UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

~~APSEE~~-Reuse: um Meta-Modelo para Apoiar a Reutilização de Processos de Software

por
RODRIGO QUITES REIS

Tese submetida à avaliação como requisito parcial
para obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Daltro José Nunes
Orientador

Porto Alegre, julho de 2002

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Reis, Rodrigo Quites
APSEE-Reuse: um meta-modelo para apoiar a reutilização de processos de software / por Rodrigo Quites Reis. – Porto Alegre: PPGC da UFRGS, 2002.
215f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Nunes, Daltro José.

1. Modelos de processos de software 2. Reutilização de processos de software. 3. Ambientes de desenvolvimento de software. 4. Especificação algébrica. I. Nunes, Daltro José. II. Título.

Agradecimentos

Agradeço a Deus e à minha padroeira Nossa Senhora de Nazaré por ter tido tranqüilidade e saúde para concluir esse trabalho.

Agradeço à minha mulher Carla, minha fonte de inspiração e companhia inseperável durante todo o trabalho, sempre compartilhando as (muitas) alegrias e (poucas) tristezas do período. Sou muito grato a ti pela força e tranqüilidade que tu sempre que passaste. Espero passar muitos anos da minha vida ao teu lado, contando com o teu amor e carinho.

Aos meus pais e avós, agradeço por todo o estímulo, pelo exemplo de perseverança, e todo esforço em tornar tranqüila a nossa vida em Porto Alegre. A saudade de Belém foi suplantada pelas suas orações e pela alegria nos nossos contatos à distância.

Agradeço ao meu orientador, Professor Daltro José Nunes, pela amizade e confiança depositada no meu trabalho. Ter trabalhado sob a tua orientação no Doutorado e Mestrado foi uma oportunidade ímpar de aprendizado para a vida acadêmica.

Agradeço aos membros do grupo PROSOFT pelas sugestões e incentivo para a realização desse trabalho. Em especial, eu gostaria de agradecer ao Srs. Heribert Schlebbe, do Instituto de Informática da Universidade de Sttutgart e Marcelo Abreu, pela eficiência demonstrada na evolução do PROSOFT e no auxílio para implementação dos alicerces básicos sistema APSEE. Meu agradecimento também aos bolsistas Leonardo Viccari e Fausto Piovesan pelo auxílio na implementação do *SearchEngine*.

Agradeço à nossa melhor amiga Alessandra Dahmer, que nos acolheu como membros de sua família. As conversas, jantares, e os telefonemas noturnos para perguntar "se estávamos bem" ficarão marcados na minha memória. Ao Lincoln, agradeço pelo incentivo, pelas estórias sempre bem humoradas, e pela companhia na torcida pelo Flamengo nas "jornadas esportivas" das tardes de domingo.

À Idy, agradeço pelas conversas sempre bem humoradas, pela amizade e incentivo. Agradeço também a todos os amigos que fizemos nesse período em que moramos no Sul.

Agradeço aos colegas da "sala 245" que acompanharam o dia-a-dia do meu trabalho. Um especial abraço ao amigo Adenauer Yamin, cuja simpática companhia foi responsável por transformar aquele nosso "aquário" (sem janelas para o mundo exterior) em um excelente ambiente de trabalho. As conversas que tivemos contigo ficaram marcadas como verdadeiras lições de experiência sobre o mundo acadêmico e científico ao nosso redor.

Agradeço a CAPES e a Universidade Federal do Pará pelo apoio financeiro fundamentais à realização desse trabalho.

Finalmente, agradeço aos meus colegas do Departamento de Informática da Universidade Federal do Pará pelo estímulo à continuidade dos meus estudos.

Sumário

Lista de Abreviaturas

ADS	Ambiente de Desenvolvimento de Software
ATO	Ambiente de Tratamento de Objetos
CASE	*Computer-Aided Software Engineering*
DDL	*Data Definition Language*
DML	*Data Modeling Language*
GG	Gramáticas de Grafos
HTML	*HyperText Markup Language*
ICS	Interface de Comunicação do Sistema
NSD	*Nassi-Schneidermann Diagram*
PPGC	Programa de Pós-Graduação em Computação
RUP	*Rational Unified Process*
PSEE	*Process-Centered Software Engineering*
SADT	*Structured Analysis and Design Technique*
SW-CMM	*Capability Maturity Model for Software*
UFRGS	Universidade Federal do Rio Grande do Sul
UML	*Unified Modeling Language*

Lista de Figuras

Lista de Tabelas

Resumo

Dentre as principais áreas que constituem a Ciência da Computação, uma das que mais influenciam o mundo atual é a Engenharia de Software, envolvida nos aspectos tecnológicos e gerenciais do processo de desenvolvimento de software. Software tornou-se a base de sustentação de inúmeras organizações dos mais diversos ramos de atuação espalhados pelo planeta, consistindo de um elemento estratégico na diferenciação de produtos e serviços atuais. Atualmente, o software está embutido em sistemas relacionados a infindável lista de diferentes ciências e tecnologias.

A Tecnologia de Processo de Software surgiu em meados da década de 1980 e representou um importante passo em direção à melhoria da qualidade de software através de mecanismos que proporcionam o gerenciamento automatizado do desenvolvimento de software. Diversas teorias, conceitos, formalismos, metodologias e ferramentas surgiram nesse contexto, enfatizando a descrição formal do modelo de processo de software, para que possa ser automatizado por um ambiente integrado de desenvolvimento de software. Os modelos de processos de software descrevem o conhecimento de uma organização e, portanto, modelos que descrevem experiências bem sucedidas devem ser continuamente disseminados para reutilização em diferentes projetos. Apesar da importância desse tópico, atualmente apenas uma pequena porção do conhecimento produzido durante o desenvolvimento de software é mantido para ser reutilizado em novos projetos.

Embora, à primeira vista, o desafio de descrever modelos reutilizáveis para processos de software pareça ser equivalente ao problema tratado pela tradicional área de reutilização de produtos software, isso é apenas parcialmente verdade, visto que os processos envolvem elementos relacionados com aspectos sociais, organizacionais, tecnológicos e ambientais. A crescente complexidade da atual modelagem de processos vem influenciando a investigação de tecnologias de reutilização que sejam viáveis nesse campo específico. A investigação conduzida nesse trabalho culminou na especificação de um meta-modelo que tem como objetivo principal aumentar o nível de automação fornecido na reutilização de processos, apoiando a modelagem de processos abstratos que possam ser reutilizados em diferentes contextos.

O meta-modelo proposto por esse trabalho - denominado *APSEE*-Reuse - fornece uma série de construtores sintáticos que permitem que os diferentes aspectos desse contexto sejam descritos segundo múltiplas perspectivas, complementares entre si, contribuindo para diminuir a complexidade do modelo geral. A solução proposta destaca-se por fornecer um formalismo para modelagem de processos, o qual é integrado à uma infraestrutura de automação de processos de software, permitindo que a reutilização esteja intimamente relacionada com as outras etapas do ciclo de vida de processos.

Os diferentes componentes envolvidos na definição do modelo *APSEE*-Reuse proposto foram especificados algebricamente, constituindo uma base semântica de alto

nível de abstração que deu origem a um conjunto de protótipos implementados no ambiente PROSOFT-Java.

O texto ainda discute os experimentos realizados com o meta-modelo proposto na especificação de diferentes estudos de casos desenvolvidos a partir de exemplos retirados na literatura especializada, e de processos que fornecem soluções em contextos e necessidades específicas de projetos desenvolvidos no PPGC-UFRGS. Finalmente, são apresentadas considerações acerca dos trabalhos relacionados, os elementos críticos que influenciam a aplicabilidade do modelo e as atividades adicionais vislumbradas a partir do trabalho proposto.

Palavras-chave: reutilização de processos de software, especificação algébrica, modelos de processos de software, PROSOFT.

TITLE: "*APSEE*-REUSE: AUTOMATED SUPPORT FOR SOFTWARE PROCESS REUSE"

Abstract

Software Engineering constitutes an important field in the modern Computer Science by combining different management and technological strategies to support software development. Software is the base for a number of organizations involved in different activities around the planet, consisting on a strategic element to differentiate current products and services. Nowadays, software is embedded in a number of systems related to a wide selection of sciences and technologies.

Software Process Technology emerged in the 1980s, aiming to improve software quality through automated management tools. Thus, tools, languages and methodologies were developed to improve the software process models adopted by industry. Process Technology is based on the notion of formal and precise description of the software development process, which allows the development of automated tools to support process management. Process models describe the knowledge of an enterprise and successful models must be continually disseminated for reuse in different projects. In spite of the importance of this topic, only a small portion of the knowledge produced by software processes is retained and reused in new projects.

At a first sight, the challenge of providing reusable software process models seems to be analogous to the conventional software artifact reuse domain. However, this is only partially true since a process model captures a multi-dimensional reality that mixes social, organizational and technological issues. The increasing complexity on current software process modeling implies on the investigation of novel technologies to support the specific requirements for process reuse. The research on this topic described by this text evolved to the specification of a meta-model which aims to increase the level of automation provided for the process model reuse task, supporting the definition of abstract process models to be reused across different contexts.

The meta-model proposed by this work provides a number of syntactical constructors to support the description of process models according to a multi-perspective approach, which has the potential to decrease the overall complexity. In addition, the proposed solution provides a modeling formalism that is integrated to a process enactment mechanism, allowing the inclusion of reuse as a first order phase in the process lifecycle.

The required software components for the proposed meta-model were specified through algebraic specification techniques which, in turn, were used to derive Java-based prototypes.

This text also discusses the experiments that based the evaluation of system feasility and the conformance of proposed language constructs with respect to a number of case studies, including context-specific processes and standard examples extracted

from the specialized literature. Finally, it is discussed how this proposal relates to the current technological state-of-the-art and the critical elements that can influence its applicability and effectiveness.

Keywords: software process reuse, algebraic specification, software process model, PROSOFT.

1 Introdução

Dentre as principais áreas que constituem a Ciência da Computação, uma das que mais influenciam o mundo atual é a Engenharia de Software, envolvida nos aspectos tecnológicos e gerenciais do processo de desenvolvimento de software (ou simplesmente processo de software, segundo Gimenes [GIM94]). Software tornou-se a base de sustentação de inúmeras organizações dos mais diversos ramos de atuação espalhados pelo planeta, consistindo no elemento estratégico da diferenciação de produtos e serviços atuais. Atualmente, o software está embutido em sistemas de uma infindável lista de diferentes ciências e tecnologias e, segundo Pressman [PRE2001], será o propulsor dos novos avanços que influenciam uma ampla gama de indústrias, envolvendo desde a Educação Elementar até corporações envolvidas com a Engenharia Genética.

Apesar dos inúmeros avanços recentes nesta disciplina, muito ainda é discutido acerca da baixa qualidade e produtividade da indústria mundial de software, refletindo-se na insatisfação dos seus usuários e em prejuízos financeiros de enormes proporções. Os computadores estão rapidamente tornando-se componentes comuns do dia-a-dia das pessoas que, por sua vez, apontam requisitos de complexidade cada vez maiores.

O fenômeno descrito na literatura especializada como "Crise do Software" é um reflexo da incapacidade da indústria de software em atender plenamente às necessidades de um mercado consumidor cada vez mais exigente. Atualmente, segundo Pressman [PRE2001], cerca de 70% dos investimentos da área são realizados com o objetivo de manter produtos desenvolvidos anteriormente. Segundo Yourdon [YOU95], "a indústria de software mundial caminha a passos largos para a estagnação".

Com o objetivo de solucionar esses problemas, várias tecnologias vêm sendo experimentadas no sentido de apoiar o ciclo de vida do software. Um dos esforços mais significativos corresponde à definição de metodologias voltadas a disciplinar o processo de desenvolvimento através do estabelecimento de etapas bem definidas, proporcionando, desta forma, um mecanismo de controle para o processo. Como conseqüência, muitas organizações de desenvolvimento de software buscam a maturidade no processo de software, usando medidas como o *SEI-Capability Maturity Model for Software* (SW-CMM) [PAU94] para estruturar as iniciativas de melhoria de processo.

O modelo SW-CMM propõe cinco estágios, sendo que uma organização de desenvolvimento de software pode se encaixar desde o nível inicial, onde o desenvolvimento é caótico, até o nível otimizado, onde o gerenciamento de software é ideal. Nesse modelo, algumas áreas-chave são identificadas para que uma organização passe de um nível ao outro do modelo, assumindo um papel importante na verificação da qualidade do software produzido. Uma das áreas-chave determinadas pelo SW-CMM, para que uma organização possa obter um aumento na maturidade dos seus processos, consiste na definição de processos reutilizáveis que determinem as

estratégias gerenciais adotadas durante o desenvolvimento de qualquer produto de software pela organização.

Segundo Mili em [MIL95], a experiência prática com a reutilização de produtos de software[1] sugere que, intuitivamente, exista a noção de que ganhos são obtidos com a reutilização pois os componentes reutilizados não precisam ser desenvolvidos desde o início. Além disso, a qualidade global do produto é melhorada se componentes de qualidade são reutilizados. Muitas das aplicações sendo desenvolvidas e mantidas atualmente são similares entre si, surgindo assim muitas oportunidades para se aproveitar a experiência adquirida durante o desenvolvimento destas aplicações em projetos futuros.

No campo de processos de software, a reutilização de processos pode trazer economias, tanto no custo quanto no prazo de entrega na produção do software em série, quando comparado com produtos independentemente produzidos. Atualmente, muitos processos similares são executados por diferentes organizações de desenvolvimento de software, surgindo a oportunidade de reutilizar a experiência adquirida durante o planejamento e condução de projetos anteriores, a fim de determinar melhores estratégias gerenciais em projetos futuros.

Segundo Osterweil, citado por Kellner em [KEL98], "efetivas descrições de processo de software constituem um dos mais importantes recursos que nós, como sociedade, possuímos". Apesar da importância desse tema, apenas uma pequena parte do conhecimento, produzido a partir das atividades do ciclo de vida de software, é registrado para ser posteriormente recuperado em novos projetos. Além das especificações de requisitos, projetos, programas, documentos, casos de teste e outros tipos de artefatos comumente produzidos, o conhecimento adquirido sobre o domínio da aplicação a partir das decisões gerenciais tais como a política de alocação de recursos e o escalonamento das atividades deve ser registrado para definir uma base de conhecimento que seja reutilizável em projetos futuros.

Embora pareça, à primeira vista, que o desafio de descrever elementos reutilizáveis para processos de software seja equivalente ao problema de reutilização de software tradicional (i.e., reutilização de produtos de software), percebe-se tratar-se de uma meia verdade, visto que um modelo de processo de software envolve elementos relacionados com aspectos sociais, organizacionais, tecnológicos e ambientais. A crescente complexidade da modelagem atual de processos de software implica na investigação de tecnologias de propósito geral que sejam viáveis nesse campo específico, tal como descrito nas seções a seguir.

Esse texto apresenta, assim, um meta-modelo como contribuição para a Tecnologia de Processo de Software, especificamente com o aumento do nível de automação fornecido na reutilização de modelos, consolidando estudos anteriormente realizados pelo autor no tópico (em especial, o Trabalho Individual [REI99] e o Exame de Qualificação[2] [REI2000]), dentro do Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (PPGC-UFRGS). Dessa forma, as

---

[1] Este texto adota simplesmente "reutilização de software" para denotar especificamente tópicos relacionados com a reutilização de produtos de software.

[2] O Exame de Qualificação teve como temas de abrangência a "Tecnologia de Processo de Software", e profundidade em "Reutilização de Processos de Software", tendo sido avaliado pelos Professores Doutores Maria Lúcia Blanck Lisbôa e Carlos Alberto Heuser em fevereiro de 2000.

seções a seguir apresentam o contexto, o problema tratado, os objetivos traçados, a abordagem escolhida para atingir os objetivos, e a organização do texto.

1.1 O Contexto

A Tecnologia de Processo de Software surgiu em meados da década de 1980 e representou um importante passo em direção à melhoria da qualidade de software através de mecanismos que proporcionam o gerenciamento automatizado do desenvolvimento de software [FEI93]. Diversas teorias, conceitos, formalismos, metodologias e ferramentas surgiram nesse contexto, enfatizando a descrição de um modelo de processo de software que é automatizado por um ambiente integrado de desenvolvimento de software.

Informalmente, o processo de software pode ser compreendido como o conjunto de todas as atividades necessárias para transformar os requisitos do usuário em software [HUM89] [OST87]. Um processo de software é formado por um conjunto de passos de processo parcialmente ordenados, relacionados com conjuntos de artefatos, pessoas, recursos, estruturas organizacionais e restrições tendo como objetivo produzir e manter os produtos de software finais requeridos [LON93] [DOW91].

A Tecnologia de Processo de Software emergiu como conseqüência da maior ênfase, na comunidade internacional, da avaliação da qualidade de software a partir da investigação dos processos adotados no seu desenvolvimento. Seus principais objetivos são:

- Definir ferramentas para descrever os processos e acompanhar a sua execução, controlando a realização de atividades que ocorrem no desenvolvimento de software, registrando as ocorrências e detectando anomalias que possam ser corrigidas em projetos correntes e futuros;
- Facilitar a adoção de uma estratégia de melhoria da maturidade dos processos de uma organização, a partir da prescrição, monitoração e registro automatizado de eventos;
- Permitir o registro do conhecimento produzido acerca dos processos bem sucedidos na organização, para ser reutilizado em contextos similares no futuro;
- Proporcionar um controle preciso na alocação e consumo de recursos durante o desenvolvimento de software;
- Coletar métricas dos projetos da organização e torná-las disponíveis para consultas posteriores.

Desse modo, inúmeros ambientes e ferramentas foram propostos nos últimos anos na literatura em Engenharia de Software para "definir, observar, analisar, aperfeiçoar e executar processos de software" e constituem o que hoje denominamos Tecnologia de Processo de Software [DER99]. As subseções a seguir aprofundam-se na definição da terminologia usada nesse trabalho.

1.1.1 Modelagem do Processo de Software

O processo de software envolve atividades complexas desempenhadas por pessoas (agentes) com as mais diversas capacidades, sendo controladas por gerentes de

desenvolvimento. Assim, um modelo de processo de software é uma descrição formal do processo de software. Vários tipos de dados são integrados em um modelo de processo para indicar quem, quando, onde, como e por que os passos são realizados [LON93]. Para representar um modelo de processo de software é freqüentemente adotada uma *Process Modelling Language* (PML[3]), a qual deve oferecer recursos para descrever e manipular os passos do processo.

As PMLs constituem uma categoria específica de linguagens de especificação e programação de computadores voltada à definição e automação do processo de software. Assim sendo, as PMLs possuem algumas características próprias, que as distanciam das linguagens de programação de propósito geral, incluindo, por exemplo, a necessidade de gerenciar atividades "semi-automáticas", ou seja, atividades que necessitam da interação humana para serem executadas.

Diversas classificações são encontradas na literatura com o objetivo de facilitar o estudo das PMLs, envolvendo aspectos díspares como o paradigma computacional usado para modelagem e a adequação da linguagem às diferentes etapas do meta-processo de software.

A grande maioria das abordagens existentes adota o paradigma orientado a atividades, que possui como vantagem significativa o fato de fornecer um estilo intuitivo para especificação da ordenação dos eventos, o que facilita a execução e acompanhamento dos processos. A evolução recente aponta para abordagens híbridas, a partir da constatação que nenhuma abordagem mono-paradigma satisfaz todos os requisitos de modelagem de processos. Assim, as soluções atuais caminham para estender o paradigma orientado a atividades com objetivo de superar as suas principais limitações relacionadas por Franch [FRA2000].

A literatura especializada distingue três tipos principais de modelos de processo (segundo Ribó e Franch [FRA2002], Derniame [DER99] e Feiler [FEI93]):

- **Modelos Abstratos** (também denominados *patterns* ou *templates*), que fornecem moldes de solução para um problema comum, em um nível de detalhe que idealmente não está relacionado a uma organização específica. Um processo abstrato é um modelo de alto nível que é projetado para regular a funcionalidade e interações entre cargos de desenvolvedores, gerentes, usuários e ferramentas em um PSEE [WAN2000];
- **Modelos Instanciados** (ou executáveis[4]) são modelos prontos para execução, podendo ser submetidos à execução por uma máquina de processo. O modelo instanciado é considerado uma instância de um modelo abstrato, com objetivos e restrições específicos, envolvendo agentes, prazos, orçamentos, recursos e um processo de desenvolvimento (o encadeamento de atividades necessárias para atingir o objetivo);

---

[3] Segundo Heimbigner (em [HEI91]), linguagens de programação de processos (*process programming languages*) se diferenciam por permitir a execução automática das instâncias criadas segundo a sintaxe da linguagem. Em virtude da recente ênfase em aproximar as PMLs das linguagens de programação de processos (fornecendo semântica para execução de PMLs, sendo que estas também são chamadas PMLs de segunda geração [SUT97]), este texto segue uma abordagem recente, também defendida por [FRA00] e [SUT97], que não distingue as linguagens de modelagem daquelas voltadas à programação de processos.

[4] Do original em inglês *enactable* [RIB2002].

- **Modelos em Execução**[5] ou **Executados** registram o histórico da execução de um processo, incluindo os eventos e desvios (modificações realizadas no modelo relacionado).

Assim, a modelagem de processos de software é freqüentemente relacionada à sua execução, descrita na seção a seguir.

1.1.2 Execução de Modelos de Processo

Conforme introduzido pela seção anterior, os modelos de processo de software devem ser executados a fim de automatizar a gerência do desenvolvimento. Na fase de execução, devem ser levadas em consideração as questões de coordenação de múltiplos usuários e a interação desses com as ferramentas disponíveis. Para isso, é necessário que se obtenha um modelo de processo executável (ou instanciado), ou seja, um modelo que descreva o processo com tal nível de detalhe que permita a sua execução por uma máquina. Portanto, a execução de modelos de processo de software deve levar em consideração as questões de coordenação de múltiplos usuários, assim como a interação entre as ferramentas e os agentes (profissionais envolvidos no processo).

A modelagem de processos de software não consiste tão somente em escrever programas que automatizem completamente o processo de desenvolvimento de software; tampouco descreve tudo o que as pessoas do processo devem fazer [REI99]. Enquanto os programas de computador são escritos para definir o comportamento de uma máquina determinística, os programas de processo são escritos para definir possíveis padrões de comportamento entre elementos não-determinísticos (pessoas) e ferramentas automatizadas [TUL89]. Por conseqüência, um PSEE deve permitir que as pessoas envolvidas no processo recebam orientação automatizada e assistência na realização de suas atividades, sem interferência no processo criativo [NGU97]. Além disso, processos ainda podem ser modificados dinamicamente (i.e., durante a execução), em resposta a estímulos organizacionais ou mudança nos requisitos do software em desenvolvimento.

PSEEs (*Process-Centered Software Engineering Environments*) ou Ambientes de Desenvolvimento de Software Orientados ao Processo [LIM98]) constituem um tipo especial de ambientes de desenvolvimento de software. Surgiram nos últimos anos para apoiar a definição rigorosa de processos de software, objetivando automatizar a gerência do desenvolvimento. Tais ambientes geralmente provêem serviços para análise, simulação, execução e reutilização das definições de processos, que cooperam no aperfeiçoamento contínuo de processos.

A arquitetura de um PSEE usualmente define como componente central a Máquina de Processo (*process engine*, segundo Derniame [DER99]) que auxilia na coordenação das atividades realizadas por pessoas e por ferramentas automatizadas, sendo responsável pela interpretação/execução dos modelos de processos descritos com PMLs. Segundo Lima Reis em [LIM98], uma máquina de processos é responsável por: ativar automaticamente atividades sem intervenção humana através de uma integração com as ferramentas do ambiente; apoiar o envolvimento cooperativo dos desenvolvedores; monitorar o andamento do processo e registrar o histórico da sua execução. Ainda segundo [LIM98], a máquina de processo também deve garantir a execução das atividades na seqüência definida no modelo de processo (fluxo de

[5] Do original em inglês *enacting* [RIB2002].

controle); a repetição de atividades; a informação de *feedback* sobre o andamento do processo; a gerência das informações de processo (incluindo gerência de versões); a coleta automática de métricas; a mudança do processo durante sua execução; a interação com as ferramentas do ambiente e a gerência de alocação de recursos.

