capítulo é exibida a abordagem proposta, que consiste na construção de uma máquina de processos de software baseada na tecnologia de agentes inteligentes, que atenda aos critérios definidos no segundo capítulo.

No quarto capítulo é detalhado o protótipo que implementa a abordagem proposta. Para exemplificar o uso do protótipo, é utilizado o processo que descreveu a construção do um sistema para o controle acadêmico do Programa de Engenharia de Sistemas e Computação da COPPE / UFRJ.

No quinto capítulo são apresentadas as contribuições e limitações desse trabalho, assim como os trabalhos futuros.

Capítulo 2 – Automação de Processos de Software

2.1 – Introdução

Este capítulo apresenta algumas abordagens para automação de processos de software descritas na literatura. Essas abordagens são comparadas através de critérios que servirão também para uma posterior comparação com a abordagem proposta neste trabalho. Esses critérios foram identificados através da análise das deficiências encontradas nas próprias abordagens. Essas abordagens podem ser divididas em três tipos (JOERIS et al., 1998):

- Baseadas em máquinas de estado ou redes de Petri;
- Baseadas em agentes, regras ou scripts;
- Híbridas, que combinam notação gráfica com regras.

Esses três tipos se diferenciam pela forma que representam o grafo de descrição da interação entre os processos, conhecido como *workflow*[2]. Essas representações de *workflow* podem ser vistas sob a perspectiva da proximidade delas com a representação de conhecimento utilizada por seres humanos ou por computadores, como exibido na Figura 2.1.

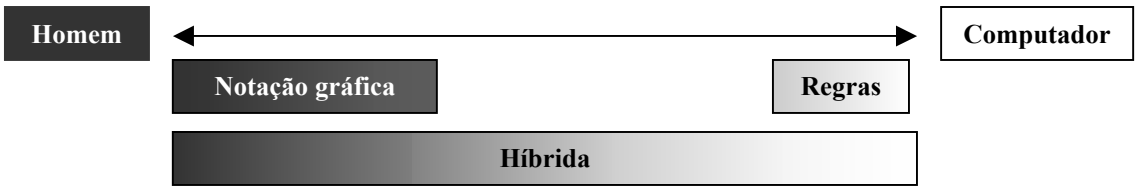

Figura 2.1: Nível de abstração das representações de processo.

A representação baseada em notação gráfica facilita a modelagem mas dificulta a execução por uma máquina, devido a falta de estrutura e formalismo da notação. A representação baseada em regras facilita a execução mas dificulta a modelagem por seres humanos, devido a dificuldade de formalização do conhecimento. A representação

[2] A WfMC (WFMC, 2002) define *workflow* como sendo a automação de um processo. Entretanto, neste trabalho utilizamos esse termo para referenciar a seqüência de atividades que compõem um processo.

híbrida utiliza algum formalismo que permita a modelagem gráfica do *workflow* combinada a um mapeamento que transforma o *workflow* modelado em regras que são interpretadas por uma máquina de inferência para executar o processo.

2.2 – Abordagens baseadas em máquinas de estado ou redes de Petri

As abordagens baseadas em máquinas de estado ou redes de Petri dão ênfase à modelagem gráfica do processo, entretanto dificultam a modelagem de dependências específicas entre atividades (JOERIS et al., 1998), pois usualmente a semântica dos seus modelos é pobre. A execução do processo normalmente ocorre no próprio *workflow*, o que aumenta o acoplamento entre os dados do processo e da sua execução.

Exemplos de abordagens desse tipo são: SPADE, Memphis e ProSoft.

2.2.1 – SPADE

A abordagem especificada no contexto do projeto SPADE (BANDINELLI et al., 1993; BANDINELLI et al., 1994) consiste em definir, analisar e executar processos de software utilizando a linguagem SLANG (SPADE *language*), que é baseada em um formalismo de alto nível para redes de Petri.

A linguagem SLANG é dividida em duas partes: ***Kernel* SLANG**, que fornece os mecanismos básicos para a modelagem de processo; e ***Full* SLANG**, ou somente **SLANG**, que fornece construtores de alto nível baseados nos construtores básicos da *Kernel* SLANG. O ambiente SPADE fornece mecanismos para a execução de modelos definidos na linguagem SLANG, através da criação e modificação de artefatos manipuláveis por processos.

A interação entre os desenvolvedores e as ferramentas ocorre através do DEC FUSE (COMPAQ, 2001). DEC FUSE é um software que fornece serviços para a integração de ferramentas através de uma interface de programação que define um protocolo para a cooperação entre as ferramentas. Como produto externo também é utilizado o banco de dados orientado a objetos O2 (DEUX, 1991) para o armazenamento dos artefatos de software. As ferramentas também podem fazer acesso ao banco de dados para compartilhar informações com os processos.

O SPADE define uma estrutura hierárquica baseada em tipos para permitir a construção dos modelos de processo. Os seus tipos são correspondentes às classes existentes no banco de dados O2 ou a combinação delas, através dos tipos compostos

também definidos no O2. Entretanto, o tipo *token* é um tipo predefinido que serve como raiz na estrutura hierárquica de tipos.

O *token* tem como subtipos os seguintes elementos: atividade, meta-tipo, cópia ativa e tipos de modelo. A atividade representa um processo composto por sub-processos; o meta-tipo representa um processo que pode manipular processos como artefatos, o que é útil para definir o processo que rege a evolução de processos; a cópia ativa permite a modificação do processo; e os tipos de modelo permitem a definição de tipos pelo usuário.

Além do *token*, existem outros elementos no modelo de processos do SPADE, que são: **lugares**, que representam repositórios de *tokens* de um determinado tipo; **transições**, que representam eventos que ocorrem em tempo zero, mas têm associados a si uma condição de guarda e uma ação; e **arcos**, que representam fluxos de um ou mais *tokens* que podem ser calculados de forma estática ou dinâmica. O cálculo dinâmico do número de *tokens* que fluem pelo arco pode ser útil para situações onde é necessário determinar um valor diferente para cada ocorrência de evento, segundo um determinado requisito.

A *Full* SLANG define alguns refinamentos sobre os elementos do *Kernel* SLANG como, por exemplo, arcos somente de leitura e o uso de hierarquias de atividades.

Desta forma, o SPADE tem como característica principal a utilização da linguagem SLANG, que é dividida em dois níveis de abstração, o que facilita uma possível extensão do ambiente. Entretanto, ele não se preocupa com questões referentes a reações pró-ativas da máquina de processo. Dentre os trabalhos futuros mencionados (BANDINELLI et al., 1994), encontra-se a possibilidade de escrever as definições das atividades usando uma linguagem declarativa. Outro trabalho realizado antes do encerramento do projeto, que ocorreu em 1995, foi a inclusão de um editor gráfico para a modelagem do processo, chamado de SLANG Editor (CATTANEO et al., 2002).

2.2.2 – Memphis

No ambiente Memphis (ROCHA et al., 1996; ARAÚJO, 1998; VASCONCELOS et al., 1998) os processos são descritos através de padrões (GAMMA et al., 1994), visando a sua reutilização. Os padrões de processos documentam informações referentes à identificação, ao contexto, à solução proposta e aos padrões relacionados. As informações do padrão sobre a solução proposta são descritas através

de um diagrama multi-nível, não cíclico, que exibe as atividades e suas dependências. A execução do processo é baseada em uma máquina de transição de estados. Os estados possíveis para cada atividade são:

- **Planejamento:** A atividade está sendo estruturada (criada) ou reestruturada (modificada) devido a necessidades encontradas durante a execução;
- **Pronta-início:** A atividade está pronta para entrar em execução, pois todos os seus pré-requisitos foram satisfeitos;
- **Ativada:** A atividade está em execução;
- **Suspensa:** A atividade, que foi previamente colocada em execução, está suspensa;
- **Pendente:** Nem todos os pré-requisitos da atividade foram satisfeitos;
- **Pronta-fim:** A atividade está terminada, com todos os documentos produzidos, entretanto espera uma ativação (aprovação da gerência);
- **Pronta:** A atividade terminou;
- **Cancelada:** A atividade foi cancelada.

Neste modelo, atividades manipulam recursos humanos, documentos (consumidos e produzidos), ferramentas e métodos. O mapeamento entre o processo modelado e o processo executável não é necessário, pois ambos usam a mesma representação interna.

A grande contribuição desse trabalho está no suporte à reutilização de processos através de padrões. Entretanto, a falta de uma separação entre a representação do processo modelado e os dados sobre a sua execução torna difícil a extensão da abordagem, pois a alteração nos dados da execução deve ser feita dentro do elemento que representa o próprio processo, e o fato de não ser possível utilizar ciclos no *workflow* de processo dificulta a modelagem de diversos ciclos de vida utilizados com freqüência no desenvolvimento de software.

2.2.3 – ProSoft

O projeto ProSoft (REIS et al., 1999; REIS et al., 2001a; REIS et al., 2001b) apresenta um ambiente de desenvolvimento de software baseado em especificações formais, onde as suas ferramentas são chamadas de ATOs (Ambiente de Tratamento de

Objetos). O ATO responsável pela máquina de processo se comunica com o ATO responsável pelo trabalho cooperativo para determinar quais artefatos podem ser acessados por determinados desenvolvedores.

O ciclo de vida de processos de software no ProSoft é composto pelas seguintes etapas: (1) O ATO da máquina de processos recebe como entrada um processo modelado e inicia a execução; (2) os desenvolvedores interagem com o ATO da máquina de processos para dar andamento a execução do processo; (3) o ATO da máquina de processos interage com o ATO de trabalho cooperativo para controlar o acesso dos desenvolvedores aos artefatos do sistema; e (4) o ATO de trabalho cooperativo interage com os outros ATOs existentes, tais como ATO Diretório, para permitir a manipulação dos artefatos pelos desenvolvedores. Como o desenvolvimento de software é uma atividade social onde pessoas interagem para atingir um único objetivo, torna-se necessário utilizar mecanismos que apóiem essa interação, que é o caso do ATO de trabalho cooperativo.

O ATO da máquina de processo foi definido utilizando método algébrico de especificação formal. Este ATO baseia a execução do processo em uma máquina de estados. Os estados existentes são:

- **Esperando:** quando a atividade está aguardando que suas antecessoras terminem;
- **Pronta:** quando todas as atividades anteriores já concluíram;
- **Ativa:** quando a atividade está em execução;
- **Parada:** quando todos os agentes envolvidos na mesma solicitaram pausa na atividade;
- **Completa:** quando todos os agentes envolvidos na mesma solicitaram seu término, ou quando a atividade é automática e completou sua função.

Os demais elementos existentes na modelagem de processos do ProSoft são: cargos (papéis), agentes, recursos, ambiente cooperativo (responsável por restringir o acesso de desenvolvedores a artefatos), projeto e processos definidos.

Como na maioria das abordagens baseadas em máquinas de estado, o ProSoft não tem mecanismos nem para facilitar a extensão do sistema, pois existe um grande acoplamento entre os dados do processo e os dados da execução, nem para fazer com que a execução do processo ocorra de forma pró-ativa, pois uma modificação de estado

do processo só ocorre caso haja algum evento externo. A execução pró-ativa existe quando, independentemente de interações externas que fazem a máquina de processos reagir, ela decide que algo deve ser feito, sendo o agente motivador da ação.

2.3 – Abordagens baseadas em agentes, regras ou scripts

As abordagens baseadas em agentes, regras ou scripts fornecem a flexibilidade necessária para a construção de qualquer tipo de *workflow*, entretanto dificultam o projeto do mesmo, pois a ausência de um ambiente gráfico para modelagem torna o seu entendimento complexo. Em outros casos, a ferramenta contém o *workflow* embutido no seu código, não permitindo a criação ou modificação do processo.

Exemplos desse tipo de abordagens são: HyperCode, Assistente inteligente ao processo do método Fusion e CAGIS.

2.3.1 – HyperCode

O HyperCode (PERRY et al., 1996) é uma ferramenta voltada para a automação do processo de inspeção de software distribuído. Como motivação para a sua construção é descrito um conjunto de características que são obtidas pelo uso de *workflow* e automação de processos:

- **Estimula o projeto e desenvolvimento concorrente:** diminui o tempo ocioso existente em um desenvolvimento puramente seqüencial;
- **Reduz o tempo gasto com papel:** Quando a informação não está incorporada no sistema, torna-se necessária uma infra-estrutura externa para manipulação dessa informação. Essa infra-estrutura, composta por lugares físicos para o armazenamento de papéis e pessoas para o controle de acesso aos papéis, usualmente cria burocracia para o acesso aos papéis;
- **Suporta equipes de desenvolvimento distribuídas:** Atualmente é comum o desenvolvimento por grupos terceirizados ou equipes geograficamente distribuídas. Essas equipes necessitam de mecanismos de comunicação que permitam o agendamento de reuniões e suporte para a execução das mesmas.

A ferramenta HyperCode foi desenvolvida para ser executada pela Internet, através de um conjunto de scripts CGI.

São fornecidas várias funcionalidades para suportar o processo de software distribuído, como: **segurança**, através de senha para acesso ao sistema; **preparação de documentos**, através de acesso à ferramenta de CVS para recuperar a documentação necessária para inspeção; **administração de processos**, permitindo atribuir código fonte para outros participantes fazerem revisão; **anotações em documentos**, possibilitando a comunicação sobre o que o participante concluiu durante a sua revisão; e **arquivamento**, mantendo informações sobre inspeções anteriores.

O ponto forte dessa abordagem está no fato da ferramenta em questão não ser um protótipo. Ela está sendo usada em ambiente real pela Lucent Technologies (LUCENT TECHNOLOGIES, 2002), que possui certificado ISO-9000. Além disso, permite o uso distribuído em ambiente global, através da Internet. Entretanto, é focada somente para o processo de inspeção, não oferecendo suporte para a modelagem de outros tipos de processos de desenvolvimento de software. Por esse motivo, a extensão da ferramenta também fica comprometida, pois o processo está descrito implicitamente no código fonte da abordagem, misturando completamente as informações do ambiente com as informações de descrição do processo.

2.3.2 – Assistente inteligente ao processo do método Fusion

O assistente inteligente para o processo de software do método Fusion (SANTANDER et al., 1997) se baseia em um sistema de planejamento de inteligência artificial. Inicialmente, o processo do método Fusion (COLEMAN et al., 1994), que é a fusão dos métodos formais, OMT, CRC e Booch, foi descrito através de um conjunto de operadores. Esses operadores contêm um objetivo, precondições, sub-objetivos, restrições e efeitos.

O objetivo dos operadores indica o que o operador adicionará como efeito quando for executado. Todavia, a sua entrada em execução só pode ser efetuada se as precondições e restrições forem satisfeitas. Quando um operador é complexo, pode ser subdividido em outros operadores através de uma estrutura hierárquica, utilizando os sub-objetivos, que serão os objetivos de cada sub-operador.

A modelagem do método Fusion, que inclui os modelos de objetos, de ciclo de vida e de operações, grafos de interação de objetos, de visibilidade e de herança e descrição das classes, foi efetuada através de 74 operadores, em 3 ou 4 níveis de abstração.

Durante o acompanhamento da execução do processo, quando um desenvolvedor deseja realizar uma atividade, o reconhecedor de planos verifica se essa atividade já foi realizada. Caso a atividade não tenha sido realizada, todas as precondições e sub-objetivos da atividade são verificados, traçando um plano de execução das pendências para que a atividade principal possa ser executada.

A abordagem foi implementada utilizando Prolog, através do mapeamento dos operadores e dos algoritmos de assistência inteligente. Esses algoritmos ajudam desenvolvedores inexperientes durante a aplicação de heurísticas e verificação da consistência dos modelos desenvolvidos para a construção de um sistema. Para exemplificar a utilização da ferramenta, foi utilizado um sistema de caixa eletrônico.

A abordagem apresenta uma estrutura pró-ativa, que verifica as pendências de uma atividade, traçando o seu plano de execução, para que seja possível permitir a execução da atividade em questão. Entretanto, apesar do processo poder ser reutilizado para vários projetos diferentes, não é possível reutilizar parte do processo como atividade em outro processo.

Para que outro processo seja utilizado pela abordagem, seria necessário definir, através dos operadores, esse novo processo e mapear essa definição para o Prolog, juntamente com os demais algoritmos. Desta forma, todo o mapeamento ocorreria de forma manual para cada novo processo utilizado ou para cada mudança ocorrida no processo atual.

Como toda a estrutura do processo está desacoplada do gerador de planos, a sua extensão é facilitada, pois é possível criar um novo operador que consulte o estado atual da execução e execute o seu objetivo sem que seja necessário fazer alterações nos predicados Prolog que representam o processo.

2.3.3 – CAGIS

Em CAGIS (WANG et al., 1999; WANG, 2000; WANG et al., 2001) é descrita uma arquitetura multi-agente para engenharia de software cooperativa. Para justificar a existência dessa arquitetura, o trabalho cooperativo é dividido em 4 categorias (WANG et al., 1999):

- **Trabalho cooperativo ad-hoc:** Atividades informais, como reuniões, *brainstorming* e aprendizado cooperativo;
- ***Workflow* predefinido:** Uso de fluxo de documentos através de um processo;

- ***Workflow* coordenado:** Implementado através de sistemas de desenvolvimento de software centralizados;
- ***Workflow* cooperativo:** Implementado através de sistemas de desenvolvimento de software descentralizados.

Dentro dessas categorias, a engenharia de software cooperativa se enquadra nas duas últimas, pois graças à Internet, as empresas estão migrando de uma estrutura centralizada para uma estrutura descentralizada, que demanda melhor suporte à comunicação, e, conseqüentemente, à cooperação. O objetivo desse trabalho foi integrar as tecnologias de processo de software e trabalho cooperativo em uma arquitetura multi-agente, visando um ganho em simplicidade e flexibilidade.

Esse ganho de simplicidade e flexibilidade ocorre através das seguintes características dos sistemas multi-agentes (WANG et al., 1999): **descentralização**, que torna mais simples a construção do sistema, a partir de um conjunto de sub-sistemas descentralizados; **reutilização**, construindo um grande sistema através da interconexão de sub-sistemas existentes, mesmo que esses sejam heterogêneos; **trabalho cooperativo**, modelando as relações de forma similar com a dos humanos, visto que um agente inteligente age de forma interativa e autônoma, representando os interesses de humanos; **flexibilidade**, sendo capaz de lidar com especificações incompletas e evolução constante, que são características próprias de sistemas distribuídos.

A arquitetura multi-agente proposta é composta de quatro componentes:

- **Agentes:** Software que age de forma autônoma, reativa, pró-ativa e interativa em busca de um objetivo. São divididos em agentes do sistema, que tratam da administração do sistema multi-agente, agentes locais, que tratam de problemas locais dos *workspaces*, e agentes de interação, que ajudam os participantes no trabalho cooperativo;
- ***Workspaces*:** Espaço compartilhado onde humanos e agentes acessam informações e recursos. Essas informações podem ser arquivos compartilhados e os recursos podem ser ferramentas de desenvolvimento de software. Desta forma, os *workspaces* são lugares onde os agentes e os humanos podem interagir. Eles podem ser implementados de forma simples, através do sistema de arquivos do sistema operacional;

- **Ágora:** Espaço em que agentes se encontram para interagir. A interação dos agentes consiste na negociação de informações e serviços. Uma ágora pode ser dividida em duas funcionalidades: comunicação entre agentes, que consiste em um espaço para agentes anunciarem as suas habilidades e procurarem outros agentes capazes de realizar determinadas tarefas, e negociação entre agentes, que consiste em um espaço que diminui os *overheads* da comunicação entre os agentes, permitindo que esses se concentrem nas suas estratégias de negociação;
- **Repositórios**: Servidor de informações com o objetivo de armazenar e recuperar dados persistentes. Dados de projetos anteriores podem ser armazenados nos repositórios para permitir a coleta de estimativas para novos projetos.

A implementação dessa estrutura multi-agente no CAGIS se chama DIAS, que significa sistema de agentes inteligentes distribuído (*Distributed Intelligent Agent System*). Essa implementação se baseia em Aglets (IBM, 2001), que é um ambiente para programação de agentes inteligentes em Java, criado pela IBM. A arquitetura proposta foi utilizada em um processo de desenvolvimento e manutenção de software de uma empresa norueguesa, identificada como AcmeSoft (WANG et al., 1999).

Esse trabalho traz o uso de agentes inteligentes para o processo de desenvolvimento e manutenção de software sob a perspectiva de representar as pessoas envolvidas em determinadas situações que demandam negociação. A abordagem não usa os agentes com o objetivo de implementar uma máquina de processos, mas sim um ambiente para a cooperação durante o ciclo de vida do processo. Desta forma, questões como a modelagem gráfica do processo ficam fora do escopo desse trabalho, além de qualquer questão relacionada com o processo propriamente dito. O foco da abordagem não está diretamente no processo de software, mas sim em como cooperar durante a execução desse processo.

Uma deficiência dessa abordagem é não ter o seu foco em processo de software, o que, em última análise, faz com que a comunicação entre ferramentas específicas para o desenvolvimento de software seja prejudicada. Também não existe uma preocupação em definir os processos de forma que incentive a reutilização.

2.4 – Abordagens híbridas

Nessas abordagens encontramos um ambiente propício para a modelagem e um mecanismo que permite a transformação do *workflow* modelado em regras de evento-condição-ação, que são mais facilmente tratadas por computadores no momento da execução do *workflow*.

Exemplo desse tipo de abordagens são: EPOS, Merlin, ProNet / ProSim, Dynamite, APEL e Mokassin.

2.4.1 – EPOS

O ambiente EPOS (JACCHERI et al., 1992) utiliza a orientação a objetos no seu modelo de dados para a representação de processos de software. Para permitir a evolução do processo modelado e das instâncias em execução do processo, é utilizado um banco de dados com suporte a versões.

A idéia principal do EPOS é a junção de modelagem de processos com gerência de configuração, através do uso de grafos e regras para a modelagem, armazenados no banco de dados EPOSDB, que fornece interface de programação para Prolog.

Os modelos de processo do EPOS são compostos por: **modelo de atividades**, que descreve tarefas que devem ser executadas por pessoas ou por ferramentas; **modelo de produto**, que estrutura o sistema; **modelo organizacional**, que representa a organização das pessoas agrupadas em equipes; e **modelo de ferramentas**, que descreve as ferramentas disponíveis.

Os modelos de processo são traduzidos em tuplas, que serão inseridas no banco de dados do EPOS. Após essa etapa, pode ocorrer a instanciação automática do processo. A instanciação do processo ocorre de forma incremental, permitindo que as associações entre os processos modelados nos grafos e os seus meta-processos aconteçam em tempo de execução. O meta-processo, segundo essa abordagem, é o processo modelado, que pode ter várias instâncias em execução.

O banco de dados com suporte a versões permite que esses processos sejam refinados, mesmo se fizerem parte de instâncias em execução. O refinamento pode ser no processo em si, que refletirá em todas as instâncias existentes, ou na instância de um determinado processo em execução, que não implicará na modificação do meta-processo e nas demais instâncias. Esse banco de dados é implementado através de uma variante do modelo entidade-relacionamento, estendido para suportar os conceitos da

orientação a objetos. O controle de transações implementado nesse banco de dados permite que transações perdurem por várias seções de aplicações.

Cada tarefa do modelo de processo é composta por estas propriedades: **pré-condições e pós-condições**, que são fórmulas de lógica de primeira ordem utilizadas para descrever o estado das tarefas tanto de forma estática, no momento da sua instanciação, quanto de forma dinâmica, antes e depois da sua execução; **código**, que é descrito de forma declarativa através do uso de Prolog, permitindo configurar o comportamento da tarefa de forma que o seu estado final fique condizente com a sua fórmula de pós-condição; **decomposição**, que fornece a rede de sub-tarefas que compõem a tarefa atual; e **produtos**, que descrevem os produtos consumidos e produzidos pela tarefa.

Quando o código de uma tarefa está vazio, a máquina de processos procura as sub-tarefas descritas pela decomposição da tarefa atual e inicia as suas execuções recursivamente.

O EPOS fornece mecanismos interessantes para a evolução do processo modelado. Esses mecanismos integram um ambiente de modelagem de processos com um gerente de configuração. Entretanto, não são descritos mecanismos que facilitem a extensão da própria máquina de processos.

Não são também fornecidos mecanismos para a reutilização dos processos modelados em contextos diferentes. A mera instanciação do processo em diversas aplicações não é suficiente para afirmar que o ferramental provê suporte para a reutilização do processo em outros contextos, que podem ser vistos como outros grafos de sub-processos.

2.4.2 – Merlin

Merlin (PEUSCHEL et al., 1992; JUNKERMANN et al., 1994) é um ambiente de desenvolvimento de software que utiliza uma base de conhecimento para suportar o controle da execução de processos.

Inicialmente, um engenheiro de processos deve definir o processo desejado para um determinado projeto, através de uma notação gráfica própria combinada com lógica de predicados e diagramas de transição de estados. Após esse passo, ocorre um mapeamento em regras e fatos Prolog, que povoarão a base de conhecimento do processo e servirão como insumo para a máquina de processo.

O meta-modelo do Merlin é composto por atividades, papéis, documentos e recursos. Desses elementos, existe um foco maior na modelagem e controle de documentos.

O núcleo do sistema contém um conjunto de regras responsáveis pela execução de processos. Na base de conhecimento ficam os fatos e regras que mapeiam os processos modelados. Segundo a concepção do projeto, esse núcleo do sistema não sofrerá mudanças e será utilizado como uma máquina de inferência especializada em processos de software.

Uma das regras definidas no núcleo do sistema trata da localização dos documentos disponíveis para um desenvolvedor que cumpre um determinado papel e está acessando o ambiente. Para que esse tipo de regra possa ser definida, é necessário que um conjunto de fatos descreva em detalhes os documentos existentes e os seus relacionamento com o restante dos elementos do ambiente. Desta forma, para definir detalhadamente os documentos, foram usados os seguintes predicados:

- **Associação de ações a documentos:** Regras que permitem associar ações a documentos que se encontram em determinados estados. Essas ações englobam a verificação de condições de consistência e execução de condições automatizadas, que serão descritas posteriormente;
- **Definição dos estados de documentos:** Fatos que permitem detalhar cada estado possível para um determinado documento. Essas possíveis variações de estados do documento também devem ser descritas no diagrama de transição de estados;
- **Associação entre ferramentas e documentos:** Fatos que permitem associar ferramentas que devem ser executadas para manipular determinados documentos. Nessa descrição também deve ser relatada a condição do artefato para permitir o uso da ferramenta (leitura e escrita ou somente leitura) e os estados iniciais e finais do documento. O ambiente executará a ferramenta e, posteriormente a sua execução, mudará o estado do artefato para um dos estados finais permitidos;
- **Associação entre tipos de documentos:** O diagrama gráfico permite que sejam criadas relações entre documentos. Para que seja possível utilizar essas relações na base de conhecimento, existe um mapeamento para elas

em fatos. Esses fatos englobam a relação, o documento de origem e o documento de destino;

- **Condições de consistência:** Fatos que permitem descrever regras de transições de estados de determinados documentos em função de outras transições de estado de outros documentos. Dentro do ambiente gráfico de modelagem do processo, é possível utilizar essas regras, que são escritas em lógica de primeira ordem;
- **Condições automatizadas:** Fatos que permitem descrever regras de transições automatizadas entre estados de documentos. Essas regras também podem ser escritas dentro do ambiente gráfico de modelagem do processo;
- **Acesso de papéis a documentos:** Fatos que permitem descrever quais papéis podem acessar quais documentos e em quais estados esses documentos devem estar para que possam ser acessados por esses papéis. Além disso, é possível determinar um conjunto de pré-condições de estado de outros documentos.

Com essas características de mapeamento é possível representar os elementos modelados em Prolog. Entretanto, o ambiente disponível para a modelagem de processo não é muito intuitivo, pois combina uma notação estendida do modelo entidade-relacionamento, que engloba o uso de lógica de predicados para descrever condições de consistências e condições automatizadas, com diagramas de transição de estados.

Um aspecto interessante dessa abordagem está na parte de trabalho cooperativo. Ela incorpora a chamada de ferramentas para a manipulação de artefatos, e, juntamente com essa chamada, são utilizados mecanismos de controle de transação. Esses mecanismos permitem a seleção do tipo de abordagem desejada no caso de documentos únicos, que pode ser a pessimista ou a otimista. No caso de contextos que englobam vários documentos, a abordagem de controle de transação tem que ser a pessimista, pois seria indesejado perder todo o trabalho por causa de um *rollback*[3] em todos os

[3] O *rollback* é um procedimento onde os dados retornam para o último estado considerado seguro. Este conceito é utilizado em transações, juntamente com os conceitos de *start transaction*, que inicia a transação marcando o estado atual como seguro, e *commit*, que autoriza a remoção da marca de antigo estado seguro, pois a transação ocorreu com sucesso (ELMASRI et al., 1994).

documentos do contexto, proveniente do uso de uma abordagem otimista. Esse controle de transação é definido no nível de abstração das regras Prolog do núcleo do ambiente.

Apesar de utilizar mecanismos sofisticados para o controle de transação, a abordagem é puramente reativa, pois não descreve a utilização de elementos pró-ativos que, seguindo um determinado objetivo, façam inferência de forma autônoma na base de conhecimento do processo. Não é também descrita nenhuma preocupação na construção de mecanismos que permitam a reutilização dos processos modelados em outros contextos.

2.4.3 – ProNet e ProSim

Em ProNet e ProSim (CHRISTIE, 1995), é descrito um modelo de processo de software centrado no fluxo de atividades. O modelo é composto por:

- **Atividades:** elementos principais do modelo. Podem ser executáveis ou um sub-processo, contendo um outro conjunto de atividades;
- **Agentes:** são entidades responsáveis por executar uma atividade. Um agente não precisa ser necessariamente uma pessoa (ex.: um compilador);
- **Artefatos:** podem ser consumidos ou produzidos por atividades. São obtidos de outra atividade do processo, externamente ao processo ou de um repositório (descrito a seguir);
- **Condições:** possuem valores booleanos (true ou false) e podem preceder ou suceder a uma atividade. Sua função é controlar a possibilidade de execução de uma atividade;
- **Combinações:** fazem combinações lógicas entre entidades do modelo. As combinações possíveis são CA (‘e’ convergente), CO (‘ou’ convergente), DA (‘e’ divergente) e DO (‘ou’ divergente). As combinações convergentes servem para atribuir a uma atividade mais de um pré-requisito, e as combinações divergentes servem para atribuir múltiplas saídas após a execução de uma atividade;
- **Repositórios:** Permitem o armazenamento de elementos do modelo para que sejam reaproveitados em um outro momento da execução;
- **Restrições:** diferentemente das condições, não são representadas como restrições booleanas. Seu objetivo é indicar como alguns procedimentos devem ser executados.

A partir de um processo modelado de acordo com a notação definida pelo ProNet, o ProSim pode ser utilizado. O ProSim é responsável pela execução do processo. Entretanto, para que o ProSim execute o processo, tem que existir uma transformação do processo gráfico em fatos Prolog. O mapeamento em fatos Prolog não é automático, tendo que ser feito manualmente. Para cada situação possível, existe uma regra que deve ser seguida para permitir esse mapeamento.

A atribuição dos desenvolvedores aos papéis também deve ser feita manualmente, inserindo fatos Prolog na base de conhecimento. A interface de execução utilizada pelo ProSim é o próprio ambiente Prolog, e não tem o suporte das ferramentas existentes em um ambiente de desenvolvimento de software.

Apesar do ProSim utilizar Prolog, que é uma linguagem própria para a construção de bases de conhecimento, ele mistura a execução com a definição do processo, pois o processo não é descrito através de fatos Prolog independentes dos fatos que descrevem o mecanismo de execução. Eles são construídos como regras que contêm tanto a definição quando a ordem de execução do processo, tornando difícil a implementação de predicados que permitam a inferência do estado atual da base ou mesmo a extensão da ferramenta para suportar, por exemplo, a coleta de métricas de execução.

2.4.4 – Dynamite

O projeto Dynamite (HEIMANN et al., 1996) utiliza a combinação da facilidade provida por um ambiente gráfico para a modelagem de processos com o poder de uma linguagem para a representação e execução do processo modelado. A linguagem utilizada para esse fim é a Progres (SCHÜRR, 1989). Com o uso dessa linguagem, são alcançados alguns requisitos de dinâmica de processos:

- **Encaminhamento do desenvolvimento:** permitir a configuração de dependências de execução de tarefas em função de artefatos produzidos em tarefas anteriores;
- **Engenharia concorrente:** com o objetivo de encurtar o ciclo de desenvolvimento, devem ser utilizados métodos que estimulem o desenvolvimento concorrente de software, produzindo artefatos o mais rápido possível para permitir que outras tarefas possam entrar em execução de forma concorrente;

- **Retorno:** permitir a utilização de fluxos de retorno, caracterizando um ciclo na rede de processo, em situações onde alguma tarefa que já foi executada deve ser executada novamente. Um exemplo desse caso seria a reimplementação de um módulo devido a um erro encontrado na tarefa de teste.

A abordagem Dynamite permite a utilização do conceito de interface e realização, amplamente utilizado na orientação a objetos, para a separação entre o que a tarefa deve fazer e como ela deve fazer. Desta forma, as dependências entre tarefas podem passar a ser descritas como dependências de uma tarefa com uma interface, e várias outras tarefas podem realizar (responder por) essa interface. Esse tipo de característica facilita a reutilização de tarefas em contextos diferentes.

Esta abordagem também prevê a existência de versões para as tarefas de uma rede. Essas versões podem existir combinadas a um fluxo de retorno, que obriga a re-execução de uma determinada tarefa. A notação utilizada para representar as várias versões é a sobreposição das tarefas, com as últimas versões desenhadas por cima do desenho das anteriores, com um pequeno deslocamento nos eixos X e Y. Além do aspecto das versões, a abordagem também leva em conta o contexto da sub-tarefa dentro de uma rede de tarefas. Esse contexto é importante para situações onde a tarefa A tem como sub-tarefa a tarefa B, mas B, em algum ponto da sua estrutura de sub-tarefas tem uma referência a tarefa A (por exemplo, A poderia ser uma sub-tarefa de B). Assim não seria possível analisar isoladamente a tarefa B. Seria necessário saber o contexto em que o B está sendo analisado, como por exemplo: A->B ou A->B->A->B, onde X->Y indica que Y é sub-tarefa de X. Esse tipo de discussão sobre o contexto de uma tarefa é necessário para permitir a construção de hierarquias recursivas de sub-tarefas, como exibido na Figura 2.2.

Para que fosse possível a definição de sub-tarefas, esta abordagem classifica as tarefas como atômicas ou complexas. As tarefas complexas contêm uma rede das sub-tarefas que as compõem. A relação entre tarefas e suas sub-tarefas pode ser vista sob 3 perspectivas: (1) a entrada de uma tarefa complexa é passada para as entradas de suas sub-tarefas; (2) a saída de uma tarefa em uma rede é passada para a entrada da tarefa subseqüente; (3) a saída de uma sub-tarefa é passada para a saída da sua super-tarefa.

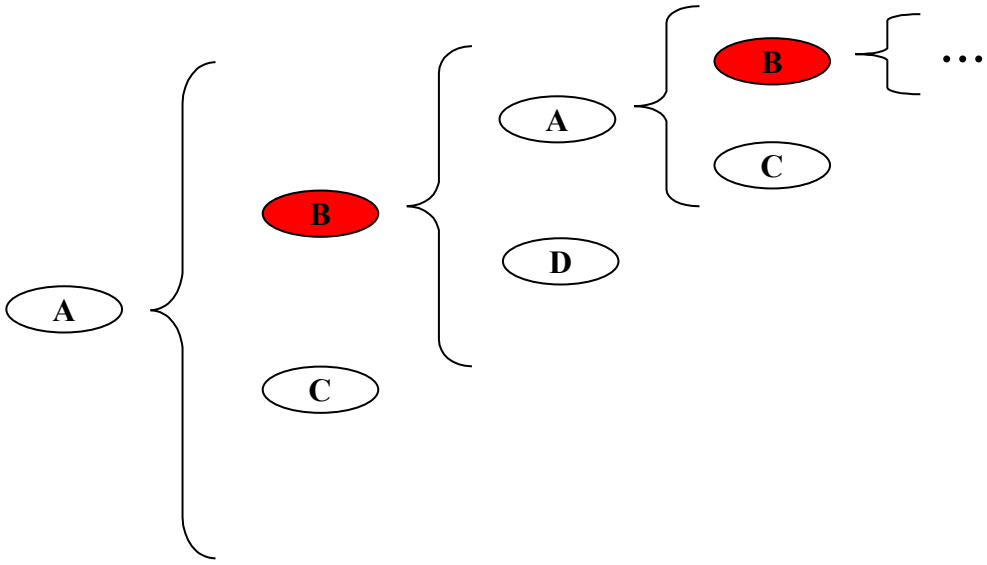

Figura 2.2: Exemplo de hierarquia recursiva com atividade B dependente de contexto.

Esta abordagem também usa uma estrutura fixa de estados para guiar a sua máquina de processos. Os estados existentes são: em definição, quando as tarefas estão sendo criadas; esperando, quando as tarefas foram definidas e podem entrar em execução; ativa, quando as tarefas estão em execução; suspensa, quando as terefas foram interrompidas; executada, quando a execução ocorreu com sucesso; e falhada, quando não foi possível chegar ao final da execução. É permitido que tarefas que estejam no estado 'esperando' possam voltar ao estado 'em definição'. Com isso, é possível fazer mudanças na rede de tarefas durante a execução do processo.

É notada uma grande preocupação com o poder de representação do processo através do uso de fluxos de retorno, versões e contextualização de tarefas. Entretanto, um ponto negativo da abordagem é a não existência de protótipo implementado. Existe somente um modelo formal descrevendo o processo em termos da linguagem Progres.

Apesar de existirem mecanismos para contornar o problema do conjunto de estados fixos, a extensão da ferramenta não se dá de forma natural, pois inexiste uma infra-estrutura voltada para esse fim.

2.4.5 – APEL

APEL (DAMI et al., 1998) fornece uma abordagem que utiliza um ambiente gráfico para a modelagem de processos e transforma esse modelo em uma descrição textual. Essa descrição textual deve seguir a gramática definida por alguma linguagem de processo de software. A partir desse momento, a execução do processo transformado ocorrerá através do ambiente dono da gramática escolhida.

O significado da sigla APEL é linguagem para máquina de processos abstrata (*Abstract Process Engine Language*). Isso indica que na verdade o APEL não é uma máquina de processos, mas sim uma linguagem gráfica que deve ser traduzida para uma linguagem textual formal e executada por uma máquina de processo real. A máquina de processo utilizada pelo APEL é a ADELE (BELKHATIR et al., 1993).

Como motivação para a construção do APEL, é argumentada a necessidade de um alto nível de abstração para a modelagem dos conceitos de processos, a necessidade de um mecanismo de execução do processo que seja altamente configurável e a necessidade do desacoplamento entre a modelagem e a execução, permitindo que o ambiente de modelagem seja independente do formalismo e da plataforma necessária para a execução.

Desta forma, o APEL é composto dos seguintes elementos: (1) ambiente de modelagem; (2) tradutor do modelo gráfico para um modelo textual; (3) compilador do modelo textual para um formalismo executável (atualmente o formalismo utilizado é o ADELE); (4) uma máquina de processo que execute o formalismo compilado (atualmente é utilizado o ADELE); e (5) um conjunto de serviços acoplados à máquina de processos que permitam desde a interação com usuários até o monitoramento do processo em execução (ainda não implementado).

O modelo de processos utilizado no APEL é simples, contendo como elementos a atividade, que pode ser atômica ou composta, produtos de entrada e de saída e o agente que executará a atividade.

Para a modelagem dos processos são utilizados três aspectos:

- **Fluxos de controle:** que indicam quais atividades devem ser executadas após a execução de uma determinada atividade;
- **Fluxos de dados:** que indicam onde os artefatos necessários para uma determinada atividade foram produzidos;
- **Diagramas de estado:** que indicam como agentes, produtos e atividade evoluem com o passar do tempo.

Os estados possíveis para uma atividade são: inativa, ativa, suspensa, abortada e terminada. Para cada atividade, um conjunto de funcionalidades para a manipulação dos artefatos é habilitado. Através do paradigma de desktop, o engenheiro de processos pode restringir essas funcionalidades, não permitindo, por exemplo, a edição de um

arquivo de código fonte durante uma atividade de teste. As atividades podem ser executadas tanto de forma automatizada quanto por pessoas.

Os artefatos seguem um modelo de circulação de dados que prevê os seguintes modos: **transferência**, onde o artefato é movido de uma atividade para outra, não estando mais disponível na atividade de origem; **compartilhamento**, onde o artefato passa a estar disponível na atividade destino e na atividade origem, sendo necessário o controle de concorrência; **cópia**, onde o artefato é copiado para a atividade destino, ficando disponível tanto na atividade origem quanto na atividade destino, mas sem necessitar de controle de concorrência.

O APEL utiliza o paradigma GQM[4] (BASILI et al., 1994) para permitir a coleta de métricas referentes ao produto e ao processo. Para essa coleta são definidos: o objeto, o propósito, o foco, o ponto de vista, e o ambiente. APEL fornece um diagrama para a construção do modelo GQM, que engloba objetivos, questões e métricas.

Apesar do APEL levantar algumas questões sobre reutilização, este é postergado para trabalhos futuros. A grande ênfase do ambiente está na possibilidade de modelar tanto o fluxo de controle das atividades, que é a essência de uma abordagem guiada por atividades, quanto o fluxo de dados das atividades, que é a essência de uma abordagem guiada por dados. Outra habilidade notável do APEL é a de ser extensível, pois separa as necessidades de modelagem das necessidades da máquina de processos, permitindo que toda a máquina de processo seja substituída por outra através da substituição do tradutor do processo gráfico para o processo textual.

Como a APEL depende de uma máquina de processo externa, o tratamento pró-ativo do processo só será obtido caso essa máquina tenha uma estrutura que favoreça o uso de agentes inteligentes ou outra abordagem autônoma.

2.4.6 – Mokassin

O projeto Mokassin (JOERIS, 2001) descreve uma abordagem que utiliza orientação a objetos para a modelagem e execução de *workflows*, visando atingir um conjunto de requisitos (JOERIS et al., 1998):

---

[4] GQM (*Goal, Question, Metric*) é uma abordagem para medição que se baseia na convicção de que é necessário primeiro especificar os objetivos a serem alcançados, relacionar esses objetivos com dados reais das medições para, finalmente, prover um *framework* para interpretação desses dados (GOMES et al., 2001b).

- **Adaptabilidade a processos heterogêneos:** Visto que vários processos de negócio são heterogêneos, algumas vezes bem estruturados, outras vezes desestruturados, podendo ser executados por pessoas ou de forma automática, e descritos através de diagramas ou regras, a integração desses elementos torna-se muito importante para permitir a construção de *workflows* mais completos.
- **Flexibilidade:** A *priori*, devem existir mecanismos flexíveis de controle para permitir o uso de diferentes políticas de cooperação, levando em conta questões como controle de versões. A *posteriori*, a evolução do *workflow* deve ocorrer de forma flexível, refletindo as modificações nos processos reais, que são as instâncias em execução.
- **Distribuição:** O elemento chave para permitir a escalabilidade de sistemas de execução de *workflow* é a distribuição. A combinação de distribuição com heterogeneidade é o ingrediente necessário para a interoperabilidade entre *workflows* de diferentes empresas.
- **Reutilização:** Com o crescente aumento da complexidade dos modelos de processo, o suporte a reutilização torna-se necessário. Para reutilizar processos, os mesmos devem ser classificados como livres ou dependentes de contexto. Um processo livre de contexto pode ser reutilizado em qualquer situação. Já um processo dependente de contexto só poderá ser reutilizado dentro do contexto em que ele foi projetado. Por exemplo, o processo "Inspeção" pode ser utilizado tanto para especificações de análise quanto para código, entretanto "Inspeção de código" pode ser utilizado somente em um contexto.
- **Requisitos arquiteturais e de sistema:** Para prover requisitos como distribuição, pode ser necessária a construção de uma arquitetura própria para isso. Um exemplo de arquitetura para a construção de *workflows* distribuídos dentro da orientação a objetos seria CORBA.
- **Facilidade de uso:** Para que seja possível um uso real do sistema de gerenciamento de *workflows*, é necessário que a linguagem para descrição de *workflows* seja fácil de usar, permitindo a modelagem em um alto nível de abstração.

Para que seja possível, ao mesmo tempo, ser fácil de usar e fornecer os recursos necessários para a modelagem de *workflows* complexos, esta abordagem permite a descrição de processos através da combinação de regras e grafo de sub-processos.

Cada *workflow* é definido como atômico ou complexo. Um *workflow* atômico contém a descrição do que ele deve fazer, seja um programa, para *workflows* automáticos, ou um roteiro, para ser interpretado pelas pessoas que o executarão. Um *workflow* complexo é composto por um grafo que descreve a interação entre seus sub-processos.

As características de separação entre classes e objetos da orientação a objetos foi utilizada por essa abordagem para permitir a separação entre os níveis de definição de processos e execução de processos. No nível de execução de processos, ocorre a instanciação de elementos descritos no nível de definição de processos.

Dentro do escopo do projeto, existe uma proposta para a integração de agentes inteligentes no suporte à execução do processo (JOERIS et al., 1997a; JOERIS et al., 1997b). Essa proposta visualiza a construção de 3 tipos de agentes: **agente de processo**, responsável pela execução do processo; **agente de repositório**, responsável pelo acesso ao repositório de informações sobre execuções de processos; e **agente de recursos**, responsável por intermediar os acessos de atores ou ferramentas à máquina de processos.

Os agentes propostos devem ser autônomos, sociais, reativos e pró-ativos. Entretanto, essa proposta descreve em alto nível de abstração como os agentes deveriam interagir através de um *framework*, e coloca como trabalhos futuros a implementação desse *framework*.

Esta abordagem, apesar de levantar a importância de reutilização na elaboração do meta-modelo de processos, tem como pontos negativos a não implementação de um protótipo com o *framework* de agentes inteligentes e o fato de não ser focada para processos de desenvolvimento de software.

2.5 – Comparação entre as abordagens

As abordagens descritas neste capítulo são aqui comparadas segundo os seguintes critérios:

- **Modelagem do processo:** Indica se a abordagem contém mecanismos para que o processo seja modelado. Essa modelagem pode ocorrer de forma gráfica ou através de linguagens próprias para descrição de

processos. Quando uma abordagem não atende a esse critério significa que ela tem o seu processo modelado dentro da sua própria especificação, não permitindo a criação de outro processo diferente do inicialmente previsto nem a sua adaptação.

- ***Workflow* com ciclos ou recursão:** Indica se a abordagem permite que o modelo de processo faça uso de ciclos ou recursão na sua representação. Um ciclo é descrito através de um fluxo para uma atividade anterior no fluxo de execução (*workflow*) e uma recursão é descrita através de uma referência no *workflow* a um processo pai na árvore de sub-processos. Ambos ocasionam a repetição da execução de atividades.
- **Mapeamento automático:** Indica se é necessária a intervenção do desenvolvedor de software após a modelagem gráfica do processo, para permitir o seu mapeamento para uma representação executável. A maioria das abordagens usa representações diferentes para a modelagem e a execução dos processos. A não automação desse procedimento pode gerar a introdução de erros e tornar a execução do processo muito trabalhosa para a equipe de desenvolvimento.
- **Interação pró-ativa:** Indica se a abordagem toma a iniciativa de interagir com o desenvolvedor, fornecendo soluções para determinadas situações sem que seja explicitamente solicitada. Este critério visa distinguir abordagens que só reagem a ações do desenvolvedor das abordagens que, com uso de Inteligência Artificial, prevêem situações que ainda não foram vislumbradas de forma autônoma.
- **Processos reutilizáveis:** Indica se a abordagem se preocupa, mesmo que de forma limitada, com a construção de processos que possam ser reutilizados em outras situações. Essa preocupação pode ser verificada, por exemplo, através da implementação de mecanismos que não obrigam o uso de um único nível de abstração para o processo, permitindo que ele seja reutilizado tanto como um processo global quanto como uma atividade específica de outro processo.
- **Extensão flexível:** Permite que a extensão da máquina de processos seja flexível, sendo possível, por exemplo, coletar uma nova métrica sem que

seja necessário implementar esta característica dentro do núcleo do sistema. Para que esse critério seja atendido, a abordagem deve prover algum mecanismo que permita uma extensão dos seus recursos de forma desacoplada.

- **Foco em processos de software:** Indica se a abordagem se restringe a automação de processos de software ou se é aplicável a qualquer tipo de processo. Quando uma abordagem não está focada em processos de software, várias características existentes nos mesmos são desprezadas e características desnecessárias para software, mas necessárias para processos de negócio são incorporadas, dificultando o uso da abordagem no contexto de desenvolvimento de software.
- **Protótipo implementado:** Indica se a abordagem apresenta um protótipo ou se a implementação foi postergada para trabalhos futuros. Abordagens com protótipos implementados usualmente são mais aderentes à realidade, pois mostram como as idéias são passíveis de construção. Obviamente que a partir dessa construção seria necessário realizar validações para se obter indícios de que as mesmas trazem benefícios.

Os critérios acima foram obtidos através do estudo das contribuições e limitações das abordagens descritas nesse capítulo. O fato de um critério não ser atendido por uma determinada abordagem não significa que essa abordagem é pior que as demais, pois para o seu objetivo esse critério pode ser irrelevante. Esses critérios, apesar de não formarem um conjunto completo nem necessariamente suficiente, servem como um guia para a elaboração de uma abordagem que se proponha a ser abrangente e que, se possível, traga novas contribuições em relação às abordagens atualmente disponíveis na literatura, que é o caso da abordagem proposta nesse trabalho.

A Tabela 2.1 classifica as abordagens de acordo com os critérios apresentados. Os campos preenchidos com ✓ indicam que os critérios em questão são cobertos totalmente pela abordagem. Os campos preenchidos com ✗ indicam que os critérios em questão não são atendidos pela abordagem. Os campos preenchidos com ? indicam que não foi possível inferir se o critério é atendido pela abordagem através da documentação disponível.

Tabela 2.1: Quadro comparativo entre as abordagens descritas neste capítulo.

Critérios	Máquinas de estado ou redes de Petri			Agentes, regras ou scripts			Híbridas (Gráfica + Regras)					
	SPADE	Memphis	ProSoft	HyperCode	A.I. P.M. Fusion	CAGIS	EPOS	Merlin	ProNet / ProSim	Dynamite	APEL	Mokassin
Modelagem do processo	✓	✓	✓	✗	✓	?	✓	✓	✓	✓	✓	✓
***Workflow* com ciclos e recursão**	✓	✗	?	✓	?	✗	?	?	✓	✓	✓	✓
Mapeamento automático	✗	✓	✓	✓	✗	?	✓	✓	✗	✗	✓	✓
Interação pró-ativa	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓
Processos reutilizáveis	✗	✓	?	✗	✗	✗	✗	✗	✗	✓	✗	✓
Extensão flexível	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓	?
Foco em processos de software	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗
Protótipo implementado	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗

2.6 – Conclusões

Após análise da Tabela 2.1 podemos chegar a algumas conclusões. Essas conclusões não podem ser consideradas sempre verdadeiras, visto que a tabela não é completa, mas pode servir como heurística para o projeto de uma abordagem mais abrangente a ser proposta:

- As abordagens baseadas em máquinas de estado ou redes de Petri não possuem "Interação pró-ativa", pois a estrutura utilizada não fornece mecanismos que estimulem a ferramenta a buscar pelo seu objetivo sem que o usuário efetue uma ação;
- As abordagens baseadas em máquina de estados são usualmente inflexíveis do ponto de vista de extensão, pois não separam o nível meta do nível instância, o que dificulta a construção de um *framework* que permita a extensão da ferramenta através de módulos não pertencentes ao seu núcleo;
- As abordagens baseadas em agentes, regras ou scripts normalmente não são focadas na reutilização do processo modelado, mas sim no poder de

representação da linguagem formal de modelagem de processos utilizada;

- O simples fato de estar usando regras Prolog não credencia o critério "Interação pró-ativa" à abordagem. Apesar de Prolog ser uma linguagem propícia para construção de bases de conhecimento e para execução de inferências nessas bases, é necessário que exista algum mecanismo externo ao Prolog, que funcione como um temporizador, motivando a execução das inferências que visam o objetivo da ferramenta mesmo que nenhum evento tenha ocorrido por parte dos usuários;
- É possível que uma ferramenta não permita a modelagem do seu processo, mas execute o mapeamento automático, quando essa modelagem do processo existe dentro do próprio código da ferramenta, o que torna desnecessário um mapeamento manual por parte dos usuários para que se dê início à execução;
- Nenhuma das abordagens que têm protótipo implementado atende aos requisitos de "Interação pró-ativa" e "Processos reutilizáveis" concomitantemente.

Além das abordagens descritas neste capítulo existem várias outras, de igual importância, tal como ADELE, ALF, APPL/A, DesignNet, Entity, FunSoft, HFSP, Marvel, MVP-L e Oikos. Comparações envolvendo algumas das abordagens descritas neste capítulo com as demais abordagens podem ser encontradas em *surveys* existentes na literatura (ARMENISE et al., 1993).

A partir desta avaliação, concluímos que nenhuma das abordagens exibidas neste capítulo atende completamente aos critérios propostos anteriormente, o que motiva a definição de uma nova abordagem, que será apresentada no Capítulo 3.

Capítulo 3 – A Máquina de Processos Charon

3.1 – Introdução

Visando a especificação e implementação de uma máquina de processos que atenda aos critérios levantados no capítulo 2, que são: modelagem do processo, *workflow* com ciclo e recursão, mapeamento automático, interação pró-ativa, processos reutilizáveis, extensão flexível e foco em processos de software, optamos pela utilização da tecnologia de agentes inteligentes.

A tecnologia de agentes inteligentes foi inicialmente concebida nos anos 70 (HEWITT, 1977) por pesquisadores da área de Inteligência Artificial, sofrendo influências da programação orientada a objetos, sistemas concorrentes orientados a objetos e projeto de interface homem-máquina (NWANA, 1995; JENNINGS et al., 1998). Recentemente, foi despertado na Engenharia de Software um grande interesse em agentes inteligentes devido a possibilidade de utilizar a já existente estrutura de abstração, decomposição e organização dos agentes para lidar com o desenvolvimento de sistemas complexos (JENNINGS, 2000). Entretanto, essa tecnologia é utilizada, em algumas situações, de forma imprecisa, não caracterizando claramente a diferença entre um agente inteligente e um programa convencional (FRANKLIN et al., 1996).

A partir da combinação das diversas definições de agentes inteligentes existentes na literatura (WOOLDRIDGE et al., 1994; RUSSELL et al., 1995; WOOLDRIDGE et al., 1995; MAES, 1995; HAYES-ROTH, 1995; FRANKLIN et al., 1996; INGHAM, 1997; GARCIA et al., 2001) consideramos agente como uma entidade que capta informações do ambiente em que está inserido, processa essas informações levando em conta informações próprias, planos e objetivos, e responde ao ambiente. A resposta que será dada ao ambiente é construída através da seleção de planos para a manipulação do conhecimento. Essa seleção deve ser feita de tal forma que minimize o esforço para que o objetivo seja alcançado

As características dos agentes permitem que a máquina de processos atenda especificamente a critérios como pró-atividade e facilidade de extensão. Entretanto, a construção de agentes inteligentes requer uma infra-estrutura de apoio.

Neste capítulo detalharemos a abordagem proposta para uma máquina de processos de software e todos os elementos que fazem parte dessa abordagem. Ao final

do capitulo exibiremos como a máquina de processos pode ser estendida para contemplar a adição de um novo requisito.

3.2 – Abordagem proposta

Uma visão inicial da abordagem proposta é exibida na Figura 3.1. A abordagem foi concebida para permitir que o processo seja modelado graficamente dentro de um ambiente de desenvolvimento de software (ADS), como exibido na parte (a) da Figura 3.1. Após a modelagem do processo, é necessário fornecer mecanismos para a sua instanciação, que ocorre através da tradução do modelo gráfico para uma linguagem executável, como exibido na parte (b) da Figura 3.1. A linguagem executável escolhida para esse mapeamento foi o Prolog[5], por ser uma linguagem declarativa e baseada em inferência. Desta forma, a abordagem proposta se caracteriza como uma abordagem híbrida (JOERIS et al., 1998), pois combina a modelagem gráfica com o mapeamento desse modelo para uma notação textual.

Após o mapeamento, toda a informação sobre o processo está descrita em uma base de conhecimento, e pode ser acessada por qualquer entidade que conheça a linguagem utilizada na sua descrição. A linguagem utilizada para o mapeamento do processo em Prolog serve para a comunicação entre os elementos que desejam interagir com a base de conhecimento para executar os processos. Essa linguagem, que define como os elementos do processo devem ser descritos na base de conhecimento para que seja possível a representação inequívoca do processo modelado, é conhecida como ontologia[6]. Desta forma, neste trabalho, o termo "ontologia" será utilizado para

---

[5] Das linguagens baseadas em inferência, a mais conhecida e utilizada é Prolog. A linguagem Prolog, abreviatura de Programming in Logic, foi criada no início dos anos 70 (ROUSSEL, 1975), para implementar os conceitos de lógica em programação (KOWALSKI, 1974). Prolog é uma linguagem declarativa, com uma fácil manipulação de símbolos, grande poder de expressão e formalismo, que utiliza um sub-conjunto da lógica de primeira ordem (cláusulas de Horn) para a representação dos programas (DUTRA, 2001). Um programa Prolog é composto por fatos e regras. Fatos são declarações sempre verdadeiras e regras são declarações verdadeiras caso um conjunto de restrições seja satisfeito (SILVA, 1999).

[6] A ontologia é a especificação de uma conceitualização (GRUBER, 2002). Uma conceitualização pode ser vista como a estruturação dos conhecimentos de um domínio de conhecimento. No caso do domínio de automação de processos, a ontologia define como é possível representar processos de forma coerente e estruturada.

expressar a conceitualização utilizada para a representação dos processos de software utilizando Prolog.

Para que o processo contido na base de conhecimento possa ser executado, agentes inteligentes devem se conectar à base, alterando-a se necessário, como exibido na parte (c) da Figura 3.1.

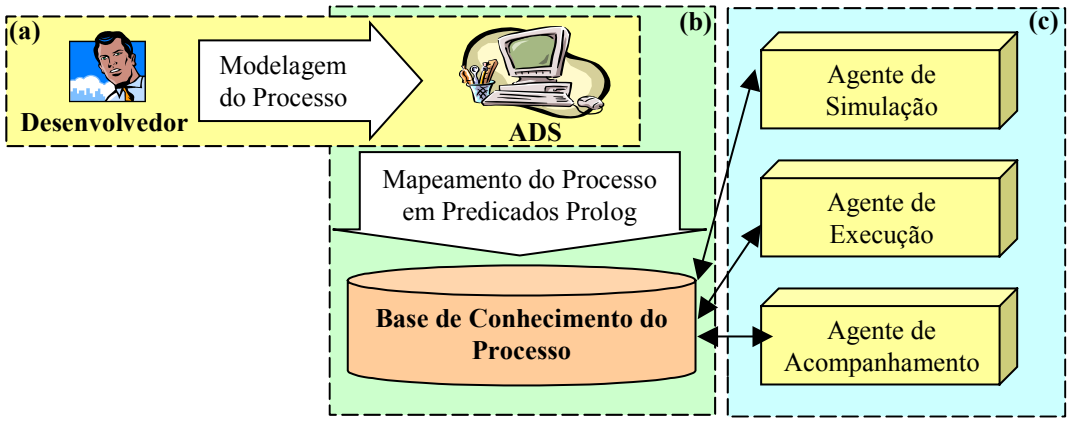

Figura 3.1: Abordagem Proposta.

Os fatos Prolog formam a base de conhecimento do processo, que é o local através do qual agentes inteligentes irão interagir para possibilitar a simulação, execução e acompanhamento do processo em questão. Inicialmente, três agentes básicos são necessários:

- **Agente de Simulação:** Responsável por verificar o processo modelado quanto a sua corretude, permitindo ou não a sua entrada em execução. Este agente será detalhado na seção 3.7;
- **Agente de Execução:** Responsável por verificar o estado da base de conhecimento, procurando por atividades finalizadas ou decisões que foram tomadas, para permitir o andamento do processo. Este agente será detalhado na seção 3.8;
- **Agente de Acompanhamento:** Responsável por interagir com o desenvolvedor, indicando as atividades que estão pendentes e as decisões que devem ser tomadas. Também permite a finalização dessas atividades ou a tomada dessas decisões. Este agente será detalhado na seção 3.9.