1.2 O Problema

O termo reutilização de processos de software (denominação adotada nesse trabalho a partir da expressão *software processes reuse*, comumente usada pela literatura especializada em inglês) define uma ampla área de estudo e utilização prática relacionada aos diferentes aspectos envolvidos com a reutilização do conhecimento adquirido na condução de projetos anteriores. A evolução dessa área foi decisivamente influenciada pelos avanços nos estudos da reutilização de produtos de software e da Tecnologia de Processo de Software. Além disso, intuitivamente, a noção de que processos de software sejam reutilizáveis tomou forma a partir do surgimento das primeiras metodologias de desenvolvimento de software, após meados de 1980, metodologias essas que buscavam definir abordagens sistemáticas e repetíveis para guiar os desenvolvedores e gerentes.

Modelos de processos de software são estruturas complexas, que interrelacionam diferentes aspectos tecnológicos, organizacionais e sociais para descrever o processo de desenvolvimento de software. A modelagem de processos requer profissionais altamente especializados, denominados projetistas de processos (*process designers*), que segundo Derniame [DER99] são responsáveis por fornecer o modelo de processo e adaptar o ambiente PSEE que apóia a execução do processo. Assim, a complexidade no desenvolvimento de modelos de processos motivou recentemente o surgimento de uma ampla área de pesquisa em reutilização de processos, envolvendo a definição de formalismos e ferramentas para facilitar a construção de modelos de processos a partir de processos e componentes já existentes.

A comunidade internacional vem realizando um importante esforço nos últimos anos para incrementar o nível de flexibilidade fornecido pelas linguagens de modelagem de processos e os ambientes de execução (PSEEs) relacionados[6]. Indubitavelmente a tecnologia de reutilização de processos de software adiciona complexidade a esse contexto pois lida com requisitos contraditórios. Enquanto do ponto de vista gerencial, representações de processos tendem a ser executáveis (possuindo, portanto, informações detalhadas que prescrevem o comportamento desejado para o desenvolvimento de software), as descrições reutilizáveis, por sua vez, tendem a ser abstratas e genéricas, estimulando propositadamente múltiplas interpretações. Portanto, muitos aspectos ainda necessitam de aprofundamento, permanecendo como desafios importantes para a comunidade de Engenharia de Software.

Na primeira geração de PSEEs e PMLs, a reutilização de processos era tratada, principalmente, por meio da habilidade de armazenar componentes de processos que poderiam ser recuperados posteriormente, de alguma maneira. Modelos de processos eram descritos individualmente, sem preocupação com elementos comuns, assim como com o potencial de reutilização em contextos diferentes do original. Dessa forma, a principal idéia era fornecer operações para "copiar e modificar" elementos previamente

[6] Finkelstein [FIN94] e Derniame [DER99] apresentam uma análise ampla sobre o assunto.

descritos no desenvolvimento de novos modelos. Embora "copiar e modificar" seja importante, pois minimiza o esforço na construção de um novo modelo, essa prática é menos útil para o aperfeiçoamento da organização e dos seus processos [JØR2001], já que, por meio dela, não são registradas informações que permitiriam acompanhar a evolução dos processos e seus componentes.

O tópico "reutilização de processos de software" recebeu muito mais atenção de pesquisadores e da indústria na segunda metade da década de 1990 e, atualmente, mecanismos aperfeiçoados vêm sendo pesquisados e experimentados para o armazenamento e a definição de elementos reutilizáveis de processos. Essa evolução vem sendo motivada principalmente pela necessidade de novas abordagens no sentido de entender os processos correntes em uma organização, assim como no sentido de promover de novas estratégias gerenciais [ELL96].

Algumas das mais críticas limitações atuais, adaptadas de posições dispersas em contribuições de Franch e Ribó [FRA99], Jørgensen [JØR2001], Perry [PER96], Ellmer et al. [ELL96], e observações pessoais do autor em [REI99] incluem:

- Poucas PMLs possuem base semântica formal e capacidade de lidar com detalhes de granulosidade fina, fornecendo simultaneamente notação gráfica de alto nível;
- A avaliação prática das abordagens atualmente disponíveis demonstra que estas não foram projetadas para apoiar a modelagem e reutilização como etapas primordiais do meta-processo de software, conforme descrito por Franch e Ribó em [FRA99];
- Não existe atualmente uma PML ou ao menos uma notação gráfica padrão amplamente aceita pela comunidade internacional;
- A maioria dos modelos de processos é estritamente restritiva no sentido de lidar com múltiplas interpretações e desvios gerados a partir de decisões tomadas durante a sua instanciação e execução;
- Segundo Zirbes, em [ZIR95], um aspecto importante na reutilização de produtos de software "é a sua possível transcendência aos limites de uma única organização". De forma análoga, desde o surgimento das primeiras metodologias para disciplinar o desenvolvimento de software na década de 1980, há um estímulo à busca de padrões organizacionais. Entretanto, os aspectos organizacionais são freqüentemente descritos de forma precária nas abordagens propostas no contexto de processo de software. Em especial, a experiência bem sucedida dos ambientes *workflow* é um indício de que esse é um aspecto crítico para o sucesso na automação de processos complexos [REI99];
- A experiência recente, descrita pelo autor em [REI2001d] e [REI2002b], mostra que processos existentes na literatura freqüentemente descrevem de maneira informal estratégias gerenciais que não são suportadas adequadamente pelos paradigmas atuais;
- Os paradigmas de modelagem atuais fornecem construtores derivados de linguagens de programação convencionais (i.e., orientadas a objetos, baseadas em regras e/ou procedimentais) para definir a modularidade de processos de software. Entretanto, os processos de software são usualmente grandes e envolvem múltiplas dimensões inter-relacionadas, as quais

implicam na coexistência de estilos de modelagem especializados. Ainda: a modularidade fornecida pelas PMLs atuais não fornece mecanismos para expressar estilos de modelagem que sejam comuns a diferentes instâncias. Dessa forma, importantes estratégias gerenciais precisam ser repetidamente especificadas, o que eleva a complexidade na criação de manutenção dos modelos construídos.

O presente trabalho trata de pontos específicos desse contexto, como descrito na seção a seguir.

1.3 Objetivos

Considerando as vantagens fornecidas pelas ferramentas e ambientes de gerência do processo de software, assim como os desafios atualmente enfrentados na descrição de modelos de processos de software que sejam reutilizáveis em diferentes contextos, o objetivo desse trabalho é incrementar o nível de automação fornecido por ferramentas que apóiam a reutilização de processos de software, avançando no estado da arte tecnológico. Em face ao exposto, são objetivos gerais desse trabalho:

- Especificar um meta-modelo conceitual que auxilie na construção e manipulação de modelos genéricos de processos de software, apoiando a modelagem de processos reutilizáveis, segundo múltiplas perspectivas, que cooperem entre si. Além disso, pretende-se adotar um modelo uniforme e integrado, que permita a integração no futuro de mecanismos que automatizem a adaptação de elementos recuperados em processos executáveis que atendam necessidades específicas;
- Demonstrar a viabilidade do modelo conceitual proposto através de um protótipo, isto é, uma implementação limitada que forneça a funcionalidade básica descrita pelo modelo conceitual;
- Mostrar que alguns exemplos significativos descritos informalmente na literatura especializada são suportados pelo modelo proposto e que, a partir deles seja possível reutilizar elementos que sejam úteis na construção de novos processos.

1.4 A abordagem escolhida para solução do problema

O trabalho aqui apresentado foi desenvolvido no contexto do projeto PROSOFT [NUN92] [NUN94] que, desenvolvido no PPGC-UFRGS e na Universidade de Stuttgart (Baden-Württemberg, Alemanha) sob orientação do Prof. Dr. Daltro José Nunes, está envolvido com a construção de um ambiente integrado de desenvolvimento de software. Em linhas gerais, o PROSOFT objetiva fornecer ferramentas que auxiliem na utilização de métodos de especificação formal em todas as fases do ciclo de vida de software.

Uma recente linha de pesquisa do projeto PROSOFT envolve estudantes de pós-graduação e pesquisadores das instituições envolvidas na definição de ferramentas para apoiar a modelagem [REI2001a], execução [LIM98], simulação [SIL2001], e visualização [SOU2002] de processos cooperativos de desenvolvimento de software, constituindo uma infraestrutura integrada para automação processos de software

denominada APSEE[7] ([REI2001a] e [LIM2001a]). Portanto, o trabalho aqui proposto está inserido no projeto APSEE-PROSOFT, e busca definir um conjunto de ferramentas que possam, integradas às demais, presentes no ambiente, apoiar especificamente a modelagem de processos reutilizáveis.

1.5 Organização do texto

O texto está organizado como segue. O capítulo 2 discute a reutilização de processos de software, e as necessidades para o desenvolvimento de soluções automatizadas nesse contexto. Os requisitos apresentados servem para situar o meta-modelo proposto em função do espectro de soluções descritas na literatura.

O capítulo 3 apresenta uma visão geral do meta-modelo proposto nesta tese.

O capítulo 4 descreve formalmente o meta-modelo proposto através da combinação de diferentes métodos algébricos.

O capítulo 5 discute aspectos do protótipo implementado em PROSOFT-Java, como uma implementação simplificada do meta-modelo APSEE-Reuse proposto.

O capítulo 6 descreve a experiência prática no uso do modelo proposto através da especificação de processos reutilizáveis. Assim, são apresentados exemplos extraídos da literatura, assim como modelos desenvolvidos para contextos específicos.

Finalmente, o capítulo 7 apresenta as considerações finais, discute o modelo proposto de acordo com aspectos importantes, comparando-o com as abordagens existentes na literatura. Além disso, são incluídos os trabalhos futuros vislumbrados a partir do presente trabalho.

[7] APSEE é um acrônimo para "A Process-Centered Software Engineering Environment".

2 Reutilização de Processos de Software: Requisitos para Apoio Automatizado

A evolução recente das tecnologias e ferramentas de gerência de processos de software, gerência de *workflow* e linguagens de modelagem de processos motivou a discussão acerca da necessidade de apoiar, de forma efetiva, a reutilização de processos. Esse capítulo, portanto, descreve requisitos descritos pela literatura especializada, que norteiam o desenvolvimento das abordagens automatizadas, para auxiliar na reutilização de processos de software. Esses requisitos são genéricos e classificados de forma a permitir a comparação do modelo proposto nesse trabalho com abordagens correlatas.

Como, segundo Godart et al. [GOD99], existem múltiplas maneiras para modelar um mesmo processo de software, de forma análoga existem diferentes abordagens que almejam fornecer apoio automatizado à reutilização de processos. É importante observar que a literatura apresenta inúmeras classificações de requisitos para PMLs e PSEEs segundo os mais diversos aspectos e isto é uma conseqüência natural da multidisciplinaridade do tópico. Assim, por exemplo, esse capítulo trata do tema **Execução de Processos de Software** de uma forma unicamente voltada aos aspectos que possam influenciar decisivamente a reutilização dos modelos executados.

2.1 Terminologia adotada

Para identificar as características do problema de reutilização de processos de software, é importante distinguir as diferentes fases do ciclo de vida do processo. O desenvolvimento de modelos de processos de software executáveis está relacionado a um ciclo de vida que a comunidade convencionou chamar **meta-processo de software**. O meta-processo de software vem sendo descrito pela literatura a partir da analogia com o ciclo de vida clássico para software, em conseqüência da semelhança proposital entre processos e software. Assim, esse trabalho descreve o meta-processo de software adotado para o ambiente *APSEE*, sendo formado por atividades comumente aceitas pela comunidade internacional e que contextualizam o problema sendo tratado.

O ciclo inicia-se com a fase de **Análise de Requisitos do Processo** que descreve requisitos para uma descrição inicial do processo a ser desenvolvido (envolvendo detalhes do processo e/ou do produto a ser desenvolvido). Em seguida, a fase de **Projeto** ou **Modelagem** resulta em um modelo de processo abstrato através de uma PML. Durante a modelagem de processos, informações e modelos de processos podem ser recuperados, consistindo na **Reutilização de Processo**, que pode fornecer modelos e fragmentos adequados à situação atual. A **adaptação** (*tailoring*) de projetos e definições de processos e seus componentes para contextos organizacionais e tecnológicos específicos é uma peculiaridade envolvida na reutilização de processos [FEI93].

A **Instanciação de Processo** é encarregada de produzir um processo instanciado a partir de um modelo abstrato, ou seja, com recursos e agentes alocados e cronograma definido. Antes da execução, o processo instanciado pode passar por uma fase de **Validação**, que freqüentemente é relacionada à noção de **Simulação de processo** de software, permitindo antever problemas de alocação e estimar a sua duração. Como resultado da fase de validação, pode ser necessário retornar a fases anteriores para aperfeiçoamento do modelo, até que o modelo esteja pronto para execução.

Durante a **Execução do Processo**, a máquina de processos coordena a interação com gerentes e desenvolvedores e obtém *feedback* sobre o andamento do processo. Em decorrência da natureza complexa e dinâmica do contexto de execução de processos, freqüentemente é necessário realizar modificações no modelo durante a sua execução, o que leva à uma abordagem de modelagem incremental: às modificações realizadas em um processo em execução é dado o nome **Evolução do Processo**. Assim, as fases de projeto, instanciação e validação podem ocorrer concorrentemente com a execução para porções independentes de um mesmo processo. Após execução, o modelo de processo é enviado à fase de **Avaliação do Processo**, onde novos requisitos podem ser gerados a partir do registro das ocorrências da execução. Processos executados também podem ser enviados à fase de **Generalização de Processo**[8] a qual que alimenta o repositório de histórico de processos. Ainda, a generalização é especialmente útil no contexto de reutilização de processos de software por extrair, a partir de processos anteriormente executados, propriedades comuns que possam ser úteis em diferentes contextos.

Em geral, as soluções automatizadas para reutilização de processos de software desenvolvidas atualmente podem ser classificadas de duas formas, complementares entre si, e que influenciam o desenvolvimento das soluções automatizadas na área:

- A abordagem *top-down*, que visa apoiar a definição de processos genéricos que consistem em modelos de processo de software abstratos (podendo ser usados como ponto de partida para modelagem em situações ou organizações similares, isto é, contextos que compartilhem propriedades comuns) [KRU96]. Um processo genérico pode ser adaptado para compor processos específicos que são mais detalhados e rigorosos, levando em consideração aspectos relacionados com características específicas do domínio de aplicação e uso desejado [LON93].
- A abordagem *bottom-up*, que visa apoiar a definição de componentes reutilizáveis que possuam interface bem definida e sejam responsáveis pela especificação de elementos específicos de um modelo de processo.

Nesse trabalho, **reutilização de processos** de software é a denominação genérica para designar as atividades do meta-processo de software, atividades essas voltadas à reutilização de conhecimento gerencial produzido anteriormente nas organizações de desenvolvimento de software e ainda, que seja útil na modelagem de novos processos. Tal conhecimento é encontrado na literatura especializada de várias formas, variando principalmente quanto ao formalismo adotado (de estratégias gerenciais narrativas até modelos de processos de software formais). Portanto, para registrar tal conhecimento, os esforços vêm se concentrando na tecnologia de

[8] Também denominada *harvesting* [JØR01] [RIB2002]

reutilização de produtos de software, em especial na investigação da aplicabilidade de conceitos análogos às idéias amplamente difundidas (tais como, as bibliotecas e protocolos para composição de componentes, os *frameworks*, e *design patterns*).

A tecnologia de reutilização de processos de software se distancia da reutilização de produtos de software, principalmente, por estar voltada a apoiar a construção de modelos que integra uma grande variedade de componentes, os quais não são diretamente relacionados ao encadeamento de atividades que formam o núcleo de um processo. Assim, aspectos relacionados a estruturas organizacionais, envolvendo, por exemplo, cargos e profissionais específicos ou a ambientes tecnológicos, incluindo ferramentas e recursos computacionais adotados, estão freqüentemente embutidos na definição de um processo. Essa característica, assim como outras, específicas ao contexto tratado, são discutidas sob o ponto de vista do ciclo de reutilização de processos de software, descrito na seção a seguir.

2.2 Meta-processo de Software Voltado à Reutilização

A figura 2.1 apresenta o ciclo de vida de processos de software simplificado e orientado de acordo com a ótica da reutilização dos modelos (adaptado de Jørgensen [JØR2001]). Assim, nem todas as etapas descritas na seção anterior foram incluídas, por não estarem diretamente relacionadas com o tópico reutilização de processos. Na figura, as setas rotuladas denotam etapas, enquanto que vértices rotulados em itálico representam estados. Assim, a etapa **Modelagem de Processos Visando a Reutilização** busca definir através de uma PML modelos de processos e componentes genéricos e abstratos que possam ser, idealmente, reutilizados em diferentes contextos. A etapa **Recuperação e Adaptação de Processos** define critérios, estratégias e mecanismos para auxiliar um projetista na seleção, adaptação e instanciação de processos genéricos que forneçam soluções para o problema sendo tratado. A etapa rotulada como **Execução de Processos** descreve todas as ocorrências que ocorrem em um modelo de processo desde o início de sua execução, ou seja, a partir do momento que o software começa a ser produzido (ou parte do modelo iniciou a execução). Finalmente, a **Generalização e Avaliação de Processos Encerrados** fornece modelos de processos reutilizáveis a partir da experiência adquirida com processos bem sucedidos.

O ciclo de vida (adaptado de uma proposta Jørgensen [JØR2001] e Ellmer et al [ELL96]) é utilizado nesse texto para classificar as soluções automatizadas disponíveis na literatura. Assim, as seções a seguir discutem os requisitos relacionados às ferramentas envolvidas com cada etapa do ciclo de vida proposto.

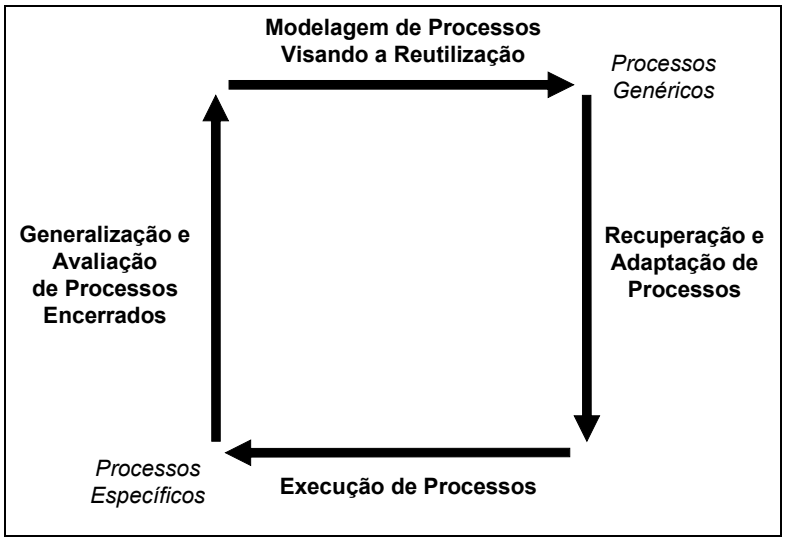

FIGURA 2.1 - Cenário geral para reutilização de processos de software (adaptado de [JØR2001])

2.3 Requisitos atuais para automação da reutilização de processos de software

A classificação apresentada na tabela 2.1 abaixo - e detalhada nas seções a seguir - leva em consideração contribuições e experiências do autor na descrição de processos reutilizáveis a partir de metodologias informais descritas na literatura [REI2001a], e elementos dispersos em [PER96], [KLE98] e [JØR2001] para definição de ambientes e linguagens que auxiliem na reutilização de processos. Alguns requisitos possuem objetivos sobrepostos e/ou inter-relacionados, e isto é evidenciado quando necessário no texto a seguir.

TABELA 2.1 - Requisitos para apoiar a reutilização de processos em ambientes automatizados (PSEEs) e linguagens de modelagem (PMLs)

Categorias	**Requisitos**	
	Id	**Nome**
1. Modelagem de Processos Visando a Reutilização	R1.1	Mecanismos de Abstração na Modelagem de Processos.
	R1.2	Separação de Detalhes em Múltiplas Perspectivas.
	R1.3	Semântica Formal para Modelos de Processos.
	R1.4	PML Simples.
	R1.5	Representação Gráfica para Modelos de Processos.
2. Recuperação e Adaptação de Processos	R2.1	Estruturação de Grandes Bases de Processos Abstratos.
	R2.2	Registro da Aplicabilidade de Processos Reutilizáveis.
	R2.3	Extração de Visões de Modelos de Processos.
	R2.4	Definição Extensível de Tipos da PML.
	R2.5	Adaptação para Contextos Específicos.
3. Execução de Processos	R3.1	Apoio à Evolução de Processos.
	R3.2	Composição Dinâmica de Elementos de Processos.
4. Generalização e Avaliação de Processos Encerrados	R4.1	Generalização de Processos Encerrados.
	R4.2	Comparação de Modelos de Processos.
	R4.3	Conexão de Instâncias aos Processos Abstratos Generalizados.

2.4 Modelagem de Processos Visando a Reutilização

Com o intuito de obter modelos genéricos de processos de software que sejam realmente úteis, esses devem ser empacotados e disponibilizados em estruturas de dados uniformes freqüentemente denominados processos abstratos ou *templates*. Assim, *templates* são modelos genéricos e reutilizáveis que estabelecem um ponto de início para a construção de um novo modelo. A construção de *templates* de processos de software é influenciada pelos requisitos descritos nas sub-seções a seguir.

2.4.1 Requisito 1.1 - Mecanismos de Abstração na Modelagem de Processos

Os mecanismos de abstração tradicionais tais como os operadores de **composição** e **herança** (baseados no paradigma de Orientação a Objetos) são importantes e estão se tornando comuns no projeto de uma PML. Enquanto a composição é importante para descrever modelos e *templates* a partir de diferentes partes menores, a herança é especialmente útil por permitir a descrição de hierarquias de *templates* e seus componentes.

Perry discute em [PER96] a necessidade de mecanismos de abstração adicionais para auxiliar na definição de processos genéricos:

- A **parametrização** permite a definição abstrata de valores, tipos, objetos, atividades e todos os outros componentes de modelos de processos de software, de forma a viabilizar a reutilização de diferentes contextos;
- ***Primitivation*** é uma forma de abstração que requer elaboração antes da execução, adiando a instanciação dos componentes de processos para o momento da execução, sendo norteado por uma gramática formalmente definida, por um conjunto de restrições, ou pela existência de elementos de construção. Esse tipo de abstração permite prover orientação sem pré-julgamento da solução, sendo importante na definição de processos genéricos;
- A **estratificação** é uma forma de abstração que define processos em camadas, permitindo a definição de processos incompletos (somente com suposições e objetivos especificados, deixando a implementação a cargo do seu usuário). Perry recomenda ainda que, a fim de propiciar uma melhor reusabilidade, sejam removidas informações sobre ferramentas e métodos das descrições genéricas de processos. Portanto, deve ser permitida a construção de modelos de processos com múltiplos níveis de detalhe, a partir de composição de elementos simples e tolerando a coexistência de múltiplas perspectivas e diferentes implementações.

2.4.2 Requisito 1.2 - Separação de Detalhes em Múltiplas Perspectivas

Diferentes paradigmas sustentam as inúmeras PMLs apresentadas na literatura. Muitas das PMLs atuais são híbridas (i.e., multiparadigma), permitindo a definição dos diferentes aspectos tratados em um processo usando diferentes paradigmas de modelagem. Como a realidade expressa por um modelo de processos é complexa, a modelagem consiste na integração de diferentes detalhes que demandam formalismos especializados [REI99]. A representação adequada de elementos de processos pode

facilitar o entendimento e tem o potencial de incrementar a reusabilidade dos modelos [REI2000], [REI2001a]. Assim, apoio automatizado adequado deve fornecer mecanismos de integração desses diferentes paradigmas, de forma que eles cooperem harmonicamente.

Deve ser possível definir processos reutilizáveis com diferentes níveis de detalhe, com o objetivo de facilitar a sua aplicação em diferentes contextos. Em geral, quanto maior o detalhamento de um processo, menor será o espectro possível de implementações. Além disso, deve ser possível estabelecer estratégias gerenciais que serão ativadas em virtude de condições verificáveis durante a execução ou em resposta à modificações dos modelos.