Para que a abordagem seja extensível, optamos pela definição de uma infra-estrutura, detalhada na seção 3.6, que provesse um agente genérico com as propriedades desejáveis de agentes inteligentes. Para construir um novo agente basta especializar o

agente genérico, definindo os planos e objetivos, e utilizando a ontologia para tornar possível a comunicação do novo agente com os demais agentes existentes através da base de conhecimento.

As propriedades desejáveis de agentes inteligentes podem ser divididas em básicas e específicas. As propriedades básicas, que devem ser implementadas em qualquer agente, são (WOOLDRIDGE et al., 1995; GARCIA et al., 2001):

- **Autonomia**: Capacidade do agente de operar sem intervenção externa (do local em que está inserido, de humanos ou de outros agentes), tendo controle de suas ações.
- **Interação**: Capacidade do agente de interagir (com o local em que está inserido, com humanos e com outros agentes), de forma reativa ou pró-ativa. A interação reativa ocorre através de respostas a eventos externos. A interação pró-ativa ocorre através da busca ao objetivo, mesmo que não haja nenhum evento externo.
- **Adaptação**: Capacidade do agente de modificar seu estado mental (em função do local em que está inserido, de ações de usuários ou de outros agentes).

Além dessas propriedades básicas, outras propriedades podem estar presentes em agentes inteligentes (FRANKLIN et al., 1996; GARCIA et al., 2001):

- **Aprendizado**: Capacidade do agente de aprender utilizando experiências anteriores quando interage com o local em que está inserido.
- **Mobilidade**: Capacidade do agente de se transportar do local em que está inserido para outro local através da rede.
- **Colaboração**: Capacidade do agente de cooperar com outros agentes para cumprir o seu objetivo e o objetivo do sistema como um todo.

O local onde o agente está inserido também pode ser classificado segundo as seguintes propriedades (RUSSELL et al., 1995):

- **Acessível**: Indica se o local pode ser completamente acessível pelos sensores dos agentes. Se sim, torna-se desnecessária a manutenção das crenças do agente, pois estas podem ser obtidas a qualquer momento

através do uso dos sensores. Ambientes de execução de processos de software são acessíveis.

- **Determinístico**: Indica que o próximo estado do local pode ser determinado a partir do estado atual e de ações selecionadas nos agentes. Ambientes de execução de processos de software são determinísticos.
- **Episódico**: Indica que o local pode ser dividido em episódios, não guardando dependência entre ações tomadas em episódios anteriores com o atual. Desta forma, independentemente do que aconteceu anteriormente, o estado atual, juntamente com a ação efetuada, determinam o novo estado do local. Ambientes de execução de processos de software são episódicos.
- **Estático**: Indica que o local não muda com o passar do tempo. Ambientes de execução de processos de software são dinâmicos.
- **Discreto**: Indica que o número de elementos contidos no local é finito. Ambientes de execução de processos de software são discretos.

No contexto da automação de processos de desenvolvimento de software, podemos identificar as seguintes atividades, apresentadas esquematicamente na Figura 3.2: (1) modelagem, (2) instanciação, (3) simulação, (4) execução, (5) acompanhamento, (6) monitoramento e (7) evolução do processo. A modelagem do processo consiste na estruturação das atividades necessárias para a sua execução, identificando os recursos necessários para que essas atividades possam ser executadas. A instanciação do processo consiste na seleção de um processo modelado e início da sua simulação. A simulação do processo consiste na verificação da corretude do mesmo, permitindo que a sua execução seja iniciada. A execução e acompanhamento do processo permitem que os desenvolvedores verifiquem a lista de trabalhos pendentes, executem esses trabalhos e, posteriormente, notifiquem o seu término à máquina de processos, para que seja possível dar andamento a outros trabalhos subseqüentes. O monitoramento do processo consiste em exibir o estado atual de sua execução para fornecer subsídios aos gerentes, para que estes possam estimar como está o andamento da execução. A evolução do processo permite que modificações no processo modelado sejam incorporadas em suas instâncias em execução.

A base de conhecimento, que contém todas as informações sobre o processo, permite que agentes se conectem a ela, efetuem suas inferências e se desconectem, possibilitando que outros agentes possam se conectar. A abordagem proposta prevê a execução concomitante de diversas instâncias de processos. Uma instância de processo representa um determinado processo em execução no desenvolvimento de um software. Assim, caso várias aplicações estejam sendo desenvolvidas utilizando o mesmo processo, existirão várias instâncias desse processo em execução, uma para cada aplicação em desenvolvimento. Cada instância de processo em execução implicará na criação de uma base de conhecimento própria. Entretanto, somente um agente poderá estar conectado a uma determinada base de conhecimento em um instante de tempo, para evitar problemas de acesso concorrente às informações. Através da propriedade de mobilidade os agentes poderão se locomover de uma base de conhecimento para outra, se desconectando da base de conhecimento anterior e se conectando na nova base de conhecimento.

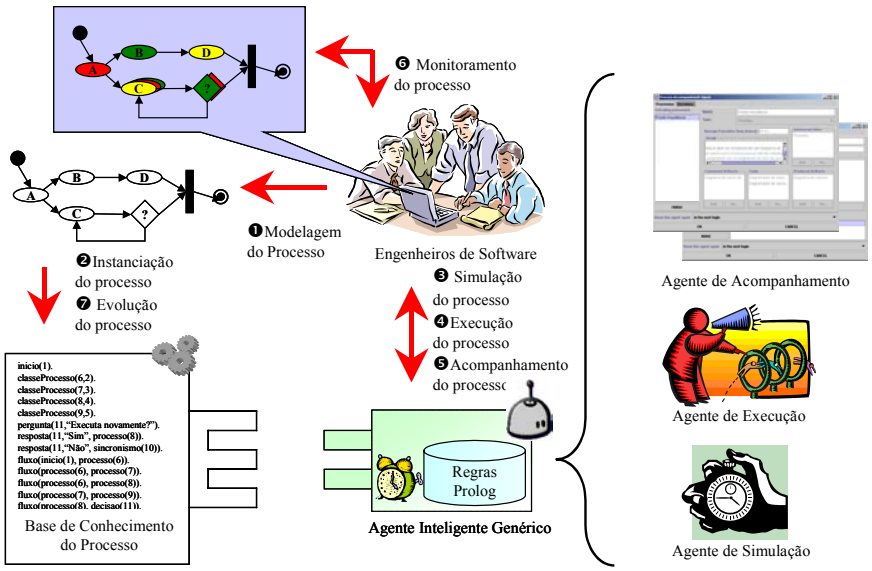

Figura 3.2: Atividades envolvidas na abordagem Charon.

Nas demais seções deste capítulo são detalhados os elementos contidos na abordagem proposta, que são o suporte para a modelagem gráfica, a instanciação do processo, a infra-estrutura para a construção de agentes inteligentes, os agentes básicos, que são voltados para a simulação, execução e acompanhamento do processo, e o monitoramento do processo. Ao final, é apresentado ainda um exemplo de extensão da máquina de processos.

3.3 – Suporte para a modelagem gráfica de processo

Processos podem ser representados através de linguagens de modelagem de processos ou meta-modelos de processos (JOERIS et al., 1998). Neste trabalho optamos por representar processos através de um meta-modelo simples, que contém os elementos comuns dos meta-modelos utilizados pelas abordagens descritas no capítulo 2.

Para que esse meta-modelo possa ser utilizado, decidimos adotar uma notação gráfica para diagramar os seus elementos. Dentre as diversas notações gráficas existentes na literatura (MAYER et al., 1995),(WFMC, 1999; OMG, 2001) optamos por utilizar uma extensão do diagrama de atividades da UML (OMG, 2001). A extensão desse diagrama tem sido foco de pesquisa por vários grupos (HEIMANN et al., 1996; OKTABA et al., 2001), pois além de ser simples, possibilita a definição de processos de software utilizando uma notação semelhante a que é utilizada no próprio desenvolvimento de software.

3.3.1 – Meta-modelo de processos

O meta-modelo de processos utilizado tem como principal vantagem a sua simplicidade. A simplicidade traz como benefício um isolamento dos detalhes inerentes ao meta-modelo, que poderiam dificultar a avaliação das contribuições e limitações reais da abordagem proposta. O foco deste trabalho está na máquina de execução de processos e não na modelagem de processos. Esse meta-modelo tem como elemento central o processo. Existem dois tipos de processos: processos primitivos e processos compostos.

Os **processos primitivos** representam as atividades atômicas que podem ser executadas dentro do ambiente. Esse tipo de processo descreve o roteiro que deve ser seguido para a sua execução, as ferramentas que devem ser utilizadas, os papéis que agrupam desenvolvedores aptos para a execução e os artefatos consumidos e produzidos pela atividade, como exibido na Figura 3.3. A Figura 3.3 e a Figura 3.4 exibem diferentes visões sobre o mesmo diagrama, onde a primeira visão descreve os relacionamentos do processo primitivo e a segunda visão, os relacionamentos do processo composto.

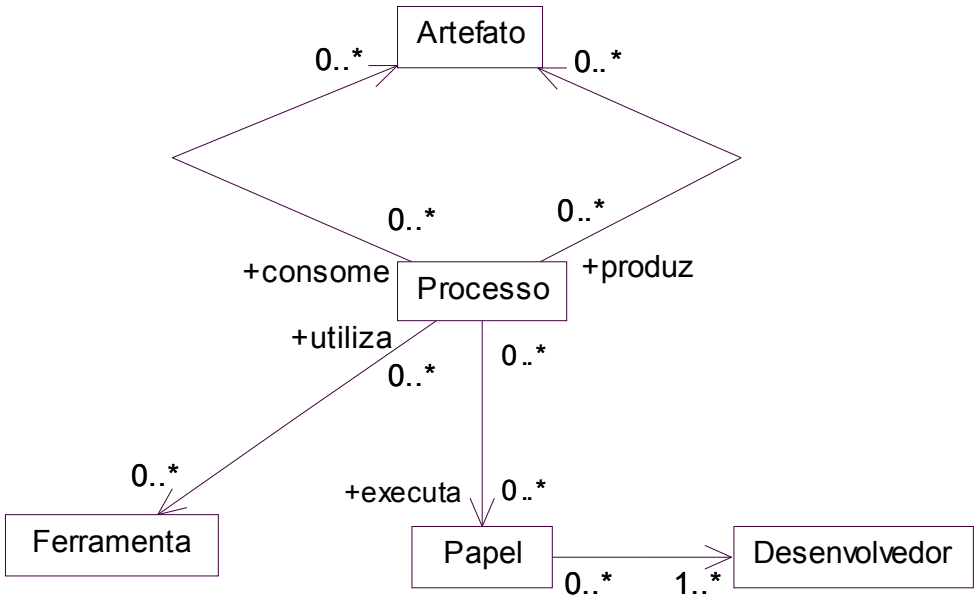

Figura 3.3: Modelo de processo primitivo utilizado pela abordagem proposta.

Os **processos compostos** representam atividades complexas, que contêm um *workflow* descrevendo como as suas sub-atividades devem ser executadas segundo a notação do diagrama de atividades da UML, como exibido na Figura 3.4. Esse *workflow* pode conter os seguintes elementos (OMG, 2001):

- **Início (●)**: Elemento onde se dará o início da execução do *workflow* do processo composto;
- **Atividade (⬭)**: Elemento principal do *workflow*, que pode ser um processo primitivo ou composto, possibilitando a decomposição recursiva de um processo;
- **Decisão (◇)**: Elemento que permite a escolha, dentre um conjunto de opções, de quais fluxos serão ativados para prosseguir a execução do *workflow* do processo composto;
- **Sincronismo (▮)**: Elemento que permite sincronizar um conjunto de fluxos de processos. O sincronismo pode ser utilizado tanto para definir que diversos fluxos de saída devem ser ativados em paralelo, quanto para definir que o próximo elemento só será executado após a ativação de todos os fluxos de chegada;
- **Término (◉)**: Elemento onde se dará o término da execução do *workflow* do processo composto;

- **Fluxo (→)**: Ligação entre dois elementos (não fluxos) do *workflow* representando a ordem de precedência necessária para a sua execução.

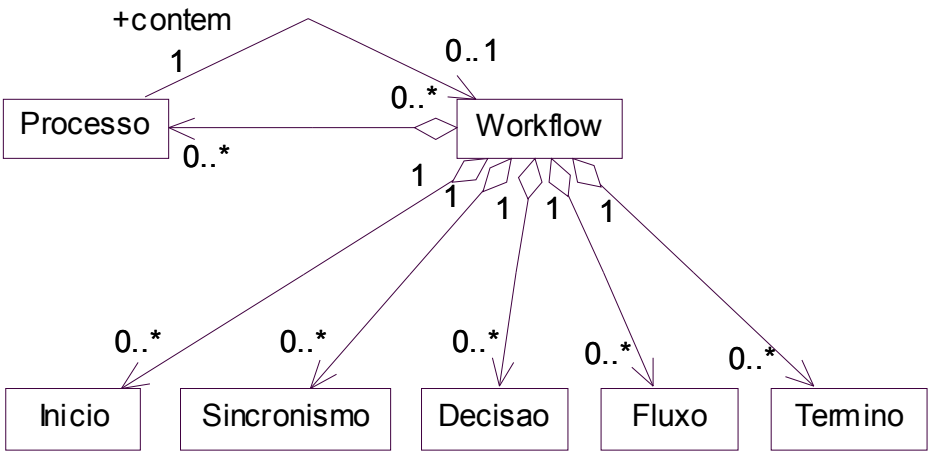

Figura 3.4: Modelo de processo composto utilizado pela abordagem proposta.

A notação utilizada nos *workflows* estende o diagrama de atividades da UML com as seguintes características:

- Um elemento qualquer pode ter vários fluxos de entrada associados, com a semântica de *OR*. Assim, quando o primeiro fluxo permitir a execução do elemento, este será executado. As exceções são para os elementos de início, que não permitem nenhum fluxo de entrada, de sincronismo, que somente serão executados quando todos os fluxos de entrada forem sincronizados, e de fluxo, que não permitem a associação com outros fluxos.
- Um elemento qualquer pode ter vários fluxos de saída associados, com a semântica de *AND*. Assim, quando terminar a execução do elemento, todos os fluxos de saída serão ativados. As exceções são para os elementos de término, que não permitem nenhum fluxo de saída, de decisão, que somente ativarão os fluxos de saída que sejam condizentes com a resposta escolhida, e de fluxo, que não permitem a associação com outros fluxos.
- Pode haver vários elementos de início, com a semântica equivalente a vários fluxos saindo de um mesmo elemento de início.

- Pode haver vários elementos de término, com a semântica equivalente a vários fluxos entrando em um mesmo elemento de término.
- Um elemento de decisão consiste em uma pergunta. Cada fluxo que sai desse elemento de decisão contém uma possível resposta para a pergunta. No momento da sua execução será selecionada uma única resposta, e o fluxo associado a essa resposta será ativado.
- Um elemento de sincronismo pode ter vários fluxos de entrada e saída associados. O seu objetivo é sincronizar todos os fluxos de entrada, fazendo com que os elementos ligados aos fluxos de saída só entrem em execução quando todos os elementos ligados aos fluxos de entrada já tiverem sido executados.
- Um processo, primitivo ou composto, ou uma decisão só entrará em execução se não existir uma instância em execução no mesmo contexto e com o mesmo identificador de instância. O contexto é o caminho percorrido do processo principal até o *workflow* do processo atual, que permite a construção de várias instâncias de um mesmo processo em diferentes *workflows*. O identificador de instância é um identificador que permite a construção de várias instâncias de um mesmo processo em um mesmo *workflow*.
- É permitida a modelagem de repetições no *workflow* de processos através de fluxos de retorno ou através de recursão. A recursão consiste na modelagem de um processo dentro do *workflow* de um de seus sub-processos, em qualquer nível. A modelagem de recursão deve seguir as regras tradicionalmente utilizadas para tratar recursão em algoritmos, onde são banidas recursões sem condição de parada.

A Figura 3.5 exibe cada uma dessas características, mostrando como a notação estendida (esquerda da igualdade) pode ser traduzida para a notação padrão do diagrama de atividades da UML.

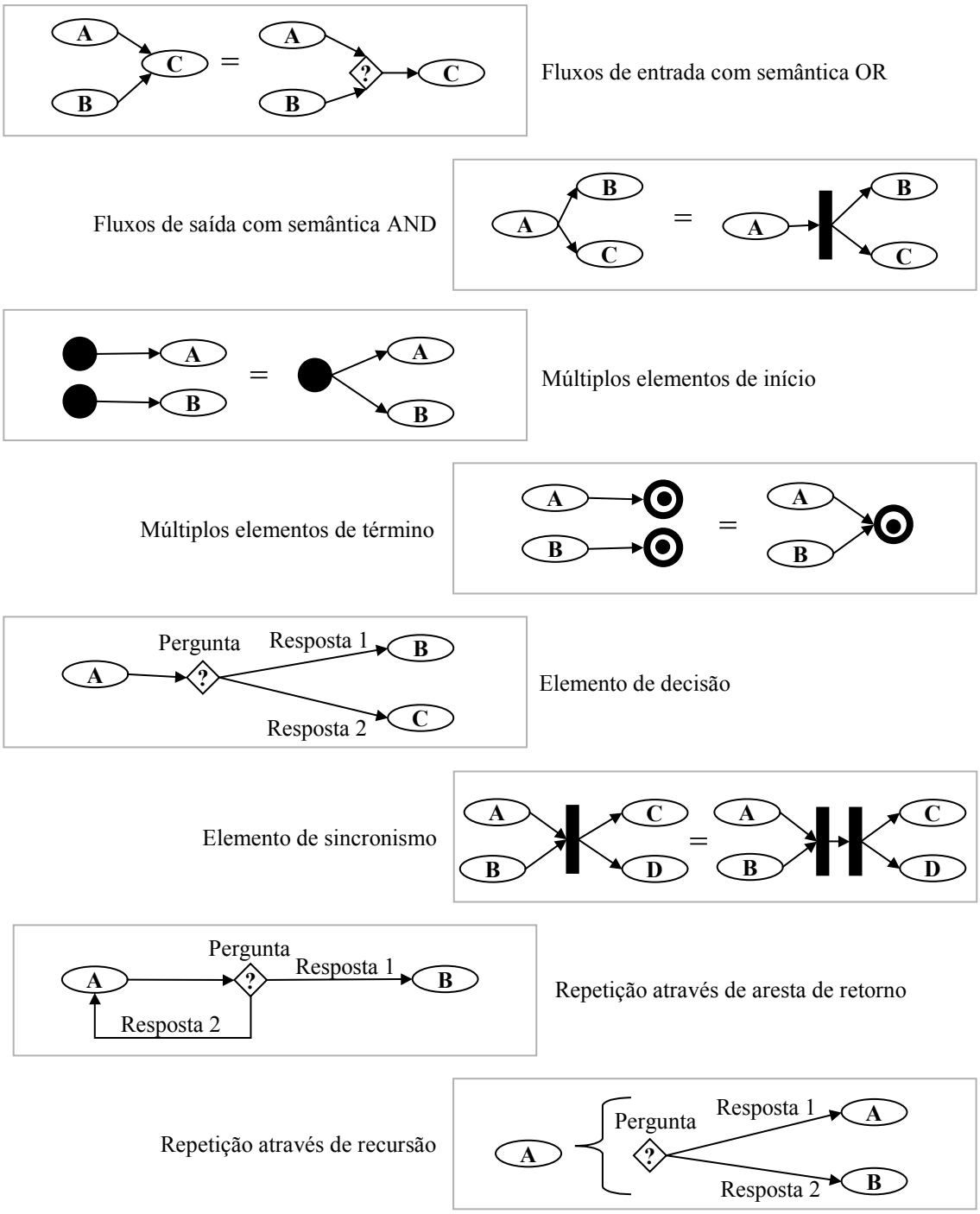

Figura 3.5: Notação do diagrama de atividades UML estendido.

3.3.2 – Apoio à reutilização de processos

Uma característica adicionada a esse meta-modelo para possibilitar a reutilização dos processos é permitir que a hierarquia de sub-processos não tenha limitação de níveis. Desta forma, um processo pode estar em diferentes *workflows*, podendo ser tratado tanto como um processo global quanto como uma atividade específica. Para

tanto, foi necessário permitir a construção de recursão dentro da hierarquia de processos.

A repetição de um processo pode ser modelada tanto com um fluxo de retorno dentro do *workflow* que o processo está desenhado quanto com uma referência recursiva para o próprio processo dono do *workflow*, ou para um outro ascendente na hierarquia de processos.

Por exemplo, supondo que o processo "projeto" seja composto pelos processos "projeto arquitetural" e "projeto detalhado", e seja desejado que, ao final do projeto detalhado, o gerente decida se deve ser executado um refinamento do projeto ou se deve ser iniciada a implementação. A modelagem dessa situação através do uso de fluxo de retorno está exibida na Figura 3.6.

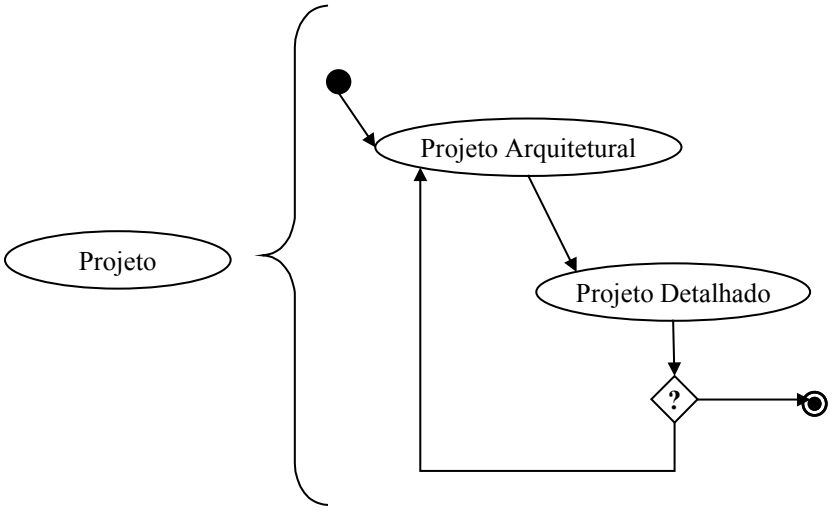

Figura 3.6: Repetição da execução através de fluxo de retorno.

Entretanto, também é possível modelar esta situação utilizando uma referência recursiva para o próprio processo, como exibido na Figura 3.7. Essa segunda modelagem força uma nova instanciação do processo principal, juntamente com a instanciação de todos os elementos pertencentes ao seu *workflow*.

Ambas as abordagens para repetição de um processo atingem o seu objetivo, que é permitir que uma atividade seja executada várias vezes, sem uma definição do número de vezes a priori. A maior diferença entre elas está na forma de representação utilizada para o monitoramento do processo. O monitoramento da primeira abordagem permitirá a visualização de todas as execuções no mesmo *workflow*, enquanto o da segunda abordagem necessitará navegação para diferentes *workflows* para que seja possível verificar como foram as execuções.

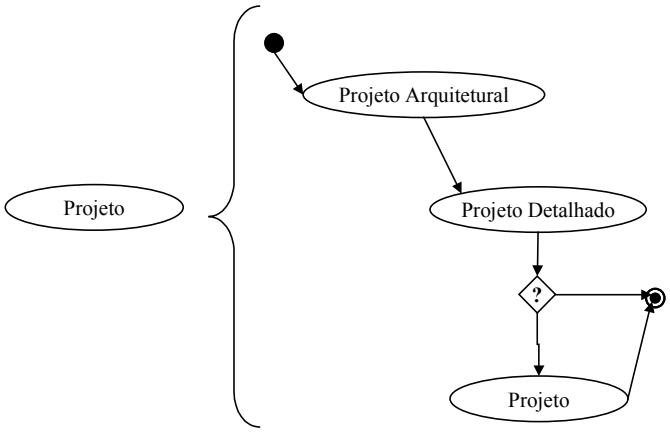

Figura 3.7: Repetição da execução através de referência recursiva.

3.4 – Suporte para a instanciação do processo

A instanciação acontece em três diferentes momentos do ciclo de vida do processo. Inicialmente, ocorre a instanciação dos processos através do mapeamento das suas informações modeladas graficamente para Prolog e da seleção de quais usuários do ambiente irão exercer quais papéis. Durante a execução do processo, cada vez que um sub-processo deve ser executado, ocorre uma instanciação dinâmica, que permite identificar unicamente a instância criada dentro do ambiente, possibilitando que existam várias instâncias de um mesmo processo em execução. O terceiro momento de instanciação ocorre quando o modelo gráfico do processo é evoluído e se deseja que essa evolução seja refletida em uma determinada instância em execução. Nas próximas seções, discutiremos em detalhes cada um desses momentos.

3.4.1 – Mapeamento do processo para Prolog

O mapeamento do processo gráfico em predicados Prolog consiste em criar um identificador numérico único para cada elemento modelado e utilizar esse identificador como chave na construção dos predicados Prolog. Para que, posteriormente, seja possível a obtenção do mapeamento inverso, é necessário manter uma tabela de mapeamento que informe que predicado Prolog representa que elemento gráfico, e vice-versa.

Cada processo modelado pode aparecer em diversos *workflows* de outros processos, tornando necessário diferenciar a definição de um processo do seu desenho em um *workflow*. Para permitir essa diferenciação, chamaremos de "classe de processo" a definição de um processo, e de "processo" ou "processo desenhado" o processo modelado em um *workflow*. Essa nomenclatura é uma analogia às definições de classe e

objeto da orientação a objetos, e serve para permitir a fatoração das características comuns de um conjunto de elementos. Desta forma, as propriedades de um processo, como nome, roteiro que descreve como o processo deve ser executado, ferramentas associadas, *workflow* associado, entre outras, serão definidas na classe do processo, pois todos os processos desenhados que pertencerem a essa classe compartilharão essas propriedades. O processo desenhado terá características próprias referentes aos seus fluxos de entrada e de saída no contexto de um determinado *workflow*.

3.4.2 – Ontologia definida

A ontologia utilizada para representar o processo em Prolog pode ser dividida em vários conjuntos para facilitar o seu entendimento. Inicialmente, as entidades referentes a definição de características comuns aos processos primitivos e compostos estão descritas na Tabela 3.1.

Tabela 3.1: Entidades referentes à classe de processos da ontologia proposta.

Entidade	Representação Prolog	Semântica
Processo primitivo	`processoPrimitivo(idC).`	Indica que uma determinada classe de processos (idC) é do tipo primitivo, e em assim sendo terá roteiro, ferramentas, papéis e artefatos associados.
Processo composto	`processoComposto(idC).`	Indica que uma determinada classe de processos (idC) é do tipo composto, e em assim sendo terá um *workflow* associado.
Processo raiz	`processoRaiz(idC).`	Indica qual é a classe do processo raiz (idC). A instanciação dinâmica utilizará essa informação para criar a instância inicial do processo.
Nome de um processo	`nome(idC, n).`	Associa um nome (n) a uma classe de processos (idC).
Simulação de um processo	`simulado(idC, s).`	Associa um tempo de simulação (s) a uma classe de processos (idC).

Caso o processo que está sendo representado seja do tipo primitivo, um outro conjunto de entidades será utilizado. Essas entidades estão descritas na Tabela 3.2. Entretanto, caso o processo modelado seja do tipo composto, as entidades descritas na Tabela 3.3 são utilizadas.

Tabela 3.2: Entidades referentes a processo primitivo da ontologia proposta.

Entidade	Representação Prolog	Semântica
Roteiro de um processo	`roteiro(idC, r).`	Associa um roteiro (r) a uma classe de processos (idC) do tipo primitivo.
Ferramenta utilizada em um processo	`ferramenta(idC, f).`	Associa uma ferramenta (f) a uma classe de processos (idC) do tipo primitivo.
Papel utilizado em um processo	`papel(idC, p).`	Associa um papel (p) a uma classe de processos (idC) do tipo primitivo.
Artefato de entrada de um processo	`artefatoEntrada(idC, a).`	Associa um artefato de entrada (a) a uma classe de processo (idC) do tipo primitivo.
Artefato de saída de um processo	`artefatoSaida(idC, a).`	Associa um artefato de saída (a) a uma classe de processo (idC) do tipo primitivo.

Tabela 3.3: Entidades referentes a processo composto da ontologia proposta.

Entidade	Representação Prolog	Semântica
Início de um *workflow*	`inicio(idC).`	Representa o elemento de início do *workflow* de uma classe de processos (idC).
Termino de um *workflow*	`termino.`	Representa o elemento de término do *workflow* de qualquer classe de processos.
Processo desenhado em um *workflow*	`processo(idP).`	Representa um processo desenhado (idP) no *workflow* de alguma classe de processos.
Classe de um processo desenhado	`classeProcesso(idP, idC).`	Associa um processo desenhado (idP) a uma determinada classe de processos (idC).
Decisão em um *workflow*	`decisao(idD).`	Representa um elemento de decisão (idD) no *workflow* de alguma classe de processos.
Simulação de uma decisão	`simulado(idD, s).`	Associa um tempo de simulação (s) a uma decisão (idD).
Pergunta de uma decisão	`pergunta(idD, p).`	Associa uma pergunta (p) a uma decisão (idD).
Resposta de uma decisão	`resposta(idD, r, idE).`	Associa uma resposta (r) a uma decisão (idD), indicando qual elemento (idE) deve ser executado caso a resposta seja selecionada.
Sincronismo em um *workflow*	`sincronismo(idS).`	Representa um elemento de sincronismo (idS) no *workflow* de alguma classe de processos.
Fluxo em um *workflow*	`fluxo(eO, eD).`	Representa um fluxo no *workflow* de alguma classe de processos. O fluxo parte do elemento de origem (eO) em direção ao elemento de destino (eD).

Além das entidades que descrevem o processo, existem outras que determinam o seu estado de execução ou os desenvolvedores que estão aptos a interagir com algum elemento durante a execução. Essas entidades estão descritas na Tabela 3.4.

Tabela 3.4: Entidades referentes à execução dos processos da ontologia proposta.

Entidade	Representação Prolog	Semântica
Usuário da máquina de processos	`usuario(u, p).`	Associa o usuário (u) a um determinado papel (p).
Elemento em execução	`executando(e, c, tI).`	Indica que um determinado elemento (e) está em execução em um determinado contexto (c) desde um tempo inicial (tI). O contexto permite que o mesmo elemento seja executado em diferentes *workflows*.
Elemento que já foi executado	`executado(e, c, tI, tF).`	Indica que um determinado elemento (e) foi executado em um determinado contexto (c) durante um intervalo de tempo (tF – tI).
Resposta selecionada em uma decisão	`respondido(idD, c, r, t, u).`	Indica que uma resposta (r) de uma decisão (idD) em execução em um determinado contexto (c) foi escolhida por um usuário (u) em um determinado momento (t).
Finalização de um processo	`finalizado(idP, c, t, u).`	Indica que um determinado processo (idP) em execução em um determinado contexto (c) foi finalizado por um usuário (u) em um determinado momento (t).

Um exemplo do uso dessa ontologia para o mapeamento de um processo modelado segundo o ciclo de vida "cascata" (PRESSMAN, 1997) é exibido na Figura 3.8. Neste exemplo, a classe de processo "Projeto", que é do tipo composto, recebe o identificador numérico "3" (cláusula processoComposto(3)). O nome do processo está associado ao identificador "3" no nível de classe de processo, fazendo com que todos os processos desenhados dessa classe tenha esse nome (cláusula nome(3, "Projeto")).

Um processo desenhado é criado como sub-processo de "Desenvolvimento". Esse sub-processo é da classe "3" ("Projeto"), recebendo o identificador "7" (cláusula classeProcesso(7, 3)). Juntamente com a sua definição, são gerados os seus fluxos de entrada e saída (cláusulas fluxo(processo(6), processo(7)) e fluxo(processo(7), processo(8))). De forma equivalente são geradas as cláusulas que representam os demais elementos.

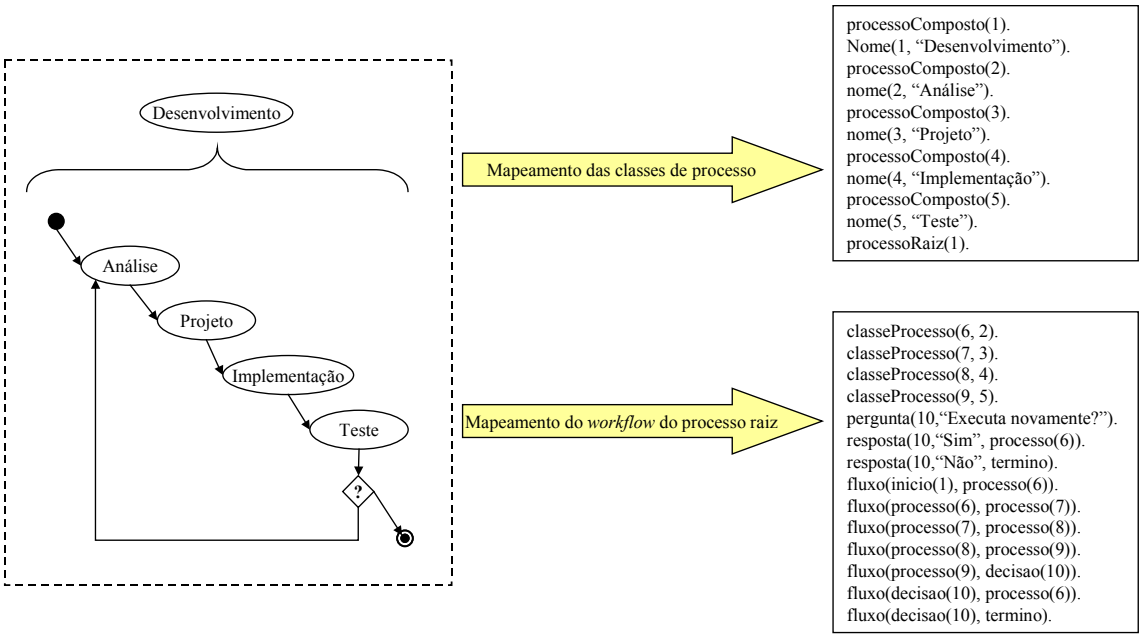

Figura 3.8: Exemplo do mapeamento de processos segundo a ontologia proposta.

3.4.3 – Instanciação dinâmica

Para permitir a construção de recursões e fluxos de retorno nos *workflows*, é necessário adotar uma política de instanciação dinâmica. Caso a instanciação fosse completa no momento do mapeamento, não seria possível determinar quando o fluxo de retorno ou a recursão param, pois essa informação está atrelada a decisões que serão tomadas pelos desenvolvedores durante a própria execução.

Sempre que for necessário instanciar um novo processo, o agente responsável pela execução de processos deverá modificar a base de conhecimento do processo em execução, adicionando os fatos referentes à nova instanciação. Esse mecanismo será detalhado na seção 3.8, que descreve o agente responsável pela execução de processos.

3.4.4 – Evolução de processos

A abordagem proposta também permite que os processos em execução evoluam com o tempo. Quando o processo é colocado em execução, acredita-se que a sua modelagem foi feita tentando refletir as reais necessidades do projeto que será guiado pelo processo. Entretanto, tanto a modelagem pode ter sido falha quanto os requisitos do projeto podem ter mudado, o que motiva a modificação do processo em execução. Uma outra motivação para evoluir um processo é a adoção de políticas que visem a sua otimização. Para permitir que a evolução de processos em execução ocorra, as cláusulas Prolog existentes foram divididas em três grupos:

- **Cláusulas de mapeamento:** são os fatos gerados pelo tradutor do processo modelado durante a instanciação inicial. Esses fatos, que representam o processo antigo, devem ser gerados novamente sempre que houver a necessidade de evolução.
- **Cláusulas de execução:** são fatos incluídos na base de conhecimento pelos agentes para permitir o controle da execução do processo. Esses fatos contêm toda a informação necessária para identificar em que estágio está a execução. Mesmo que o processo evolua, esses fatos devem ser mantidos na base para que seja possível saber o que já aconteceu e quais são os processos atualmente em execução.
- **Cláusulas de agentes:** são regras definidas dentro dos agentes, que representam os seus planos de execução. Essas regras devem ser removidas da base de conhecimento sempre que o agente se desconectar da mesma.

Esses grupos de cláusulas são manipulados através da verificação do contexto atual do ambiente, e remoção das cláusulas indesejadas. Sempre que um agente se desconecta do ambiente, as cláusulas de agentes são removidas, pois não são úteis sem o próprio agente dono das cláusulas. Quando é necessário reconstruir os dados de definição do processo, as cláusulas de mapeamento são removidas e regeradas para refletir o estado atual do processo modelado.

Desta forma, a evolução do processo consiste na reconstrução completa das cláusulas de mapeamento, e posterior junção com as cláusulas de execução existentes. Essa reconstrução das cláusulas de mapeamento tem que levar em conta os identificadores utilizados pelas cláusulas de execução, para que as novas cláusulas sejam compatíveis com as antigas.

As cláusulas de execução devem ser sempre mantidas, pois refletem o que ocorreu com o processo desde a sua criação até o momento atual da execução.

3.5 – Infra-estrutura para a construção de agentes inteligentes

Após a instanciação do processo, com a subseqüente criação de sua base de conhecimento, agentes inteligentes podem atuar sobre esta base. Contudo, a construção de agentes inteligentes, como a construção de qualquer tipo de software, é beneficiada quando existe um arcabouço com as características comuns dos elementos que estão

sendo construídos, permitindo que essas características possam ser estendidas para a construção de elementos específicos.

A orientação a objetos fornece a estruturação necessária para esse tipo de problema, permitindo que características genéricas sejam construídas em classes abstratas e que essas classes sejam estendidas e posteriormente instanciadas, facilitando a construção dos elementos específicos. Esse conjunto de classes abstratas, que contém as características genéricas e necessitam ser estendidas e instanciadas para se tornarem úteis dentro de um contexto, é denominado *framework*. Desta forma, um *framework* pode ser considerado como um projeto ou arquitetura de alto nível, consistindo de classes que são especialmente projetadas para serem refinadas e usadas em grupo (WIRFS-BROCK et al., 1990).

Um *framework* para a construção de agentes inteligentes deve definir os elementos existentes nesse domínio, levando em conta as propriedades genéricas dos agentes, que são: autonomia, interação reativa e pró-ativa, adaptação, aprendizado, mobilidade e colaboração, conforme visto na Seção 3.2. Com esse *framework* construído, a tarefa de utilizar agentes para o domínio de automação de processos se resume a criação dos agentes específicos, que reutilizarão todas as propriedades já implementadas no agente genérico do *framework*.

Outra possibilidade seria a utilização de uma infra-estrutura já existente para a construção dos agentes, como, por exemplo, Aglets (IBM, 2001). Todavia, devido às infra-estruturas existentes serem complexas, optamos por desenvolver uma infra-estrutura própria, ao invés de reutilizar alguma infra-estrutura já existente, que implicaria em um trabalho maior, referente à compreensão e adaptação dessa infra-estrutura complexa à realidade desse trabalho.

A implementação de cada uma das propriedades dos agentes demanda a utilização de recursos da orientação a objetos. Os elementos principais do *framework* são:

- **Base de Conhecimento:** representa o local onde os agentes serão inseridos. Nela residem os fatos com todas as informações sobre o mundo em que os agentes atuarão. Além disso, é na base de conhecimento que a máquina de inferência está situada. A base de conhecimento é um ambiente acessível e dinâmico, permitindo que qualquer agente possa a qualquer momento acessar ou alterar qualquer informação existente. O *framework* permite a existência de várias bases

de conhecimento, com a restrição de existir somente um agente interagindo com uma determinada base de conhecimento em um instante de tempo.

- **Agente:** elemento central do *framework* que contém uma base de regras, representando os seus planos de execução. O agente pode conectar-se a uma base de conhecimento, transferindo a sua base de regras para a máquina de inferência da base de conhecimento e fazendo consultas segundo o seu objetivo. Um agente pode estar em somente uma base de conhecimento em um instante de tempo.
- **Disparador:** elemento responsável por controlar a execução dos vários agentes. Sempre que um agente necessita ser executado, o Disparador cria uma *thread* própria para o agente, permitindo que a execução do agente não interfira na execução normal do sistema.

Esses elementos interagem segundo o diagrama exibido na Figura 3.9.

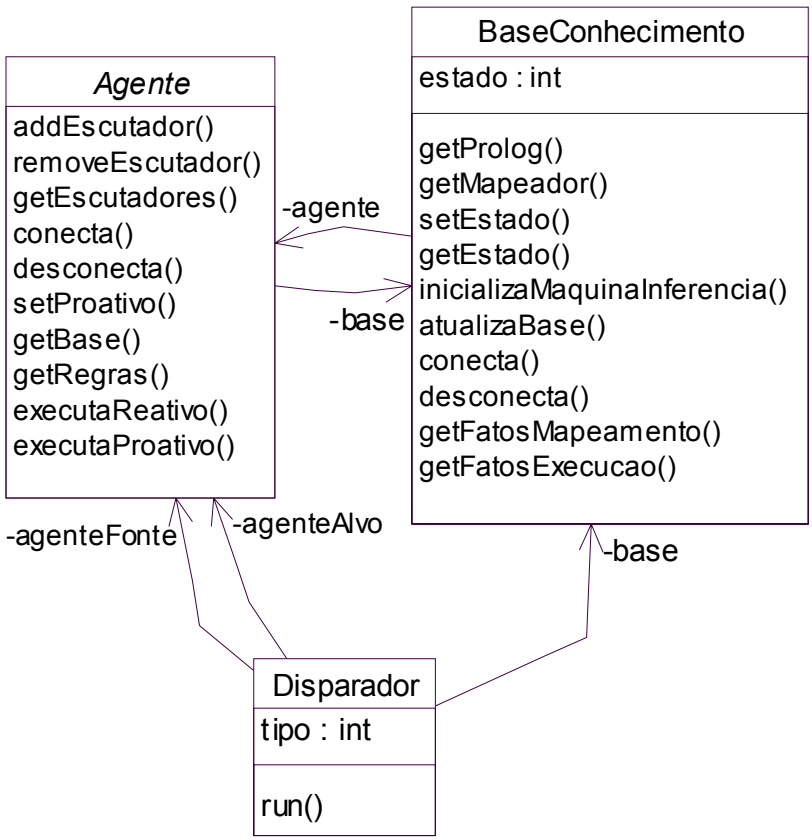

Figura 3.9: Principais elementos do *framework* de construção de agentes inteligentes.

A base de conhecimento, por ser o meio de comunicação entre os agentes, pode estar em diversos estados durante uma execução. Esses estados indicam se um determinado agente deve ou não interagir com a base nesse momento. A Figura 3.10 exibe o diagrama de transição de estados de uma base de conhecimento.

Quando a base é criada, o seu estado inicial é SIMULANDO, e somente o agente de simulação terá interesse em se conectar à base. Já o agente de execução verifica quais bases foram simuladas com sucesso, pois tem interesse em bases no estado PENDENTE, para que possa dar início a sua execução, ou no estado EXECUTANDO, para continuar a execução caso algum usuário tenha finalizado algum processo ou tomado alguma decisão. Os estados EXECUTANDO e FINALIZADO são de interesse do agente de acompanhamento, que constatam se é necessário interagir com o usuário ou se o trabalho descrito pelo processo já chegou ao fim.

A interação entre os elementos do *framework*, que são as Bases de Conhecimento, os Agentes e o Disparador, visa satisfazer as propriedades dos agentes inteligentes. Para cada propriedade, discutimos a seguir como o *framework* fornece mecanismos para a sua satisfação.

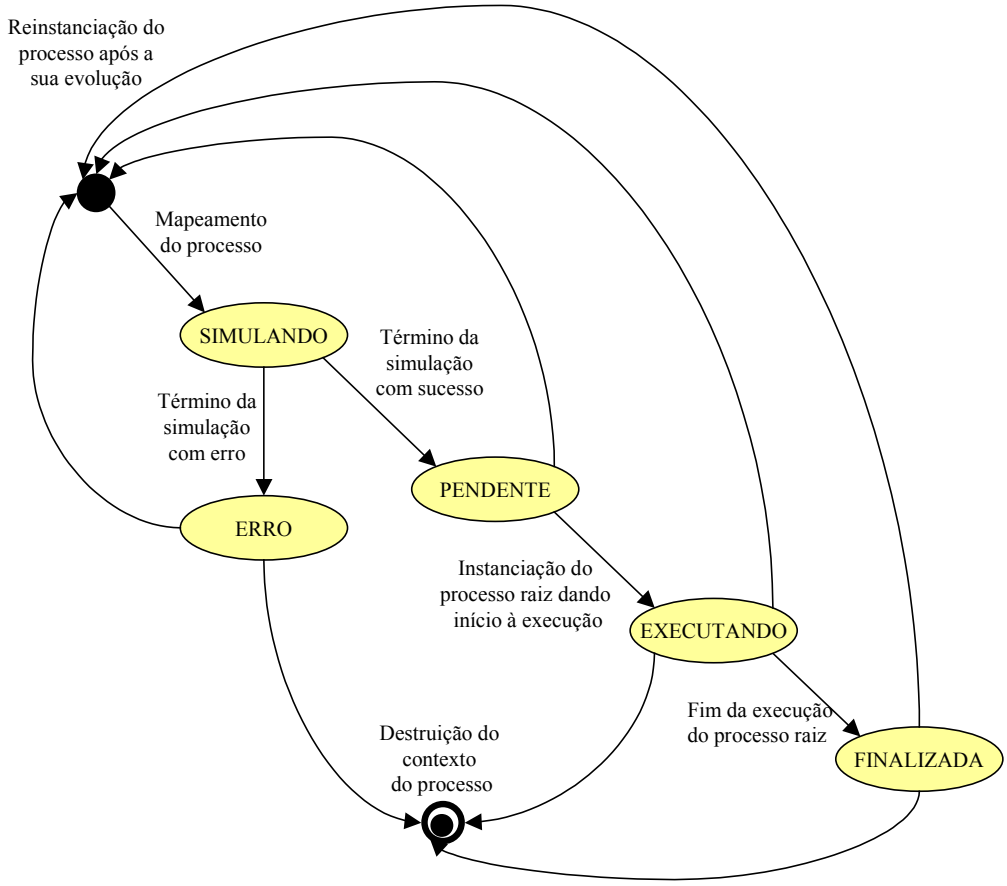

Figura 3.10: Diagrama de transição de estados da base de conhecimento.

3.5.1 – Autonomia

A autonomia dos agentes é obtida através do Disparador. O Disparador cria uma nova *thread* sempre que um agente necessita entrar em execução. Essa nova *thread* fornece os recursos computacionais necessários para que, mesmo que o sistema em que os agentes estejam inseridos pare, o agente continue executando, ou, no caso inverso, onde um agente mal projetado cause a paralisação da sua execução, não haverá interferência na execução do sistema nem na execução dos demais agentes.

3.5.2 – Interação reativa

A interação reativa consiste na motivação do agente em interagir em resposta à ocorrência de algum evento externo. Para que essa propriedade fosse contemplada, os agentes implementam o padrão *Observer* (GAMMA et al., 1994), onde todos os agentes que têm interesse em ser notificados após a ação de um determinado agente devem se cadastrar junto a esse agente.

O mecanismo consiste em monitorar o momento em que um agente se desconecta de uma base de conhecimento e em notificar a todos os outros agentes interessados que isso aconteceu. Essa notificação é acompanhada com informações sobre qual agente e qual base de conhecimento participaram do evento.

A interação reativa também ocorre quando qualquer outro elemento necessita notificar ao agente que algo aconteceu. Desta forma, o agente recebe uma mensagem com informações sobre a origem do evento e sobre a base de conhecimento que deve ser manipulada para que seja possível o tratamento desse evento.

A mensagem que representa a interação reativa é disparada através do Disparador, que constrói uma *thread* própria, permitindo que vários agentes executem em paralelo os seus tratamentos ao evento em questão.

3.5.3 – Interação pró-ativa

Para que os agentes não dependam da execução de outros agentes para entrar em execução, é utilizado um temporizador que emite um evento a cada intervalo de tempo. Esse evento motiva a execução pró-ativa dos agentes.

A execução pró-ativa se baseia na busca do objetivo do agente de forma autônoma, mesmo que nenhum evento proveniente do ambiente, de outro agente ou do usuário tenha ocorrido. Como o temporizador fica localizado dentro do próprio agente, não existe nenhum evento passando pela fronteira entre o agente e o ambiente em que o

agente está inserido. Para garantir essa autonomia, a passagem do evento oriunda do temporizador ocorre através do Disparador, que cria uma *thread* própria para o seu tratamento. Desta forma, do ponto de vista do ambiente, dos outros agentes e do usuário, o agente em questão agiu de forma pró-ativa e autônoma.

3.5.4 – Adaptação

A adaptação ocorre quando os agentes, após se conectarem a uma base de conhecimento, fazem alterações em seu estado, o que representa uma modificação nas crenças do agente.

Como a base de conhecimento representa um ambiente acessível e dinâmico, as crenças do agente são armazenadas dentro do próprio ambiente. Essas crenças podem ser modificadas, fazendo com que o agente passe a agir de forma diferente para uma mesma situação.

3.5.5 – Aprendizado

Quando o agente entra em execução, sendo de forma reativa ou pró-ativa, e se conecta a uma base de conhecimento, a sua base de regras é solicitada através de um método abstrato da classe Agente. A implementação desse método em um agente específico, através da propriedade de polimorfismo da orientação a objetos, possibilitaria a adaptação da base de regras, provocando uma mudança nos planos do agente. Essa mudança nos planos faria com que o agente chegasse a inferências diferentes para um mesmo problema. Desta forma, o aprendizado estaria no algoritmo de seleção de quais regras deveriam sair e quais regras deveriam entrar nos planos do agente. Esse algoritmo não é definido no agente genérico, ficando a cargo dos agentes específicos.

Apesar de ser possível utilizar, através do *framework*, essa propriedade, que não é considerada básica, a sua utilização pode aumentar em muito a complexidade da extensão do agente genérico, o que motivou a sua não adoção para a construção dos agentes da máquina de processos.

3.5.6 – Mobilidade

A mobilidade é usualmente considerada como a habilidade do agente em se locomover através da Internet e se conectar a um outro ambiente. Segundo essa

concepção, o *framework* proposto não provê mobilidade. Entretanto, um agente pode, dentro do *framework*, se locomover de uma base de conhecimento para outra.

Quando um agente deseja se conectar ou desconectar de uma base de conhecimento, faz uso de métodos do tipo *plug* e *unplug*. Esses métodos verificam se a conexão é possível e utilizam um mecanismo de controle de concorrência para inibir o acesso concomitante de mais de um agente a uma mesma base de conhecimento.

Após desconectado de uma base de conhecimento, todos os agentes interessados nessa desconexão são notificados e o agente em questão pode se conectar a outra base de conhecimento. O *framework* proposto fornece uma estrutura central que permite o acesso à lista de bases de conhecimento existentes, possibilitando que o agente utilize o evento da sua interação pró-ativa para percorre-las em busca de novidades que favoreçam o cumprimento de seu objetivo.

3.5.7 – Colaboração

Como os agentes não utilizam uma linguagem para a sua comunicação, e nem possuem mecanismos que permitam a sua comunicação direta, é necessário fazer uso do ambiente como meio de comunicação e, conseqüentemente, de colaboração.

O ambiente é construído segundo uma ontologia bem definida. Essa ontologia serve como vocabulário comum para os agentes, permitindo que um agente compreenda o mundo em que está inserido, e o altere, possibilitando que outros agentes notem essa alteração e dêem procedimento à busca pelo seu objetivo.

Desta forma, a colaboração entre os agentes consiste na troca de informações, através da base de conhecimento, utilizando uma ontologia comum para a comunicação. Essa ontologia deve ser definida de acordo com o domínio em que os agentes serão inseridos, como foi feito neste trabalho para o domínio de automação de processos de software.

3.6 – Agente responsável pela simulação do processo

O Agente de Simulação é um agente reativo, que inicia a simulação assim que o mapeamento do processo é finalizado. Este tipo de simulação consiste na execução do processo raiz e dos seus sub-processos, visando determinar o tempo médio de execução de cada processo composto, utilizando como entrada os tempos médios de simulação dos processos primitivos e das decisões, que são informados pelo usuário no momento

da modelagem gráfica do processo. Como a simulação executa o processo, podem ser encontrados erros de modelagem durante esse procedimento.

Para que o agente possa terminar a sua execução mesmo que o processo não esteja correto, seu mecanismo pró-ativo monitora a simulação, verificando se ela está executando por um tempo muito grande e notificando ao desenvolvedor que pode existir um erro na modelagem do processo.

Durante a simulação do processo, é verificada a modelagem quanto a sua corretude. Os seguintes erros podem ser detectados:

- *Workflow* de processo sem nenhum caminho entre algum nó de início e algum nó de término;
- Soma das probabilidades de seleção de fluxos saindo de decisão diferente de 100%;
- Possibilidade de repetição infinita, tanto através de fluxo de retorno quanto através de recursão.

O algoritmo utilizado para a simulação [7](ROSS, 1990; AUDE, 1996) consiste em criar uma lista de eventos ordenada pelo tempo em que os eventos irão ocorrer, e adicionar todos os nós de início do *workflow* na lista com tempo zero. A partir desse momento, executar, recursivamente, pegando o primeiro elemento da lista, processando-o e adicionando os elementos subseqüentes ao elemento processado, que são os elementos que dependem do elemento processado para a execução, até que o elemento de término seja encontrado. O processamento dos elementos varia em função do tipo:

- **Início:** Não há processamento. É utilizado o tempo atual como o tempo de início de simulação para os elementos subseqüentes;
- **Processo Primitivo:** É utilizado o tempo esperado de execução modelado no processo, somado ao tempo atual, como o tempo de início de simulação para os elementos subseqüentes;
- **Processo Composto:** O processamento consiste na simulação recursiva do seu *workflow* e utilização do tempo obtido, somado ao tempo atual, como tempo de início de simulação para os elementos subseqüentes;

[7] Apesar de ter sido utilizada simulação baseada em eventos, o uso de métodos de análise estática seria possível. Todavia, a complexidade da sua aplicação seria maior, devido à existência de fluxos de retorno e recursão nos *workflows*.

- **Decisão:** É sorteado, utilizando uma distribuição uniforme, um dos fluxos de saída da decisão, levando em conta as probabilidades modeladas de seleção dos fluxos, e utilizado o tempo médio de resposta modelado na decisão, somado ao tempo atual, como tempo de início de simulação para o elemento destino do fluxo sorteado;
- **Sincronismo:** É verificado se todos os elementos que devem ser sincronizados já foram executados. Caso positivo, é utilizado o tempo atual como tempo de início de simulação para os elementos subseqüentes;
- **Término:** Não há processamento. O tempo atual é utilizado como tempo da simulação do *workflow*. Como a lista de eventos está ordenada pelo tempo, não é possível existir na lista outro elemento de término com tempo de simulação menor do que o tempo atual. Desta forma, o algoritmo utiliza como tempo de simulação de um *workflow* o mínimo entre os tempos de simulação dos diferentes fluxos, caso haja paralelismo.

O agente de simulação utiliza os estados exibidos no diagrama da Figura 3.11 para fazer o seu controle de execução.

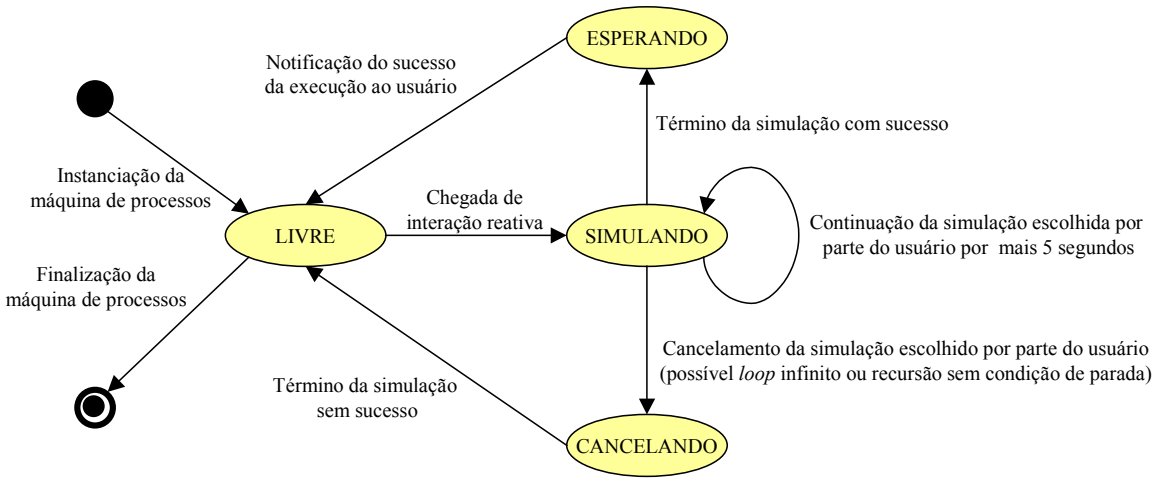

Figura 3.11: Diagrama de transição de estados do Agente de Simulação.

Inicialmente, o agente se encontra no estado LIVRE, esperando que alguma interação reativa ocorra. Após a chegada da interação reativa, o agente muda de estado para SIMULANDO e fica nesse estado até que ocorra uma interação pró-ativa ou até que termine a simulação, o que o faria passar para o estado ESPERANDO. A interação

pró-ativa verifica se o agente ainda está efetuando a simulação e avisa ao usuário que pode estar ocorrendo alguma repetição infinita por causa de fluxos de retorno ou recursão sem condição de parada, ou ainda, no caso do estado ESPERANDO, avisa que a simulação ocorreu com sucesso. Se a simulação ainda não foi concluída, em função da decisão do usuário, o agente passa para o estado CANCELANDO, ou continua no estado SIMULANDO. O estado CANCELANDO indica à interação pró-ativa que a execução da simulação deve ser cancelada.

3.7 – Agente responsável pela execução do processo

O agente de execução é o principal agente da máquina de processos. Ele conhece o que é necessário para controlar a execução de um processo, repassando o fluxo da execução para os elementos corretos quando um determinado elemento termina a sua execução.

Esse agente atua tanto de forma reativa, cadastrando-se como escutador do agente de acompanhamento e verificando se algo novo foi inserido na base de conhecimento do processo, quanto de forma pró-ativa, verificando se alguma base está pronta para entrar em execução e dando início a essa execução.

A atuação pró-ativa do agente consiste em verificar se alguma base de conhecimento já passou com sucesso pelo agente de simulação, o que faz com que seu estado seja PENDENTE. Caso sejam encontradas bases nesse estado, o agente procurará pela classe do processo raiz e criará a primeira instância desse processo, passando a base do estado PENDENTE para o estado EXECUTANDO.

Já a atuação reativa do agente consiste em tratar o evento de desconexão do agente de acompanhamento de uma base de conhecimento. O tratamento desse evento é composto pela busca de alterações na base que sofreu a desconexão. Essas alterações podem indicar que um processo primitivo foi finalizado ou uma decisão foi tomada. Em ambos os casos, o agente de execução deve verificar quais são os próximos elementos que entrarão em execução e dar prosseguimento a essas execuções. Após o tratamento do evento, é verificado se a instância inicial do processo raiz chegou ao seu término de execução, o que motivaria a mudança do estado da base de EXECUTANDO para FINALIZADA.

Desta forma, o objetivo do agente de execução é conseguir finalizar a instância inicial do processo raiz, e para isso ele faz uso dos planos descritos na Tabela 3.5, que é

composta pelo objetivo do plano, a representação do plano em regra Prolog e uma descrição textual explicando como cada plano faz para atingir o seu objetivo.

Tabela 3.5: Planos do agente de execução.

Objetivo	Representação Prolog	Semântica

Iniciar a execução de uma lista de elementos.	`inicia([],_,_).`
	
	`inicia([E|Es], C, T) :-`
	`  inicia(E, C, T),`
	`  !,`
	`  inicia(Es, C, T).`
	
	`inicia([_|Es], C, T) :-`
	`  !,`
	`  inicia(Es, C, T).`
		Caso a lista não seja vazia, tenta iniciar a execução do primeiro elemento da lista, e chama o início de execução do restante da lista, recursivamente.