Um modelo de processo de software é uma descrição formal de uma realidade complexa. Segundo Dandekar citado em [PER97], a complexidade dos processos é um problema que afeta dramaticamente tanto o seu uso quanto a sua reutilização. Um elemento que contribui significativamente para complexidade é o nível de detalhe e a forma com que esse detalhe é descrito em um formalismo de modelagem. Segundo Perry [PER97], modelos complexos e descritos em níveis de detalhes inadequados inibem definitivamente a reusabilidade.

Ainda de acordo com Perry (em [PER96]), durante a modelagem de processos, o projetista lida com uma estrutura inerentemente multi-dimensional, que mistura informações acerca do **projeto** (com informações sobre o cronograma detalhado na prescrição de um processo instanciado), da **organização** (com os cargos e recursos utilizados no processo) do **ambiente tecnológico** (incluindo as ferramentas específicas utilizadas no desenvolvimento de software), e do **processo** propriamente dito (com informações sobre as atividades e políticas gerenciais que compõem o processo). Além disso, a separação de detalhes[9] é potencialmente útil para nortear a definição de critérios de busca em bases de processos de software reutilizáveis, relacionado com o requisito 2.3 (descrito na seção 2.5.3 a seguir).

2.4.3 Requisito 1.3 - Semântica Formal para Modelos de Processos

Métodos para especificação formal vêm sendo utilizados para apoiar o desenvolvimento de sistemas de software complexos, que demandam requisitos de alta qualidade, visto que a semântica formal permite a verificação matemática de propriedades como completude e correção, assim como a definição precisa de sistemas complexos em alto nível de abstração.

Para reutilizar um processo o projetista precisa, em primeiro lugar, entender o modelo. A fim de possibilitar uma reutilização efetiva do modelo - não somente a cópia dos elementos sintáticos descritos anteriormente e armazenados em uma biblioteca mas sim a reutilização do conhecimento semântico relacionado ao modelo - as PMLs devem ser formais, possuindo uma base semântica bem definida, sendo, portanto, livres de ambigüidade. Entretanto, a experiência prática demonstra que, mesmo adotando formalismos diagramáticos, os modelos de processos de software continuam sendo estruturas complexas que requerem muito esforço dos usuários para compreensão. Assim, PMLs formalmente definidas possuem importantes vantagens adicionais, listadas abaixo:

---

[9] Tradução a partir de *separation of concerns* [LOP97].

- É possível verificar formalmente propriedades de modelos abstratos de processos (por exemplo, prevenção de *deadlock*);
- A notação é formal e precisa. Além disso, as linguagens formais destacam-se por expressar concorrência e sincronização (aspectos de extrema importância na descrição de processos de software) de uma maneira elegante e compacta;
- O entendimento dos modelos e, conseqüentemente, sua reutilização, é facilitada pela definição precisa da semântica dos modelos. Além disso, vários métodos e ferramentas estão disponíveis para simulações (*dry-runs*[10]).

2.4.4 Requisito 1.4 - PML Simples

O projeto e o conseqüente uso de uma PML deve estar baseado em conceitos simples e usuais do domínio das organizações de desenvolvimento de software [ROM2001].

2.4.5 Requisito 1.5 - Representação gráfica para Modelos de Processos

Em geral, a experiência da Engenharia de Software indica que modelos gráficos tendem a ser mais efetivos na descrição de sistemas de informação complexos, facilitando o entendimento do conhecimento armazenado em um modelo de processo e aumentando, assim, a produtividade geral.

2.5 Recuperação e Adaptação de Processos

Quando modelos e componentes de processos estão armazenados em um repositório, é importante que estejam disponíveis mecanismos para recuperar, selecionar e adaptar os componentes reutilizáveis.

A recuperação de processos de software vem despertando atenção da comunidade científica. As principais alternativas discutidas em [JØR2001] são: a busca em texto livre; a busca de palavras-chave; e a estruturação *a priori* de hierarquias de processos usando algum critério bem definido tal como em [MAL99], que combina especialização e decomposição na definição de uma estrutura navegacional.

Por sua vez, a adaptação envolve todas as modificações necessárias para reutilizar um processo recuperado para um contexto específico, envolvendo a modificação sintática do modelo, o refinamento de tipos genéricos para tipos dependentes da organização-destino, e a instanciação de agentes e recursos a partir dos tipos e restrições definidas no modelo original. O lema "única interface com diferentes implementações possíveis" amplamente difundido na Reutilização de Software pode ser trazido para o contexto de processos de software, permitindo que diferentes instâncias de processos sejam geradas a partir de um único *template.* Assim, é importante que diferentes processos possam ser gerados a partir de um mesmo *template*, coexistindo e evoluindo independentemente, tornando-se possível reconciliar tais perspectivas no futuro.

---

[10] Segundo Totland [TOT95], o termo *dry-run* denota execução de processos sem efeito direto no mundo real.

Dessa forma, essa seção discute os requisitos relacionados com a recuperação e adaptação de modelos de processos de software.

2.5.1 Requisito 2.1 - Estruturação de Grandes Bases de Processos Abstratos

A reutilização de software baseada em repositório sugere imediatamente a existência de algum mecanismo que permita recuperar componentes a partir de critérios definidos pelo usuário. Assim, diferentes abordagens são discutidas na literatura, no sentindo de tratar o problema de recuperação de componentes de software [YE2000].

Processos de software são modelos complexos que incluem muitas informações. Dessa forma, a descrição de *templates* deve permitir a busca estruturada de informações, incluindo detalhes sobre os seus objetivos, prerrogativas e recursos necessários [JØR2001]. Propriedades importantes de processos devem ser definidas para definir critérios de buscas especificados pelo usuário [REI2001f]. Um mecanismo de busca deve ser capaz de inferir sobre situações similares, retornando descrições de processos que têm por base as expectativas dos usuários.

2.5.2 Requisito 2.2 - Registro da Aplicabilidade de Processos Reutilizáveis

As descrições de processos de software devem fornecer explicitamente informações sobre os seus objetivos, pré-condições e recursos necessários. Tais propriedades devem ser estruturadas para a realização de buscas segundo critérios definidos por usuários. Além disso, da estruturação da informação é possível inferir sobre situações similares que auxiliem, assim, na identificação de modelos de processos que atendam parcialmente às expectativas dos usuários.

2.5.3 Requisito 2.3 - Extração de Visões de Modelos de Processos

Assim como na modelagem, é importante que os processos sejam descritos segundo perspectivas independentes e bem estruturadas (requisito 1.2 - seção 2.4.2). De forma análoga o contrário também é importante para auxiliar no entendimento de modelos previamente construídos. A projeção de processos, portanto, é uma característica importante que se refere à remoção de algumas das dimensões ou detalhes disponíveis em um modelo de processo. Segundo Avrilionis [AVR96a], um sistema que auxilie na visualização de processos deve ser capaz de extrair diferentes visões dos modelos de processos, partindo de propriedades bem definidas, como, por exemplo, um cargo, uma ferramenta ou um tipo de recurso. Assim, a partir de um modelo de processo completo podem ser realizadas operações que, a critério do usuário, removam recursos ou o fluxo de dados de um processo [SOU2002] a fim de facilitar o entendimento de algum aspecto essencial do modelo.

2.5.4 Requisito 2.4 - Definição Extensível de Tipos da PML

O projeto de uma PML normalmente define um conjunto de meta-tipos básicos que estão relacionados a objetos do mundo real envolvidos com a descrição do desenvolvimento de software. Processos abstratos são associados a tipos “independentes de organização ou ambiente tecnológico”, enquanto que processos executáveis devem descrever elementos sensíveis ao contexto de execução.

2.5.5 Requisito 2.5 - Adaptação para Contextos Específicos

Quando o usuário seleciona um componente de processo para ser reutilizado, esse componente invariavelmente precisa ser adaptado ao contexto tratado. Isso se deve à inerente variabilidade de contextos organizacionais e ao caráter individual dos produtos de software desenvolvidos. Como os processos são executados por humanos, algum nível de flexibilidade é tolerada na adaptação e, portanto, a literatura especializada apresenta poucos trabalhos especializados na adaptação automática de processos para contextos específicos.

2.6 Execução de Processos

A execução de processos é a maneira pela qual os usuários podem obter apoio automatizado na condução do desenvolvimento de software. Durante a execução, o modelo de processo de software pode ser modificado e, por conseqüência, o resultado final freqüentemente difere do *template* original. Tais modificações incluem a mudança de requisitos, modificação na estratégia organizacional, dentre outras. Tais modificações devem ser registradas a fim de contrastar o modelo realmente executado daquele originalmente prescrito. Assim, as sub-seções a seguir apresentam requisitos para apoiar a reutilização de processos durante a execução.

2.6.1 Requisito 3.1 - Apoio à Evolução de Processos

Modelos de processos são estruturas dinâmicas que podem sofrer alterações estruturais durante a sua adaptação. Uma vez que "a história permite que pessoas utilizem a ambigüidade como um recurso efetivo para o aprendizado a partir de discussões" (Wenger apud [JØR2001]) e é invariavelmente difícil entender, reutilizar ou aperfeiçoar um modelo processo de software tendo por base somente sua descrição formal. Serviços de gerência de configuração e registro de decisões de projeto constituem elementos úteis para armazenar as razões subliminares para eventos e modificações ocorridas em um modelo de processo. Dessa forma, devem estar disponíveis mecanismos que permitam aos usuários navegar no histórico de decisões para entender a evolução ocorrida por modificações e desvios durante a modelagem e/ou execução.

Descrições incompletas de processos têm como conseqüência o fato de que as reais instâncias a ocuparem papéis importantes nos modelos de processos somente serão definidas em tempo de execução (instanciação). Portanto, os PSEEs devem registrar ainda as alterações que ocorrem em tempo de modelagem e durante a execução dos processos (denominadas modificações dinâmicas). Além disso, deve ser possível propagar automaticamente as modificações realizadas nos *templates* para as instâncias associadas.

2.6.2 Requisito 3.2 - Composição Dinâmica de Elementos de Processos

Em virtude da dinâmica do desenvolvimento de software, é importante fornecer apoio à composição (semi[11]-)automática do modelo de processo em execução.

[11] Em conseqüência do fato de que componentes e processos recuperados usualmente necessitarem de adaptações não triviais para serem úteis em novos contextos.

Assim, para permitir a composição dinâmica de componentes, é necessário fornecer padrões para descrição de interfaces de componentes de processos, apoiando assim a definição de condições lógicas que guiem a recuperação e reconfiguração automática dos modelos evoluídos [AVR96a] [AVR96b].

2.7 Generalização e Avaliação de Processos Encerrados

Processos encerrados armazenam informação valiosa sobre o uso real de estratégias gerenciais adotadas pela organização durante o desenvolvimento de software. Por exemplo, as métricas de produto e processo coletadas durante a sua execução são elementos importantes que devem ser levados em consideração na definição de uma *baseline* e, por conseguinte, na instanciação de novos processos [NGU97]. Entretanto, a fim de propiciar a reutilização do modelo do processo encerrado, tal processo deve ser "destilado[12]", isto é, informações sobre os detalhes concretos devem ser removidas a fim de possibilitar a construção (semi-)automática de um *template* correspondente inicial. Desse modo, essa seção apresenta os requisitos úteis para apoiar a generalização de modelos de processos.

2.7.1 Requisito 4.1 - Generalização de Processos Encerrados

De acordo com Bogia e Jaccheri, citados em [JØR2001], um sistema deve incluir operações para descontextualizar[13] os processos encerrados, isto é, repor para os estados iniciais os valores do processo, removendo detalhes e substituindo instâncias por classes (tipos independentes de contexto).

2.7.2 Requisito 4.2 - Comparação de Modelos de Processos

Visto que a generalização de processos não é uma tarefa trivial, os usuários devem ser apoiados na tarefa de contrastar e comparar modelos particulares de forma que padrões similares e distintos sejam destacados, e que tal mecanismo ainda auxilie na combinação de diferentes modelos particulares em um *template* geral.

2.7.3 Requisito 4.3 - Conexão de Instâncias aos Processos Abstratos Generalizados

Com o objetivo de facilitar o aprendizado e entendimento de um *template* construído por generalização, os *templates* devem ser conectados à(s) instância(s) que serviram de inspiração. Essa informação pode ser de extrema valia na medida da efetividade e ocorrência de um específico processo numa organização de software. Além disso, tal procedimento pode facilitar o entendimento e treinamento de projetistas novatos a partir de exemplos.

2.8 Estado da arte tecnológico atual

Esta seção apresenta um resumo das soluções propostas na literatura para todos os requisitos descritos nesse capítulo. Nenhuma solução descrita na literatura - genérica

---

[12] Do original em [JØR01], *to distill*.

[13] Do original em [JØR01], *de-contextualize*

e universalmente aceita - atende totalmente os requisitos listados, mas as informações relacionadas permitem a comparação entre diferentes abordagens apresentadas na literatura. De fato, a inexistência de soluções adequadas para muitos dos requisitos levantados serve como motivação para a descrição do trabalho proposto aqui. Assim, as sub-seções a seguir discutem exemplos significativos de soluções existentes e perspectivas para o futuro.

2.8.1 Modelagem de Processos Visando a Reutilização

Do ponto de vista da **Modelagem de Processos Visando a Reutilização**, as soluções existentes são baseadas nas PMLs. Os **mecanismos de abstração** adotados são aqueles disponibilizados pelas linguagens de programação de propósito geral: assim, as unidades básicas para descrição de processos (objetos, regras ou funções, segundo os paradigmas predominantes) são relacionadas entre si através de operadores de composição, sincronização e especialização (esse último, para linguagens OO). Tanto a parametrização quanto a primitivação são formas atualmente tratadas através de estruturas OO que apóiam a ligação tardia de elementos concretos aos modelos de processos, enquanto é dada pouca atenção para a estratificação de modelos.

A **separação de detalhes em múltiplas perspectivas** é tópico de interesse em pesquisa desde a segunda metade da década de 1990, quando foi constatado que PMLs mono-paradigma possuíam limitações importantes [CON95]. A evolução recente nos paradigmas de desenvolvimento de software[14], que atualmente objetiva fornecer mecanismos de abstração adicionais, no sentido de especificar e manter separadamente aspectos inerentemente ortogonais dos elementos semânticos centrais do sistema em desenvolvimento [LOP97], é um campo promissor de idéias que podem ser aplicadas ao contexto de modelos de processos de software. É importante observar, entretanto, que o projeto da PML deve evitar a sobreposição de informações como ocorre com a linguagem APEL, onde as diferentes notações existentes fornecem perspectivas que interferem umas nas outras, podendo confundir o usuário pelos efeitos colaterais causados.

Quando à **semântica**, as PMLs variam das totalmente informais (estruturando atributos textuais para armazenar descrições narrativas) às que possuem semântica formalmente definida (e.g., linguagens baseadas em Redes de Petri, LOTOS ou Gramática de Grafos), passando por aquelas cuja semântica é definida por tradutores para linguagens de programação de propósito geral. Da mesma forma, diversos trabalhos têm se preocupado com a especificação formal de mecanismos e funções críticas para os sistemas em questão.

A **simplicidade** das PMLs é um item de difícil mensuração devido ao seu caráter subjetivo estando, tal item intimamente relacionado aos outros requisitos definidos. Assim, em geral, o projeto de PMLs modernas leva em consideração aspectos relacionados à definição de programas concisos, que combinam diferentes paradigmas e que forneçam um conjunto suficiente de construtores sintáticos básicos com semântica bem definida e não sobrepostos.

Muitas linguagens fornecem **representação gráfica**, mas o principal problema em aberto consiste na visualização adequada para os muitos elementos inter-

[14] Em especial, os paradigmas envolvidos em *separation of concerns* e programação orientada a aspectos [LOP97].

relacionados utilizados na construção dos modelos. Além disso, a falta de uma notação homogênea e universalmente aceita implica no investimento de esforços de uniformização em torno de notações bem sucedidas para modelagem de software (por exemplo, UML).

2.8.2 Recuperação e Adaptação de Processos

A **estruturação de bases de processos** é um tópico investigado pelo projeto *Handbook of Organizational Processes*, desenvolvido no contexto do *Center of Coordination Science* do *Massachusetts Institute of Technology*. Tal projeto, segundo Malone [MAL99], objetiva colecionar e organizar exemplos em um catálogo unificado, constituído pela forma como diferentes organizações realizam processos similares. Esse catálogo tem o objetivo de compartilhar idéias acerca de melhores práticas organizacionais no contexto amplo de processos de negócio. Embora não tenha sido desenvolvida especificamente para apoiar a reutilização de processos de software, essa abordagem tem se demonstrado eficiente para apoiar a classificação de processos de negócio genéricos a partir da análise de características organizacionais, organizando suas similaridades e diferenças em estruturas de especialização OO.

Quanto ao **registro da aplicabilidade**, os sistemas normalmente associam métricas à *templates* e suas implementações. O sistema CABS (Case-Based Specification System) [FUN99] destaca-se por permitir a realização de buscas em bases de processos tendo a capacidade de computar a similaridade parcial entre o elemento solicitado pelo usuário e aqueles elementos armazenados em uma base de processos. A identificação de processos e componentes similares é feita levando-se em consideração os artefatos de entrada (consumidos) e saída (produzidos) pelas atividades de um processo e a hierarquia de tipos de artefatos associada.

A **extração de visões** é uma característica que vem despertando a atenção da comunidade desde a segunda metade da década de 1990. Tais aplicações baseiam-se na definição de nodos (formados por componentes e conexões) recuperáveis a partir de critérios definidos pelo usuário. A ferramenta OPSIS, descrita por Avrilionis em [AVR96a] e [AVR96b], é um exemplo de mecanismo para extração de sub-modelos de Redes de Petri a partir de aspectos específicos do processo (por exemplo, um cargo, um produto ou uma atividade). Assim, em OPSIS, uma visão baseada em um cargo específico é um sub-modelo de processo que incorpora todas as atividades e produtos realizados e manipulados por um determinado cargo. De forma similar, uma visão baseada em um produto específico contém toda informação do modelo de processo relacionado com esse produto (por exemplo, ações que alteram o seu estado, cargos que estão envolvidos com tal produto).

A ferramenta *APSEE*-Monitor [SOU2002] também apóia a extração de visões, entretanto o seu objetivo consiste em fornecer visualizações especializadas que auxiliam no entendimento de um processo complexo multidimensional.

As PMLs orientadas a objetos tais como E^3 [JAC98] e SPELL [NGU97] introduziram a capacidade de **tipos extensíveis e definidos pelo usuário**, permitindo que as hierarquias de classes básicas sejam estendidas em virtude de necessidades organizacionais específicas. Entretanto, do ponto de vista de facilitar a reutilização e a adaptação de processos, tais abordagens não adotam mecanismos para explicitar tipos básicos (independentes de contexto organizacional ou tecnológico) daqueles específicos tratados por um contexto específico.

Finalmente, a **adaptação de processos** constitui um ponto de pesquisa em aberto atualmente. A solução apresentada por Münch em [MÜN97] propõe a descrição formal do contexto destino e a seleção de um conjunto de regras (a partir de um elenco disponível) para mapear as estruturas abstratas de origem em modelos de processos executáveis. Soluções baseadas em políticas de instanciação, tal como descritas no modelo *APSEE* [LIM2001a], adotam uma postura diferente por fornecer solução automatizada para escolha de agentes e recursos a partir de critérios genéricos definidos *a priori*. Idealmente, esse tópico pode se valer de soluções baseadas em conhecimento, com o objetivo de fornecer sugestões de adaptações em função de soluções semelhantes adotadas em contextos similares, levando em consideração as características ambientais e organizacionais correntes.

2.8.3 Execução de Processos

A execução de processos é um tópico ativo que conta com diferentes abordagens e suporte computacional diversos, conforme descrito pelo *survey* de Lima Reis em [LIM2000]. Do ponto de vista de reutilização, a **evolução de processos** em execução é um aspecto crítico, envolvendo o registro das modificações dinâmicas. Assim, sistemas de gerência de configuração vem sendo integrados à PSEEs com o objetivo de apoiar o registro das decisões de projeto que motivaram a evolução dos processos.

A abordagem OPSIS, citada anteriormente, também permite a seleção e **composição dinâmica de visões**, isto é, visões podem ser escolhidas como resultado de estratégias determinadas de acordo com o estado atual do processo. A correção do modelo integrado é garantida por um protocolo de composição baseado na semântica de sincronização de Redes de Petri, isto é, a partir da passagem de controle entre portas de entrada/saída e de transições de sincronização. Assim, ligações especiais são automaticamente incluídas para conectar os caminhos incompletos entre fragmentos da Rede gerada.

A herança de Redes de Petri é defendida por van der Aalst [VAN2000] como mecanismo para apoiar a evolução de processos a partir de modificações realizadas nos *templates* de origem associados. Assim, as modificações permitidas são restritas por regras de transformação que preservam a herança, garantindo que o novo processo não incorra em problemas como duplicação de trabalho, omissão de tarefas, *deadlocks* e *livelocks*.

2.8.4 Generalização e Avaliação de Processos Encerrados.

O tema generalização e avaliação de processos encerrados também constitui um ponto de discussão em aberto, com número limitado de soluções automatizadas, em face da recente atenção que a comunidade vem dando ao tema Reutilização no contexto de Processos de Software.

Embora Jørgensen [JØR2001] e Perry [PER96][PER97] sugerirem que a generalização de processos, isto é, a extração de visões abstratas a partir de processos encerrados seja possível, na prática isso não tem se comprovado em virtude da complexidade e diversidade de informações dependentes de contexto nos modelos encerrados.

3 Visão Geral do Modelo APSEE-Reuse

Esse capítulo apresenta uma visão geral do modelo *APSEE*-Reuse, proposto nessa Tese como uma contribuição ao campo de pesquisa em reutilização de processos de software. Esse modelo esteve em desenvolvimento no período de 1999 a 2002, e incorpora idéias que foram especificadas, integradas e experimentadas no contexto do PPGC-UFRGS.

Esse capítulo, inicialmente, detalha os objetivos específicos do modelo proposto. Tais objetivos são usados para motivar as características e inspirações para o desenvolvimento da solução proposta, enfatizando os aspectos estruturais de alto nível do modelo. A partir dos objetivos vislumbrados, são descritas as características principais do modelo proposto, por meio de uma visão geral dos seus principais conceitos.

3.1 Objetivos específicos

Levando em consideração os requisitos levantados no capítulo 2, o estado da arte tecnológico atual e as motivações pessoais do autor, são objetivos específicos desse trabalho:

- Apresentar construtores sintáticos e mecanismos que auxiliem na solução de requisitos relacionados com a categoria Modelagem de Processos Visando à Reutilização (previamente descrita na seção 2.4 desse volume). Além disso, é objetivo atender a demais requisitos específicos de outras categorias que sejam intimamente relacionados com a modelagem, incluindo: a Definição Extensível de Tipos da PML (R2.4 - seção 2.5.4), a Generalização de Processos Encerrados (R4.2 - seção 2.7.2) e a Conexão de Instâncias aos Processos Abstratos Generalizados (R4.3 - seção 2.7.3);
- Definir uma linguagem especializada na modelagem de processos abstratos (doravante denominados *templates*[15] de processos de software). Em razão das inúmeras vantagens proporcionadas pela execução automatizada para a gerência de processos de software, a linguagem proposta deve estar intimamente relacionada com uma linguagem executável, isto é, que permita a adaptação de processos abstratos para instanciados, com o objetivo de facilitar a integração da reutilização como uma das atividades primordiais do ciclo de vida para processos de software;

[15] Embora muitos trabalhos no assunto utilizem o termo *pattern* como uma denominação geral para processos abstratos, este texto propositalmente utiliza o termo *template* para distinguir explicitamente este modelo das propostas derivadas das idéias de Gamma et al. (em [GAM94]).

- Investigar a viabilidade de definir as soluções propostas a partir da extensão e integração de tecnologias que já demonstraram sua utilidade em contextos similares;
- Especificar o modelo proposto de forma precisa e em alto nível de abstração, investigando, em específico, a viabilidade de usar de métodos de especificação formal nas primeiras etapas do processo de desenvolvimento do software. Ainda, em virtude da necessidade de conduzir experimentos práticos com o modelo proposto, é importante que os formalismos escolhidos propiciem a implementação por derivação de protótipos em linguagens de programação existentes.