Iniciar a execução de um elemento de início de *workflow*.	`inicia(inicio(Id), C, T) :-`
	`  !,`
	`  finaliza(inicio(Id), C, T).`
		Finaliza a execução do elemento de início (Id) em um determinado contexto (C), e com um tempo de início (T).

Iniciar a execução de um processo primitivo.	`inicia(processo(IdP), C, T) :-`
	`  classeProcesso(IdP, IdC),`
	`  processoPrimitivo(IdC),`
	`  !,`
	`  not(executando(processo(IdP), C, _)),`
	`  assertz(executando(processo(IdP), C, T)).`
		Caso o processo (IdP) pertença a uma classe de processos primitivos e não esteja atualmente em execução, o coloca em execução em um determinado contexto (C), e com um tempo de início (T).

Iniciar a execução de um processo composto.	`inicia(processo(IdP), C, T) :-`
	`  classeProcesso(IdP, IdC),`
	`  processoComposto(IdC),`
	`  !,`
	`  not(executando(processo(IdP), C, _)),`
	`  assertz(executando(processo(IdP), C, T)),`
	`  inicia(inicio(IdC), [processo(IdP)|C], T).`
		Caso o processo (IdP) pertença a uma classe de processos compostos e não esteja atualmente em execução, o coloca em execução em um determinado contexto (C), e com um tempo de início (T), e inicia a execução do elemento de início de seu *workflow*.

Iniciar a execução de um elemento de decisão.	`inicia(decisao(IdD), C, T) :-`
	`  !,`
	`  not(executando(decisao(IdD), C, _)),`
	`  assertz(executando(decisao(IdD), C, T)).`
		Caso a decisão (IdD) não esteja atualmente em execução, a coloca em execução em um determinado contexto (C), e com um tempo de início (T).

Iniciar a execução de um elemento de sincronismo.	`inicia(sincronismo(IdS), C, T) :-`
	`  not(executando(sincronismo(IdS), C, _)),`
	`  !,`
	`  assertz(executando(sincronismo(IdS), C, T)),`
	`  finaliza(sincronismo(IdS), C, T).`
	
	`inicia(sincronismo(IdS), C, T) :-`
	`  !,`
	`  finaliza(sincronismo(IdS), C, T).`
		Caso o sincronismo (IdS) não esteja atualmente em execução, o coloca em execução em um determinado contexto (C), e com um tempo de início (T). De qualquer forma, tenta executar a sua finalização.

Iniciar a execução de um elemento de término de *workflow*.	`inicia(termino, C, T) :-`
	`  !,`
	`  finaliza(termino, C, T).`
		Finaliza a execução do elemento de término em um determinado contexto (C), e com um tempo de término (T).

Finalizar a execução de um elemento de início de *workflow*.	`finaliza(inicio(IdC), C, T) :-`
	`  !,`
	`  findall(E, fluxo(inicio(IdC), E), Es),`
	`  inicia(Es, C, T).`
		Inicia a execução, em um determinado contexto (C), e com um tempo de início (T), da lista de elementos que são destino nos fluxos em que o elemento de início (IdC) é origem.

Finalizar a execução de um processo primitivo.	`finaliza(processo(IdP), C, T) :-`
	`  classeProcesso(IdP, IdC),`
	`  processoPrimitivo(IdC),`
	`  !,`
	`  executando(processo(IdP), C, Ti),`
	`  finalizado(IdP, C, Tf, _),`
	`  Tf > Ti,`
	`  Tf <= T,`
	`  retract(executando(processo(IdP), C, Ti)),`
	`  assertz(executado(processo(IdP), C, Ti, Tf)),`
	`  findall(E, fluxo(processo(IdP), E), Es),`
	`  inicia(Es, C, Tf).`
		Caso o processo (IdP) pertença a uma classe de processos primitivos, esteja atualmente em execução e tenha sido finalizado por algum usuário, o coloca como executado em um determinado contexto (C), e com um tempo de término (T), e inicia a execução da lista de elementos que são destino nos fluxos em que o processo é origem.

Finalizar a execução de um processo composto.	`finaliza(processo(IdP), C, T) :-`
	`  classeProcesso(IdP, IdC),`
	`  processoComposto(IdC),`
	`  !,`
	`  executando(processo(IdP), C, Ti),`
	`  retract(executando(processo(IdP), C, Ti)),`
	`  assertz(executado(processo(IdP), C, Ti, T)),`
	`  findall(E, fluxo(processo(IdP), E), Es),`
	`  inicia(Es, C, T).`
		Caso o processo (IdP) pertença a uma classe de processos compostos e esteja atualmente em execução, o coloca como executado em um determinado contexto (C), e com um tempo de término (T), e inicia a execução da lista de elementos que são destino nos fluxos em que o processo é origem.

Finalizar a execução de um elemento de decisão.	`finaliza(decisao(IdD), C, T) :-`
	`!,`
	`executando(decisao(IdD), C, Ti),`
	`respondido(IdD, C, R, Tr, _),`
	`Tr > Ti,`
	`Tr <= T,`
	`retract(executando(decisao(IdD), C, Ti)),`
	`assertz(executado(decisao(IdD), C, Ti, Tr)),`
	`findall(E, resposta(IdD, R, E), Es),`
	`inicia(Es, C, Tr).`
		Caso a decisão (IdD) esteja atualmente em execução e tenha sido respondida por algum usuário, a coloca como executada em um determinado contexto (C), e com um tempo de término (T), e inicia a execução da lista de elementos que são destino nos fluxos associados com a resposta selecionada.

Finalizar a execução de um elemento de sincronismo.	`finaliza(sincronismo(IdS), C, T) :-`
	`!,`
	`executando(sincronismo(IdS), C, Ti),`
	`findall(E1,(fluxo(E1,`
	`sincronismo(IdS)), E1 \\=`
	`decisao(_), executado(E1, C, _, Tf),`
	`Tf >= Ti, Tf <= T), E1s),`
	`findall(E2,(fluxo(E2,`
	`sincronismo(IdS)), E2 =`
	`decisao(IdD), respondido(IdD, C, R,`
	`Tr, _), resposta(IdD, R,`
	`sincronismo(IdS)), Tr >= Ti, Tr <=`
	`T), E2s),`
	`append(E1s, E2s, E3s),`
	`findall(E4,(fluxo(E4,`
	`sincronismo(IdS)), E4 \\= inicio(_),`
	`E4 \\= decisao(_)), E4s),`
	`findall(E5,(fluxo(E5,`
	`sincronismo(IdS)), E5 = decisao(_)),`
	`E5s),`
	`append(E4s, E5s, E6s),`
	`E3s = E6s,`
	`retract(executando(sincronismo(IdS),`
	`C, Ti)),`
	`assertz(executado(sincronismo(IdS),`
	`C, Ti, T)),`
	`findall(E7, fluxo(sincronismo(IdS),`
	`E7), E7s),`
	`inicia(E7s, C, T).`
		Caso o sincronismo (IdS) esteja atualmente em execução e todos os elementos que são origem nos fluxos que tem o sincronismo como destino já tenham sido executados, o coloca como executado em um determinado contexto (C), e com um tempo de término (T), e inicia a execução da lista de elementos que são destino nos fluxos em que o sincronismo é origem.

Finalizar a execução de um elemento de término de *workflow*.	`finaliza(termino, [processo(IdP)|C],`
	`T) :-`
	`!,`
	`finaliza(processo(IdP), C, T).`
		Finaliza a execução do processo que é dono do *workflow* em um determinado contexto (C), e com um tempo de término (T), em que se situa o elemento de término.

3.8 – Agente responsável pelo acompanhamento do processo

O agente de acompanhamento tem como objetivo fazer a ponte de comunicação entre a máquina de processos e os usuários, informando aos usuários quais elementos estão pendentes e adicionando à base de conhecimento as ações tomadas pelos usuários.

Esse agente atua de forma reativa, através da chamada por parte do usuário para verificar quais são as pendências. Quando o usuário acessa o agente, é possível

configurar um perfil que indica como aquele usuário deseja que o agente se comporte. Dentre as opções, é possível pedir para o agente aparecer sempre que o usuário entrar no ambiente de desenvolvimento que está sendo guiado por algum processo, de tempos em tempos, ou nunca. Se o perfil do usuário indicar que o agente não deve aparecer nunca, o agente passa a ter um comportamento puramente reativo para o usuário em questão.

A execução pró-ativa do agente consiste em verificar qual é o usuário atual do ambiente e acessar o seu perfil, vendo como esse usuário deseja que o agente se comporte. Caso, segundo o perfil, esteja na hora do agente interagir com o usuário, isso acontecerá de forma análoga ao que aconteceria se o usuário selecionasse o menu de ativação do agente.

Tabela 3.6: Planos do agente de acompanhamento.

Objetivo	Representação Prolog	Semântica

Buscar os processos primitivos pendentes.	`processoPendente(U, IdP, C) :-`
	`findall(PU, usuario(U,PU), PUs),`
	`!,`
	`executando(processo(IdP), C, _),`
	`classeProcesso(IdP, IdC),`
	`processoPrimitivo(IdC),`
	`findall(PP, papel(IdC, PP), PPs),`
	`intersecao(PUs, PPs).`
		Busca por algum processo pendente (IdP) em algum contexto (C), que tem papéis compatíveis com os papéis do usuário (U).

Buscar as decisões pendentes.	`decisãoPendente(U, IdD, C, Rs) :-`
	`findall(PU, usuario(U,PU), PUs),`
	`!,`
	`executando(decisao(IdD), C, _),`
	`findall(PD, papel(IdD, PD), PDs),`
	`intersecao(PUs, PDs),`
	`findall(R, resposta(IdD, R, _),`
	`Rs).`
		Busca por algum processo pendente (IdP) em algum contexto (C), com suas possíveis respostas (Rs) que tem papéis compatíveis com os papéis do usuário (U).

Verificar a Interseção entre listas (requisito das outras cláusulas).	`intersecao([X|L1], L2) :-`
	`member(X,L2),`
	`!.`
	
	`intersecao([_|L1], L2) :-`
	`intersecao(L1, L2),`
	`!.`
		Verifica se existe uma interseção entre as listas L1 e L2. Caso positivo, retorna SIM somente na primeira resposta, e NÃO nas demais.

Durante a interação, o agente fornece a lista de processos atribuídos ao usuário que estão atualmente em execução. Caso o usuário deseje, o agente fornece informações detalhadas sobre como executar um determinado processo da lista. O agente também fornece a lista de decisões atribuídas ao usuário que estão pendentes de respostas. Caso o usuário deseje, o agente lista a pergunta e todas as respostas possíveis para cada

decisão que deve ser tomada. A construção dessas listas ocorre através da coleta de todas as respostas das consultas feitas à base de conhecimento descritas na Tabela 3.6.

O usuário pode interagir com o agente ordenando que algum processo seja finalizado ou que uma determinada resposta seja utilizada na tomada de uma decisão. Isso fará com que o agente altere a base de dados do processo, sinalizando a decisão do usuário.

Após a interação, quando o agente de acompanhamento se desconectar da base, o agente de execução será notificado desse evento e dará andamento à execução do processo levando em conta os novos fatos introduzidos na base.

3.9 – Suporte para o monitoramento de processos

Após a instanciação do processo, a sua execução é iniciada, tornando necessário o uso de uma notação que permita visualizar o estado atual desta execução. Essa notação deve ser capaz de expressar, de forma clara, quantas vezes cada processo de um *workflow* foi executado, como foi o desempenho dessas execuções em relação à previsão obtida pela simulação do processo e quais processos estão em execução no momento. Essas informações são úteis para os gerentes do projeto detectarem gargalos no processo e tomarem as devidas providências, sejam elas: redimensionar a equipe, realocar os desenvolvedores em outros papéis ou otimizar o próprio processo, minimizando os gargalos.

Para atender a esses requisitos, optamos por utilizar uma notação que se sobrepõe à notação estendida do diagrama de atividades da UML. Essa notação utiliza o diagrama construído na etapa de modelagem do processo, modificando o desenho dos elementos para passar a semântica desejada.

Para representar as várias versões de execução de um processo, utilizamos a sobreposição dos desenhos do processo (HEIMANN et al., 1996), adicionada a uma convenção de cores que indica como foi a execução, onde verde (tom de cinza intermediário, caso impresso em preto e branco) representa uma execução dentro do tempo simulado, e vermelho (tom de cinza escuro, caso impresso em preto e branco) representa uma execução com tempo pior do que o tempo simulado. Para representar os processos que estão em execução no momento, utilizamos a cor amarela (tom de cinza claro, caso impresso em preto e branco).

A Figura 3.12 exemplifica essa notação exibindo um *workflow* onde o processo A foi executado com tempo pior que o tempo previsto pela simulação. Após a sua

execução, os processos B e C entraram em execução. O processo B foi executado com tempo melhor que o tempo de simulação e deu início à execução do processo D, que ainda está em execução. Já o processo C foi executado completamente duas vezes e está em uma terceira execução. A primeira execução de C foi com um tempo melhor que o tempo de simulação e a segunda com um tempo pior. Sempre após as execuções do processo C, uma decisão tem que ser tomada. Essa decisão foi tomada duas vezes, motivando as reexecuções do processo C. A primeira decisão demorou mais do que o esperado, mas a segunda decisão foi dentro do prazo.

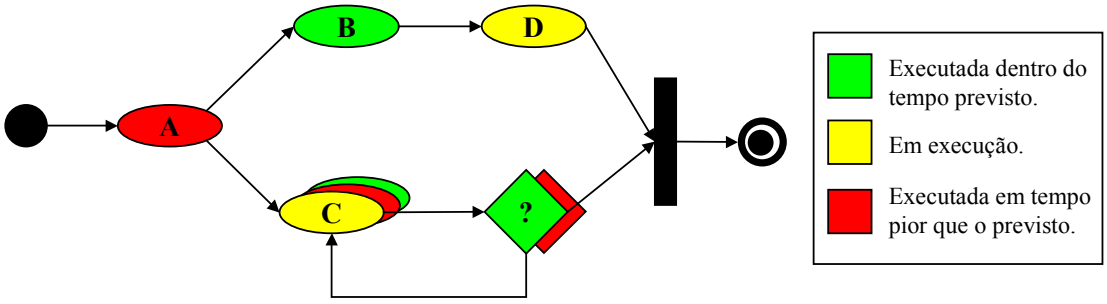

Figura 3.12: Exemplo da notação de monitoramento da execução de processos.

Caso algum processo desenhado no *workflow* seja composto, a mesma notação é aplicada para o seu *workflow*, detalhando o estado das diversas execuções de cada um de seus elementos.

Como todas as informações sobre a simulação e execução dos processos residem na base de conhecimento, o ambiente de monitoramento da execução deve fazer consultas a essa base para adquirir essas informações e processá-las.

3.10 – Exemplo de extensão da máquina de processos

A principal vantagem de construir uma máquina de processos baseada em agentes inteligentes é a facilidade de manutenção dessa máquina. A manutenção de qualquer tipo de software pode ser dividida em quatro tipos diferentes (PRESSMAN, 1997):

- **Manutenção corretiva:** consiste em corrigir erros existentes no software. Esse tipo de manutenção não será beneficiado com o uso dos agentes caso os erros encontrados estejam no próprio *framework*;
- **Manutenção adaptativa:** consiste em adaptar o software para refletir mudanças ocorridas no ambiente em que o software atua. A estruturação

modular dos agentes pode contribuir na localização do ponto em que deve ocorrer a adaptação;

- **Manutenção evolutiva:** consiste em evoluir o software para atender a novas funcionalidades desejadas pelo usuário. Esse tipo de manutenção é o principal beneficiado pelo uso de agentes, pois a extensão da máquina de processos pode ser feita através da criação de novos agentes, não necessitando alterações nos demais agentes nem na estrutura central da máquina;
- **Manutenção preventiva:** consiste em modificar a estrutura do sistema para facilitar manutenções futuras. Esse tipo de manutenção normalmente é necessário quando o software já passou por várias manutenções dos outros tipos, deteriorando sua estrutura. A estrutura dos agentes inteligentes, que predetermina um comportamento em relação à sua comunicação com os outros elementos do sistema, pode minimizar a necessidade de manutenções preventivas.

Para exemplificar a extensão da máquina de processos, vamos supor que surgiu um novo requisito, que consiste na necessidade de voltar a execução do processo para um determinado instante de tempo. Esse requisito pode ser necessário para corrigir erros de finalizações de processo indevidas ou tomadas de decisão impróprias.

Esse tipo de manutenção encaixa-se em manutenção evolutiva, pois trata-se de um requisito novo, que adiciona uma funcionalidade desejada pelo usuário à máquina de processos.

Para atendermos a esse novo requisito, é necessária a construção de um novo agente. Como todas as informações sobre a execução do processo estão na base de conhecimento, esse novo agente deve interagir com o usuário, verificando para qual instante de tempo a base deve ser deslocada e reconstruindo os fatos da base para refletirem esse deslocamento.

A reconstrução dos fatos da base é composta pela remoção dos acontecimentos que se iniciaram posteriormente à data desejada e modificação dos acontecimentos que se iniciaram antes da data desejada, mas ainda estão ativos ou foram finalizados depois da data desejada. A Tabela 3.7 exibe os planos do novo agente para o deslocamento da base no tempo.

Tabela 3.7: Planos do agente de deslocamento no tempo.

Objetivo	Representação Prolog	Semântica

Remover execuções posteriores à data desejada.	`Desloca(T) :-`
	`  executando(E, C, Ti),`
	`  Ti > T,`
	`  retract(executando(E, C, Ti)),`
	`  !,`
	`  desloca(T).`
	
	`desloca(T) :-`
	`  executado(E, C, Ti, Tf),`
	`  Ti > T,`
	`  retract(executado(E, C, Ti, Tf)),`
	`  !,`
	`  desloca(T).`
	
	`desloca(T) :-`
	`  finalizado(IdP, C, Tf, U),`
	`  Tf > T,`
	`  retract(finalizado(IdP, C, Tf, U)),`
	`  !,`
	`  desloca(T).`
	
	`desloca(T) :-`
	`  respondido(IdD, C, R, Tr, U)`
	`  Tr > T,`
	`  retract(respondido(IdD, C, R, Tr, U)),`
	`  !,`
	`  desloca(T).`
		Caso exista execução em andamento ou já finalizada com data de início posterior à data desejada (T), então remove. Caso exista pedido de finalização de processo ou resposta de decisão com data posterior à data desejada (T), então remove.

Modificar execuções com data de início anterior a data desejada, mas com data de término posterior.	`desloca(T) :-`
	`  executado(E, C, Ti, Tf),`
	`  Ti <= T,`
	`  Tf > T,`
	`  retract(executado(E, C, Ti, Tf)),`
	`  assertz(executando(E, C, Ti)),`
	`  !,`
	`  desloca(T).`
		Caso exista execução já finalizada com data de início anterior ou igual à data desejada (T) e data de término posterior, então modifica colocando a execução em andamento com a mesma data de início.

O procedimento para atender a outros requisitos de extensão da máquina de processos seria similar a esse, através da construção de um novo agente, sem alterar os demais elementos da máquina. Esse tipo de característica é possível graças à separação proporcionada pela infra-estrutura entre a funcionalidade dos agentes (controle) e as informações sobre o processo e sua execução (modelo).

3.11 – Conclusões

A abordagem proposta neste capítulo atende aos critérios levantados no capítulo 2 através das seguintes características:

- **Modelagem do processo:** A modelagem de processos ocorre através de um ambiente gráfico, utilizando uma extensão da notação de diagramas de atividades da UML. Esse processo modelado será traduzido para Prolog e constituirá a base de conhecimento do processo;
- ***Workflow* com ciclos ou recursão:** O diagrama de processos permite tanto a utilização de ciclos dentro do *workflow*, quanto a referência recursiva a um processo que faz parte da estrutura hierárquica do processo em questão;
- **Mapeamento automático:** A tradução entre o processo diagramado e o processo executável ocorre através de um mecanismo de mapeamento que cria identificadores para os processos e constrói predicados Prolog utilizando esses identificadores. Todos os predicados e processos são mantidos armazenados para uma posterior consulta sobre qual predicado representa qual processo e vice-versa;
- **Interação pró-ativa:** A máquina de processos não se limita a responder a estímulos provenientes dos desenvolvedores. Isso se dará através da utilização de agentes inteligentes. Os agentes são dotados de temporizadores que motivam o andamento da execução do processo, de tempos em tempos, mesmo que não ocorra nenhum evento externo;
- **Processos reutilizáveis:** Os processos são modelados sem a necessidade de fixar o nível de abstração, permitindo que eles sejam utilizados em qualquer *workflow* de qualquer processo modelado. Desta forma, o processo pode ser visto como um artefato reutilizável que sistematiza uma determinada tarefa. Esse artefato pode ser reutilizado em diferentes contextos que necessitem da sistematização desta tarefa;
- **Extensão flexível:** Conforme dito anteriormente, a implementação da abordagem ocorre através de agentes inteligentes. Esses agentes são construídos a partir da instanciação de um *framework*, que define as propriedades existentes em um agente genérico, e pode ser especializado para a construção de novos agentes. Com isso, a extensão da máquina de processo para atender a um novo requisito, como a necessidade de retrocesso da execução para um determinado momento, pode ser obtida através da construção de um novo agente para esse fim;

- **Foco em processos de software:** Apesar de ser possível utilizar a abordagem proposta para automatizar a execução de outros tipos de processos, como processos de negócio, o objetivo principal é prover suporte para a execução de processos de software. Para que essa característica fosse atendida, além do fato de estar aderente às características de processos de software, a máquina de processos se localizará dentro de um ambiente de desenvolvimento de software, que tem todo o ferramental necessário disponível para a realização das atividades do processo, quando a máquina de processos indicar que essas atividades podem ser executadas;

No capítulo 4 é descrito o protótipo que implementa a abordagem proposta, de acordo com as características levantadas utilizando os critérios propostos.

Capítulo 4 – Protótipo Implementado

4.1 – Introdução

Neste capítulo são descritos o protótipo, que implementa as idéias descritas no Capítulo 3, e a sua utilização, através da construção de um sistema exemplo. Esse sistema exemplo é o CtrlPESC, que se destina ao controle acadêmico do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. A escolha do CtrlPESC se deve a disponibilidade de informações que descrevem o processo quanto a estruturação das suas atividades e ao tempo médio de execução de cada atividade e ao fato de o sistema ser simples, facilitando a exibição das características do protótipo e minimizando as dificuldades inerentes à complexidade do sistema exemplo.

Inicialmente, é apresentada a estrutura interna do protótipo através de um detalhamento técnico. Posteriormente, descrevemos o processo utilizado para a construção do CtrlPESC e, finalmente, a utilização do protótipo para exemplificar como poderia ter sido desenvolvido o CtrlPESC caso o protótipo tivesse sido utilizado[8].

4.2 – Detalhamento técnico da arquitetura

O protótipo utiliza a infra-estrutura de construção de agentes inteligentes descrita no capítulo 3 para facilitar a criação de novos agentes. A implementação consiste na junção das linguagens Java e Prolog, através do uso dos agentes.

Visto que a base de conhecimento de processos representa um local acessível e dinâmico, todos os agentes conseguem ver todo o local onde estão inseridos e alterá-lo a qualquer instante. Desta forma, os agentes podem anexar seus planos, que foram mapeados em regras Prolog, à base de conhecimento e utilizar a máquina de inferência do próprio local em que estão inseridos.

Esta abordagem é simples, pois todo o conhecimento de todos os agentes pode ser obtido no local onde os agentes são inseridos. Entretanto, é necessária uma definição clara do significado de cada elemento existente nesse local para que os agentes possam

[8] O CtrlPESC foi desenvolvido antes do desenvolvimento desse trabalho, utilizando outras ferramentas de engenharia de software. O exemplo aqui exibido serve apenas como um guia de utilização da máquina de processo, pois não retrata a execução real do processo de construção do CtrlPESC.

utilizar um vocabulário comum. Essa definição foi obtida através da construção de uma ontologia para o domínio em questão.

Desta forma, teremos somente uma máquina de inferência, localizada no local onde os agentes são inseridos, e o seguinte mapeamento entre os atributos de um agente inteligente e a implementação pode ser feito:

- Crenças ↔ fatos Prolog no ambiente.
- Planos ↔ regras Prolog transportadas do agente para o ambiente.
- Objetivo ↔ tempotizador Java + regras Prolog transportadas do agente para o ambiente.
- Sensores e Atuadores ↔ Prolog seguindo a ontologia definida + local onde os agentes são inseridos.

Cada agente, segundo essa abordagem, é uma classe Java que herda da classe *Agente*. Esses agentes se conectam com as bases de conhecimento dos processos que contêm um interpretador Prolog embutido.

Para que a base de conhecimento possa ser criada, o processo deve ser instanciado e mapeado, trabalho este delegado, respectivamente, às classes *InstanciadorProcesso*, que irá criar a base para um determinado processo raiz, e *Mapeador*, que irá povoar essa base com os fatos que representam esse processo raiz e os seu demais sub-processos, em Prolog. A Figura 4.1 exibe o diagrama de classes do protótipo, que contém os elementos utilizados na sua construção.

Como vários agentes podem desejar se conectar à base de conhecimento ao mesmo tempo, gerando problemas de controle de concorrência, pois cada agente possui a sua própria *thread*, a base de conhecimento mantém um controle de qual agente está atualmente conectado nela, não permitindo a conexão concomitante de mais de um agente.

O agente de acompanhamento mantém, para cada desenvolvedor cadastrado no ambiente, uma lista de preferências, que indica como o agente deve agir na presença desse desenvolvedor.

Para facilitar o uso da máquina de processos pelos outros módulos do ambiente, seus serviços foram encapsulados na classe *GerenteProcesso*. Essa classe implementa o padrão *Facade* (GAMMA et al., 1994), que torna possível o acesso a um único ponto do sistema para a requisição de serviços. Quando algum serviço é solicitado ao *GerenteProcesso*, este o delega para a classe responsável da máquina de processos, sem

que o cliente tenha conhecimento, tornando a execução transparente e permitindo a redução da complexidade de utilização da máquina de processos.

Figura 4.1: Detalhamento técnico do protótipo.

4.3 – O sistema exemplo

O CtrlPESC visa possibilitar o controle acadêmico dos alunos do Programa de Engenharia de Sistemas e Computação da COPPE. Para que fosse possível a sua construção, foi definido um processo (BARROS, 2001) que leva em conta os seus módulos principais, relacionados com as entidades envolvidas com o controle acadêmico, que são:

- Usuários administrativos (secretaria e coordenadores);
- Professores;
- Linhas de pesquisa;
- Disciplinas;
- Alunos;
- Inscrições.

Para simplificar o processo, as entidades foram agrupadas duas a duas, segundo o seu grau de afinidade, resultando nos seguintes grupos:

- Usuários administrativos e professores;
- Linhas de pesquisa e disciplinas;
- Alunos e inscrições.

O processo selecionado para o desenvolvimento do sistema segue o ciclo de vida cascata, devido ao tamanho e simplicidade do sistema alvo. Desta forma, cada uma das entidades teve que ser especificada, projetada, implementada e testada, produzindo artefatos intermediários, através das seguintes atividades:

1. **Análise de requisitos:** Construção dos casos de uso de cada uma das entidades;
2. **Projeto arquitetural:** Construção dos diagramas de classes que descrevem a estrutura do sistema como um todo;
3. **Projeto detalhado:** Construção dos diagramas de classes, seqüência e estados que descrevem cada um dos cadastros das entidades;
4. **Implementação:** Codificação dos cadastros das entidades;
5. **Teste:** Verificação dos casos de uso em relação ao sistema codificado.

A partir do cálculo de pontos por função (IFPUG, 1999), levando em conta a média de pontos produzidos por desenvolvedor/mês, que é de 27,8 (JONES, 2000), a distribuição de esforço por atividade (JONES, 2000) e utilizando uma produção de 160 horas por homem por mês, chegamos às estimativas de tempo exibidas na Tabela 4.1 para as atividades do processo, supondo que cada atividade será executada por somente um desenvolvedor, o que aconteceu no caso do CtrlPESC.

A duração estimada de cada atividade está expressa em horas úteis, que indicam a quantidade de horas contínuas de trabalho necessárias para a execução de cada atividade. Esse valor foi obtido através da divisão de um mês em 160 horas, levando em conta uma carga de trabalho de 8 horas por dia durante 20 dias. Entretanto, para uma análise real do tempo necessário para completar cada atividade, a partir de um determinado instante, foi utilizado o conceito de horas corridas, onde um mês é composto por 720 horas, pois são consideradas as 24 horas de um dia durante 30 dias.

Do ponto de vista de análise da Tabela 4.1, o conceito de horas úteis pode ser utilizado para informar quanto tempo de trabalho ininterrupto em média é necessário para concluir o projeto, e o conceito de horas corridas pode ser utilizado para determinar o tempo real necessário para a conclusão do projeto, levando em conta que um desenvolvedor trabalha de segunda-feira a sexta-feira durante 8 horas por dia.

Tabela 4.1: Estimativas de tempo gasto nas atividades do processo.

Atividade	Pontos por Função	Estimativa de horas úteis	Estimativa de horas corridas
Análise de requisitos de usuários administrativos e professores	0,81	4,66	20,98
Análise de requisitos de linhas de pesquisa e disciplinas	0,84	4,83	21,76
Análise de requisitos de alunos e inscrições	0,96	5,52	24,86
Projeto arquitetural	2,53	14,56	65,52
Projeto detalhado de usuários administrativos e professores	2,31	13,29	59,83
Projeto detalhado de linhas de pesquisa e disciplinas	2,39	13,75	61,90
Projeto detalhado de alunos e inscrições	2,73	15,71	70,70
Implementação de usuários administrativos e professores	4,05	23,31	104,89
Implementação de linhas de pesquisa e disciplinas	4,20	24,17	108,78
Implementação de alunos e inscrições	4,80	27,63	124,32
Testes de usuários administrativos e professores	11,81	67,97	305,87
Testes de linhas de pesquisa e disciplinas	12,25	70,50	317,26
Testes de alunos e inscrições	14,00	80,57	362,59
TOTAL	63,68	366,50	1649,25

4.4 – Utilização do protótipo

A utilização do protótipo pode ser descrita através de um processo simples, composto pelas seguintes atividades:

1. Inicialmente, é necessário modelar o processo graficamente;
2. A partir de um processo modelado, é possível iniciar a construção de um domínio ou de uma aplicação através da instanciação do processo;
    a. A instanciação inicia pela seleção do processo raiz;
    b. Após a seleção, deve-se definir quais papéis serão exercidos por quais desenvolvedores;
3. Com o processo instanciado, o Agente de Simulação tenta executar a simulação e, a partir do resultado, informa se é possível dar início a execução real do processo;
4. O Agente de Execução coloca o processo raiz em execução, permitindo que o desenvolvimento possa ser iniciado;
5. Quando o desenvolvedor entrar no ambiente de desenvolvimento, o Agente de Acompanhamento irá informar quais atividades estão pendentes e quais decisões devem ser tomadas;
6. Sempre que alguma atividade pendente for finalizada ou alguma decisão for tomada, o Agente de Execução será acionado para dar continuidade à execução do processo;
7. Durante a execução:
    a. O ambiente de monitoramento do processo poderá ser acessado para verificar quais atividades já foram executadas, quantas vezes foram executadas e como foi a execução em relação à previsão gerada pelo Agente de Simulação;
    b. Um menu poderá ser acionado para verificar as atividades e decisões pendentes;
    c. Um menu poderá ser acionado para retroceder o processo para um determinado instante de tempo;
    d. Um menu poderá ser acionado para reinstanciar o processo, refletindo alterações no seu modelo;
8. Quando o processo raiz for finalizado, o Agente de Acompanhamento informará que o processo de desenvolvimento terminou. Entretanto, será

possível reiniciá-lo através de um retrocesso, perdendo as informações do que foi feito, ou através de uma reinstanciação, mantendo os dados anteriores sobre o processo.

Nas sub-seções seguintes são detalhadas as atividades de utilização do protótipo, tomando como exemplo o desenvolvimento do sistema CtrlPESC. Apesar do Ambiente Odyssey, onde o protótipo foi construído, ser um ambiente para suporte a reutilização, as atividades relacionadas com a reutilização foram omitidas para tornar o processo mais simples.

4.4.1 – Modelagem gráfica de processo

Inicialmente, o processo que será utilizado como raiz deve ser modelado. Esse processo deve descrever cada uma das atividades do ciclo de vida cascata, que também devem ser modeladas. A Figura 4.2 exibe o ambiente de modelagem de processos, onde o ciclo de vida cascata foi modelado.

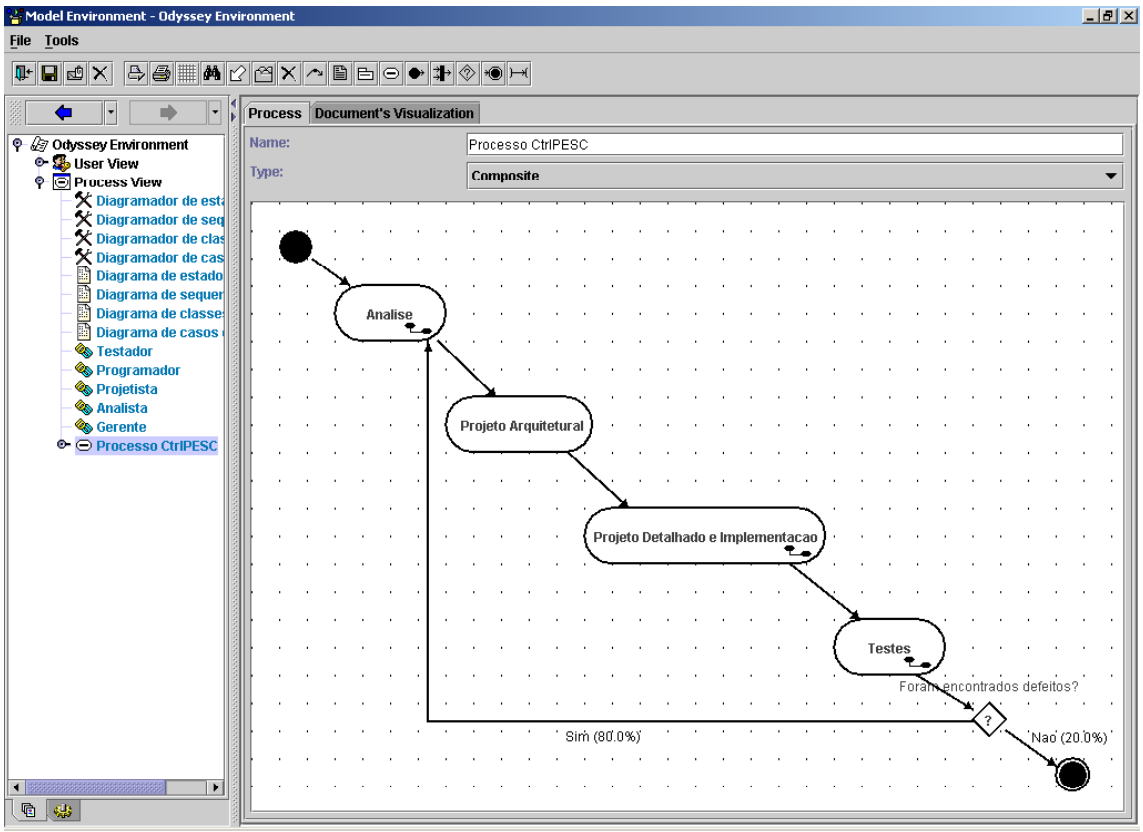

Figura 4.2: Modelagem do processo que representa o ciclo de vida cascata.

Neste processo, foi utilizada uma decisão que, após os testes, verifica se foram encontrados erros. Caso positivo, o processo é reiniciado para permitir a correção dos

erros. Caso contrário, o processo é finalizado. O objetivo desse exemplo é mostrar o uso de um fluxo de retorno. Entretanto, para o processo modelado se tornar mais condizente com a realidade, outros sub-processos deveriam estar definidos para o retorno, pois a segunda execução do sub-processo de "Análise", por exemplo, não demandará o mesmo tempo de execução da primeira iteração, e, possivelmente, também terá um roteiro de execução diferente. Desta forma, não seriam versões de um mesmo processo, mas sim, processos diferentes.

Essa decisão recebeu, durante a modelagem, os valores de probabilidade de ocorrência de 80% para a localização de erros e 20% para a não localização de erros. Esses valores, que são editáveis, foram estimados sem precisão, somente para a exemplificação das características do protótipo. Para que os valores fossem corretos, seria necessário verificar qual o número médio de erros existentes em sistemas dessa complexidade (E), qual é a porcentagem de erros encontrados na atividade de teste (EE) e qual é a taxa de geração de erros durante as atividades de manutenção (NE). Com esses valores, seria possível estimar, com alguma precisão, quantas vezes seria necessário repetir o processo até que o sistema se encontrasse em um nível aceitável de erros. Para isso, [E – N * (EE - NE)] deve ser menor ou igual ao número de erros aceitável para essa categoria de sistema, sendo N o número de repetições necessárias do processo.

Outra característica do processo modelado que deve ser ressaltada é que as atividades de projeto detalhado e implementação foram unidas em uma única atividade, para permitir a otimização do paralelismo entre as suas sub-atividades, como exibido na Figura 4.11.

O símbolo  indica que determinadas atividades são compostas, contendo um outro *workflow* que descreve as suas sub-atividades. Desta forma, "Análise", "Projeto Detalhado e Implementação" e "Testes" são atividades compostas, e "Projeto Arquitetural" é uma atividade primitiva.

Também na Figura 4.2, na lista da esquerda, estão descritos alguns dos elementos existentes no sistema. Dentro da visão de processos, podem ser encontradas quatro ferramentas, que são os diagramadores de estados, seqüência, classes e casos de uso, quatro artefatos, que são diagramas de estado, seqüência, classes e casos de uso, e cinco papéis, que são testador, programador, projetista, analista e gerente, além do processo principal do CtrlPESC, que contém os demais sub-processos. O nível de decomposição de cada elemento deve ser definido de acordo com a necessidade do

projeto. Neste exemplo, estamos generalizando os artefatos pelo seu tipo, definindo somente um artefato como, por exemplo, “diagrama de casos de uso”, ao invés de definir vários artefatos, um para cada diagrama de caso de uso existente, como, por exemplo, “Diagrama de casos de uso de usuários administrativos e professores”.

Além de modelar o processo principal, que será utilizado como processo raiz para a construção do CtrlPESC, é necessário modelar os seus sub-processos. Para exemplificar essa modelagem, utilizamos o sub-processo primitivo “Projeto Arquitetural” e o sub-processo composto “Testes”.

O processo “Projeto Arquitetural” não pode ser executado em paralelo com nenhuma outra atividade, pois a sua função necessita dos resultados completos do processo de “Análise” e os seus resultados são pré-condição para a execução do processo de “Projeto Detalhado e Implementação”.

Como todo processo primitivo, o “Projeto Arquitetural” foi descrito através de seu tempo médio de execução, que é 65,52 horas corridas, o equivalente a aproximadamente 10% de um mês de trabalho de um desenvolvedor; um roteiro, que descreve o que deve ser feito para que o processo seja executado; a lista de papéis, que indica quais tipos de desenvolvedor estão autorizados a executar esse processo; as listas de artefatos consumidos e produzidos, que indicam o que deve ser utilizado para que o roteiro possa ser cumprido; e a lista de ferramentas, que indica quais ferramentas devem ser utilizadas para que os artefatos consumidos e produzidos possam ser manipulados. Essa descrição está exibida na Figura 4.3.

Já o processo “Testes”, descreve um *workflow* que executará em paralelo os testes individuais de cada módulo, que são representados pelos processos “Teste de Usuários e Professores”, “Teste de Linhas e Disciplinas” e “Teste de Alunos e Inscrições”. Após o término do último teste de módulo, é iniciado o teste do sistema como um todo, que visa verificar se a comunicação entre os módulos está correta em função da reação do sistema descrita nos casos de uso. A Figura 4.4 exibe o ambiente de modelagem de processos com o processo “Testes” sendo modelado.

Após a modelagem de todos os sub-processos necessários para a execução do processo principal do CtrlPESC, é possível iniciar a instanciação do processo principal.

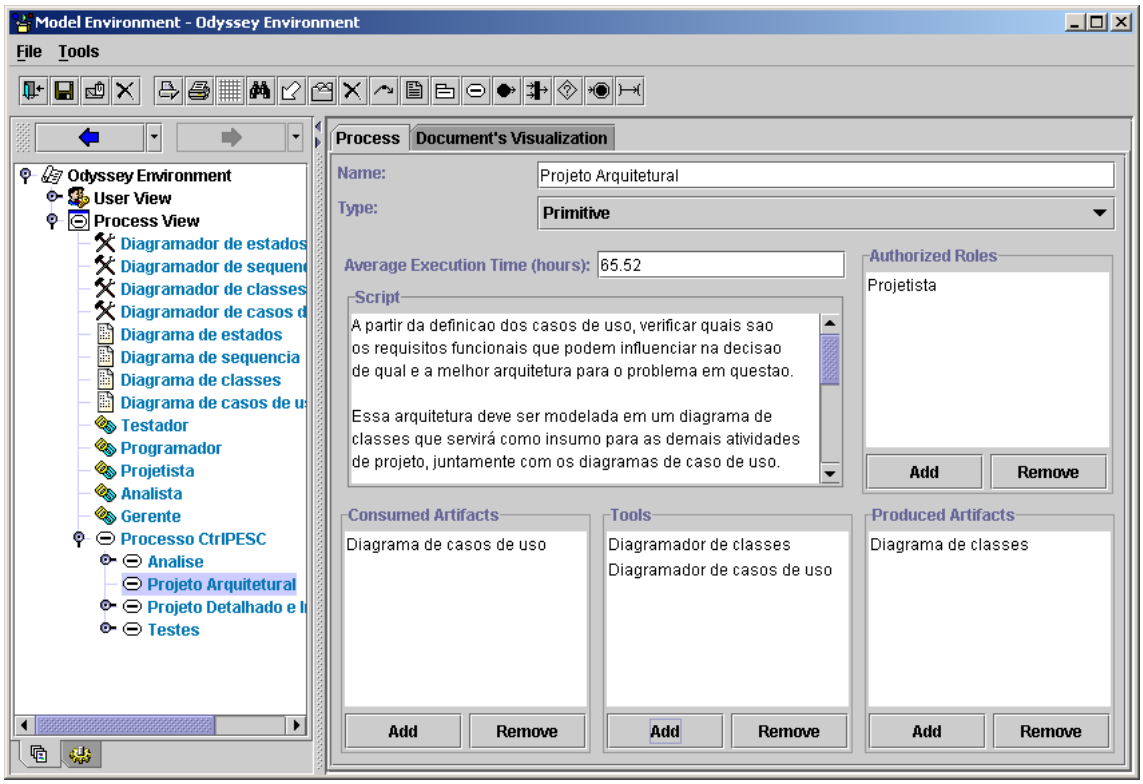

Figura 4.3: Modelagem do processo que representa a atividade de projeto detalhado.

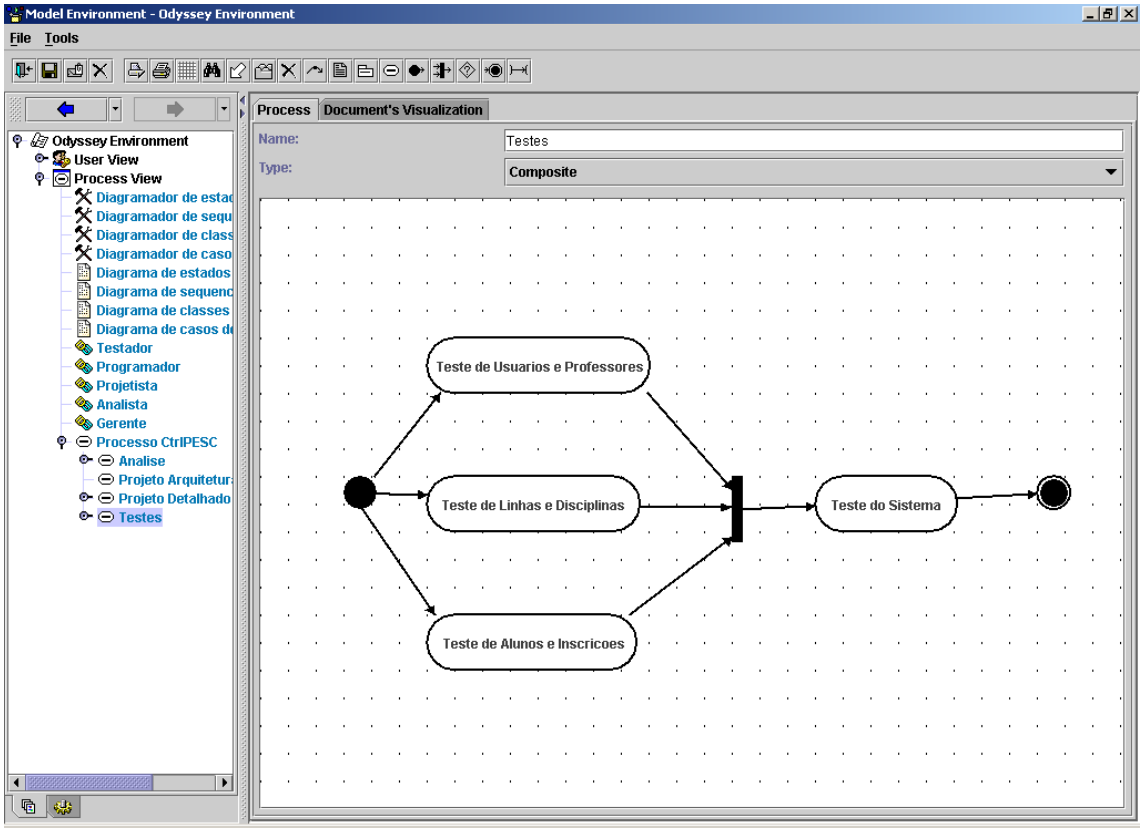

Figura 4.4: Modelagem do processo que representa a atividade de testes.

4.4.2 – Instanciação do processo

A instanciação do processo, que ocorre no momento da criação do domínio ou da aplicação no ambiente Odyssey, consiste inicialmente da seleção do processo raiz, como exibido na Figura 4.5.

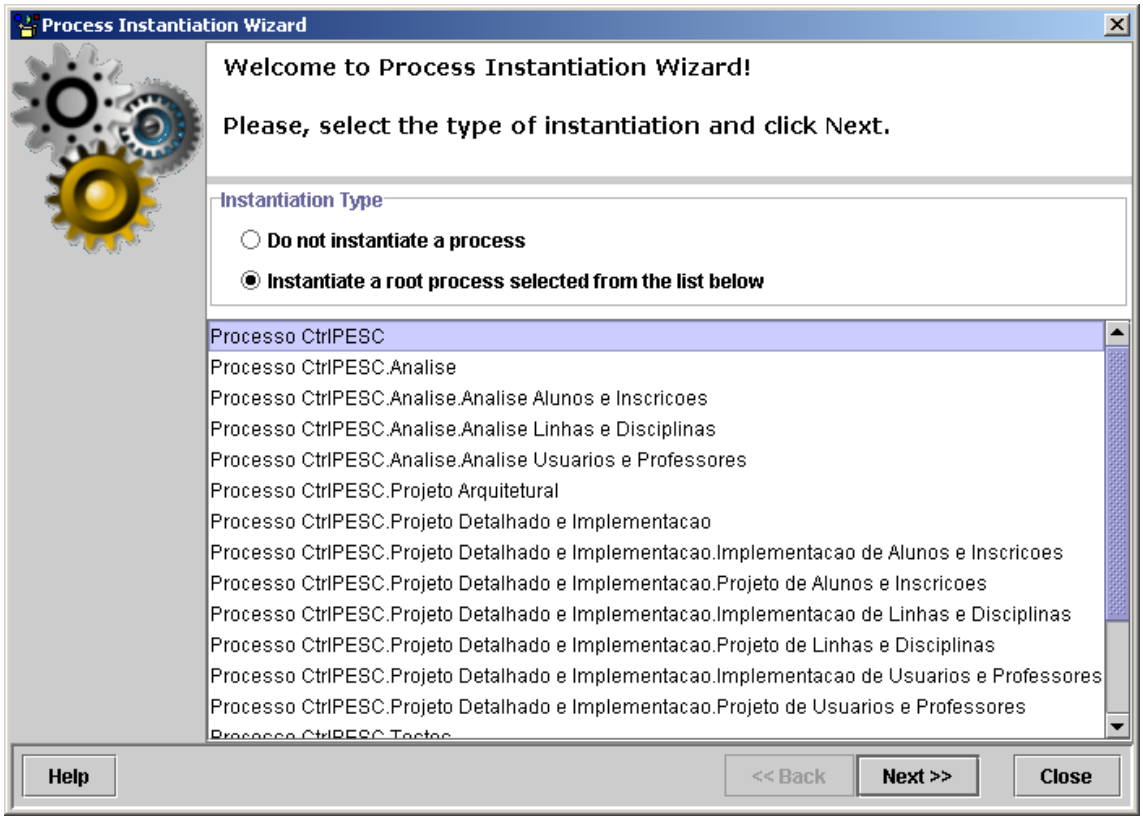

Figura 4.5: Seleção do processo raiz durante a instanciação do processo.

O domínio ou a aplicação que está sendo criado no Odyssey pode não assumir um processo. Neste caso, o desenvolvimento será *ad-hoc* ou utilizará uma máquina de processos externa ao ambiente, o que, em ambos os casos, é ruim para o desenvolvimento, pois deixa de utilizar todo o potencial de integração entre a máquina de processos e o ambiente de desenvolvimento de software.

Caso se deseje instanciar um processo, essa opção deve ser escolhida, juntamente com o nome do processo raiz da aplicação, que no caso do CtrlPESC é o "Processo CtrlPESC".

Após a seleção do processo raiz, ocorre o mapeamento do processo selecionado em fatos Prolog (exibidos no Apêndice A) e a seleção dos papéis pendentes, para que seja possível informar à máquina de processos quais desenvolvedores estão autorizados a atuar em quais processos, exercendo quais papéis, como exibido na Figura 4.6.

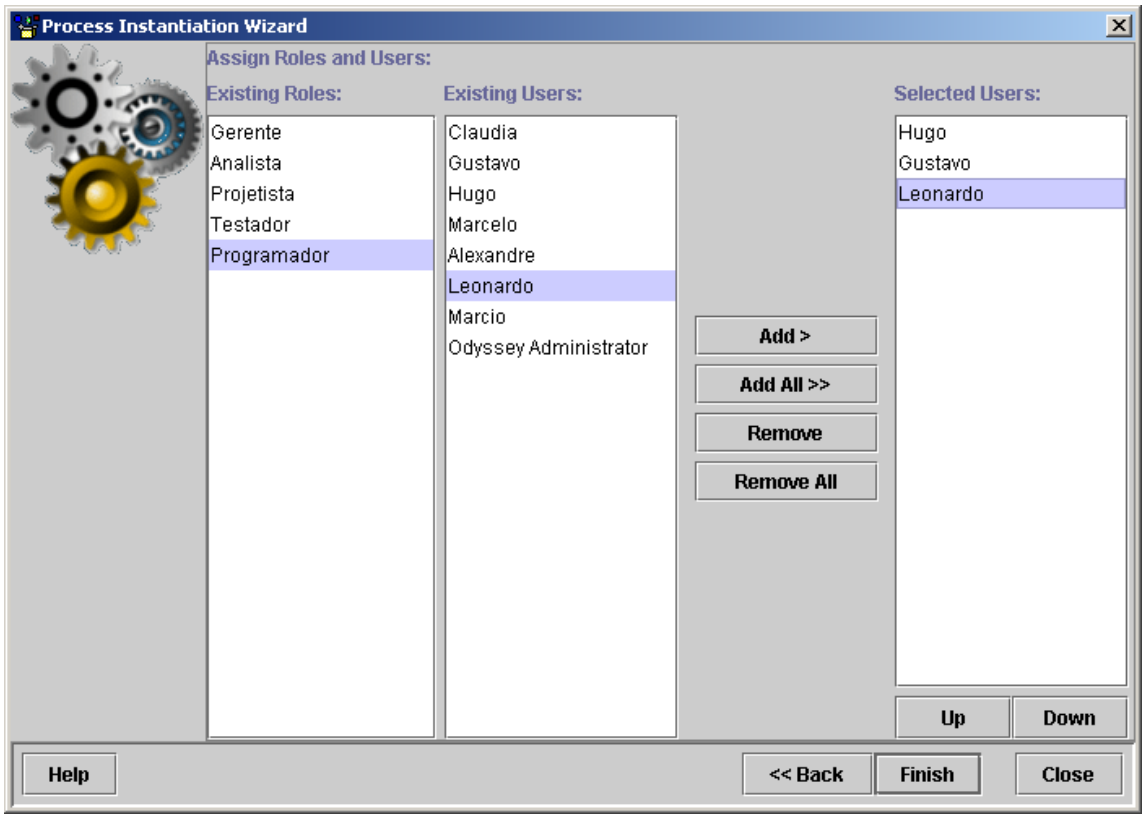

Figura 4.6: Associação entre papéis e usuários durante a instanciação do processo.

Vale ressaltar que cada papel tem que estar associado a pelo menos um desenvolvedor. Entretanto, um papel pode estar associado a vários desenvolvedores, que trabalharão em equipe para possibilitar o desenvolvimento do sistema. No exemplo, os desenvolvedores Hugo, Gustavo e Leonardo estão sendo associados com o papel programador.

Após a instanciação, o processo será simulado para verificar se existem erros de modelagem que inviabilizem a sua execução. Caso nenhum erro seja encontrado, o processo poderá entrar em execução.

4.4.3 – Simulação do processo

A simulação do processo ocorre em paralelo com a etapa de associação entre os papéis e os desenvolvedores, pois para a simulação não é importante saber quem vai executar cada tarefa. O importante é verificar se a tarefa pode entrar em execução sem que ocorram erros que impossibilitem o seu término.

Outra função da simulação é descobrir os tempos estimados de execução das atividades compostas. Esses tempos variam em função das dependências entre suas sub-

atividades e das probabilidades de seleção dos fluxos que saem de decisões, sendo calculados como a média de um conjunto de execuções simuladas.

Caso a simulação consiga ser executada com sucesso, não detectando erros nem repetições infinitas, como ocorreu com o CtrlPESC, a tela da Figura 4.7 será exibida, indicando que o desenvolvimento do sistema pode ser iniciado. Isso significa que o Agente de Execução colocou o processo raiz em execução e, recursivamente, todos os processos que são destino de fluxos que partam dos elementos de início do seu *workflow*. No caso do sistema CtrlPESC, esses processos são "Análise de Usuários e Professores", "Análise de Linhas e Disciplinas" e "Análise de Alunos e Inscrições".

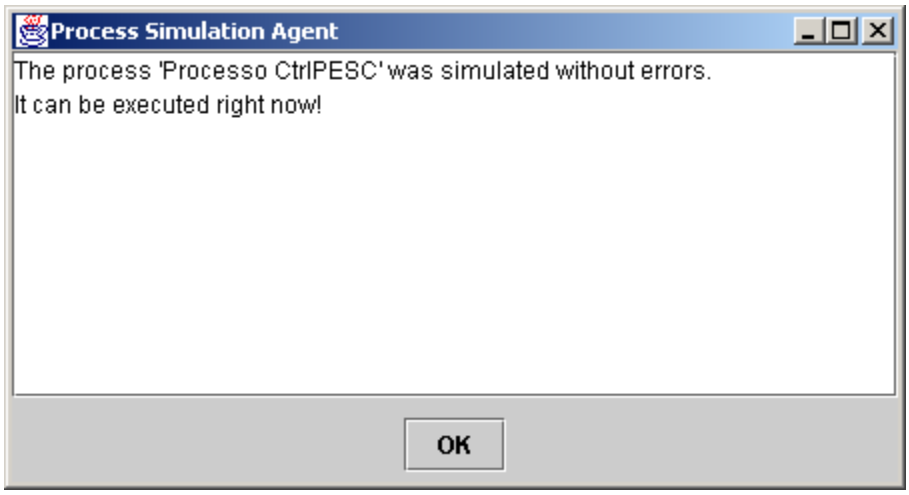

Figura 4.7: Notificação de simulação ocorrida com êxito.

Entretanto, caso a simulação ocorra por um tempo superior a cinco segundos, o Agente de Simulação notifica ao usuário, através da tela exibida na Figura 4.8, que pode estar ocorrendo uma repetição infinita devido a um fluxo de retorno ou a uma recursão sem condição de parada.

Nesse momento, o usuário pode optar por continuar a simulação por mais cinco segundos ou cancelar a simulação. Caso a simulação seja continuada por mais cinco segundos e não chegue ao seu fim, uma nova tela como essa será exibida, permitindo que alguma ação seja tomada. Caso a simulação seja cancelada, o usuário deverá reestruturar o processo, corrigindo o ponto onde não existia condição de parada, e reinstanciá-lo, como será descrito posteriormente. Só assim o processo entrará em execução.

Do momento em que a simulação foi terminada com sucesso em diante, sempre que algum desenvolvedor entrar no ambiente de desenvolvimento do CtrlPESC, será possível receber informações sobre o estado atual do processo.

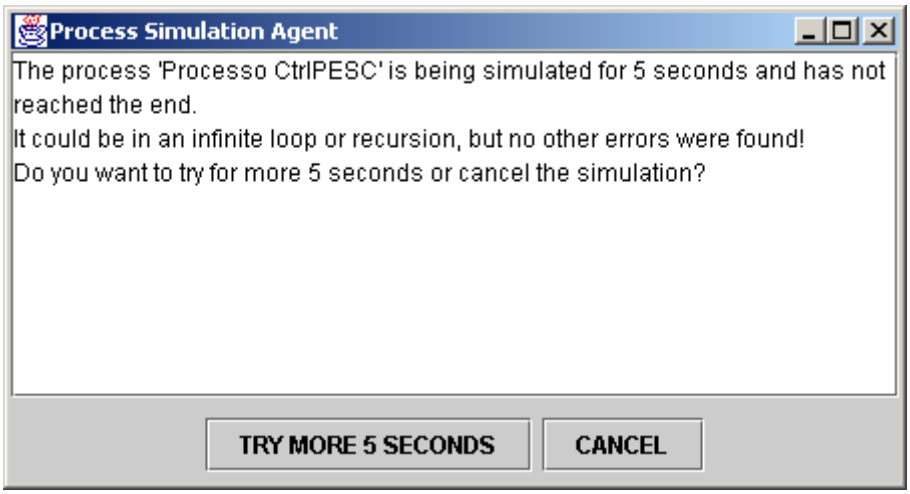

Figura 4.8: Notificação de possível repetição infinita durante a simulação.

4.4.4 – Acompanhamento do processo

O acompanhamento do processo funciona como um guia, que norteia o desenvolvedor, indicando as tarefas pendentes e decisões que devem ser tomadas.

Assim que algum desenvolvedor entra pela primeira vez no ambiente de desenvolvimento do CtrlPESC, serão informados quais atividades estão pendentes para ele, em função dos papéis associados. Essas atividades são a lista de trabalho do desenvolvedor e devem ser executadas para permitir que outras atividades entrem em execução, agendando trabalho para outros desenvolvedores. A Figura 4.9 exibe o Agente de Acompanhamento com a lista de atividades pendentes para um determinado desenvolvedor. A atividade "Projeto Arquitetural" está exibida na lista como pendente para o desenvolvedor atual. Os dados dessa atividade estão exibidos em cinza claro, pois seu modo atual é de somente visualização, não sendo possível a sua alteração.

Todos os processos dessa lista são processos primitivos, pois quando existe algum processo composto em execução, a sua execução se resume a execução dos seus processos primitivos.

Quando um processo é selecionado nessa lista, suas informações são exibidas no painel da direita. O desenvolvedor deve consultar essas informações para saber quais recursos devem ser utilizados no desenvolvimento do sistema segundo o processo. Além disso, quando o processo tiver sido cumprido, é possível notificar à máquina de processos que isso ocorreu, através de um botão "Finaliza" (*FINISH*) existente na própria janela.