3.2 A infraestrutura APSEE para Automação de Processos de Software

Essa seção apresenta, resumidamente, a evolução histórica do ambiente *APSEE* como um meta-modelo unificado para automação da gerência de processos de software, o qual serviu de base para o modelo aqui proposto.

Durante as primeiras etapas do desenvolvimento do trabalho proposto por essa Tese, foi investigada a possibilidade de se adotar algum PSEE disponível na literatura, incluindo EPOS [NGU97], E3 [JAC98], OPSIS [AVR96a] [AVR96b] e Endeavors [BOL99]. Afinal, à primeira vista, a idéia de se valer de um ambiente já implementado e discutido na literatura é atraente, pois poderia facilitar o desenvolvimento dos mecanismos aqui propostos em uma solução já existente. Entretanto, a escolha por um ambiente em desenvolvimento acabou sendo justificada por vários motivos, a saber:

- A quase totalidade dos PSEEs disponíveis exige plataformas de hardware e software especializadas, as quais não são disponíveis no Instituto de Informática da UFRGS em razão de custos ou opção tecnológica alternativa;
- Embora a maioria dos PSEEs permita a integração de diferentes ferramentas (através da definição de uma interface pública para componentes escritos pelo usuário), muitos dos PSEEs - do ponto de vista do suporte à modelagem e execução de processos - são sistemas monolíticos, nos quais é muito difícil propor mudanças na infraestrutura básica de modelagem de processos;
- Muitos dos PSEEs suportam somente uma PML, isto é, não permitem estender ou substituir a PML existente;
- A documentação disponível acerca das implementações existentes é pobre, dificultando sobremaneira a extensão e até mesmo o uso prático dos ambientes.

Desse modo, optou-se por propor uma solução que fosse integrada a um ambiente que, quando do início desse trabalho (em 1998), estava sendo desenvolvido pelo grupo PROSOFT[16], sob orientação do Prof. Dr. Daltro José Nunes. O ambiente em

[16] PROSOFT é um ambiente integrado de desenvolvimento de software resultante de um projeto de pesquisa homônimo estabelecido em cooperação do PPGC-UFRGS e a *Fakultät Informatik* da *Universität*

questão - denominado *APSEE* - é uma infraestrutura de software que evoluiu a partir de um mecanismo de gerenciamento de processos de software proposto na dissertação de mestrado de Lima Reis para o PROSOFT em [LIM98], no PPGC-UFRGS. No modelo Gerenciador de Processos (GP) proposto àquela altura, a preocupação primordial era a de definir um meta-modelo para execução automatizada de processos de software descritos em uma linguagem primitiva.

Nos últimos anos, o mecanismo GP foi estendido, com a introdução de novos recursos para apoiar de forma mais adequada as características evolucionárias e dinâmicas do processo de software, dando origem ao modelo que se convencionou chamar *APSEE* [LIM2001b]. Assim, em 1999, uma versão aperfeiçoada do meta-modelo GP foi utilizada por Silva no desenvolvimento de um simulador de processos de software baseado em conhecimento ([SIL99] e [SIL2001]). A partir daí, o modelo evoluiu para apoiar de forma mais consistente outras etapas do meta-processo de software, envolvendo a gerência de recursos de apoio [LIM2001a] [LIM2002b], a definição de políticas de processos ([REI2001c] e [REI2002b]), a visualização/monitoração [SOU2002] e a execução flexível dos modelos de processos construídos [LIM2002d]. Em 2002 surgiu uma nova implementação distribuída para o modelo, servindo de base para as implementações atuais de ferramentas para o mesmo. Atualmente, *APSEE* é um projeto de pesquisa que envolve estudantes e pesquisadores de diferentes instituições no desenvolvimento de ferramentas que atuam sobre o mesmo meta-modelo uniforme.

O diagrama da figura 3.1 apresenta os principais componentes do sistema *APSEE* atual, organizados em três camadas (interação com usuário, mecanismos para gerência de processos, e repositório com os meta-tipos fornecidos), as quais são descritas nas sub-seções a seguir.

Stuttgart (Alemanha). Nos capítulos a seguir serão apresentados detalhes adicionais sobre o ambiente e o seu paradigma para desenvolvimento de software.

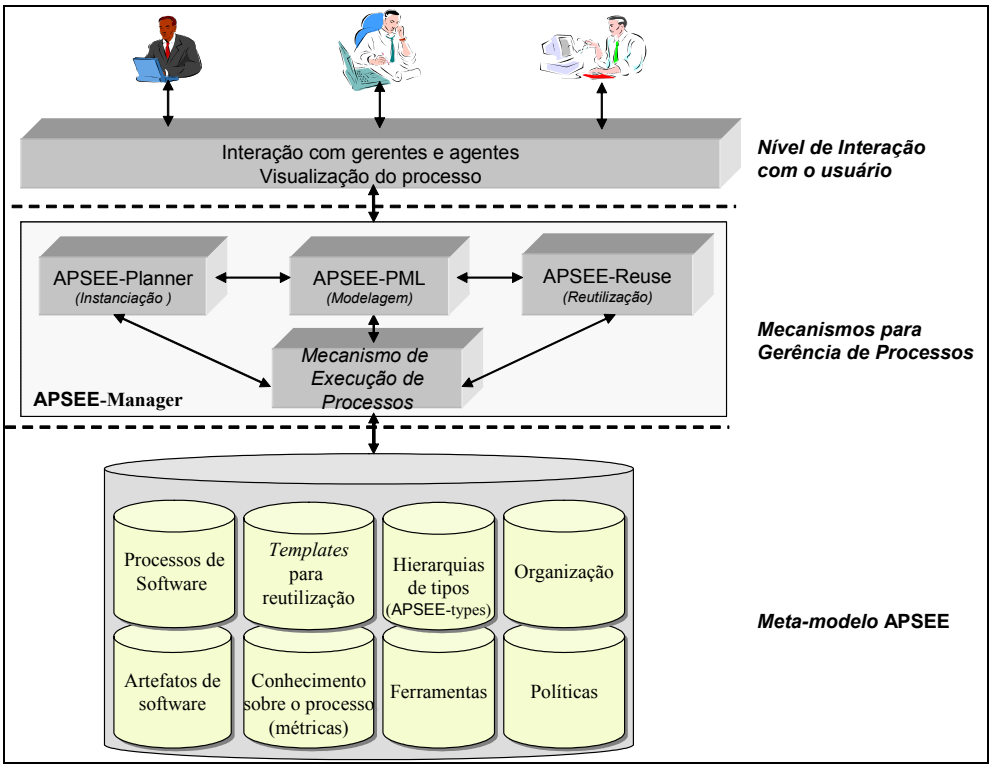

FIGURA 3.1 - Visão geral dos componentes principais do modelo APSEE

3.2.1 Interação com o usuário

A interação com o usuário é fornecida através de uma interface gráfica, sendo a mesma fornecida pelo ambiente PROSOFT-Java, integrando serviços fornecidos pela linguagem de programação hospedeira do ambiente. A figura 3.2 apresenta um exemplo, com duas visões para a execução de um processo, por meio da console do gerente (ao fundo da figura) e da agenda do desenvolvedor (à frente). Atualmente, diferentes ferramentas - entre as quais, o *APSEE*-Monitor [SOU2002] - vêm sendo propostas com o objetivo de fornecer visões especializadas para aspectos estáticos e dinâmicos dos modelos de processo de software.

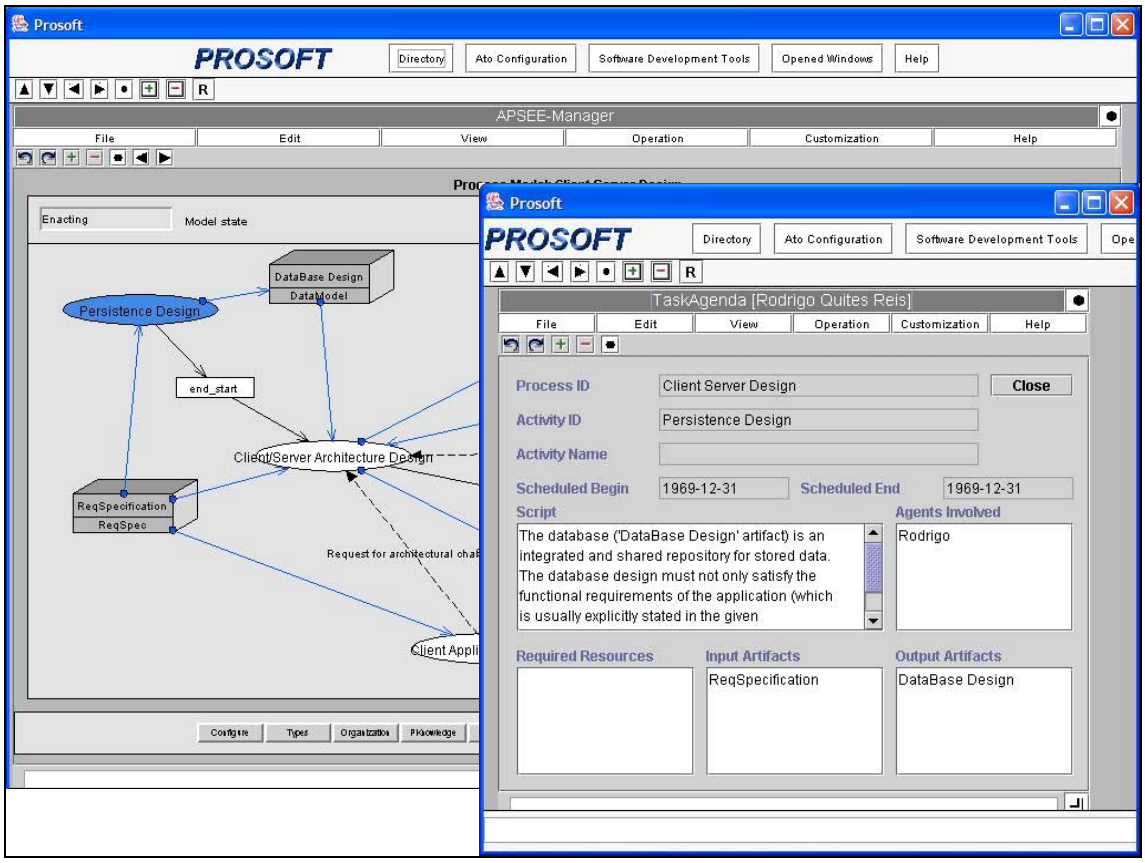

FIGURA 3.2 - Execução distribuída de processos de software no ambiente APSEE: gerência do desenvolvimento (ao fundo) e agenda do desenvolvedor (à frente)

3.2.2 Mecanismos para Gerência de Processos

A camada **mecanismos para gerência de processos** descreve um conjunto de serviços que constituem o componente denominado *APSEE*-Manager existente na implementação atual do sistema.

O mecanismo para execução de processos constitui a base que interpreta os modelos de processos (descritos com o editor *APSEE*-PML), e exerce um papel central na organização dos mecanismos existentes. Desse modo, o mecanismo de execução é responsável por fornecer dados sobre a dinâmica da execução de processos que são úteis para o mecanismo de instanciação *APSEE*-Planner. Além disso, o mecanismo de execução fornece informação sobre processos abstratos, instanciados e executados para o componente responsável pela descrição de processos reutilizáveis.

O componente *APSEE*-Planner é responsável por fornecer assistência automática para instanciação de recursos e agentes durante a execução de processos de software através de políticas programáveis pelo usuário (sendo descrito por Lima Reis em [LIM2002b] e [LIM2002c]).

Finalmente, o editor da linguagem *APSEE*-PML auxilia a descrição de processos e *templates*, sendo também útil para habilitar políticas em modelos de processos de software.

3.2.3 O Meta-modelo APSEE

O meta-modelo *APSEE*, camada mais inferior do diagrama da figura 3.1, apresenta alguns dos tipos de dados mais importantes da infraestrutura proposta. Assim, estão dispostos elementos inter-relacionados, a saber:

- '*Templates*' e 'Modelos de Processos de Software' (instâncias descritas com editor *APSEE*-PML);
- As 'Hierarquia de tipos do ambiente' (*APSEE*-Types), conforme descrito posteriormente na seção 3.4;
- A 'Organização', representando as pessoas envolvidas, incluindo seus cargos, habilidades e grupos, e os recursos de apoio utilizados;
- Os 'Artefatos de Software', descrevendo os objetos de software utilizados e produzidos pelos processos;
- O item 'Conhecimento sobre o processo' (descrevendo métricas resultantes da execução dos processos definidos);
- As 'Ferramentas' utilizadas no desenvolvimento de software;
- As 'Políticas', descrevendo elementos reutilizáveis usados na composição de processos e *templates*.

Muitos dos componentes descritos acima foram incorporados em função do trabalho aqui descrito, sendo descritos em maior detalhe nas seções a seguir.

3.3 Características gerais do modelo APSEE-Reuse

Como descrito nas seções anteriores, o modelo *APSEE*-Reuse está centrado em apoiar a modelagem de processos, sendo decisivamente influenciado pela necessidade de fornecer apoio à separação de detalhes na modelagem de processos. Assim, o meta-modelo proposto combina uma série de elementos sintáticos que auxiliam na descrição de processos reutilizáveis, elementos esses descritos nessa seção. Como a implementação das ferramentas envolveu a interação de pesquisadores do Brasil e da Alemanha, as classes e os atributos descritos nesse trabalho são rotulados com os seus nomes originais em inglês.

O projeto do modelo aqui apresentado evoluiu a partir do modelo original PROSOFT-GP, influenciando e sofrendo influência dos diferentes trabalhos desenvolvidos no contexto do projeto *APSEE* (a saber: [LIM2002d], [SIL2001] e [SOU2002]), em virtude do inter-relacionamento entre os componentes e a proposital homogeneidade de conceitos nas diferentes fases do ciclo de vida de processos. Assim, essa seção é responsável por descrever as principais características dos componentes propostos pelo autor no sentido de apoiar a modelagem de processos reutilizáveis.

O modelo proposto incorpora uma série de elementos que constituem uma camada adicional que atua, especificamente, na definição de processos abstratos no modelo *APSEE*. A unidade principal para a modelagem de processos é denominada ***ProcessTemplate*** (ou simplesmente *Template*), definindo um processo abstrato como um conjunto parcialmente ordenado de atividades (instâncias do tipo *Activity*) orientadas ao desenvolvimento de software. O construtor *ProcessTemplate* é proposto para apoiar a descrição de processos abstratos e genéricos, estando relacionado a

elementos abstratos que são independentes de organizações ou tecnologias específicas (i.e., meta-tipos).

Conceitualmente, *template* é muito similar ao *framework* do paradigma de objetos, de forma que este último tem o objetivo de fornecer uma abstração genérica para um domínio de aplicações. Tal como no desenvolvimento de *frameworks*, um *template* de processo de software "deve ter capacidade de adaptar-se a um conjunto de aplicações diferentes" [SIL2000]. Ainda segundo Silva [SIL2000], "para construir um *framework*, é fundamental que se disponha de modelagens de um conjunto significativo de aplicações do domínio", o que ocorre de forma análoga com relação à modelagem de *templates*.

Segundo a classificação de Estublier [EST96], a PML fornecida pelo modelo de *Template* proposto segue uma abordagem orientada a atividades, com **atividades representadas por objetos**. Assim, as atividades são tipadas, isto é, organizadas através de uma hierarquia de generalização-especialização. Portanto, o modelo aqui proposto se aproxima das abordagens defendidas pelas linguagens APEL [DAM98] e JIL [SUT97], representando para o usuário processos e *templates* como redes parcialmente ordenadas de atividades, escondendo do usuário a complexa definição interna dos tipos de dados relacionados.

O modelo proposto foi projetado para apoiar a construção de processos genéricos a partir da composição de elementos independentes definidos em diferentes dimensões. Tanto as abordagens *top-down* quanto *bottom-up* são habilitadas na construção de *templates*, visto que elementos podem ser adicionados e refinados em tempo de modelagem. Para tanto, a modularidade é apoiada por quatro **mecanismos de abstração** principais, fornecidos aos usuários:

- O construtor de generalização-especialização (para definir hierarquias de tipos de componentes de processos, conforme detalhado na seção 3.4 a seguir);
- O construtor de composição (todo-parte) que está disponível para a descrição de atividades[17], assim como entre grupos e agentes;
- A associação de diferentes componentes, entre os quais: agentes e cargos, cargos e habilidades, recursos requeridos por uma atividade normal, e ferramentas requeridas pelas atividades.
- A habilitação de Políticas às instâncias, em conformidade com interface definida pelas Políticas selecionadas (descrita na seção 3.5.6, sobre Políticas de processos, a seguir).

Representações gráficas são propostas para os *templates* e demais elementos da linguagem proposta, descrevendo notações especializadas para cada detalhe sendo modelado.

A figura 3.3 apresenta uma visão geral dos pacotes de classes envolvidos na descrição do modelo de *Templates* usando a notação de pacotes fornecida pela UML [BOO98]: esse diagrama, assim como todos os outros diagramas de classe apresentados nesse capítulo são simplificações[18] do modelo APSEE-Reuse, obtidos a partir de

[17] Uma atividade decomposta (fragmento) define recursivamente um novo processo de software.

[18] Omitindo algumas classes e atributos existentes no modelo completo descrito no capítulo 4.

engenharia reversa manual das classes PROSOFT usadas no projeto do modelo, tendo como objetivo apresentar de forma compacta os tipos de dados principais envolvidos, facilitando o entendimento geral da proposta.

Como observado pela figura, *ProcessTemplates* é um pacote complexo, composto por seis pacotes interdependentes, rotulados como *SoftwareDimension*, *PeopleDimension*, *ProcessDimension*, *PolicyDimension*, *ResourceDimension* e *ToolDimension*. Todos os pacotes de *ProcessTemplates* dependem da hierarquia de tipos do ambiente *APSEE*.

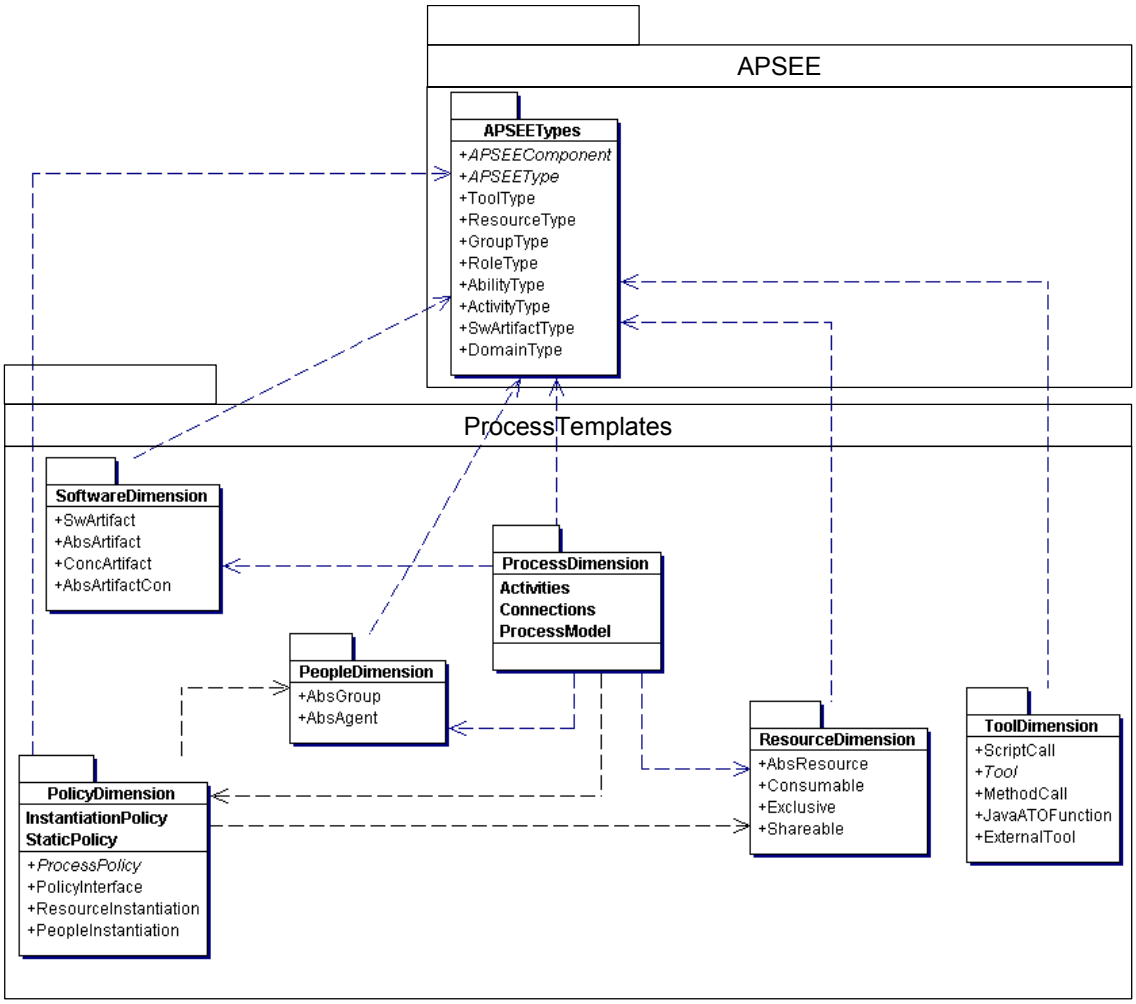

FIGURA 3.3 - Diagrama de Dependência de Pacotes de Tipos para *ProcessTemplates*

As seções a seguir discutem alguns dos aspectos principais do modelo proposto, os quais são mais precisamente descritos na especificação de projeto do modelo no capítulo 4.

3.4 A Hierarquia de Tipos

Um dos pilares básicos do modelo proposto relaciona-se à existência de hierarquias generalização-especificação para os tipos dos diferentes componentes da linguagem em questão, sendo que esse aspecto em específico foi inspirado pela hierarquia de tipos fornecida pelo ambiente E^3 [JAC98]. Assim, hierarquias de tipos estão disponíveis para os principais componentes do modelo, a saber: *Resource* (recurso), *Agent* (agente), *Ability* (habilidade), *Role* (cargo), *Group* (grupo), *Tool*

(ferramenta), *Policy* (política), *SwArtifact* (artefato de software) e *Application Domain* (domínio de aplicação).

A figura 3.4 fornece uma representação simplificada para o pacote *APSEETypes*, o qual engloba o relacionamento entre as classes *APSEEComponent*, *APSEEType* e as hierarquias de tipos (especializações de *APSEEType*). Assim, um componente de processo (i.e., um objeto de alguma especialização da classe abstrata *APSEEComponent*) está associado à exatamente um tipo. Os tipos são organizados em árvores, aonde um nodo pode possuir zero (para nodo-raiz) ou um ancestral (para todos os demais).

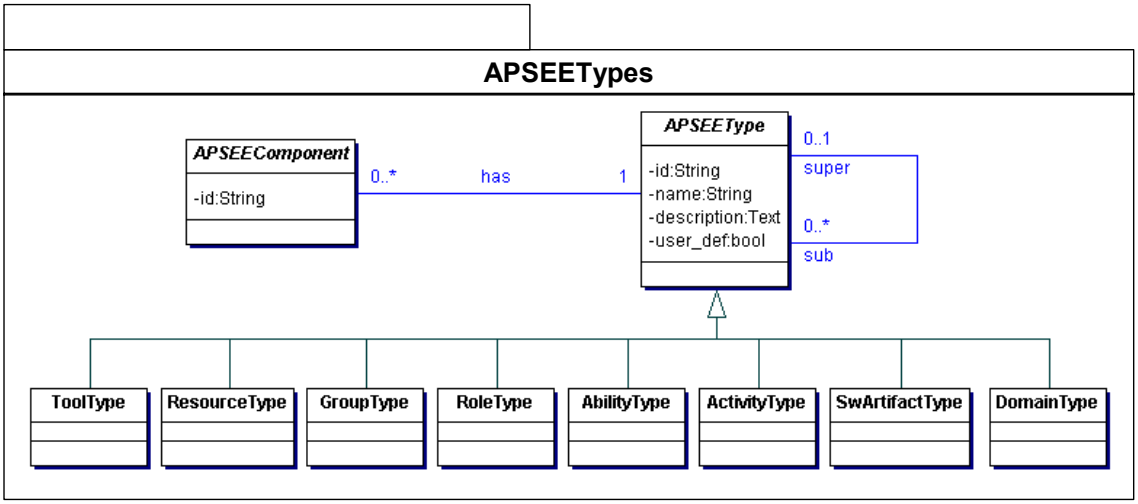

FIGURA 3.4 - Hierarquia de tipos para componentes de processos

Esta organização em hierarquias exerce um papel fundamental na **estratificação** e **definição de soluções genéricas**, visto que *templates* podem ser definidos com associação a elementos genéricos, permitindo a definição tardia para estruturas de processos. Tal fato adia, por conseqüência, a definição de estruturas concretas até o momento de execução do processo.

3.5 As dimensões propostas pelo modelo APSEE-Reuse

As dimensões definidas para separação de detalhes foram projetadas com o objetivo de encapsular os elementos usados na modelagem de processos. De fato, o modelo *APSEE*-Reuse estende a proposta de Perry [PER96] - descrita preliminarmente na seção 2.4.2 - sob os seguintes aspectos:

- As dimensões são definidas de uma maneira mais rigorosa, sendo diretamente relacionadas às classes e atributos do meta-modelo adotado, enquanto que a proposta original de Perry é descrita através de uma narrativa em inglês (em [PER96]);
- A dimensão originalmente rotulada como ambiente tecnológico (*environment* em [PER96]) foi mapeada em duas dimensões distintas: **Recursos** (*Resources*), representando os recursos físicos e financeiros de apoio ao desenvolvimento de software, e **Ferramentas** (*Tools*), descrevendo as ferramentas de software usadas nas diferentes etapas do processo de software;

- As dimensões **Software** e **Políticas** (*Policies*) foram adicionadas em relação à proposta de Perry, constituindo uma importante contribuição do modelo proposto.

As seções a seguir descrevem em detalhe as dimensões propostas.

3.5.1 A Dimensão 'Processo'

A dimensão Processo descreve os elementos fundamentais que atuam na composição de *templates* de processos de software. A figura 3.5 apresenta a decomposição e interdependência de pacotes para os tipos de dados definidos. Desse modo, a definição da dimensão Processo é dividida em Modelo de Processo (*Process Model*), Atividades (*Activities*) e Conexões (*Connections*).

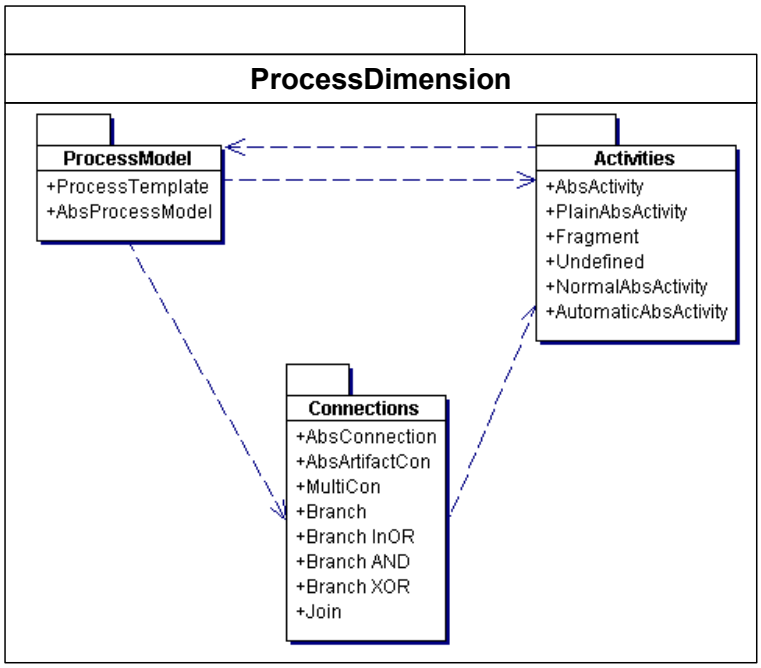

FIGURA 3.5 - O pacote *ProcessDimension*

O pacote Modelo de Processo (*ProcessDimension.ProcessModel*) - mostrado na figura 3.6 - estabelece que um *Template* é aplicável a um tipo de domínio de problema (*APSEETypes.DomainType*), e define um modelo de processo abstrato de software (*AbsProcessModel*). Um *template* está associado a um tipo (que, na realidade é um tipo de atividade). O tipo *AbsProcessModel*, por sua vez, define os requisitos para o processo (através do atributo textual *requirements*) e um *flag* booleano que identifica se as políticas estáticas habilitadas estão satisfeitas (atributo *staticPolicyOK*). Além disso, um modelo de processo é definido por um conjunto de atividades (classe *AbsActivity* do pacote *Activities*) e um conjunto de conexões (*Connections.AbsConnection*) - estas últimas definindo a ordenação de atividades do processo.

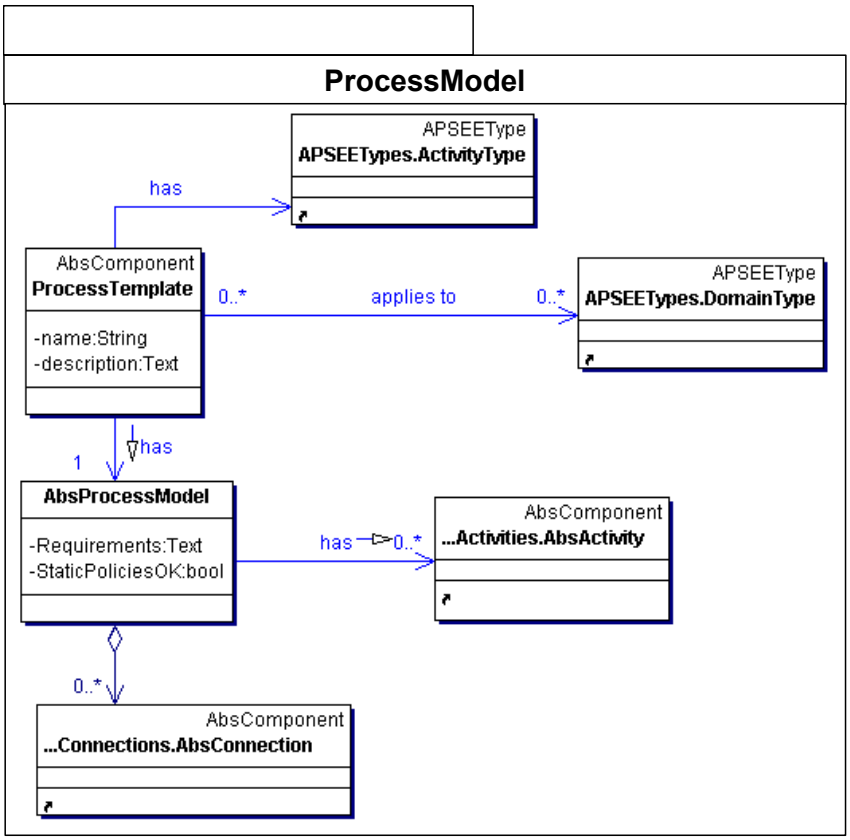

FIGURA 3.6 - O pacote *ProcessDimension.ProcessModel*

O pacote *ProcessDimension.Activities* - apresentado na figura 3.7 - descreve os diferentes tipos de atividades usadas na composição de um modelo de processo. As atividades descrevem as ações realizadas no decorrer do desenvolvimento de software e estão disponíveis em quatro tipos:

- **Atividades normais** (tipo *NormalAbsActivity*) são nodos-folha da árvore de composição de um *template* e demandam a participação de agentes (pessoas) para a sua realização, sendo alocadas às suas respectivas agendas em tempo de execução. As atividades normais são descritas através de um campo textual *Script*, que contém as instruções fornecidas aos agentes durante a execução dos processos;
- **Atividades automáticas** (tipo *AutomaticAbsActivity*) descrevem chamadas às operações disponibilizadas pelo ambiente operacional do programa (incluindo métodos de classes, funções de ferramentas, e chamadas aos serviços de sistema operacional) que não requerem intervenção de usuário para serem executadas. Assim como as atividades normais, as atividades automáticas constituem atividades-folha (i.e., o tipo *AutomaticAbsActivity* também é uma especialização de *PlainAbsActivity*);
- **Atividades decompostas (fragmentos ou sub-processos)**, que descrevem recursivamente um novo modelo de processo ou um outro *template*. No pacote *ProcessDimension.Activities*, as atividades decompostas são descritas pela classe fragmento, a qual está associada a uma alternativa entre a definição recursiva de um novo modelo de processo (i.e., um novo objeto de *AbsProcessModel*) ou um novo *Template*.

- As atividades ainda podem ser **indefinidas** (***undefined***), ou seja, deixando o seu usuário livre para definir o seu detalhamento em tempo de modelagem.

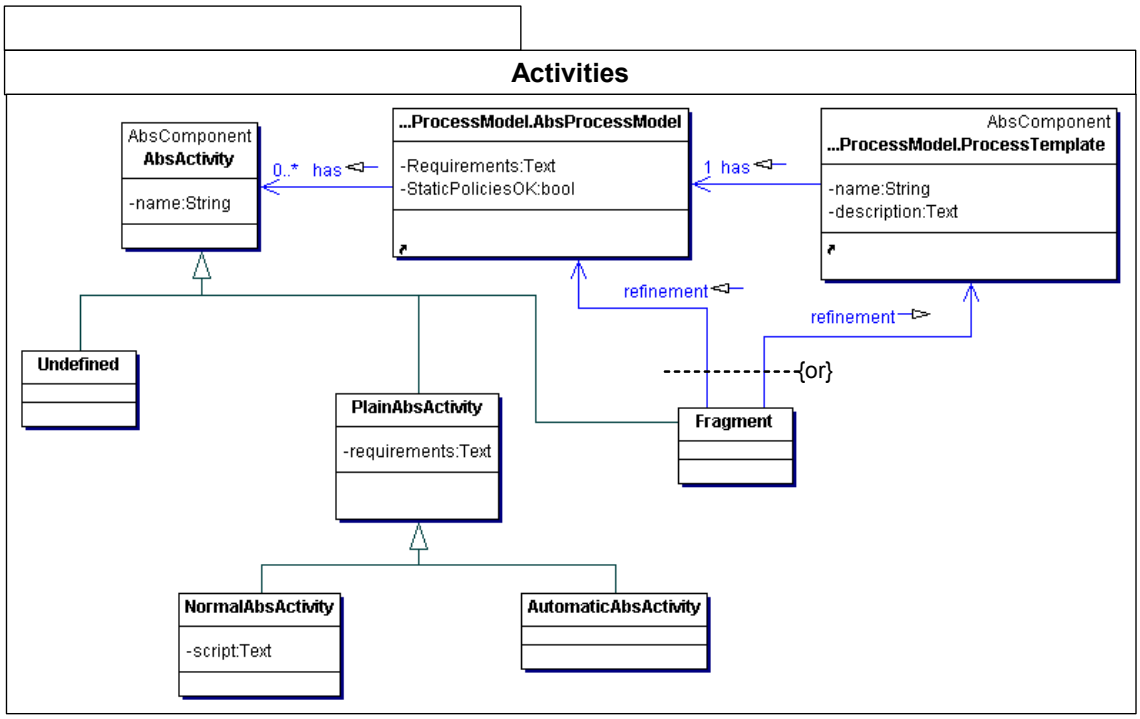

FIGURA 3.7 - O pacote *ProcessDimension.Activities*

Templates descrevem a dependência temporal para atividades através de conexões. As conexões são associadas a tipos de dependências temporais, com semântica de execução definida por [LIM2002a], estando descritas abaixo:

- Dependência ***end-start***: o término da origem habilita o início da execução do destino;
- Dependência ***end-end***: o término da origem habilita o término do destino;
- Dependência ***start-start***: o início da execução da origem habilita o início da execução do destino.
- A dependência **indefinida** (graficamente representada por um ponto de interrogação) permite que o projetista adie a definição exata da dependência para o momento de execução de um processo.

Conexões entre duas atividades estão disponíveis em diferentes tipos, conforme descrito a seguir:

- **Conexão Simples:** são conexões entre duas atividades (uma origem e um destino), associadas a um tipo de dependência temporal;
- **Conexão de *Feedback*:** prescrevem a possibilidade de retorno condicional para um ponto do processo já previamente executado;

As conexões múltiplas envolvem múltiplas origens e destinos (de forma que origem e destino são atividades ou conexões múltiplas). As conexões múltiplas estão disponíveis em dois tipos:

- **Conexão Múltipla do tipo *Branch*:** envolve um único ponto de origem e vários destinos. No caso de conexões *Branch* do tipo *XOR* ou *InOR*, uma condição é definida para habilitar a execução de cada destino;
- **Conexão Múltipla do tipo *Join*:** envolve múltiplos pontos de origem e um único destino. Da mesma forma que nas conexões *Branch*, condições lógicas são requeridas para *Join XOR* ou *InOR*.

Finalmente, a conexão de artefato (*SoftwareDimension.AbsArtifactCon*) está associada ao tipo atividade e *MultiCon*: as conexões de artefato descrevem artefatos produzidos e consumidos por atividades, ou ainda fornecidos como entrada para conexões múltiplas.

A figura 3.8 descreve o pacote *ProcessDimension.Connections*, com todos os tipos definidos. Os seguintes elementos merecem destaque:

- A condição associada às conexões múltiplas *Branch InOR* e *XOR* e de *feedback* é definida por um tipo associado às políticas estáticas (classe *Condition*, relacionada, por sua vez, ao tipo *PolicyDimension.StaticPolicy.PolExpression*);
- O tipo *Branch* - por permitir somente uma origem - restringe a composição do nodo *from* a uma atividade ou uma conexão múltipla. De forma análoga, o tipo *Join* impede a ocorrência simultânea das associações *to activity* e *to connection*.

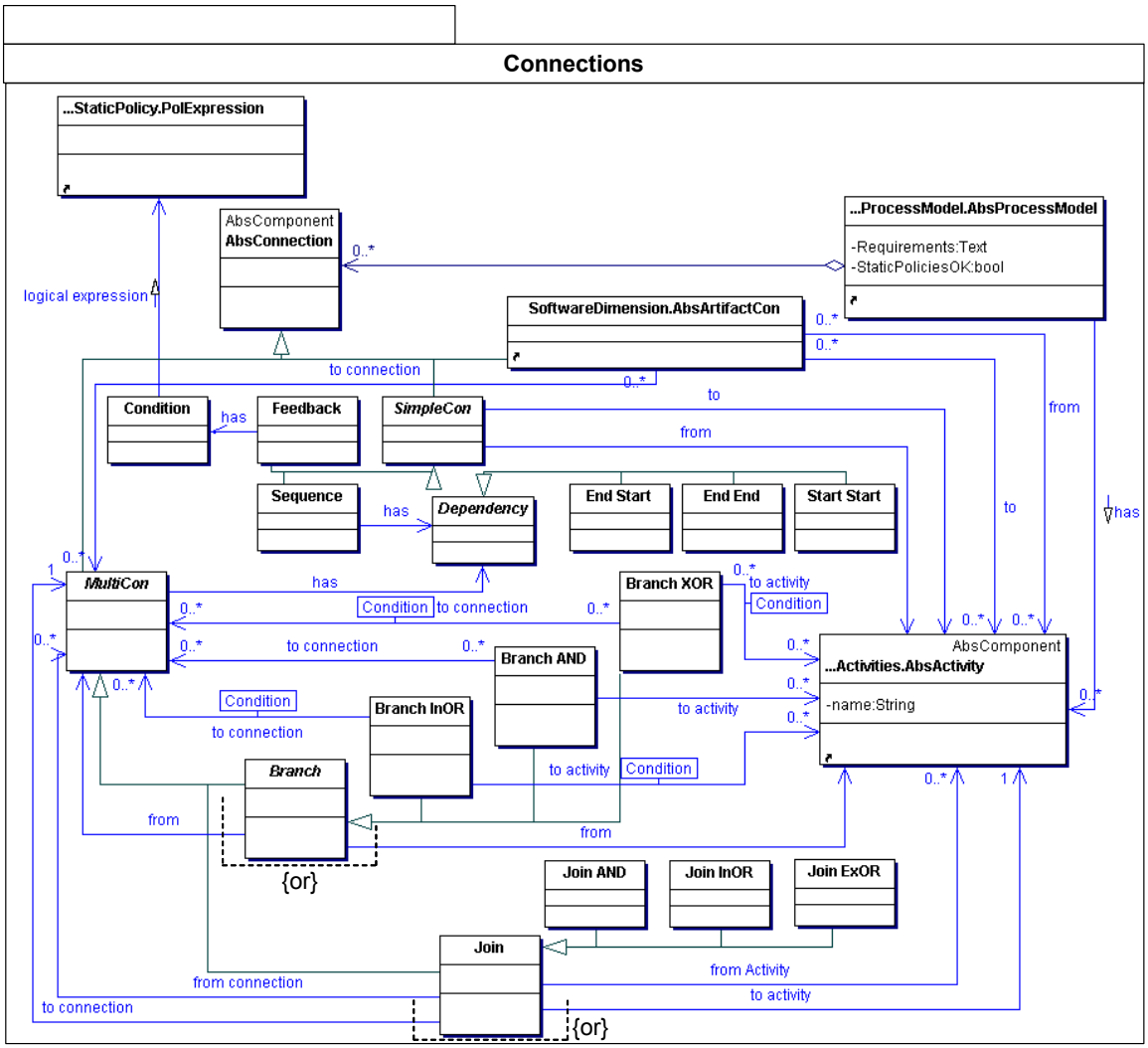

FIGURA 3.8 - O pacote *ProcessDimension.Connections*

A figura 3.9 apresenta os ícones usados para representação gráfica de cada elemento de um *template*, segundo a dimensão Processo. A notação fornece representações gráficas distintas para cada um dos tipos de atividades disponíveis (a saber, fragmento, atividade folha-normal e atividade folha-automática), conforme ilustrado na parte superior da figura. Para todos os tipos de atividades,

As conexões simples (i.e., aquelas que possuem exatamente uma origem e um destino) estão representadas na parte central da figura 3.9. A conexão de seqüência inclui o rótulo de dependência, o qual pode assumir os valores *start_start*, *end_end*, *end_start* ou *?* (esse último para conexões indefinidas). As conexões de *feedback*, por sua vez, são representadas graficamente por linhas pontilhadas, as quais possuem um rótulo que identifica a condição lógica necessária para habilidar a re-execução da atividade destino.

As conexões múltiplas - cuja notação é representada na parte inferior da figura 3.9 - podem possuir múltiplas origens (para conexões *Join*) ou múltiplos destinos (para conexões *Branch*). O retângulo que representa uma conexão múltipla possui as seguintes informações: uma letra indicando se a conexão é *Branch* (B) ou *Join* (J), o tipo (*AND, OR* ou *XOR*) e dependência temporal relacionada.

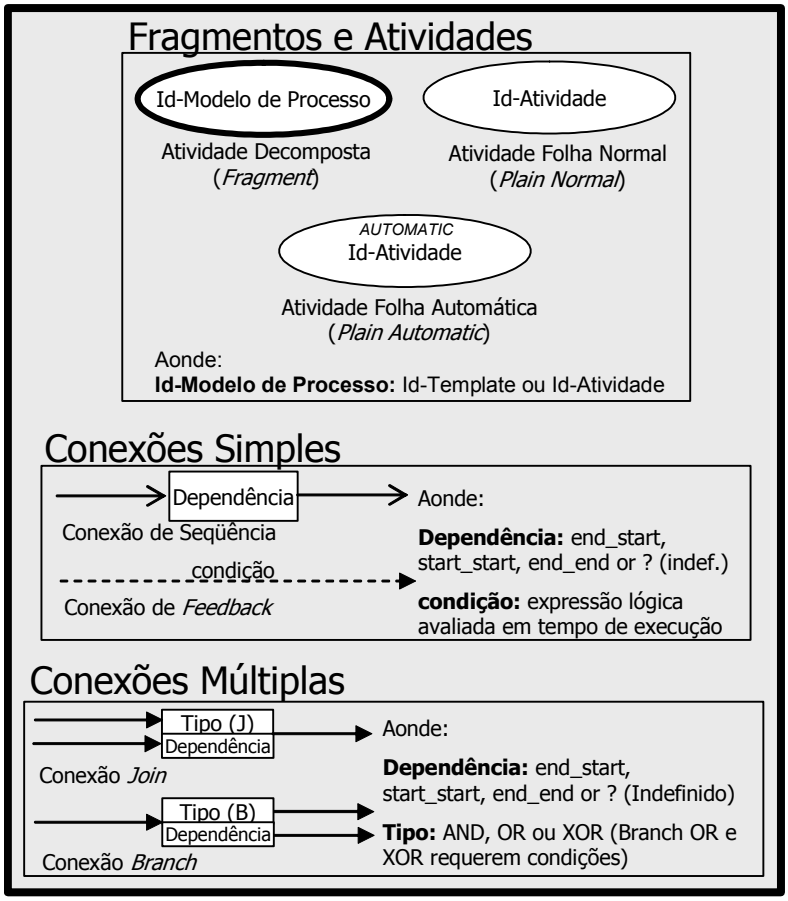

FIGURA 3.9 - Representação gráfica para elementos usados na constituição de *templates* na dimensão processo

A figura 3.10 apresenta a representação gráfica para um *template* envolvendo as atividades e conexões simples e de *feedback* necessárias para uma parte do modelo de processo. O modelo descreve uma seqüência que é iniciada na atividade *Analyse the Problem* e é encerrada na atividade *Refine the System Definition*. O modelo inclui ainda, além das conexões simples de seqüência, três conexões de feedback (a saber, entre as atividades *Evaluation* e *Analyse the Problem*, entre *Manage the Scope of the System* e *Understand Stakeholder Needs*, e entre *Refine the System Definition* e *Define the System*).

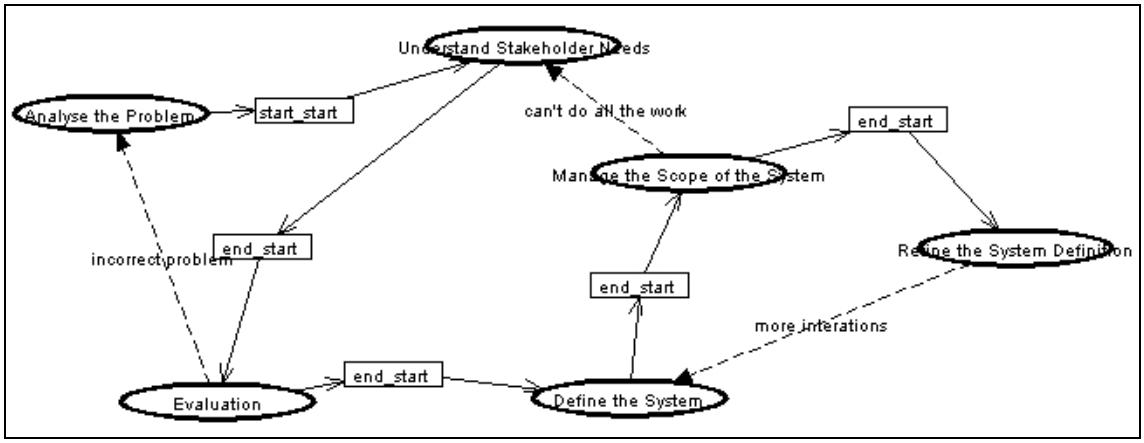

FIGURA 3.10 - O *template* RUP: exemplo de atividades e conexões em um *template* descrito com a notação proposta

O capítulo 4 descreve algebricamente os tipos de dados e apresenta as regras para boa formação de *templates* (i.e., composição de atividades e conexões).

3.5.2 A Dimensão 'Software'

A dimensão 'Software' define de forma abstrata os artefatos consumidos, transformados e produzidos pelos processos de software. Os artefatos são classificados de duas formas: artefatos **concretos** referem-se a elementos imutáveis, que são consumidos durante o desenvolvimento de software (por exemplo, *frameworks*, manuais com estilos de codificação, bibliotecas de software e *design patterns*); artefatos **abstratos** são referências aos elementos de dados que são instanciados somente em tempo de execução.