No exemplo do CtrlPESC, o Agente de Acompanhamento indica que o desenvolvedor atual, que é um projetista, deve executar a atividade de "Projeto

Arquitetural". Essa informação é obtida através de inferências na base de conhecimento do processo.

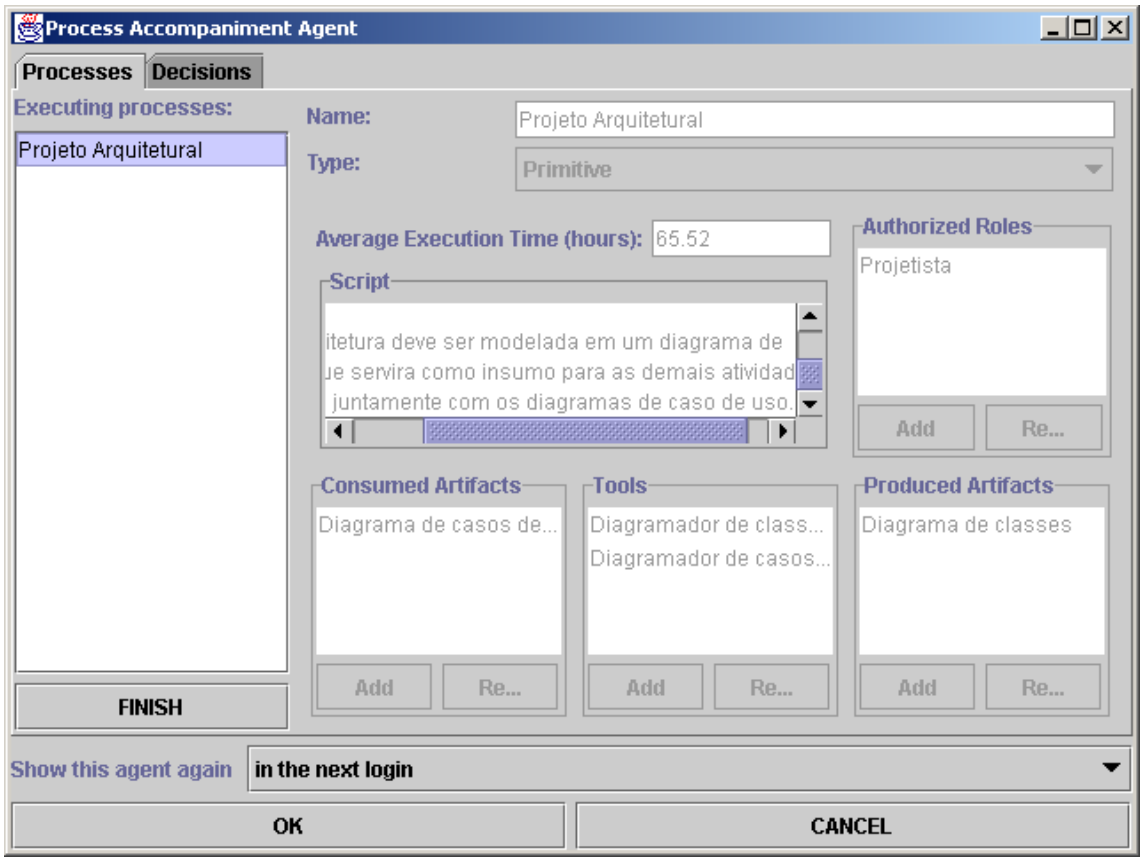

Figura 4.9: Finalização de processo em execução.

Similarmente a processos, o Agente de Acompanhamento fornece todas as decisões pendentes para o desenvolvedor corrente, como exibido na Figura 4.10. Ao selecionar uma determina decisão, são exibidas as respostas possíveis, e fornecida a opção de efetuar a escolha de uma das respostas.

No exemplo do CtrlPESC, o usuário atual, que é o gerente do projeto, deve informar se foram encontrados defeitos na atividade de "Testes" que inviabilizem a geração da versão de produção do CtrlPESC. A resposta selecionada foi "Sim", o que indica que o processo será novamente executado para que os erros possam ser corrigidos.

Esse Agente de Acompanhamento também fornece ao usuário a opção de selecionar qual a freqüência de seu aparecimento. A configuração padrão indica que o agente deve aparecer sempre que o desenvolvedor entrar no ambiente de desenvolvimento do CtrlPESC. Entretanto, o acesso ao Agente de Acompanhamento também está disponível via um menu do ambiente de desenvolvimento.

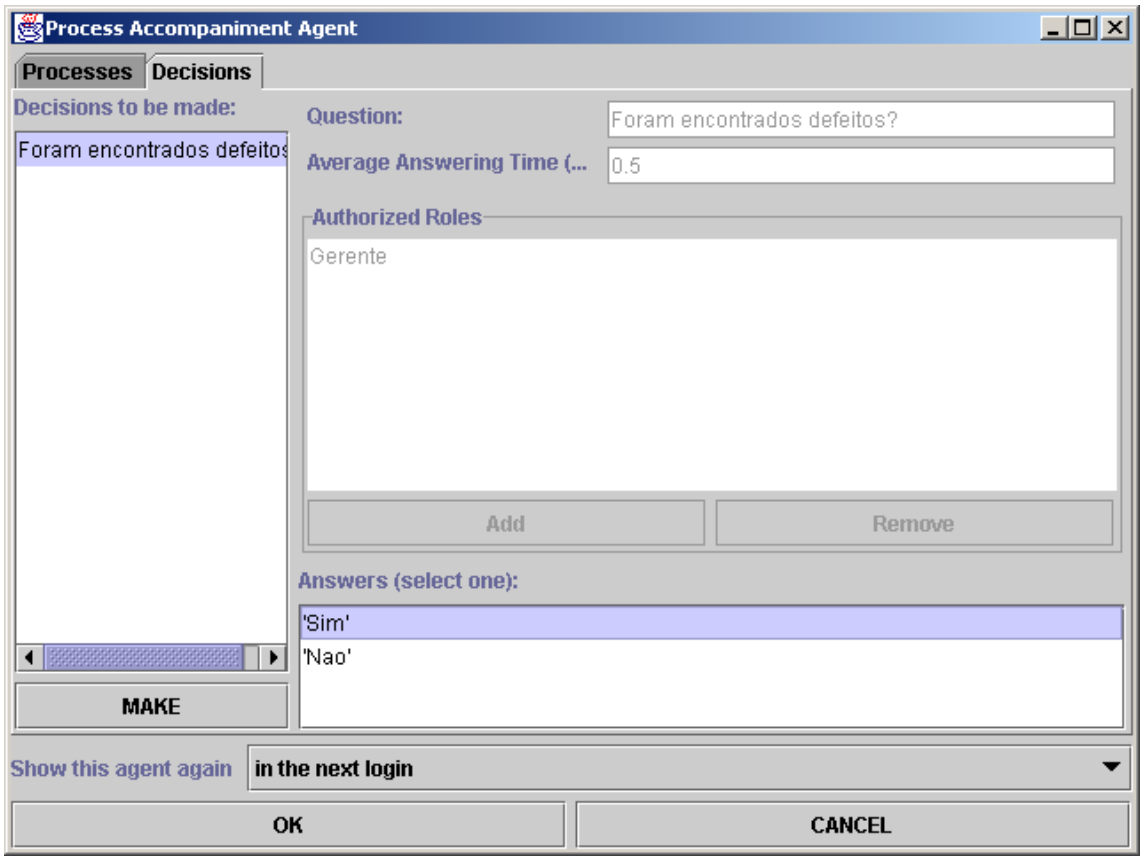

Figura 4.10: Finalização de decisão em execução.

4.4.5 – Monitoramento de processos

Além de permitir o acompanhamento do processo, que fornece uma visão individual das pendências de processos e decisões para cada desenvolvedor, é possível acessar o ambiente de monitoramento do processo em execução para obter uma visão global do processo. A Figura 4.11 exibe o ambiente de monitoramento com o processo de "Projeto Detalhado e Implementação" em execução.

Nesse processo, todos os seus sub-processos já foram executados duas vezes, menos o processo "Implementação de Linhas e Disciplinas" que está dependendo do "Projeto de Linhas e Disciplinas" que ainda está com a sua segunda execução em andamento, assim como o processo "Implementação de Usuários e Professores".

Apesar do processo "Implementação de Alunos e Inscrições" já ter terminado, o processo "Projeto Detalhado e Implementação" não pode ser finalizado, pois existe um sincronismo que força com que todos os sub-processos de implementação terminem para que o processo que os contém seja finalizado.

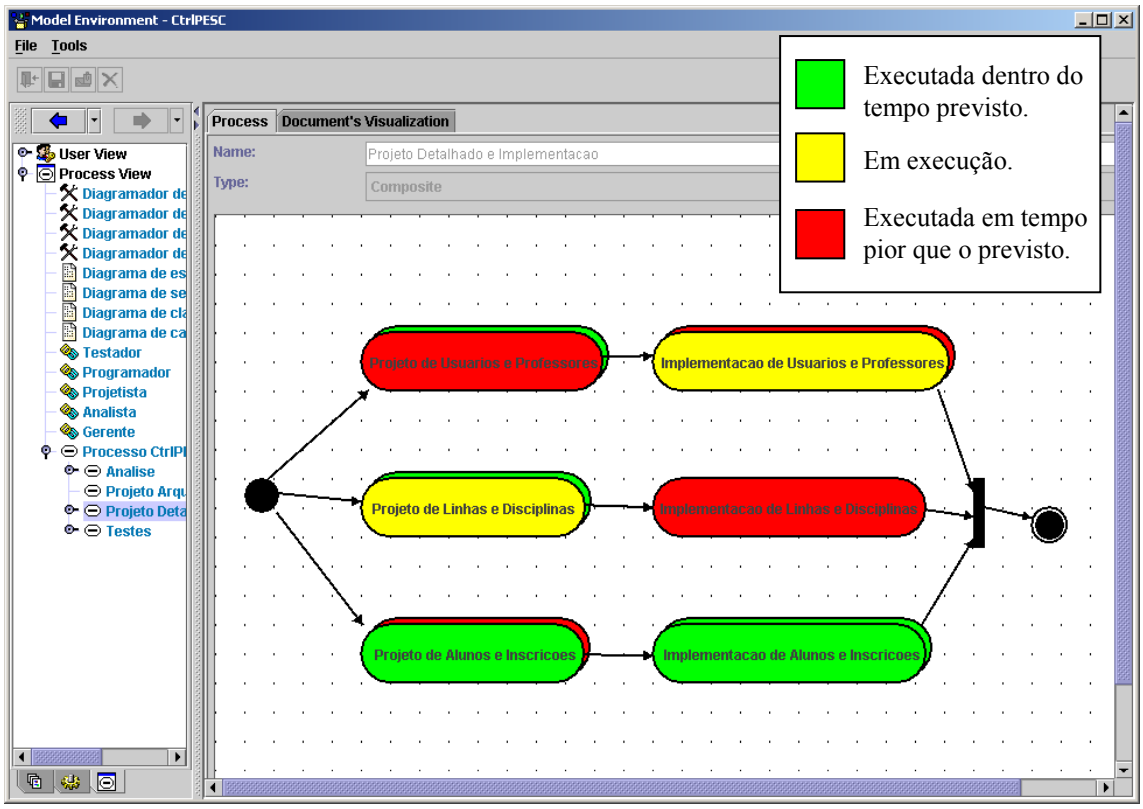

Figura 4.11: Monitoramento do processo de projeto detalhado e implementação.

Outro exemplo do ambiente de monitoramento se refere ao processo principal, como exibido na Figura 4.12. Este processo, que descreve o ciclo de vida cascata, já foi executado duas vezes completas e está na sua terceira execução. A atividade de "Análise" teve as suas duas primeiras execuções piores que o esperado, contudo a sua terceira execução ficou dentro do tempo estimado pela simulação. Atualmente, a execução do processo está na atividade de "Projeto Arquitetural", que teve as suas duas primeiras execuções dentro do esperado. As atividades de "Projeto Detalhado e Implementação" e "Testes" passaram por duas execuções, sendo que somente a última execução de "Testes" ocorreu dentro do tempo esperado.

Sempre que a atividade de "Testes" termina, o gerente é questionado sobre a existência de defeitos no sistema. Esse questionamento ocorreu duas vezes até então, e nas duas ocorrências o gerente optou por reiniciar o processo, possibilitando a remoção desses defeitos. Como o processo está na sua terceira execução, o questionamento será efetuado novamente, motivando ou não uma nova execução, assim que a atividade de "Testes" terminar. As duas tomadas de decisões foram efetuadas dentro do tempo previsto pela simulação.

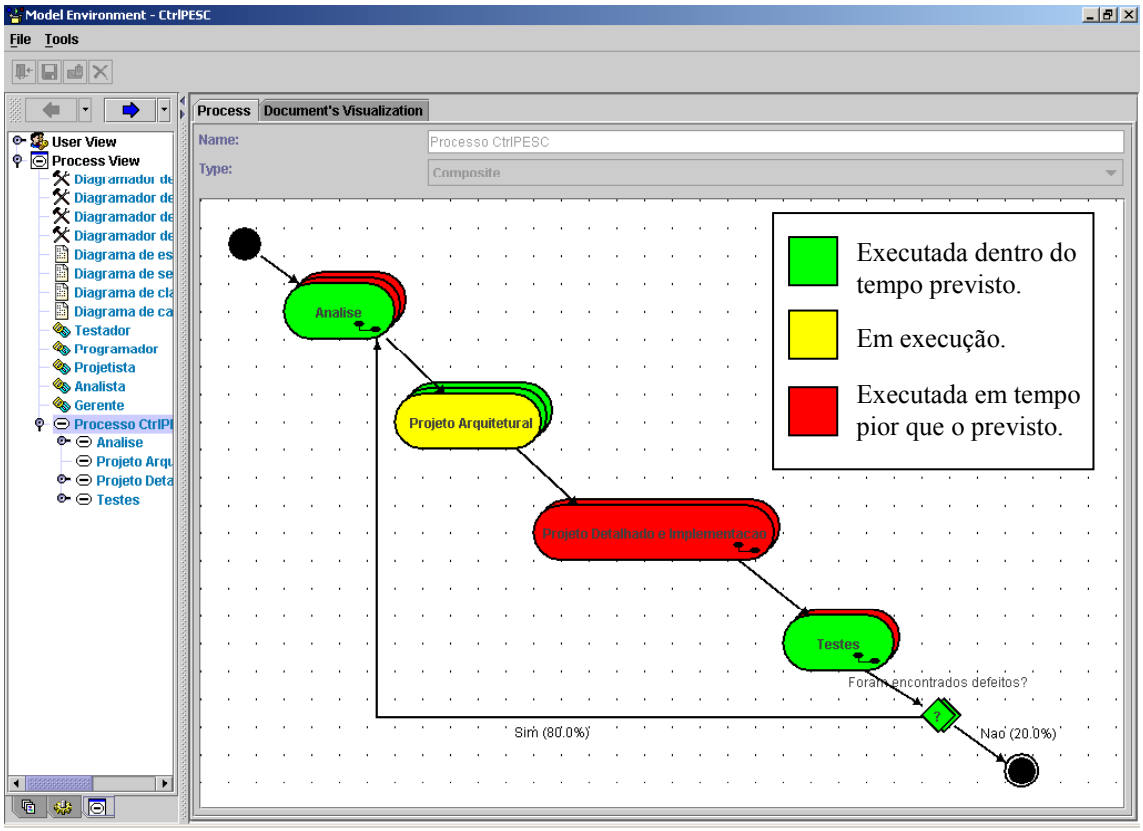

Figura 4.12: Monitoramento do processo principal.

4.4.6 – Retrocesso do processo

Quando um processo é finalizado ou uma decisão é tomada, outros desenvolvedores recebem novas tarefas nas suas listas de trabalho. Entretanto, pode ser detectado que o término de um processo ou uma tomada de decisão não ocorreu no momento oportuno, sendo necessário retroceder a execução do processo para que um novo fluxo seja escolhido a partir daquele momento.

Para atender a esse requisito, o Agente de Retrocesso fornece a tela exibida na Figura 4.13. Nessa tela, é informada a data atual e perguntado em quantas horas o processo deve ser retroagido. De posse dessa informação, o Agente de Retrocesso manipula a base de conhecimento do processo, fazendo com que a data atual menos o número de horas escolhido passe a ser a última data de interação entre qualquer usuário e a máquina de processos.

Como todas as informações sobre o processo e sobre sua execução estão nessa base de conhecimento, não é necessário comunicar a nenhum outro agente ou módulo do sistema para efetuar essa correção. Quando algum desenvolvedor acessar o Agente de Acompanhamento ou o ambiente de monitoramento do processo em execução, verá

que o processo não tem mais nenhuma informação a partir da data estabelecida, pois todos os agentes ou módulos do sistema constroem suas interfaces com os usuários de forma dinâmica, acessando a base de conhecimento.

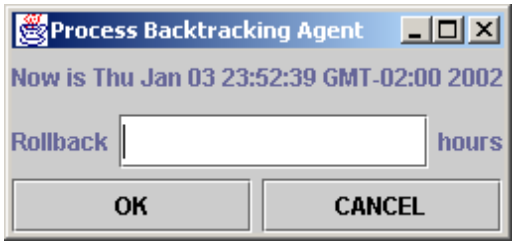

Figura 4.13: Retrocesso na execução do processo.

4.4.7 – Reinstanciação do processo

Outro evento possível de acontecer durante a execução do processo é a detecção de falhas semânticas na modelagem do processo. As falhas sintáticas são detectadas pelo Agente de Simulação. Entretanto, as falhas semânticas, que podem ser, por exemplo, a ordem invertida entre os testes individuais dos módulos e o teste global do sistema, não podem ser detectadas automaticamente, pois pertencem ao domínio de conhecimento do responsável pela definição do processo, e não ao domínio de conhecimento de automação de processos.

Quando, durante a execução do processo, uma falha desse tipo é encontrada, é necessário corrigí-la no ambiente de modelagem e reinstanciar o processo. Para que o processo possa ser reinstanciado, é necessário que todas as atividades que estão atualmente em execução sejam mantidas inalteradas no modelo. Com isso, pode ser preciso retroceder o processo antes de efetuar a sua reinstanciação. A Figura 4.14 exibe o menu com as opções de solicitação do Agente de Acompanhamento, reinstanciação do processo e solicitação do Agente de Retrocesso.

A partir da solicitação de reinstanciação do processo, a tela exibida na Figura 4.6 será reexibida, permitindo que uma nova associação entre os papéis e os desenvolvedores seja feita. O processo raiz não pode ser trocado, pois sem ele não seria possível manter referências dos contextos onde os sub-processos foram executados. A necessidade de uma nova associação entre papéis e desenvolvedores vem da possibilidade da troca dos papéis durante a manutenção do processo, onde novos papéis poderiam ser inseridos sem que houvesse desenvolvedor anteriormente associado.

No caso do CtrlPESC, quando a última atividade do processo principal, que representa a resposta “Não” à pergunta “foram encontrados defeitos?”, é executada, o

processo é dado como terminado e só voltará à execução caso haja alguma intervenção no Agente de Retrocesso, ou caso seja reinstanciado, o que motivará o reinício da atividade de "Análise". Quando a reinstanciação ocorre com o processo em andamento, sua execução continua no ponto em que estava.

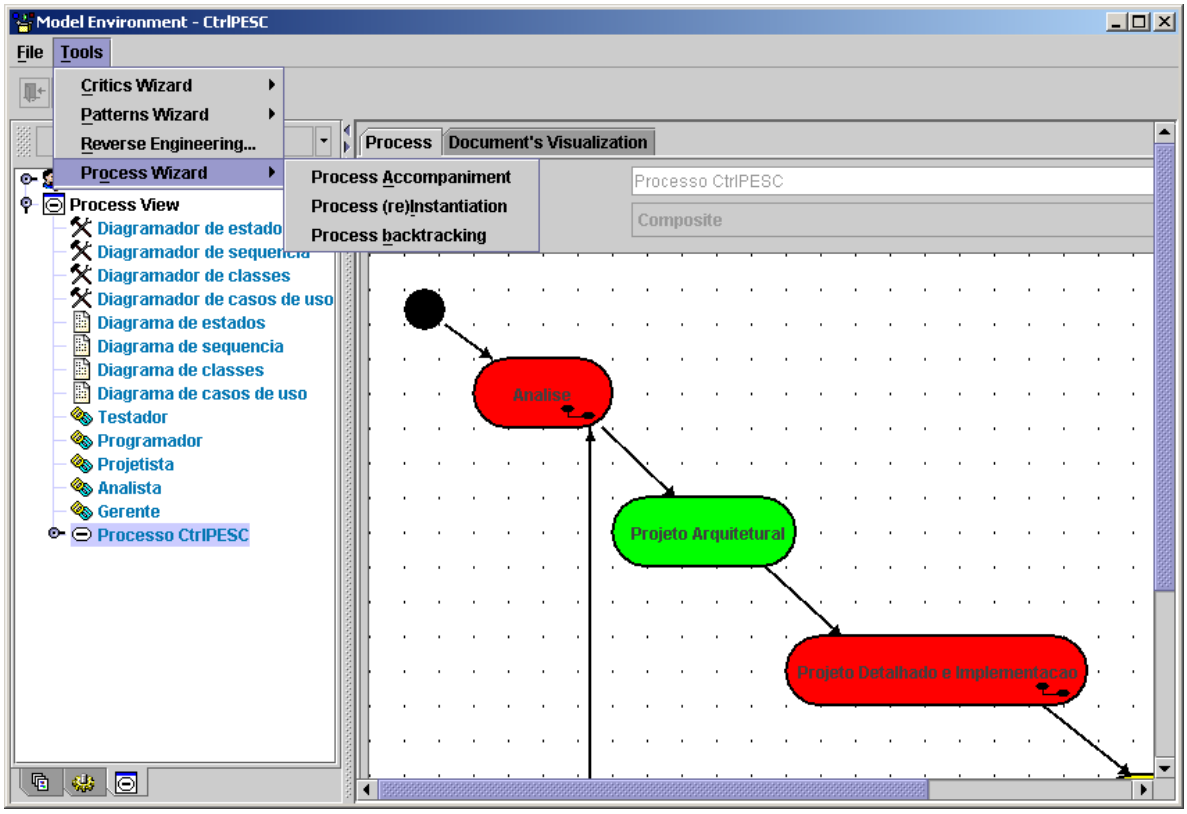

Figura 4.14: Menu de reinstanciação de processos.

4.5 – Conclusões

Neste capítulo, descrevemos o contexto em que o protótipo foi construído, o Ambiente Odyssey, que visa suportar o desenvolvimento de software baseado em reutilização. Também descrevemos a sua utilização através do desenvolvimento de um sistema exemplo, o CtrlPESC, sistema para auxílio no controle acadêmico utilizado pelo Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ.

O protótipo foi implementado utilizando a linguagem Java (SUN MICROSYSTEMS, 2001), que também é a linguagem utilizada no Ambiente Odyssey. Entretanto, os agentes inteligentes foram implementadas utilizando, além de Java, o JIP Prolog (CHIRICO, 2001), que é uma implementação Prolog em Java que permite a comunicação bidirecional entre Java e Prolog. A Tabela 4.2 exibe o tamanho de cada módulo da máquina de processos e o tamanho do Ambiente Odyssey, antes e após a

inclusão da máquina de processos. Esses tamanhos são medidos em número de classes e em linhas de código (LOCs).

Tabela 4.2: Métricas referentes à implementação do protótipo.

Elemento	Classes	LOCS
Núcleo da máquina de processos	10	1870
Agentes Inteligentes	13	2086
Elementos para a modelagem de processos	22	1973
Protótipo Completo	45	5929
Ambiente Odyssey antes da inclusão da máquina de processos	446	95185
Ambiente Odyssey após a inclusão da máquina de processo	491	101114

O número de linhas de códigos exibido inclui também as linhas despendidas na documentação do código.

Algumas das alterações necessárias no Ambiente Odyssey, tais como inclusão de menus, criação de componentes para a navegação nos diagramas, etc. não foram contabilizadas como parte do protótipo. Também não foi contabilizada a biblioteca JIP Prolog, que teve que ser inserida no Ambiente Odyssey para permitir a construção dos agentes inteligentes da máquina de processos.

O Ambiente Odyssey com a máquina de processos está disponível em http://www.cos.ufrj.br/~odyssey para execução através do Java WebStart (SUN MICROSYSTEMS, 2002), que é um mecanismo que possibilita a execução de aplicações Java através da Internet.

Capítulo 5 – Conclusões

5.1 – Contribuições

Conforme discutido nesse trabalho, grande parte das abordagens tradicionais para a automação de processos de software agem de forma reativa, respondendo a eventos gerados pelos desenvolvedores. Essas abordagens geralmente não têm preocupações referentes à reutilização de processos e à extensão da própria máquina de processos, permitindo que novos requisitos sejam atendidos com facilidade.

Para suprir essas necessidades, especificamos e construímos uma máquina de processos, que fornece as seguintes características:

- Modelagem gráfica de processos, que permite a utilização de ciclos e recursões, segundo uma notação estendida do diagrama de atividades da UML. A modelagem permite a separação de processos de software em primitivos e compostos, onde os processos primitivos são descritos através de roteiros, papéis, ferramentas e artefatos consumidos e produzidos, e os processos compostos são descritos através de um *workflow* de sub-processos, segundo a mesma notação de modelagem;
- Possibilidade de reutilização de um mesmo processo em diferentes contextos e níveis de abstração. O suporte à reutilização da abordagem proposta ocorre de forma simplificada, não levando em conta todas as questões necessárias para a aplicação sistemática e abrangente da reutilização;
- Mapeamento automático do processo da notação gráfica para uma base de conhecimento Prolog, segundo uma ontologia definida;
- Simulação do processo modelado antes do início da execução, permitindo a detecção de erros sintáticos de modelagem;
- Mecanismo para a execução do processo, descrito em regras de inferência Prolog;
- Acompanhamento do processo através de *worklists* que descrevem quais atividades ou decisões estão pendentes para um determinado desenvolvedor;

- Monitoramento da execução do processo através de uma convenção de cores, para denotar o estado da execução, e sobreposição, para denotar o número de execuções anteriores dos processos;
- Evolução de processos via um mecanismo de reinstanciação que permite que modificações em processos que estejam em execução sejam refletidas em suas versões instanciadas sem reiniciar-los;
- Suporte a pró-atividade e extensão da máquina de processos, através da implementação de uma infra-estrutura para a construção de agentes inteligentes;
- Implementação da máquina de processos dentro do Ambiente Odyssey, utilizando a linguagem Java combinada com a máquina de inferência Prolog JIP, permitindo a instanciação de processos para a engenharia de domínio e engenharia de aplicação através da seleção do processo raiz e da associação entre os papéis e os desenvolvedores.

Estas características permitem que a abordagem Charon atenda aos critérios definidos na Seção 2.5 do Capítulo 2, como exibido na Tabela 5.1.

Tabela 5.1: Quadro comparativo entre as abordagens

Critérios	Máquinas de estado ou redes de Petri			Agentes, regras ou scripts			Híbridas (Gráfica + Regras)						
	SPADE	Memphis	ProSoft	HyperCode	A.I. P.M. Fusion	CAGIS	EPOS	Merlin	ProNet / ProSim	Dynamite	APEL	Mokassin	**Charon**
Modelagem do processo	✓	✓	✓	✗	✓	?	✓	✓	✓	✓	✓	✓	✓
***Workflow* com ciclos e recursão**	✓	✗	?	✓	?	✗	?	?	✓	✓	✓	✓	✓
Mapeamento automático	✗	✓	✓	✓	✗	?	✓	✓	✗	✗	✓	✓	✓
Interação pró-ativa	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓
Processos reutilizáveis	✗	✓	?	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓
Extensão flexível	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓	?	✓
Foco em processos de software	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓
Protótipo implementado	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓

A Seção 3.11 do Capítulo 3 descreve detalhadamente como cada um dos critérios é atendido pela abordagem Charon.

Podemos dizer que o principal objetivo desse trabalho, que consistia em construir uma máquina de processos pró-ativa e extensível, foi atingido através do uso de agentes inteligentes, que executam autonomamente, podendo entrar em execução de forma completamente independente dos eventos de usuário. Esses agentes representam os requisitos da máquina de processos, e contém planos de execução que permitem que esses requisitos sejam atendidos. Desta forma, a extensão da máquina de processos consiste na criação de novos agentes, com novos planos para manipular a base de conhecimento de processos, visando atender a determinados requisitos.

5.2 – Limitações e trabalhos futuros

Algumas limitações e trabalhos futuros puderam ser detectados durante esse trabalho, dentre eles: melhorias no suporte a reutilização de processos, melhorias no mecanismo de instanciação de processos, controle e monitoramento de ferramentas, ambiente para a geração de relatórios gerenciais, verificação de consistência das bases de conhecimento e verificação empírica das conclusões obtidas, conforme descrito a seguir.

5.2.1 – Melhorias no suporte a reutilização de processos

O suporte a reutilização de processo poderia ser melhorado segundo três perspectivas:

- Definição de um novo tipo de processo: o processo abstrato;
- Criação de processos dependentes de domínio;
- Utilização de padrões para a documentação de componentes de processo;

A criação do tipo de **processo abstrato** consiste em permitir que a decisão de qual processo utilizar em um determinado contexto possa ser postergada para o momento da instanciação do processo raiz. Um processo abstrato é um elemento que não pode entrar em execução, mas pode ser substituído por um processo primitivo ou composto no momento da instanciação. Assim, seria possível descrever em um determinado *workflow* a necessidade de um tipo de atividade, sem definir a priori qual a tecnologia a ser utilizada nesta atividade.

Por exemplo, seria possível construir um processo que representasse o ciclo de vida cascata, contendo, entre outras atividades, a atividade "Análise". Contudo, caso a atividade "Análise" fosse representada por um processo abstrato, seria possível indicar quais processos, simples ou compostos, poderiam ser selecionados para cumprir essa atividade, como "Análise Estruturada", "Análise Essencial" ou "Análise Orientada a Objetos".

A idéia de um tipo de processo abstrato é semelhante à definição de classes abstratas na orientação a objetos, onde uma classe abstrata nunca pode ser instanciada, necessitando a sua herança por outra classe para que seja possível a instanciação de um objeto daquele tipo. Desta forma, podemos falar, por exemplo, que "Análise Orientada a Objetos" é um tipo de análise "Análise".