Tal como mostrado pelo pacote *SoftwareDimension* (figura 3.11) os artefatos são associados às conexões de artefato (*Connections.AbsArtifactCon*)

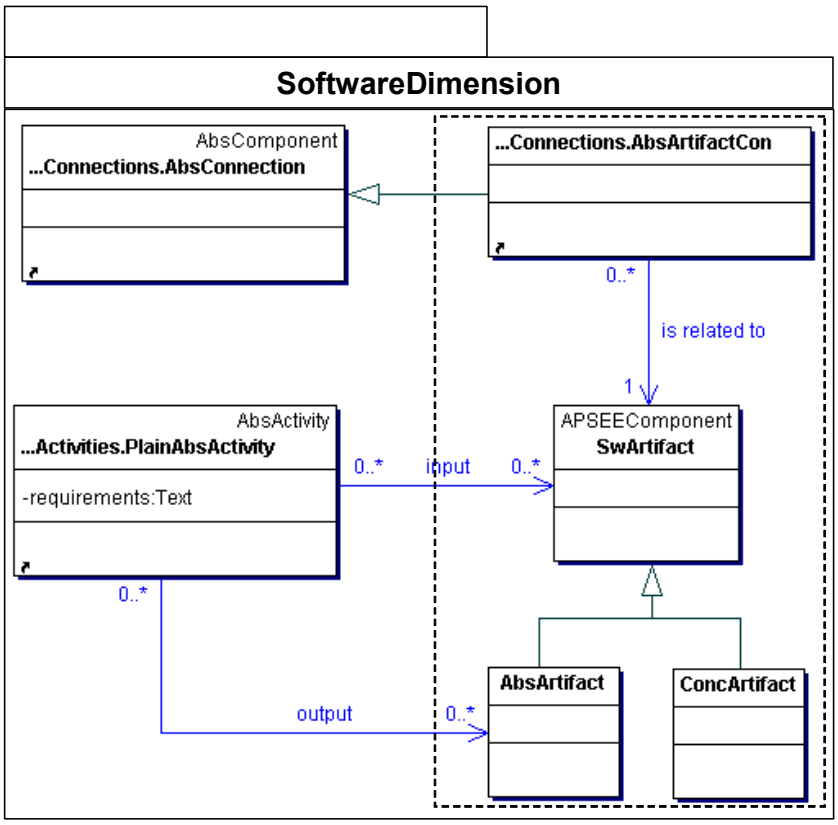

FIGURA 3.11 - A dimensão 'Software'

A figura 3.12 apresenta a notação gráfica adotada para conexões de artefato e artefatos propriamente ditos, utilizada na descrição dos *templates*.

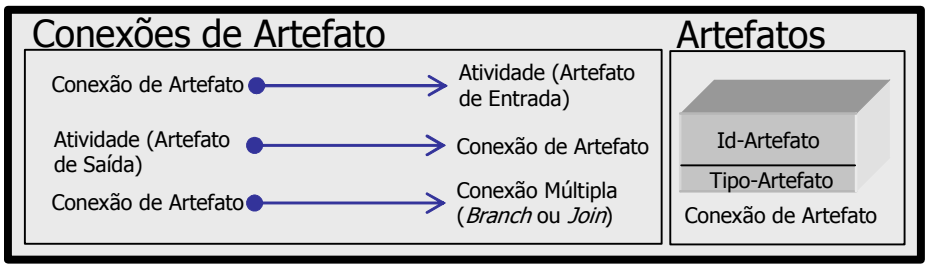

FIGURA 3.12 - Notação para conexões de artefato

A figura 3.13 apresenta um exemplo de um *template* parcial na notação *APSEE*, incluindo uma atividade (*Requirements Acquisition*), duas conexões de artefato (*ProductLiterature* e *ReqSpecification*) e uma conexão múltipla *Branch AND*.

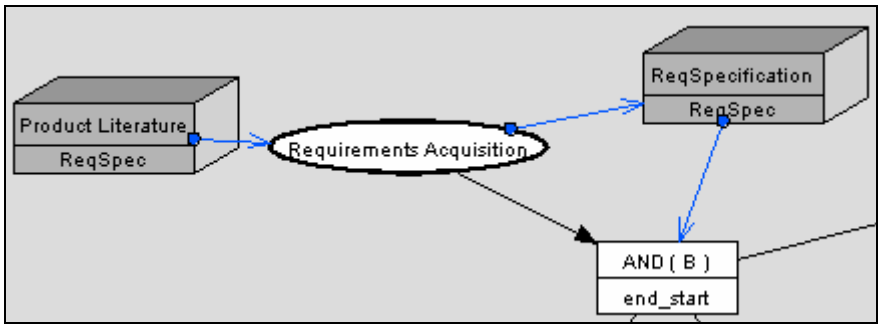

FIGURA 3.13 - Exemplo parcial da notação de um *template* contendo duas conexões de artefato

3.5.3 A Dimensão 'Pessoal'

A dimensão 'Pessoal' descreve o fator humano envolvido na definição de atividades normais abstratas, como descrito na figura 3.14. Na definição de uma atividade normal de *template*, seu projetista pode definir os tipos de cargos (*RoleType*) requeridos. Além disso, grupos abstratos podem ser definidos a partir dos seus tipos (*GroupType*). Todos os elementos descritos possuem correspondentes na definição concreta, envolvendo as classes *Group*, *Agent*, *Role* e *Ability* [LIM2002d]. A área circundada da figura 3.14 destaca os elementos que compõem a dimensão Pessoal.

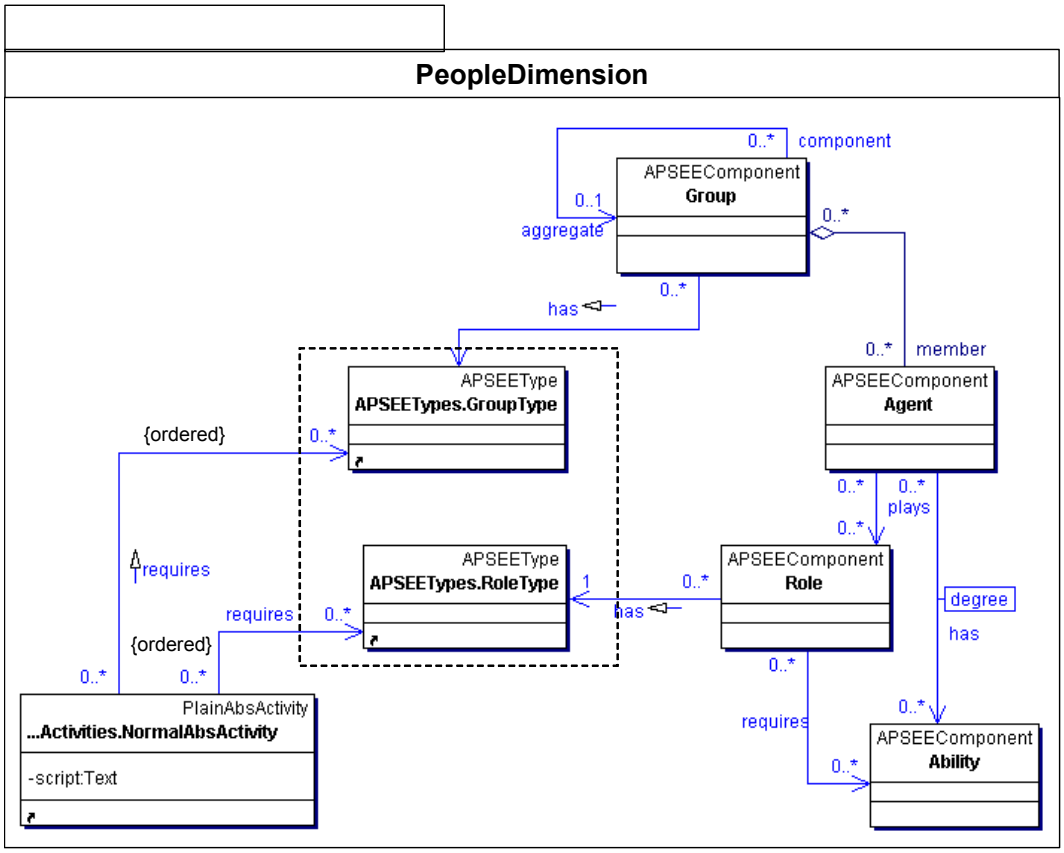

FIGURA 3.14 - A dimensão 'Pessoal'

3.5.4 A Dimensão ‘Ferramentas’

A dimensão Ferramentas descreve a perspectiva de alocação de ferramentas de software no desenvolvimento de software. Em um *template*, tal como ilustrado pela figura 3.15, os tipos de ferramentas (instâncias de *APSEETypes.ToolType*) ocupam lugar de destaque na definição de uma atividade automática. Além disso, as ferramentas são alocadas por atividades normais, produzindo e consumindo software (isto é, instâncias de *SoftwareDimension.SwArtifact*).

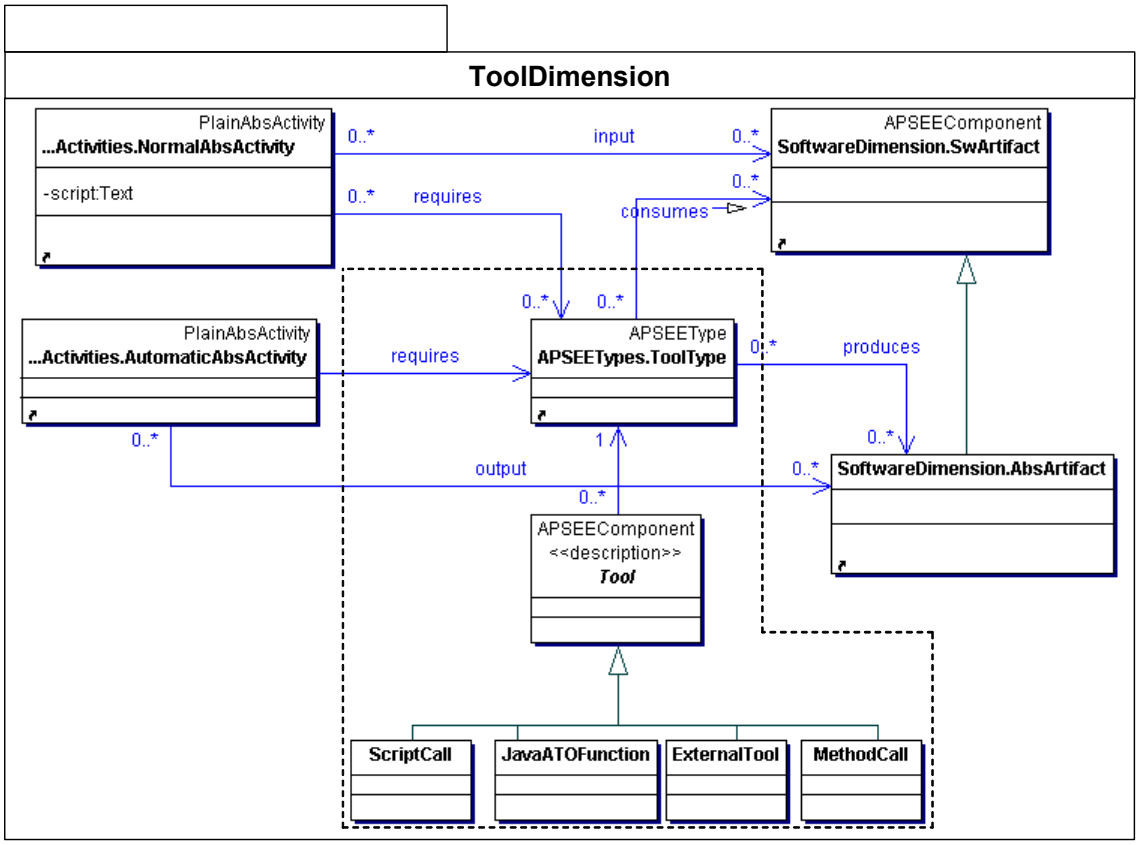

FIGURA 3.15 - A dimensão ‘Ferramentas’

3.5.5 A Dimensão ‘Recursos’

A dimensão Recursos descreve os recursos de apoio utilizados direta ou indiretamente no desenvolvimento de software. Nesse texto, recursos de apoio são aqueles elementos passivos que apóiam a realização de uma atividade da organização e podem ser compartilhados, tal como impressoras; consumidos, tal como recursos financeiros; ou de uso exclusivo como, por exemplo, salas e recursos computacionais [LIM2001a].

No modelo de *templates*, os recursos são requeridos pelas atividades normais (*NormalAbsDesc*). Conforme ilustrado pela figura 3.16, os recursos são tipados (ou seja, são associados a instâncias da classe *APSEETypes.ResourceType*) e três nodos *default* são definidos como compartilháveis (*shareable*), consumíveis (*consumable*) ou de uso exclusivo (*exclusive*), fornecendo uma hierarquia de tipos *default* homogênea com o adotado pelo ambiente *APSEE* (descrito em [LIM2002b] e [LIM2002c]).

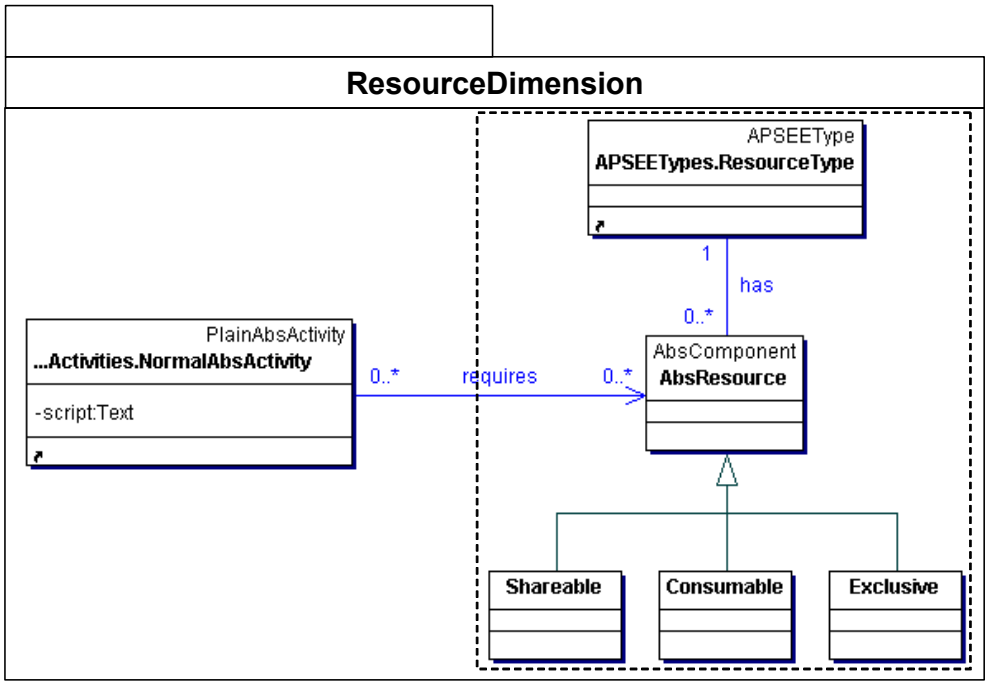

FIGURA 3.16 - A dimensão 'Recursos'

A figura 3.17 apresenta um exemplo para a hierarquia de recursos, organizados como especializações dos três tipos básicos de recursos (*Consumable*, *Exclusive* e *Shareable*).

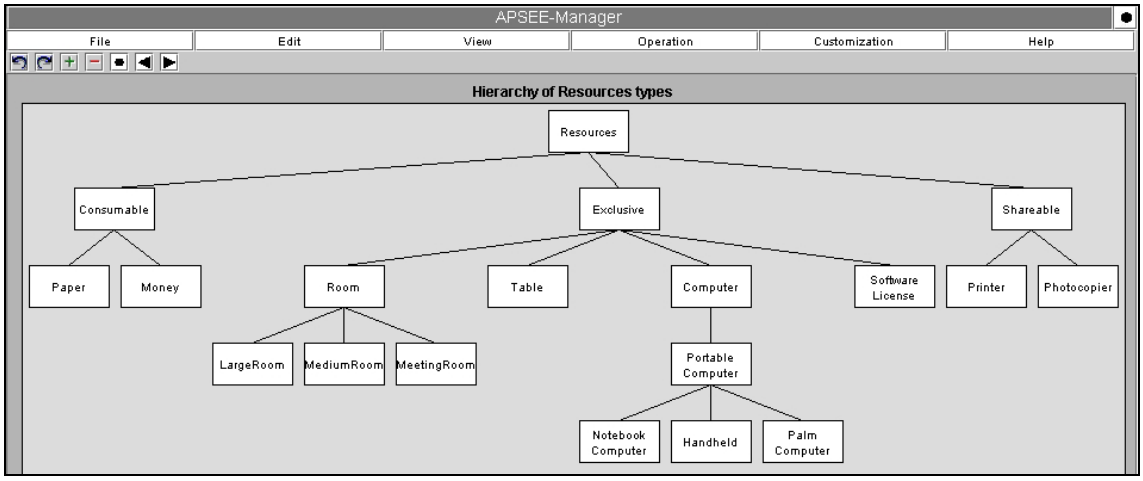

FIGURA 3.17 - Exemplo de hierarquia de tipos de recursos

3.5.6 A Dimensão 'Políticas de Processos'

Segundo Feiler [FEI93], Políticas de Processos de Software (*Process Policies*) consistem em "princípios que conduzem o desenvolvimento e/ou a execução de processos de software". No modelo aqui proposto, uma política pode ser informalmente descrita como um conjunto de propriedades que atuam na formação e execução de modelos de processos de software, representando um conhecimento gerencial genérico e reutilizável para diferentes contextos. Uma política pode estar habilitada em um componente de processo (por exemplo, uma atividade, um recurso ou um agente), em um fragmento de processo (estando habilitada em todos os componentes), ou em uma organização (isto é, em todos os processos existentes na organização), permitindo a sua reutilização em diferentes contextos.

De fato, no modelo APSEE, Políticas foram propostas com o objetivo de complementar os *Templates* na descrição de elementos reutilizáveis para processos de software, podendo ser úteis para expressar conhecimento relacionado com a gerência de processos de software de uma maneira computável e compacta. De fato, Políticas foram projetadas a partir da constatação de que importantes estratégias gerenciais definidas por metodologias e práticas comuns difundidas para a gerência de processos de software, não podiam ser expressas formalmente através das PMLs existentes.

No modelo APSEE, as Políticas estendem a PML proposta, complementando as dimensões descritas nas seções anteriores. Inicialmente, foi proposto o construtor que apóia a definição e verificação das **Políticas Estáticas** - objeto de estudo desse trabalho - com o objetivo de apoiar a modelagem de processos de software, encapsulando regras sintáticas, independentemente definidas, que atuam na formação dos modelos, podendo ser úteis para fazer cumprir boas práticas de gerenciamento de projetos adotadas pela organização. Assim, o construtor de Políticas Estáticas define uma linguagem compacta e declarativa, que permite aos projetistas de processos a definição de novas instâncias.

A figura 3.18 apresenta um esquema abstrato que ilustra a habilitação de uma Política Estática em um Componente, mostrando graficamente os componentes de uma Política e o seu relacionamento com o tipo de dados definido (colocado entre parênteses abaixo do rótulo do componente). De forma simplificada[19], a definição de uma Política Estática é dividida em três partes principais:

- A **Interface** especifica o tipo de objeto APSEE que será tratado pela Política, definindo um rótulo (*label*), o qual é usado na definição das propriedades da Política para referenciar o componente externamente definido;
- As **Propriedades de Habilitação** (*Enabling Properties*) constituem as pré-condições lógicas que devem ser satisfeitas para que uma política seja executada (relacionadas ao tipo definido na interface da Política). As propriedades correspondem essencialmente às chamadas de métodos disponíveis para o tipo relacionado com a interface da Política;
- As **Propriedades** (*Properties*) expressam características sintáticas que são verificadas sempre que as propriedades para habilitação são satisfeitas. Caso alguma destas propriedades não seja satisfeita, o mecanismo de verificação de processos adota o comportamento especificado pelo tipo de reação atribuído à política. Assim, no caso de uma Política obrigatória, a verificação é interrompida e uma mensagem de erro é exibida ao usuário. Para Políticas opcionais, somente um aviso é incluído no *log* de verificação das Políticas do processo, e a verificação prossegue.

[19] A descrição completa para a sintaxe e semântica do construtor de Políticas Estáticas está apresentada no capítulo 4 a seguir.

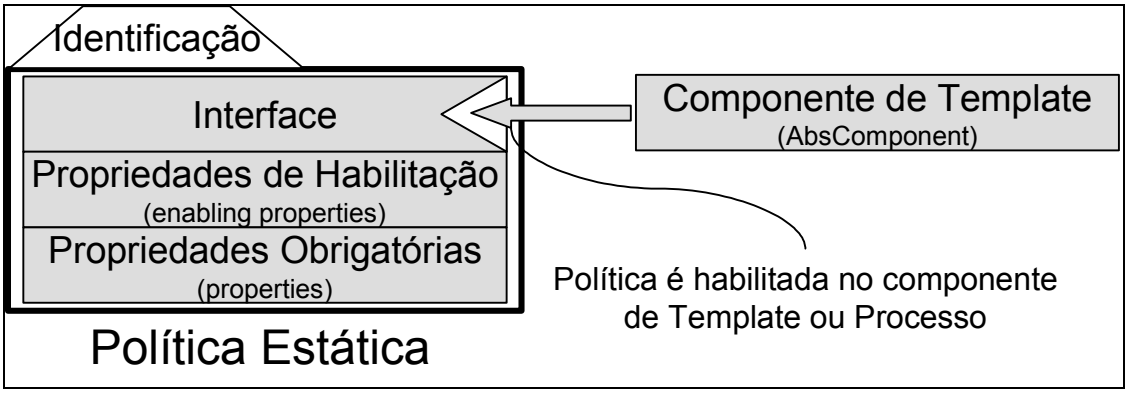

FIGURA 3.18 - Esquema ilustrativo para habilitação de Políticas em componentes APSEE

Políticas Estáticas são verificadas durante a modelagem de processos, quando o projetista de processo solicita a verificação do modelo (um passo obrigatório no ciclo de vida de processos *APSEE* para habilitar execução de um modelo de processo instanciado). Assim, por exemplo, supondo que a política "*Development Produce Artifacts*" descrita na figura 3.19 esteja habilitada em um *template* teremos que, para toda atividade componente do *template*, desviar o fluxo de execução para verificar se as *Enabling Properties* são satisfeitas (i.e., o item {1} da figura 3.19). Desse modo, se a atividade avaliada for do tipo "*Development*" (ou um subtipo) a verificação prossegue executando o item {2}.

ID: "Development produce artifacts"
Description: "A Development activity that does not have output artifacts is forbidden".
Mandatory: True
Interface: a: Activity;
Enabling Properties: a.get_type() **sub_type_of** "Development" {1}
Properties: a.get_output_artifacts().get_size() >= 1 {2}

FIGURA 3.19 - Exemplo de Política Estática: "*Development produce artifacts*"

A experiência positiva na definição e utilização das Políticas Estáticas em diferentes situações[20] levou ao desenvolvimento, por outros membros do grupo, de outros dois tipos de linguagens especializadas na instanciação de agentes e recursos. Estas são denominadas **Políticas de Instanciação**, sendo descritas por Lima Reis em [LIM2001a] e [LIM2002b], cuja sintaxe é baseada na proposta aqui apresentada, assim como fornece formalismos especializados para expressar critérios definidos pelo usuário, formalismos esses que atuam na descrição de estratégias genéricas para restrição e ordenação na instanciação de recursos e agentes. Tais Políticas podem ser ativadas automaticamente durante a execução de processos, considerando a disponibilidade atual e futura para os recursos e agentes envolvidos.

A classificação proposta para Políticas de Processos distingue tipos de dados especializados que compartilham um objetivo comum: complementar um modelo de processo com estratégias para gerenciamento independentemente definidas. A prática demonstrou que esses dois tipos de políticas podem ser complementares. Por exemplo, é possível expressar em uma Política Estática restrições básicas para a alocação de agentes (verificadas durante a modelagem do processo) que é refinada através de Políticas de Instanciação (executadas durante a execução do processo).

[20] A experiência na utilização prática de Políticas Estáticas para *templates* reais é resumida no capítulo 6.

O pacote descrito graficamente pela figura 3.20[21] descreve o relacionamento entre as Políticas (tipo abstrato *ProcessPolicy*, na figura) e os componentes de um processo APSEE (*APSEEComponent*). Uma política possui uma condição de disparo (*firing condition*, condição esta denominada *enabling property* em Políticas Estáticas) e uma *Interface*, a qual descreve um rótulo e um tipo APSEE, restringindo a aplicabilidade da política ao tipo associado. A descrição completa para os elementos sintáticos de uma Política Estática é complexa (requerendo mais de vinte classes), sendo portanto apresentada no capítulo 4 (mais especificamente, na seção 4.4).

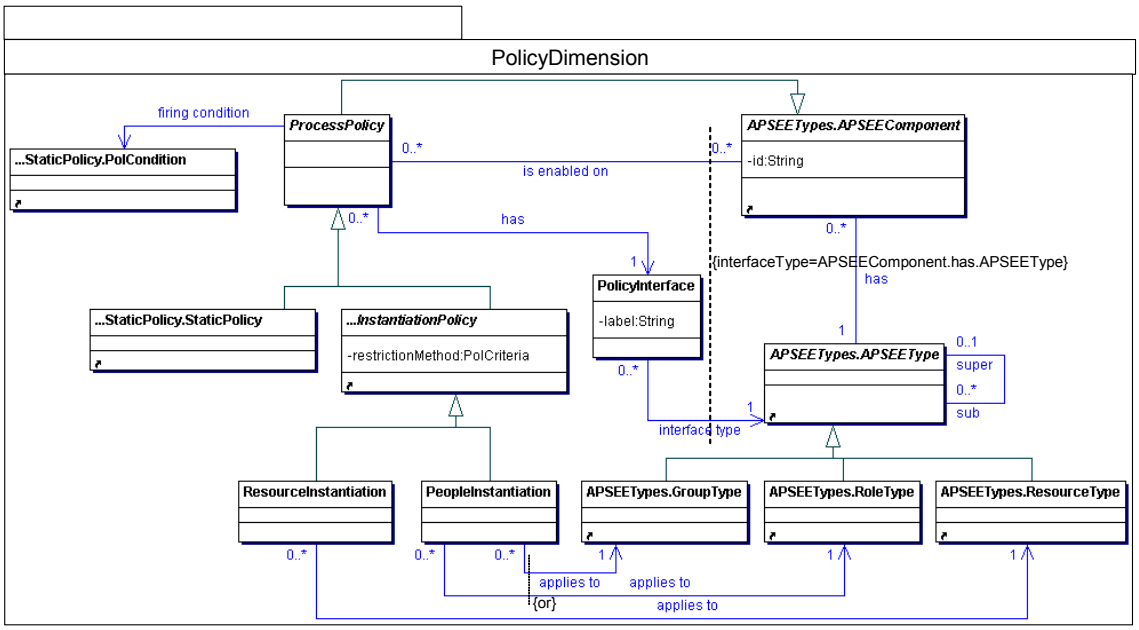

FIGURA 3.20 - A dimensão 'Políticas de Processos'

3.5.7 Agrupando todas as dimensões no projeto de um *Template*

Na construção de um *template*, o projetista faz uso de diferentes editores e ferramentas especializadas na modelagem das diferentes dimensões propostas, as quais devem ser agregadas na constituição de novos modelos de processo. Assim, por exemplo, um projetista pode especificar grupos, cargos e habilidades definidas na dimensão Pessoal que serão relacionados às atividades normais de um *template*. De forma análoga, Políticas de processos podem ser projetadas levando-se em consideração os novos tipos de agentes disponíveis, sendo então habilitadas em componentes específicos do *template* em processo de especificação.

A figura 3.21 apresenta um esquema gráfico para um *template* arbitrário. Cada camada do diagrama representa uma das dimensões propostas, isto é, as dimensões Pessoal, Ferramentas, Recursos, Software, Processos e Políticas são graficamente dispostas. No caso, os relacionamentos intra e interdimensionais para as camadas horizontalmente dispostas são propositadamente omitidos, com o objetivo de simplificar o entendimento do diagrama. A dimensão Políticas é disposta perpendicularmente às demais, a fim de evidenciar o caráter ortogonal das Políticas como elementos de modelagem de *templates*.

[21] O diagrama da figura 3.20 ainda contém detalhes acerca das Políticas de Instanciação que estão fora do escopo deste trabalho

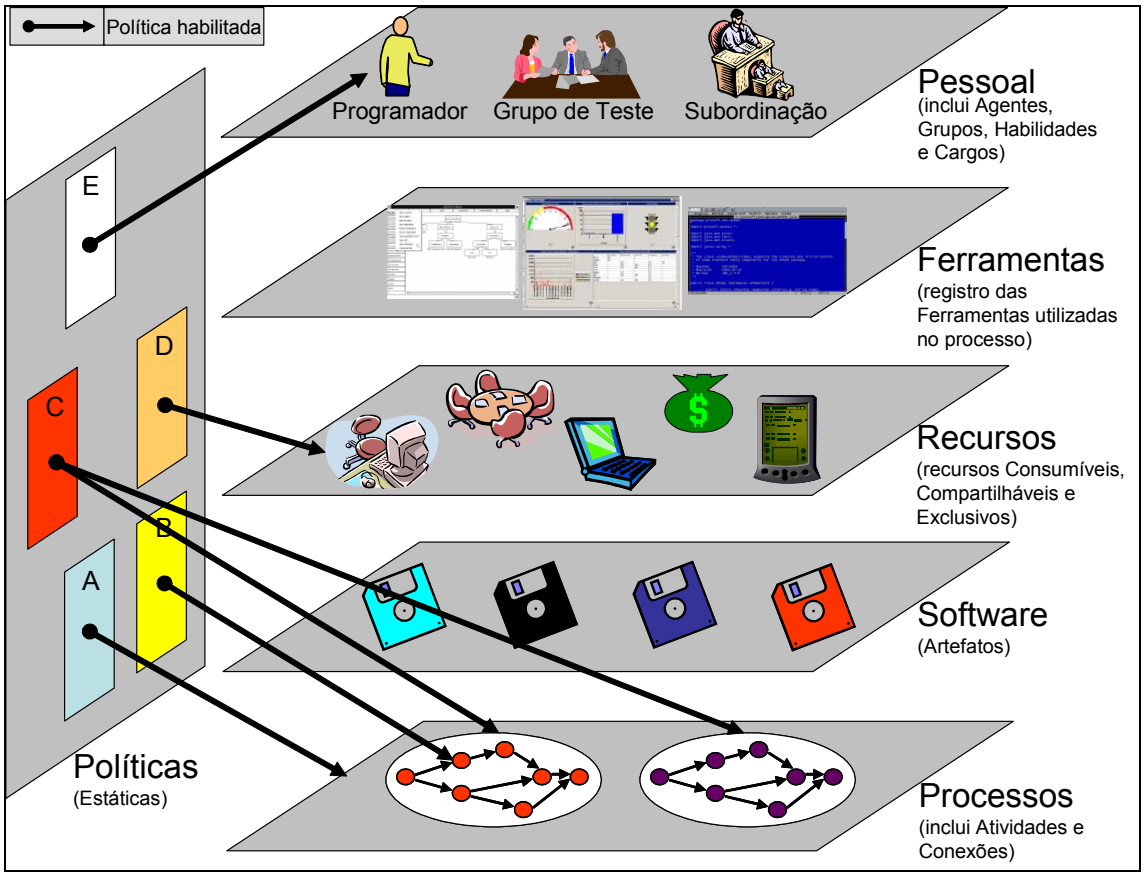

FIGURA 3.21 - Esquema para composição de elementos em um *template*

3.6 Relacionamento entre *Templates* e demais estados de um Processo

Vale destacar, para os propósitos desse texto, a existência, no modelo APSEE, de três estados principais para um processo, representando o diferente nível de detalhamento necessário para descrever o processo de software. ***ProcessTemplate*** - construtor proposto nesse trabalho - fornece uma descrição genérica e abstrata, apropriada para reutilização futura (podendo ser reutilizado em contextos organizacionais e/ou tecnológicos diferentes). O ***InstantiatedProcessModel***, por sua vez, consiste de um modelo com informações já instanciadas, definindo a alocação de recursos e agentes para as atividades do processo, sendo que todas as informações necessárias estão disponíveis para habilitar a sua execução. Por fim, o ***EnactedProcessModel*** é o estado em que o modelo foi previamente executado, sendo que o tipo de dados registra as informações sobre as ocorrências da execução do processo.

É importante observar que um projetista de processos, com as ferramentas disponíveis, pode descrever alternativamente um *template* ou um modelo instanciado (permitindo desde descrições essencialmente abstratas como os *ProcessTemplates* até descrições executáveis na forma de modelos instanciados), adicionando informações no modelo de processo de acordo com a necessidade. Além disso, o ambiente permite a existência de processos no estado *Mixed*, ou seja, em que seus componentes estejam em estados diferentes.

Há ainda uma semelhança proposital entre as estruturas disponíveis para instâncias de *Templates*, *Instantiated* e *Enacted*. Em linhas gerais, as representações mais concretas (*instantiated* e *enacted*) somente adicionam atributos que estão relacionados especificamente com o contexto de processos instanciados e em execução. Tal uniformidade tem o potencial de facilitar a adaptação de *templates*. Assim, todas as atividades e componentes de processos e *templates* são associados a identificadores e, portanto, podem ser recuperados para compor novos modelos (abstratos ou instanciados). A seção 3.7, a seguir, discute os aspectos relacionados com a adaptação, segundo a abordagem proposta.

3.7 Protocolo para Composição e Adaptação de Processos a partir de *Templates* Recuperados

Durante a modelagem de processos é comum enfrentar situações em que um ou mais *templates* (ou sub-processos) precisam ser recuperados, adaptados e combinados para a composição de um novo processo. A adaptação é necessária por diferentes razões, tais como: aspectos tecnológicos específicos do problema tratado, a falta de experiência da organização no uso de uma determinada técnica proposta por um *template*, ou a natureza inerentemente específica do software sob desenvolvimento. Assim, um protocolo de composição é necessário para guiar a adaptação de modelos de processos gerados a partir de *templates* reutilizáveis.

Esse tópico, sob a ótica da reutilização de software, lida com requisitos contraditórios: enquanto de um lado a descrição de protocolos para componentes de software deve ser precisa e rígida, por outro lado alguma flexibilidade é necessária para permitir a adaptação de componentes em um maior número de contextos. Quanto à adaptação de *templates* de processos de software, não existe uma abordagem universalmente aceita. De fato, este é um tópico pouco explorado na literatura especializada (citada em [ELL96], [PER96] e [JØR2001]). Visto que - ao contrário de componentes de software, executados por máquinas - os processos de software são modelados e executados por humanos, temos um consenso de que modificações em componentes recuperados são toleradas [PER96] [JØR2001].

Apesar do objetivo do trabalho aqui apresentado se limitar à especificação de um meta-modelo, o qual permite a definição de modelos reutilizáveis para processos de software, a adaptação de tais modelos é um tópico importante, sendo tratado aqui com o objetivo de facilitar a aplicação, experimentação prática e análise inicial da viabilidade do meta-modelo proposto.

O projeto atual de protocolos de composição e adaptação de processos deve levar em consideração a expectativa do surgimento, em futuro próximo, de ferramentas que automatizem a recuperação e adaptação automática de *templates* e seus componentes. Assim, partindo-se das características dos protótipos existentes atualmente (tais como a ferramenta *ProTail* descrita por Münch em [MÜN97]), deve-se levar em conta que tais ferramentas exigem que o usuário forneça como entrada o(s) *template*(s) candidato(s) à adaptação e a descrição rigorosa das características do contexto destino, obtendo como resultado um processo instanciado sintaticamente válido, pronto para ser executado ou aperfeiçoado ainda mais pelo projetista. Portanto, em primeiro lugar, as modificações permitidas na adaptação e composição de *templates* devem ser derivadas das regras gerais para boa formação sintática de modelos de processos executáveis.

Nesse trabalho, devido à proposital correspondência sintática entre muitos dos elementos de *templates* (tipo de dados apresentado nesse texto) e processos instanciados (descritos por Lima Reis em [LIM2002a]), acredita-se que a adaptação automática seja possível em um futuro próximo, fornecendo como resultado um processo não-instanciado. Entretanto, qualquer modificação sintática introduzida pelo usuário durante a adaptação manual de um *template* - modificação esta não relacionada simplesmente com a instanciação ou especialização de um componente do *template* para o processo derivado - tem o potencial de distanciar o processo obtido da semântica e intenções originais previstas pelo projetista do *template* original. Portanto, o modelo aqui proposto oferece três tipos de restrições a serem escolhidas pela organização-usuária: a adaptação totalmente livre, a adaptação guiada por políticas, ou a adaptação que preserva os elementos sintáticos originais. Tais tipos de restrições foram especificadas no modelo (capítulo 4) e estão incorporadas na implementação atual do editor de processos *APSEE* (capítulo 5), sendo informalmente descritas pelas sub-seções a seguir.

3.7.1 Adaptação livre de *templates*

A **adaptação livre** (*free adaptation*, na terminologia usada na descrição do modelo no capítulo 4) não apresenta qualquer restrição na adaptação de um *template* recuperado. Desta forma, na construção de um processo derivado a partir de um *template*, o projetista é livre para remover ou adicionar quaisquer elementos, desde que o resultado final seja um processo sintaticamente válido.

A figura 3.22 apresenta um *template*-exemplo usado nessa e nas seções posteriores para discussão acerca dos tipos de adaptação fornecidos[22]. Esse exemplo é composto por duas atividades: Modelagem de Dados (atividade normal do tipo *DataModeling*) e Projeto de Bases de Dados (tipo *DatabaseDesign*). Os artefatos envolvidos são: Modelo de Dados (produzido pela atividade Modelagem de Dados e fornecido à atividade de Projeto), Requisitos (artefato fornecido como entrada para as duas atividades do modelo), e Projeto de Bases de Dados (produzido pela atividade de Projeto). Finalmente, a Política Estática "*Development Produce Artifacts*" (previamente descrita na seção 3.5.6) está habilitada no *template*. Por fim, a atividade Modelagem de Dados possui uma conexão simples do tipo *end-start*, tendo como destino a atividade Projeto de Bases de Dados.

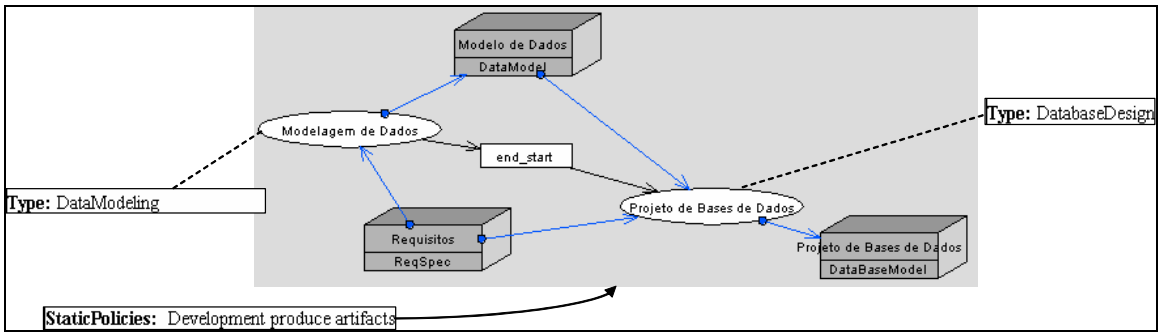

FIGURA 3.22 - Exemplo de *template* fornecido para adaptação

[22] Nesse exemplo, os rótulos das atividades e artefatos estão em Português.

A figura 3.23 apresenta seis processos candidatos (com rótulos numéricos)[23], apresentando adaptações realizadas a partir do *template* original. No caso da adaptação livre, somente a adaptação de número quatro (4) não é válida, por incluir uma conexão simples que introduz um ciclo no processo (conexão *end-start* entre as atividades Revisão e Modelagem de Dados).

[23] Este exemplo admite que os tipos das atividades originais permanecem os mesmos, assim como a política habilitada ("*Development produce artifacts*") permanece habilitada nas adaptações da atividade Modelagem de Dados.

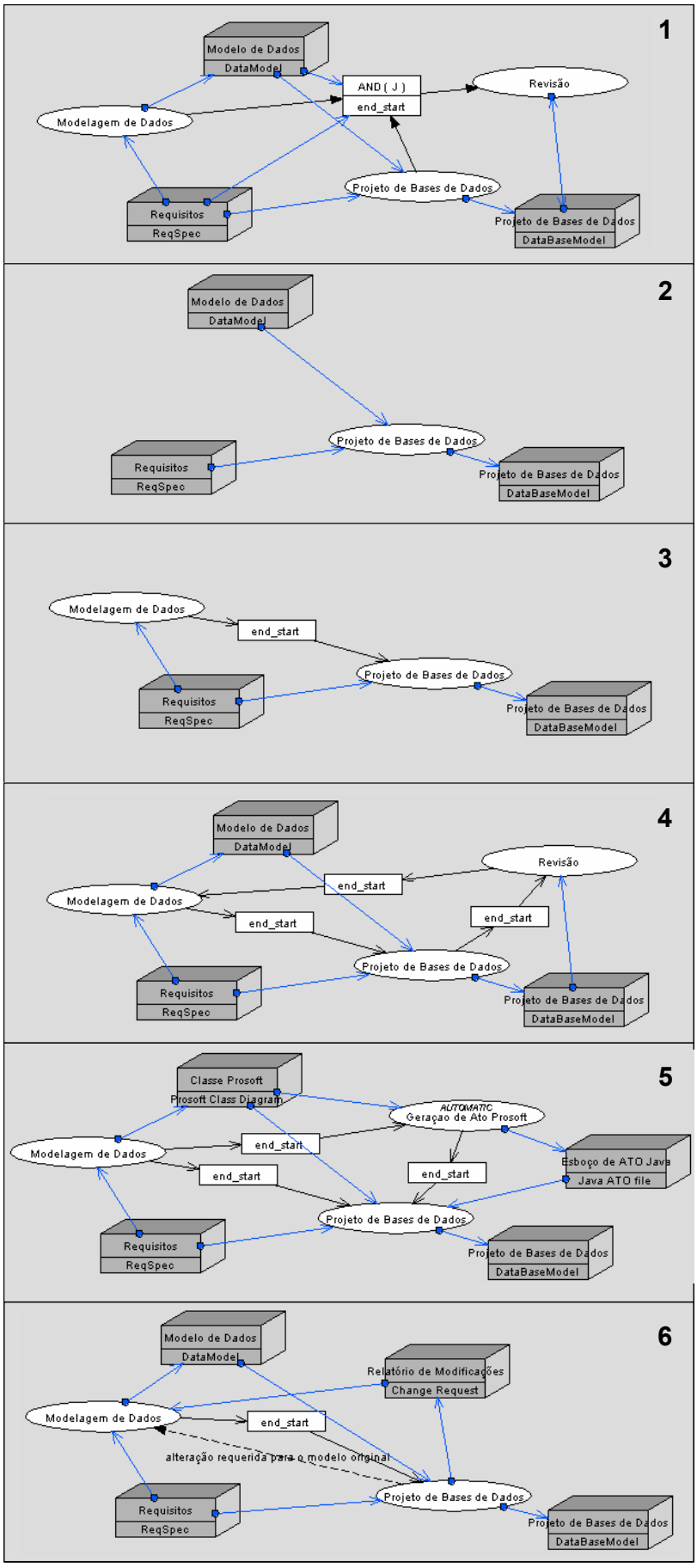

FIGURA 3.23 - Exemplos de Adaptação do *Template* exemplo

3.7.2 Adaptação de *templates* guiada por políticas

A **adaptação guiada por políticas** (*policy-based adaptation*, no capítulo 4) está baseada no fato que políticas expressam propriedades gerais e ortogonais aos *templates*, devendo ser respeitadas mesmo na utilização dos *templates* em novos contextos. Assim, na adaptação guiada por políticas, as políticas habilitadas no *template* original não podem ser removidas nos processos derivados e, por conseguinte, o processo resultante será restrito ao expresso pelas políticas originais.

Considerando o mesmo exemplo apresentado na seção anterior, e seguindo a definição acima, considera-se que a adaptação dois (2) constitui um exemplo inválido (por não satisfazer a política habilitada), além do item 4 (por não satisfazer a adaptação livre).

3.7.3 Adaptação de *templates* que preserva os elementos sintáticos originais

A **adaptação que preserva os elementos sintáticos originais** (*restricted adaptation*, no capítulo 4) segue uma linha que ainda privilegia a flexibilidade, mas impede que os principais componentes inseridos por um projetista de *template* sejam removidos nas adaptações posteriores. Desta forma, tal abordagem restringe a adaptação de *templates* e componentes recuperados, enquanto que os elementos sintáticos originais são preservados. São características dessa abordagem:

- São válidas as restrições existentes na adaptação livre (i.e., o modelo final deve ser consistente), e na adaptação guiada por políticas (i.e., políticas habilitadas no modelo original não podem ser removidas);
- De forma análoga à instanciação de *frameworks* [SIL2000], na adaptação de um *template* completo é permitido ao usuário adicionar livremente novos elementos sintáticos[24]. Portanto, em geral, as alterações permitidas são restritas à instanciação ou especialização de tipos para os componentes originalmente fornecidos. Há uma exceção relativa à alteração dos *scripts* das atividades normais (livre);
- As conexões de artefatos e temporais (simples, múltiplas e de *feedback*) presentes no *template* original não podem ser removidas nas adaptações;
- Uma atividade pode ter o seu nível de detalhe alterado em função de novos requisitos ou facilidades tecnológicas existentes. Assim, uma atividade-folha pode ser transformada em fragmento, assim como uma atividade originalmente definida como normal pode ser transformada em automática (e vice-versa).

Considerando os candidatos apresentados na figura 3.23 e a hierarquia exemplo para tipos de artefatos da figura 3.24, somente os itens 5 e 6 são considerados as adaptações válidas, pois os itens de 1 a 3 removem elementos sintáticos descritos no *template* original. Como nos outros tipos de adaptação fornecidos, a adaptação número 4 foi excluída por constituir um exemplo de processo sintaticamente incorreto. No caso da adaptação de número 5, a conexão de artefato originalmente rotulada como ‘Modelo

[24] Os elementos inseridos em um processo devem levar em consideração as regras para boa formação de processos (por exemplo, a ausência de ciclos em processos), tal como descrito no capítulo 4.

de Dados' foi renomeada para 'Classe Prosoft', artefato cujo tipo - *Prosoft Class Diagram* - é uma especialização do tipo original *DataModel*.

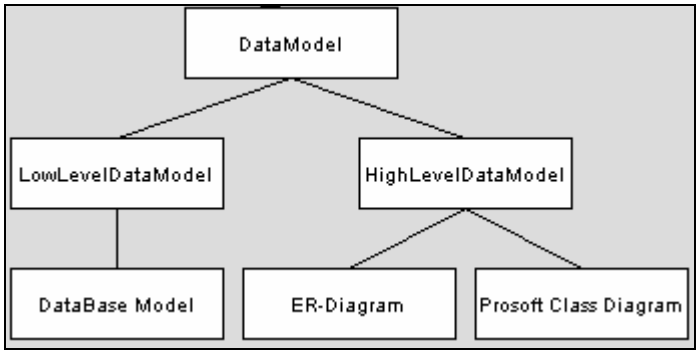

FIGURA 3.24 - Hierarquia exemplo para artefatos no *template*: especializações para o tipo *DataModel*

4 Especificação do Modelo APSEE-Reuse

Esse capítulo é o responsável por descrever em detalhe o modelo *APSEE*-Reuse proposto, o qual englobou a especificação de mais de 40 tipos de dados, o que derivou a implementação de um protótipo de editor diagramático que define regras de consistência para *templates*, políticas de processos e processos adaptados.

O desenvolvimento de um sistema de software de tal complexidade demanda a escolha de métodos de Engenharia de Software que sejam gerenciáveis, permitindo a especificação precisa e viável dos componentes do sistema. Atualmente, a literatura descreve um amplo leque de métodos que apóiam o desenvolvimento sistemático de software em diferentes níveis de abstração e usando notações específicas.