A criação de **processos dependentes de domínio** consiste em fornecer um ambiente para a modelagem de processos dentro de um domínio de aplicação, onde processos serão gerados levando em conta as características do domínio em questão. No momento da criação de uma aplicação dentro desse domínio, ocorreria a junção entre os processos genéricos e os processos dependentes de domínio, permitindo a construção do processo raiz da aplicação. Atualmente, é possível criar processos dependentes de domínio no mesmo local onde são criados os processos genéricos, o que quebra o encapsulamento do domínio em questão, pois artefatos (processos) referentes ao domínio estarão situados em uma área externa ao domínio no ambiente.

Assim, um processo dependente de domínio só poderia ser reutilizado dentro do domínio em que ocorreu a sua definição, permitindo que processos genéricos fossem mais facilmente refinados para refletir características próprias do domínio em questão.

A **documentação de componentes de processo** pode ser apoiada pelo uso de padrões, como definido no Ambiente Memphis (WERNER et al., 1996). Para viabilizar essa documentação dentro do Ambiente Odyssey, a ferramenta FrameDoc (MURTA, 1999; MURTA et al., 2001) poderia ser utilizada. A FrameDoc possibilita a criação de padrões de documentação para determinados tipos de componentes existentes no Ambiente Odyssey. Cada padrão tem um conjunto de campos que devem ser preenchidos no momento da criação do componente, e servem para apoiar a seleção desse componente para reutilização em diferentes contextos.

A ferramenta Charon está totalmente integrada com a ferramenta FrameDoc, tornando necessária, somente, a definição dos padrões de documentação dentro do

ambiente de configuração do FrameDoc, para que os componentes de processos possam fazer uso desse recurso.

5.2.2 – Melhorias no mecanismo de instanciação de processos

Atualmente, o mecanismo de instanciação de processo interage com o desenvolvedor para obter a informação de quais desenvolvedores cumprirão quais papéis. Entretanto, a tarefa de definir uma equipe de desenvolvimento é complexa e seria interessante um suporte automatizado para facilitar a sua execução.

Uma possível solução para essa automatização seria fornecer um ambiente para o cadastro dos desenvolvedores que levasse em conta as suas habilidades, como, por exemplo, a de lidar com pessoas ou com raciocínio lógico. Juntamente com esse ambiente, permitir a definição dos requisitos de habilidades para as atividades e construir um sistema de programação linear que levasse em conta a disponibilidade de cada desenvolvedor e fizesse uma sugestão de quais desenvolvedores deveriam ser alocados em quais papéis.

5.2.3 – Controle e monitoramento do uso das ferramentas

A versão atual do protótipo que implementa a abordagem não interfere na execução das ferramentas, pois atua somente como um guia que sugere qual ferramenta deve ser executada em um determinado momento. Entretanto, seria interessante controlar o estado do ambiente através da informação de quais ferramentas estão sendo executadas por quais desenvolvedores.

Esse controle seria útil para permitir o monitoramento do que cada desenvolvedor fez em cada ferramenta e restringir, caso desejado, o uso de uma ferramenta por um determinado período de tempo. Além disso, o controle de ferramentas poderia permitir que uma ferramenta qualquer pudesse ser iniciada sem a necessidade de desenvolvedores presentes no ambiente, através de um servidor que interpretasse scripts com o conjunto de ações que devem ser efetuadas na ferramenta. Até mesmo no caso de ferramentas interativas, seria possível definir um roteiro de ações que deveriam ser executadas na ferramenta para que a sua iniciação fosse possível. As ações das ferramentas interativas também seriam úteis para permitir o monitoramento de exatamente o que cada desenvolvedor fez no ambiente e possibilitar que determinadas ações sejam desfeitas.

5.2.4 – Ambiente para geração de relatórios gerenciais

Todas as informações referentes à execução do processo são armazenadas na base de conhecimento do processo. Entretanto, a única externalização dessas informações ocorre através do diagrama que permite o monitoramento da execução do processo.

Seria interessante a construção de um agente para a especificação de relatórios gerenciais, onde o próprio gerente utilizasse uma linguagem declarativa para descrever que tipo de informações deseja em cada relatório. Caso essa linguagem declarativa fosse o próprio Prolog, a consulta seria processada pela máquina de inferência do ambiente e enviada a um *parser*, que seria responsável pela formatação final do relatório.

Desta forma, seria possível obter uma visão micro da execução do processo, em contrapartida à visão macro existente através do diagrama de monitoramento da execução do processo.

5.2.5 – Verificação de consistência das bases de conhecimento

O *framework* proposto permite que outros agentes possam ser construídos para cumprir determinadas tarefas. Com isso, a população de agentes pode crescer, tornando complexo o controle de corretude das suas atividades.

As bases de conhecimento dos processos são alteradas com o passar do tempo por esses diversos agentes. Entretanto, não existe nenhuma garantia que o seu estado está consistente, pois um determinado agente pode corromper a base incluindo informações que contradizem outras informações já existentes.

Seria desejável a inclusão de um mecanismo de verificação do estado da base. Esse mecanismo, implementado através de um novo agente, poderia verificar uma base e assegurar o seu estado atual como consistente ou, ao detectar uma inconsistência, retroceder a base para o último estado consistente. Uma outra abordagem seria a verificação da base antes da inclusão das novas cláusulas, assegurando que a mesma nunca entre em um estado inconsistente.

5.2.6 – Verificação empírica das conclusões obtidas

Ao final desse trabalho, concluímos que o uso da arquitetura baseada em agentes pode facilitar a manutenção evolutiva da máquina de processo através da criação de um novo agente para cada novo requisito. Contudo, essa conclusão poderia ser vista como

uma premissa, que deve ser verificada através de estudos empíricos (WOHLIN et al., 2000).

A verificação poderia ocorrer através do uso da abordagem proposta em ambiente real. Esse uso implicaria no surgimento de novos requisitos. Os requisitos deveriam ser implantados tanto na abordagem proposta quanto em abordagens convencionais e, através da medição dos custos e do tempo de implantação, poderiam surgir indícios que sustentassem essa premissa.

Referências Bibliográficas

ARAÚJO, M. A. P., 1998, *Automatização do Processo de Desenvolvimento de Software nos Ambientes Instanciados pela Estação TABA*, Dissertação de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

ARMENISE, P., BANDINELLI, S., GHEZZI, C., et al., 1993, "A Survey and Assessment of Software Process Representation Formalisms", *International Journal of Software Engineering and Knowledge Engineering*, v. 3, n. 3, pp. 401-426.

AUDE, J. S., 1996, "Notas de aula de Circuitos Lógicos". *DCC, IM, UFRJ*, Rio de Janeiro, Brasil.

BANDINELLI, S., FUGGETTA, A., 1993, "Computational Reflection in Software Process Modeling: the SLANG Approach". In: *15th Int. Conf. on Software Engineering*, pp. 144-154, Maryland, USA.

BANDINELLI, S., FUGGETTA, A., GHEZZI, C., et al., 1994, *SPADE: An Environment for Software Process Analysis, Design and Enactment*. Research Studies Press.

BARROS, M., 2001, *Gerenciamento de Projetos Baseado em Cenários: Uma Abordagem de Modelagem Dinâmica e Simulação* , Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

BASILI, V. R., CALDIERA, G., ROMBACH, H. D., 1994, "Goal-Question-Metric Paradigm", *Encyclopedia of Software Engineering*, v. 1, pp. 469-476.

BELKHATIR, N., ESTUBLIER, J., MELO, W. L., 1993, "Software process model and work space control in the Adele system". In: *2nd Int'l Conf. on the Software Process*, Berlin, Germany.

CATTANEO, F., CODA, F., 2002, "SPADE-1: Software Process Analysis, Design and Enactment". In: http://roma.dbis.informatik.uni-frankfurt.de/~door/docs/spadecourse/index.htm, Accessed in 23/01/2002.

CHIRICO, U., 2001, "JIP: Java Internet Prolog". In: http://www.geocities.com/jiprolog/, Accessed in 08/11/2001.

CHRISTIE, A., 1995, *Software Process Automation: The Technology and Its Adoption*, Berlin, Springer-Verlag Publishing.

COLEMAN, D., ARNOLD, P., BODOFF, S., et al., 1994, *Object Oriented Development: The Fusion Method*, Prentice-Hall.

COMPAQ, 2001, "DEC FUSE: Software Product Description". In: http://www.tru64unix.compaq.com/fuse, Accessed in 03/12/2001.

DAMI, S., ESTUBLIER, J., AMIOUR, M., 1998, "APEL: A graphical yet executable formalism for process modelling", *Automated Software Engineering: An International Journal*, v. 5, n. 1, pp. 61-96.

DEUX, O., 1991, "The O2 System", *Comm.of the ACM*, v. 34, n. 10.

DUTRA, I. C., 2001, "Lógica em Programação". In: http://www.cos.ufrj.br/~ines/courses/LP.html, Accessed in 23/10/2001.

ELMASRI, R., NAVATHE, S. B., 1994, *Fundamentals of Database Systems*. second ed., Redwood, CA, Addison Wesley.

FRANKLIN, S., GRAESSER, A., 1996, "Is it an Agent, or Just a Program? - A Taxonomy for Autonomous Agents". In: *Third International Workshop on Agent Theories, Architectures and Languages*, pp. 21-35, Budapest, Hungary.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Massachusetts, Addison Wesley.

GARCIA, A. F., SILVA, V. T., LUCENA, C. J. P., et al., 2001, "An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems". In: *XV Simpósio Brasileiro de Engenharia de Software*, pp. 177-192, Rio de Janeiro, Brasil, October.

GOMES, A., MAFRA, S., OLIVEIRA, K., et al., 2001a, "Avaliação de Processos de Software na Estação Taba". In: *XV Simpósio Brasileiro de Engenharia de Software - Seção de Ferramentas*, pp. 344-349, Rio de Janeiro, Brasil.

GOMES, A., OLIVEIRA, K., ROCHA, A. R., 2001b, "Avaliação de Processos de Software Baseada em Medições". In: *XV Simpósio Brasileiro de Engenharia de Software*, pp. 84-99, Rio de Janeiro, Brasil.

GRUBER, T. R., 2002, "What is an Ontology?". In: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html, Accessed in 07/01/2002.

HAYES-ROTH, B., 1995, "An architecture for adaptive intelligent systems", *Artificial Intelligence: Special Issue on Agents and Interactivity*, v. 72, pp. 329-365.

HEIMANN, P., JOERIS, G., KRAPP, C., et al., 1996, "DYNAMITE: Dynamic Task Nets for Software Process Management". In: *18th International Conference on Software Engineering*, pp. 331-341, Berlin, Germany.

HEWITT, C., 1977, "Viewing control structures as patterns of passing messages", *Artificial Intelligence*, v. 8, n. 3, pp. 323-364.

IBM, 2001, "Aglets". In: http://www.trl.ibm.co.jp/aglets, Accessed in 20/11/2001.

IFPUG, 1999, *Function Point Counting Practices Manual, Release 4.1*, Westerville, OH, International Function Point Users Group.

INGHAM, J., 1997, *What is an Agent?* Centre for Software Maintenance, University of Durham.

JACCHERI, L., LARSEN, J., CONRADI, R., 1992, "Software Process modeling and Evolution in EPOS". In: *4th International Conference on Software Engineering and Knowledge Engineering (SEKE'92)*, pp. 574-581, Capri, Italy.

JENNINGS, N. R., 2000, "On agent-based software engineering", *Artificial Intelligence*, v. 177, n. 2, pp. 277-296.

JENNINGS, N. R., SYCARA, K., WOOLDRIDGE, M. J., 1998, "A Roadmap of Agent Research and Development", *Journal of Autonomous Agents and Multi-Agent Systems*, v. 1, n. 1, pp. 7-38.

JOERIS, G., 2001, "Mokassin". In: http://www.informatik.uni-bremen.de/grp/mokassin, Accessed in 18/11/2001.

JOERIS, G., HERZOG, O., 1998, *Towards Object-Oriented Modeling and Enacting of Processes*. Center for Computer Technologies, University of Bremen.

JOERIS, G., KLAUCK, C., HERZOG, O., 1997a, "Dynamical and Distributed Process Management based on Agent Technology". In: *6th Scandinavian Conference on Artificial Intelligence (SCAI'97)*, pp. 187-198, Helsinki, Finland.

JOERIS, G., KLAUCK, C., WACHE, H., 1997b, "A Flexible Agent-Based Framework for Process Management". In: *3rd Knowledge Engineering Forum*, Kaiserslautern, Germany.

JONES, C., 2000, *Software Assessments, Benchmarks, and Best Practices*, Reading, MA, Addison-Wesley Publishing Co.

JUNKERMANN, G., PEUSCHEL, B., SCHÄFER, W., et al., 1994, "Merlin: Supporting Cooperation in Software Development through a Knowledge-based Environment ". In Nuseibeh, B., Finkelstein, A., and Kramer, J., Taunton, Inglaterra, John Wiley and Sons.

KOWALSKI, R. A., 1974, *Logic for Problem Solving*. Dept of Articial Intelligence, University of Edinburgh.

LUCENT TECHNOLOGIES, 2002, "Lucent Technologies". In: http://www.lucent.com, Accessed in 04/01/2002.

MAES, P., 1995, "Artificial Life Meets Entertainment: Interacting with Lifelike Agents", *Communications of the ACM, Special Issue on New Horizons of Commercial and Industrial AI*, v. 38, n. 11, pp. 108-114.

MAYER, R. J., MENZEL, C. P., PAINTER, M. K., et al., 1995, *Information Integration for Concurrent Engineering (IICE) IDEF3 Process Description Capture Method Report*. Knowledge Based Systems, Inc.

MURTA, L. G. P., 1999, *FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis*, Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, Brasil.

MURTA, L. G. P., BARROS, M. O., WERNER, C. M. L., 2001, "FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis". In: *IV International Symposium on Knowledge Management/Document Management (ISKM/DM'2001)*, Curitiba, Brasil.

NWANA, H. S., 1995, "Software Agents: An Overview", *Knowledge Engineering Review*, v. 11, n. 2, pp. 205-244.

OKTABA, H., ESQUIVEL, C., 2001, "Process Diagrams: An Extension of UML Activity Diagrams for the Modeling of Software Processes". In: *IV Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software (IDEAS 2001)*, pp. 31-40, Heredia, Costa Rica.

OMG, 2001, "OMG Unified Modeling Language Specification version 1.3". In: http://www.omg.org/cgi-bin/doc?formal/01-03-01.pdf, Accessed in 06/12/2001.

PERRY, D. E., PORTER, A., VOTTA, L. G., et al. , 1996, "Evaluating workflow and process automation in wide-area software development". In: *European Workshop on Software Process Technology*, pp. 188-193, Berlin, Germany.

PEUSCHEL, B., SCHÄFER, W., 1992, "Concepts and Implementation of a Rule-based Process Engine". In: *14 th International Conference on Software Engineering*, pp. 262-279, Melbourne, Australia.

PRESSMAN, R. S., 1997, *Software Engineering: A Practitioner's Approach*. 4 ed., McGraw-Hill.

REIS, C. A., REIS, R. Q., SCHLEBBE, H., et al. , 2001a, "A Abordagem APSEE para Modelagem e Gerência de Recursos em Ambientes de Processos de Software". In: *XV Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, Brasil.

REIS, R. Q., REIS, C. A., NUNES, D. J., 2001b, "APSEE-StaticPolicy: verificação de políticas estáticas em modelos de processos de software". In: *XV Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, Brasil.

REIS, R. Q., REIS, C. A. N. D. J., 1999, "Ambiente de Desenvolvimento de Software PROSOFT: Evolução e Estágio Atual". In: *Semana de Informática (SEMINF'99)*, Universidade Federal do Pará.

ROCHA, A. R., 1997, "Notas de Aula de Engenharia de Software". *IM/DCC, UFRJ*, Rio de Janeiro, Brasil.

ROCHA, A. R., WERNER, C. M. L., TRAVASSOS, G. H., et al., 1996, "Memphis - Um Ambiente de Desenvolvimento de Software baseado em Reutilização". In: *X Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas*, São Carlos, Brasil.

ROCHA, H., 2002, "A Divina Comédia". In: http://www.ibpinetsp.com.br/helder/dante, Accessed in 03/04/2002.

ROSS, S. M., 1990, *A Course in Simulation*, Nova York, Macmillan Publishing Company.

ROUSSEL, P., 1975, *Prolog: Manuel de Reference et d'Utilisation*. Groupe d'Intelligence Articielle, Marseille-Luminy.

RUSSELL, S., NORVIG, P., 1995, *Artificial Intelligence: A Modern Approach*. 1 ed., Englewood Cliffs, NJ, Prentice Hall.

SANTANDER, V. F. A., GIMENES, I. M. S., MASIERO, P. C., 1997, "Assistência Inteligente ao Processo de Engenharia de Software". In: *XI Simpósio Brasileiro de Engenharia de Software*, pp. 147-161, Fortaleza, Brasil.

SCHÜRR, A., 1989, "Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language". In: *15th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 151-165, Castle Rolduc, Netherlands.

SILVA, J. C. P., 1999, "Notas de Aula de Logica em Programação". *DCC/IM, UFRJ*, Rio de Janeiro, Brasil.

SILVA, S. A. D., NUNES, D. J., 1999, *Proposta de uma Ferramenta de Simulação de Processos de Software Baseada em Conhecimento para o Ambiente PROSOFT*. IV Semana Acadêmica do PPGC - UFRGS.

SUN MICROSYSTEMS, 2001, "Java". In: http://www.java.sun.com, Accessed in 11/05/2001.

SUN MICROSYSTEMS, 2002, "Java WebStart". In: http://java.sun.com/products/javawebstart, Accessed in 04/01/2002.

VASCONCELOS, F. M., WERNER, C. M. L., 1998, "Organizing the Software Development Process Knowledge: An Approach Based on Patterns", *International Journal of Software Engineering and Knowledge Engineering*, v. 8, n. 4, pp. 461-482.

WANG, A. I., 2000, "Using Software Agents to Support Evolution of Distributed Workflow Models". In: *International ICSC Symposium on Interactive and Collaborative Computing (ICC'2000)*, Wollongong, Australia.

WANG, A. I., CONRADI, R., LIU, C., 1999, "A Multi-Agent Architecture for Cooperative Software Engineering". In: *The Eleventh International Conference on Software Engineering and Knowledge Engineering (SEKE'99)*, pp. 1-22, Kaiserslautern, Germany.

WANG, A. I., HANSSEN, A. A., NYMOEN, B. S., 2001, "Design Principles For A Mobile, Multi-Agent Architecture For Cooperative Software Engineering". In: http://citeseer.nj.nec.com/342416.html, Accessed in 20/11/2001.

WERNER, C. M. L., BRAGA, R. M. M., MATTOSO, M. L. Q., et al., 2000, "Infra-estrutura Odyssey: estágio atual". In: *XIV Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas*, pp. 366-369, João Pessoa, Brasil.

WERNER, C. M. L., TRAVASSOS, G. H., ROCHA, A. R., et al., 1996, *Memphis: Um Ambiente para Desenvolvimento de Software Baseado em Reutilização: Definição da Arquitetura*. COPPE/UFRJ.

WFMC, 1999, "Interface 1: Process Definition Interchange". In: http://www.wfmc.org/standards/docs/TC-1016-P_v11_IF1_Process_definition_Interchange.pdf, Accessed in 06/12/2001.

WFMC, 2002, "Workflow Management Coalition". In: **www.wfmc.org**, Accessed in 26/03/2002.

WIRFS-BROCK, R. J., JOHNSON, R. E., 1990, "Surveying Current Research in Object-Oriented design", *Communications of the ACM*, v. 9, n. 33, pp. 104-124.

WOHLIN, C., RUNESON, P., HÖST, M., et al., 2000, *Experimentation in Software Engineering - An Introduction*, Norwell, Massachusetts, Kluwer Academic Publishers.

WOOLDRIDGE, M. J., JENNINGS, N. R., 1994, "Agent Theories, Architectures, and Languages: A Survey". In: *Workshop on Agent Theories, Architectures and Languages*, pp. 1-32, Amsterdam, Netherlands.

WOOLDRIDGE, M. J., JENNINGS, N. R., 1995, "Intelligent agents: Theory and practice", *The Knowledge Engineering Review*, v. 10, n. 2, pp. 115-152.

Apêndice A – Código Prolog do processo do CtrlPESC

Neste apêndice é exibido o código gerado após o mapeamento do processo gráfico do CtrlPESC para Prolog:

```
processoComposto(0).
pergunta(1,'Foram encontrados defeitos?').
simulado(1,'0.5').
papel(1,'Gerente').
resposta(1,'Nao',termino).
processoComposto(3).
processoPrimitivo(5).
roteiro(5,'Construir os diagramas de usuarios administrativos e
      professores.').
simulado(5,'20.98').
ferramenta(5,'Diagramador de casos de uso').
papel(5,'Analista').
artefatoSaida(5,'Diagrama de casos de uso').
nome(5,'Analise Usuarios e Professores').
classeProcesso(4,5).
fluxo(processo(4),sincronismo(6)).
processoPrimitivo(8).
roteiro(8,'Construir os diagramas de linhas de pesquisa e de
      disciplinas.').
simulado(8,'21.76').
ferramenta(8,'Diagramador de casos de uso').
papel(8,'Analista').
artefatoSaida(8,'Diagrama de casos de uso').
nome(8,'Analise Linhas e Disciplinas').
classeProcesso(7,8).
fluxo(processo(7),sincronismo(6)).
fluxo(sincronismo(6),termino).
fluxo(inicio(3),processo(4)).
fluxo(inicio(3),processo(7)).
processoPrimitivo(10).
roteiro(10,'Construir os diagramas de alunos e de inscricoes.').
simulado(10,'24.86').
ferramenta(10,'Diagramador de casos de uso').
papel(10,'Analista').
artefatoSaida(10,'Diagrama de casos de uso').
nome(10,'Analise Alunos e Inscricoes').
classeProcesso(9,10).
fluxo(inicio(3),processo(9)).
fluxo(processo(9),sincronismo(6)).
nome(3,'Analise').
classeProcesso(2,3).
resposta(1,'Sim',processo(2)).
fluxo(decisao(1),termino).
fluxo(decisao(1),processo(2)).
processoPrimitivo(12).
roteiro(12,'A partir da definicao dos casos de uso, verificar quais
      sao os requisitos funcionais que podem influenciar na decisao de
      qual e a melhor arquitetura para o problema em questao. Essa
      arquitetura deve ser modelada em um diagrama de classes que
```

```
      servira como insumo para as demais atividades de projeto,
      juntamente com os diagramas de caso de uso.').
simulado(12,'65.52').
ferramenta(12,'Diagramador de classes').
ferramenta(12,'Diagramador de casos de uso').
papel(12,'Projetista').
artefatoEntrada(12,'Diagrama de casos de uso').
artefatoSaida(12,'Diagrama de classes').
nome(12,'Projeto Arquitetural').
classeProcesso(11,12).
fluxo(processo(2),processo(11)).
fluxo(inicio(0),processo(2)).
processoComposto(14).
processoPrimitivo(16).
roteiro(16,'testar o modulo de usuarios administrativos e
      professores.').
simulado(16,'305.87').
ferramenta(16,'Ambiente de Teste').
papel(16,'Testador').
artefatoEntrada(16,'Codigo fonte').
artefatoSaida(16,'Casos de teste').
nome(16,'Teste de Usuarios e Professores').
classeProcesso(15,16).
fluxo(inicio(14),processo(15)).
processoPrimitivo(18).
roteiro(18,'testar o modulo de linhas de pesquisa e disciplinas.').
simulado(18,'317.26').
ferramenta(18,'Ambiente de Teste').
papel(18,'Testador').
artefatoEntrada(18,'Codigo fonte').
artefatoSaida(18,'Casos de teste').
nome(18,'Teste de Linhas e Disciplinas').
classeProcesso(17,18).
fluxo(inicio(14),processo(17)).
processoPrimitivo(20).
roteiro(20,'testar o modulo de alunos e inscricoes.').
simulado(20,'362.59').
ferramenta(20,'Ambiente de Teste').
papel(20,'Testador').
artefatoEntrada(20,'Codigo fonte').
artefatoSaida(20,'Casos de teste').
nome(20,'Teste de Alunos e Inscricoes').
classeProcesso(19,20).
fluxo(inicio(14),processo(19)).
fluxo(processo(15),sincronismo(21)).
fluxo(processo(17),sincronismo(21)).
fluxo(processo(19),sincronismo(21)).
processoPrimitivo(23).
roteiro(23,'testar o sistema como um todo.').
simulado(23,'0.0').
ferramenta(23,'Ambiente de Teste').
papel(23,'Testador').
artefatoEntrada(23,'Casos de teste').
artefatoEntrada(23,'Codigo fonte').
artefatoSaida(23,'Casos de teste').
nome(23,'Teste do Sistema').
classeProcesso(22,23).
fluxo(processo(22),termino).
fluxo(sincronismo(21),processo(22)).
nome(14,'Testes').
classeProcesso(13,14).
```

```
fluxo(processo(13),decisao(1)).
processoComposto(25).
processoPrimitivo(27).
roteiro(27,'Construir os diagramas detalhados de classes, sequencia e
      estado para usuarios administrativos e professores.').
simulado(27,'59.83').
ferramenta(27,'Diagramador de estados').
ferramenta(27,'Diagramador de sequencia').
ferramenta(27,'Diagramador de classes').
ferramenta(27,'Diagramador de casos de uso').
papel(27,'Projetista').
artefatoEntrada(27,'Diagrama de casos de uso').
artefatoSaida(27,'Diagrama de estados').
artefatoSaida(27,'Diagrama de sequencia').
artefatoSaida(27,'Diagrama de classes').
nome(27,'Projeto de Usuarios e Professores').
classeProcesso(26,27).
processoPrimitivo(29).
roteiro(29,'codificar os casos de uso de usuarios administrativos e
      professores.').
simulado(29,'104.89').
ferramenta(29,'Diagramador de estados').
ferramenta(29,'Diagramador de sequencia').
ferramenta(29,'Diagramador de classes').
ferramenta(29,'Diagramador de casos de uso').
ferramenta(29,'Ambiente de Programacao').
papel(29,'Programador').
artefatoEntrada(29,'Diagrama de estados').
artefatoEntrada(29,'Diagrama de sequencia').
artefatoEntrada(29,'Diagrama de classes').
artefatoEntrada(29,'Diagrama de casos de uso').
artefatoSaida(29,'Codigo fonte').
nome(29,'Implementacao de Usuarios e Professores').
classeProcesso(28,29).
fluxo(processo(26),processo(28)).
processoPrimitivo(31).
roteiro(31,'Construir os diagramas detalhados de classes, sequencia e
      estado para linhas de pesquisa e disciplinas.').
simulado(31,'61.9').
ferramenta(31,'Diagramador de estados').
ferramenta(31,'Diagramador de sequencia').
ferramenta(31,'Diagramador de classes').
ferramenta(31,'Diagramador de casos de uso').
papel(31,'Projetista').
artefatoEntrada(31,'Diagrama de casos de uso').
artefatoSaida(31,'Diagrama de estados').
artefatoSaida(31,'Diagrama de sequencia').
artefatoSaida(31,'Diagrama de classes').
nome(31,'Projeto de Linhas e Disciplinas').
classeProcesso(30,31).
processoPrimitivo(33).
roteiro(33,'codificar os casos de uso de linhas de pesquisa e
      disciplinas.').
simulado(33,'108.78').
ferramenta(33,'Ambiente de Programacao').
ferramenta(33,'Diagramador de estados').
ferramenta(33,'Diagramador de sequencia').
ferramenta(33,'Diagramador de classes').
ferramenta(33,'Diagramador de casos de uso').
papel(33,'Programador').
artefatoEntrada(33,'Diagrama de estados').
```

```
artefatoEntrada(33,'Diagrama de sequencia').
artefatoEntrada(33,'Diagrama de classes').
artefatoEntrada(33,'Diagrama de casos de uso').
artefatoSaida(33,'Codigo fonte').
nome(33,'Implementacao de Linhas e Disciplinas').
classeProcesso(32,33).
fluxo(processo(30),processo(32)).
processoPrimitivo(35).
roteiro(35,'Construir os diagramas detalhados de classes, sequencia e
      estado para alunos e inscricoes.').
simulado(35,'70.7').
ferramenta(35,'Diagramador de estados').
ferramenta(35,'Diagramador de sequencia').
ferramenta(35,'Diagramador de classes').
ferramenta(35,'Diagramador de casos de uso').
papel(35,'Projetista').
artefatoEntrada(35,'Diagrama de casos de uso').
artefatoSaida(35,'Diagrama de estados').
artefatoSaida(35,'Diagrama de sequencia').
artefatoSaida(35,'Diagrama de classes').
nome(35,'Projeto de Alunos e Inscricoes').
classeProcesso(34,35).
processoPrimitivo(37).
roteiro(37,'codificar os casos de uso de alunos e inscricoes.').
simulado(37,'124.32').
ferramenta(37,'Ambiente de Programacao').
ferramenta(37,'Diagramador de estados').
ferramenta(37,'Diagramador de sequencia').
ferramenta(37,'Diagramador de classes').
ferramenta(37,'Diagramador de casos de uso').
papel(37,'Programador').
artefatoEntrada(37,'Diagrama de estados').
artefatoEntrada(37,'Diagrama de sequencia').
artefatoEntrada(37,'Diagrama de classes').
artefatoEntrada(37,'Diagrama de casos de uso').
artefatoSaida(37,'Codigo fonte').
nome(37,'Implementacao de Alunos e Inscricoes').
classeProcesso(36,37).
fluxo(processo(34),processo(36)).
fluxo(sincronismo(38),termino).
fluxo(processo(28),sincronismo(38)).
fluxo(processo(32),sincronismo(38)).
fluxo(processo(36),sincronismo(38)).
fluxo(inicio(25),processo(26)).
fluxo(inicio(25),processo(30)).
fluxo(inicio(25),processo(34)).
nome(25,'Projeto Detalhado e Implementacao').
classeProcesso(24,25).
fluxo(processo(11),processo(24)).
fluxo(processo(24),processo(13)).
nome(0,'Processo CtrlPESC').
processoRaiz(0).
```