Assim, inicialmente o texto apresenta os formalismos adotados para a especificação do modelo, discutindo suas características e os aspectos fundamentais que nortearam a escolha. A apresentação da especificação do modelo proposto é dividida em duas partes principais: na seção 4.2 são descritos os tipos de dados e algumas das funções para a modelagem de *templates* e seus componentes; por sua vez, a seção 4.3 descreve as regras para a linguagem visual de modelagem de *templates*; finalmente, a seção 4.4 apresenta o mecanismo de interpretação de políticas estáticas.

4.1 Formalismos utilizados na especificação do modelo proposto

O desenvolvimento de software atual conta com diferentes ferramentas, técnicas e metodologias que vem evoluindo a partir da crescente complexidade do software produzido. Em função das características do software em desenvolvimento, a seleção de tecnologias adequadas constitui um fator decisivo para o sucesso do processo. Assim sendo, a seção a seguir discute sucintamente as características do modelo proposto do ponto de vista específico da especificação e implementação do modelo, discussão esta que motivou a investigação da aplicabilidade de formalismos específicos.

4.1.1 Características do problema tratado sob ponto de vista da exeqüibilidade da especificação e prototipação do modelo

O modelo *APSEE*-Reuse, descrito informalmente no capítulo anterior, é um sistema de software complexo, possuindo características próprias. De uma forma geral, a ampla diversidade de aspectos envolvidos na definição do meta-modelo completo implica no uso de métodos que especifiquem precisamente os seus tipos de dados e algoritmos, e que tal especificação possa ser continuamente utilizada como base semântica que auxilie no entendimento e na evolução futura da proposta.

O meta-modelo proposto requer ainda um conjunto de tipos de dados inter-relacionados que possuem diferentes representações gráficas. Ainda, o modelo define

um conjunto de regras sintáticas para serem tipicamente aplicadas a partir de um editor gráfico[25], que combine os diferentes elementos sintáticos para a descrição de um *template* válido. A aderência a regras que guiem as transformações sintáticas bem formadas é, portanto, um elemento importante nesta especificação.

A dimensão Políticas de Processos possui algumas especificidades importantes: o construtor é baseado na definição de um texto. Assim, o suporte computacional requerido para verificação das Políticas Estáticas definidas pelo usuário é essencialmente um interpretador. Tal interpretador seria responsável por executar as chamadas de funções definidas pelo usuário, associando os objetos de um modelo de processo de software como parâmetros reais para as interfaces definidas no corpo de uma política.

A experimentação prática do modelo proposto constitui um objetivo importante. Para tanto, é necessário desenvolver protótipos que implementem a funcionalidade básica prevista pelas diferentes partes do modelo proposto. O tamanho e complexidade do meta-modelo, entretanto, impelem à implementação segmentada em diferentes componentes independentemente definidos. Além disso, a prototipação (combinada com a simulação de modelos de alto nível) freqüentemente fornece *feedback* com novos requisitos ou necessidades de aperfeiçoamentos que devem ser reconciliados ao modelo original.

4.1.2 Características gerais dos formalismos adotados

Em face às características do problema expostas acima, foi realizada uma avaliação dentre as metodologias existentes, levando também em consideração a experiência do autor[26] e dos demais membros do grupo PROSOFT na especificação de arquiteturas de software de complexidade similar.

A evolução do ambiente PROSOFT está intimamente ligada com o objetivo de construir ferramentas que apóiem o uso de métodos formais no ciclo de vida do software. As técnicas formais de especificação são aquelas que fornecem uma sintaxe e semântica formais para descrever a função e o comportamento de um sistema, permitindo a descrição de comportamento complexo através de conceitos e entidades abstratas [NIS99]. De forma geral, segundo Alencar [ALE99], uma linguagem de especificação formal é composta por:

- Uma sintaxe que define a notação própria da especificação e que permite que os requisitos sejam interpretados de forma única;
- Uma semântica que ajuda a definir o universo de objetos que serão utilizados na descrição do sistema; e
- Um conjunto de relações que define as regras que indicam que objetos satisfazem a especificação.

No caso específico do PROSOFT, as especificações formais constituem uma base sólida que guia a implementação de protótipos ou sistemas de software completos.

---

[25] Constituindo, segundo Bardohl [BAR00], um editor gráfico dirigido por sintaxe.

[26] Em específico, a experiência do autor na utilização combinada do método Algébrico [WAT91] e LOTOS [ISO87] na especificação de um *middleware* para aplicações de trabalho cooperativo síncrono para ambiente PROSOFT [REI98].

Além disso, segundo Wang e King [WAN2000], o campo de automação de processos de software é um terreno fértil para o desenvolvimento de soluções baseadas em métodos formais: a literatura especializada apresenta experiências com diferentes formalismos, incluindo LOTOS [YAS94], Redes de Petri [BAN94], CCS [JAC98], o Método Algébrico [MUR96] e Gramáticas de Grafos (GG) [KRA98].

As seções a seguir apresentam as principais características do PROSOFT-Algébrico (seção 4.1.3) e das Gramáticas de Grafos (seção 4.1.4), formalismos esses usados na especificação dos principais componentes do modelo proposto. Alguns argumentos foram decisivos para a escolha dos formalismos citados. De uma forma geral, a especificação formal permitiu a descrição de modelos de dados e algoritmos complexos de uma forma compacta e em alto nível de abstração. A adoção do PROSOFT-Algébrico (para especificação dos tipos abstratos de dados) proporcionou uma derivação direta para implementação no ambiente PROSOFT-Java[27], visto que há uma correspondência semântica entre os elementos usados na especificação e implementação dos componentes de software descritos nesse paradigma.

Por sua vez, a escolha de se usar GG foi decisivamente influenciada pela possibilidade de se construir a sintaxe abstrata da linguagem visual proposta através de um formalismo gráfico baseado em regras que utiliza os mesmos ícones da sintaxe concreta da APSEE-PML. Partes significativas da especificação em GGs ainda foram submetidas ao simulador *Attributed Graph Grammar System* (AGG, desenvolvido pela *Technischen Universität Berlin*) [AGG2002], o que permitiu constatar certas propriedades do modelo logo no início do desenvolvimento.

Vale ressaltar que, embora o uso de métodos formais possibilite a realização de provas e verificações matemáticas acerca de propriedades do modelo, tal característica não constitui objetivo do trabalho aqui apresentado sendo, entretanto, um recurso importante que pode ser aproveitado em trabalhos futuros.

4.1.3 PROSOFT-Algébrico

O formalismo denominado PROSOFT-Algébrico (descrito em [NUN92] e [NUN94]) permite a descrição de tipos abstratos de dados através de um paradigma algébrico baseado em objetos. Segundo Nunes [NUN94], esse paradigma "adota uma abordagem *data-driven* para o desenvolvimento de software", isto é, estimula o desenvolvimento de software inicialmente através da composição dos tipos de dados necessários.

O PROSOFT-Algébrico é, portanto, uma técnica orientada a propriedades que define um objeto matemático baseado nas relações entre as operações desse objeto [COH86]. Cada tipo de dados é instanciado a partir de um ATO - Ambiente de Tratamento de Objetos. Assim, a figura 4.1 apresenta o esquema gráfico de um sistema arbitrário de software desenvolvido sob o paradigma do PROSOFT, sendo composto por um número de ATOs.

Cada ATO é dividido em três partes como mostrado na figura 4.1. A primeira parte tem como objetivo definir uma instanciação do tipo através de uma linguagem gráfica (**classe**). A instanciação é uma árvore cujas folhas são referências a outros tipos

[27] PROSOFT-Java é a denominação para o ambiente escolhido para experimentação do modelo APSEE, sendo descrito na seção 5.1.

de dados e cujos nodos são tipos de dados compostos. A segunda parte especifica a funcionalidade (**interface**) das (novas) operações. A terceira parte define a semântica das **funções** (axiomas) que atuam sobre o objeto.

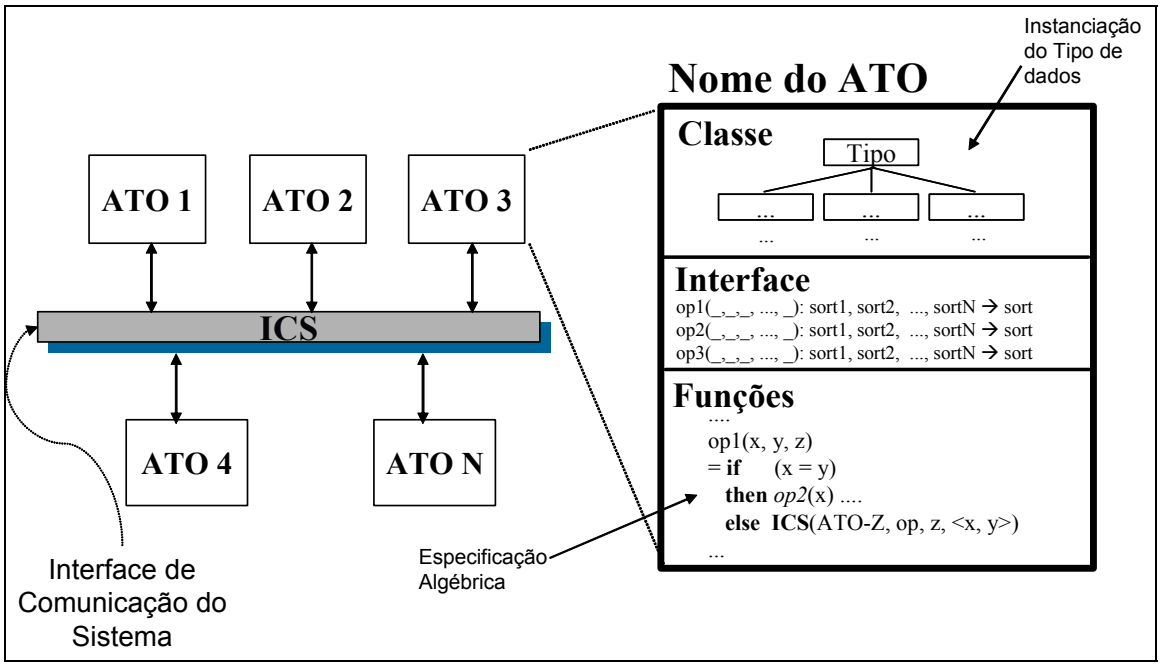

FIGURA 4.1 - Composição de ATOs Algébricos na descrição de software no PROSOFT

Os tipos de dados PROSOFT são classificados como primitivos (Integer, String, Real, Date e Boolean), compostos (Conjunto, Lista, Mapeamento, Registro e União Disjunta) ou definidos pelo usuário. Os tipos primitivos e compostos são incluídos automaticamente em cada ATO e, portanto, podem ser imediatamente usados. Quando uma operação de um ATO faz referência a uma operação de um outro ATO, então o mecanismo de inclusão não se aplica. A referência nesse caso deve ser feita através do mecanismo de envio de mensagem usando a Interface de Comunicação do Sistema (ICS):

ICS(<nome do ATO>, <operação>, <seletor>, <argumentos*>)[28]

Cada ATO trata apenas de termos do *sort* definido pelo próprio ATO. Assim, se um ATO mandar uma mensagem via ICS, então, o ATO receptor da mensagem procura a operação contida na mensagem, substitui os parâmetros pelos seus respectivos argumentos, aplica a operação e devolve, novamente via ICS, para o ATO emissor, o resultado. Cada operação do ATO possui um parâmetro que é do tipo do *sort* definido no ATO. Como uma operação pode possuir várias definições, um dos argumentos do tipo do *sort* definido pelo ATO - denominado seletor - é usado para encontrar a operação.

Cada ATO especifica algebricamente um tipo abstrato de dado. Qualquer termo desse tipo é chamado de objeto e, segundo Nunes [NUN94], não é similar com o conceito de objeto das linguagens de programação orientadas a objetos[29]. Segundo

[28] Onde, argumentos* representa os argumentos da operação ordenados em uma lista.

[29] Vale ressaltar que o conceito de Classe empregado no PROSOFT-Algébrico difere do conceito de classes em linguagens de programação orientadas a objetos. Segundo Körbes [KÖR96], uma classe

Daudt [DAU92], “as instâncias de uma classe (objetos) são entidades passivas, que não têm capacidade de responder a estímulos (mensagens)”. Portanto, os termos PROSOFT armazenam dados mas não podem manipulá-los: toda manipulação de dados é descrita através de funções definidas no escopo de um ATO.

No PROSOFT-Algébrico é usada a linguagem gráfica para instanciação de tipos de dados derivada da notação de Jackson [JAC83], conforme exemplificado nas figuras a seguir. Assim, a classe facilita o desenvolvimento das especificações algébricas, fornecendo uma notação gráfica que é uma instanciação do tipo definido. Uma classe representa a hierarquia de composição de tipos de dados, aonde o nodo superior descreve o nome do Tipo definido e os nodos-folha são relacionados aos tipos primitivos ou definidos pelo usuário.

A figura 4.2 apresenta um exemplo de instanciação textual e gráfica de um tipo abstrato Lista. Nesse exemplo, *ATOLISTOFPARAMETERS*, *LISTS*, e *PARAMETER* são especificações e *AtoListOfParameters*, *List*, *Item*, *Parameter* e *AtoPolExpression* são *sorts*. *AtoPolExpression* é um *sort* dado definido pela especificação *Item*. O objetivo da figura é mostrar uma instanciação da especificação LISTS que define o *sort* List, com a especificação ITEM que define o *sort AtoPolExpression*, usando-o no lugar do parâmetro formal *Component*. Como *AtoListOfParameters* instancia uma lista, os construtores e operadores para Listas são válidos (apresentados no Anexo 1), e permitem a manipulação dos itens de dados inseridos na lista instanciada.

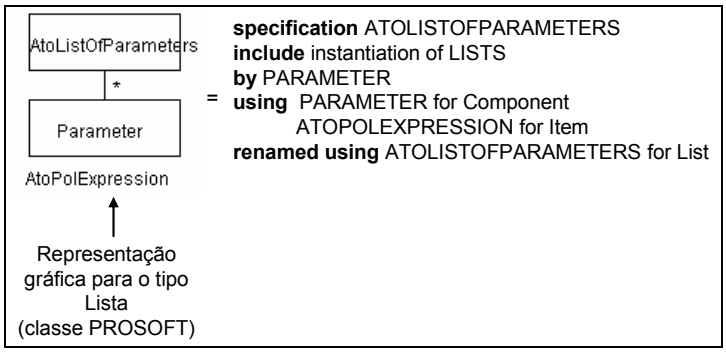

FIGURA 4.2 - Exemplo da representação gráfica usada na composição de tipos de dados PROSOFT

Exemplos adicionais com os outros tipos de dados compostos (conjunto, mapeamento, união disjunta e registro) são apresentados na figura 4.3.

PROSOFT é uma estrutura de dados que corresponde ao que normalmente é chamado de atributos nas linguagens orientadas a objetos.

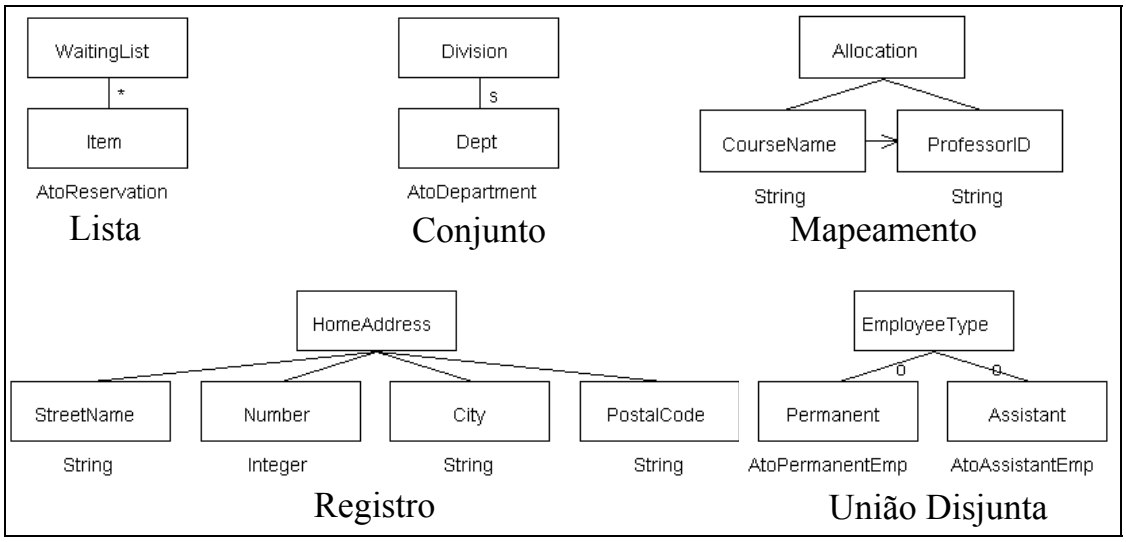

FIGURA 4.3 - Notação gráfica para definição de tipos compostos

Comparando-se a especificação algébrica convencional (como o método descrito por David Watt [WAT91]) e o PROSOFT-Algébrico, pode-se dizer que a assinatura da especificação corresponde à instanciação e à interface do ATO. Os axiomas, por sua vez, correspondem às operações do ATO, aonde segundo Moraes [MOR97] "a funcionalidade de cada operação é definida na forma de equações". A vantagem fundamental do PROSOFT-Algébrico é que, em função do mecanismo de inclusão automática de tipos compostos e primitivos em um ATO PROSOFT, não é necessário explicitar a "importação" das especificações dos tipos externos como é feito tradicionalmente nos métodos algébricos. Portanto, a comunicação entre ATOs é feita explicitamente através da ICS, em um estilo muito similar à troca de mensagens do paradigma de objetos, aonde os componentes são encapsulados e descrevem interfaces bem definidas.

Finalmente, ressalta-se o papel desempenhado por esse formalismo na especificação do modelo proposto: Classes foram construídas para descrever os seus tipos de dados (na seção 4.2.1), enquanto que o mecanismo de interpretação de Políticas Estáticas foi totalmente especificado com o PROSOFT-Algébrico (seção 4.4).

4.1.4 Gramáticas de Grafos (GGs)

As pesquisas na área de Gramáticas de Grafos iniciaram nos anos 1970 e métodos, técnicas e resultados nesta área já foram estudados e aplicados em uma grande variedade de campos da informática [RIB2000] [DEH2000]. Gramáticas de grafos estão baseadas no processo de transformação que um grafo pode sofrer em função de um conjunto de regras previamente definidas. Assim, um sistema de software é especificado em termos de estados (que são modelados por grafos), e mudanças de estados (modeladas por regras ou derivações).

Existem várias abordagens diferentes para Gramáticas de Grafos. A abordagem seguida por esse trabalho é algébrica, em que são fornecidos mecanismos de tipagem de dados. A notação usada é derivada do alfabeto real da linguagem concreta para definição de *templates*. Um grafo tipo [DEH2000] é usado para representar os tipos de nodos e todos os arcos do sistema, sendo que o grafo inicial e os derivados das transformações devem ser compatíveis com o grafo tipo.

O fato de Gramáticas de Grafos serem formais e intuitivas ao mesmo tempo, e de poderem tratar com simplicidade aspectos de concorrência e distribuição de sistemas influenciou na sua escolha para a definição das regras sintáticas para a composição de *templates*.

No projeto APSEE, as GG foram usadas por Lima Reis para especificação do mecanismo de execução de processos [LIM2002d]. Portanto, em virtude da proposital semelhança entre os tipos de dados usados para descrição de processos abstratos (*templates*, aqui propostos) e executáveis (processos instanciados, propostos por [LIM2002d]), esse texto evidencia na seção 4.3 os elementos comuns e os detalhes específicos das especificações citadas.

4.2 Especificação dos tipos básicos para *ProcessTemplates*

A descrição do meta-modelo que define *ProcessTemplates* é apresentada nesta seção. Assim, são descritas as classes PROSOFT envolvidas na definição dos tipos abstratos de dados necessários, e são definidas as funções para generalização e adaptação de *templates* em processos instanciados.

4.2.1 Os tipos de dados

Esta seção descreve a especificação dos tipos abstratos de dados usados para armazenar instâncias de *Templates* de Processos de Software, apresentando de uma forma mais detalhada aquilo previamente descrito na seção 3.5 do capítulo anterior.

A experiência do grupo PROSOFT com o uso do formalismo Algébrico (descrita em diferentes dissertações e teses citadas por Rangel [RAN2002]) sugere que, na especificação de um sistema complexo, as classes devem ser particionadas em tipos de dados que não tenham grande profundidade (i.e., com número de nodos reduzido). A decomposição em diferentes ATOs também é justificada por simplificar as funções especificadas, as quais são baseadas em chamadas ICS para os seus componentes.

4.2.1.1 O tipo *APSEE*

Como ressaltado nas seções anteriores, o componente APSEE-Reuse (e, conseqüentemente, os *Templates*) faz parte da infraestrutura APSEE para automação de processos de software. A figura 4.4 apresenta, então, a estrutura de dados principal do ambiente APSEE[30]. Nesta classe, são agrupados em um registro os diferentes elementos usados, a saber:

- *Configuration*: descreve informações sobre o armazenamento físico do repositório de dados APSEE, além da identificação única do sistema que permite o acesso remoto;
- *ApseeTypes*: armazena as hierarquias de tipos para componentes APSEE;
- *Organization*: descreve os agentes, cargos e recursos existentes em uma organização específica;

[30] Os nomes de tipos, assim como os seus atributos são apresentados em itálico neste texto. Convém observar que o prefixo Ato - automaticamente inserido na construção de tipos definidos pelo usuário no PROSOFT – é suprimido do texto quando da referência ao nome de classes.

- *PKnowledge*: descreve as métricas, habilidades e estimativas (i.e., a base de conhecimento acerca da execução de processos);
- *Processes*: descreve os modelos de processos instanciados e executados na organização;
- *SwArtifacts*: armazena referências para os artefatos concretos manipulados pelos processos executados;
- *Tools*: descreve as ferramentas usadas nas atividades dos processos;
- *Policies*: descreve as Políticas Estáticas e de Instanciação;
- *ProcessReuse*: descreve os componentes do APSEE-Reuse, relacionadas com o trabalho aqui proposto.

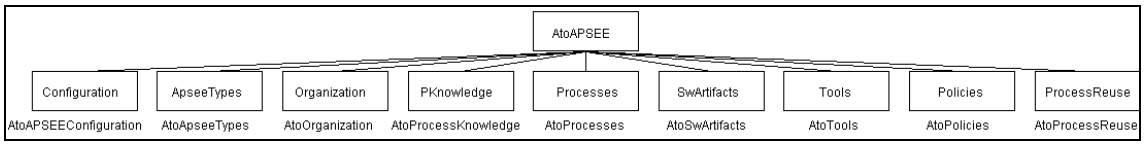

FIGURA 4.4 - A classe APSEE

4.2.1.2 Os tipos *ApseeTypes* e *Types*

A figura 4.5 apresenta as classes *ApseeTypes* e *Types*, usadas para armazenar os nodos de hierarquias de tipos[31]. Assim, no mapeamento de *ApseeTypes*, o domínio - representado pelo atributo *TypeName* - identifica o nome da hierarquia de tipos, o que é mapeado para uma árvore de tipos, isto é, o nodo imagem (*TypeTree*) é refinado no tipo *Types*. O tipo *Types*, por sua vez, define um novo mapeamento em que o domínio denota um identificador único para o nodo existente na árvore (*Id*), cuja imagem inclui o Nome (atributo *Name*), o identificador do super-tipo (i.e., o tipo ancestral identificado pelo atributo *Super_Id*), uma descrição textual (*Description*) e um *flag* booleano que identifica se o tipo é específico da organização de software instalada (*UserDefinedType*[32]).

[31] A seta pontilhada - tais como aquela existente entre o nodo *TypeTree* e o tipo *Types* na figura 4.5 - não faz parte da notação de classe PROSOFT e foi incluída em diferentes figuras deste texto com o objetivo de facilitar o entendimento do relacionamento das classes apresentadas.

[32] O atributo *UserDefinedType* é útil para distinguir aqueles nodos das hierarquias de tipos que são específicos de uma organização.

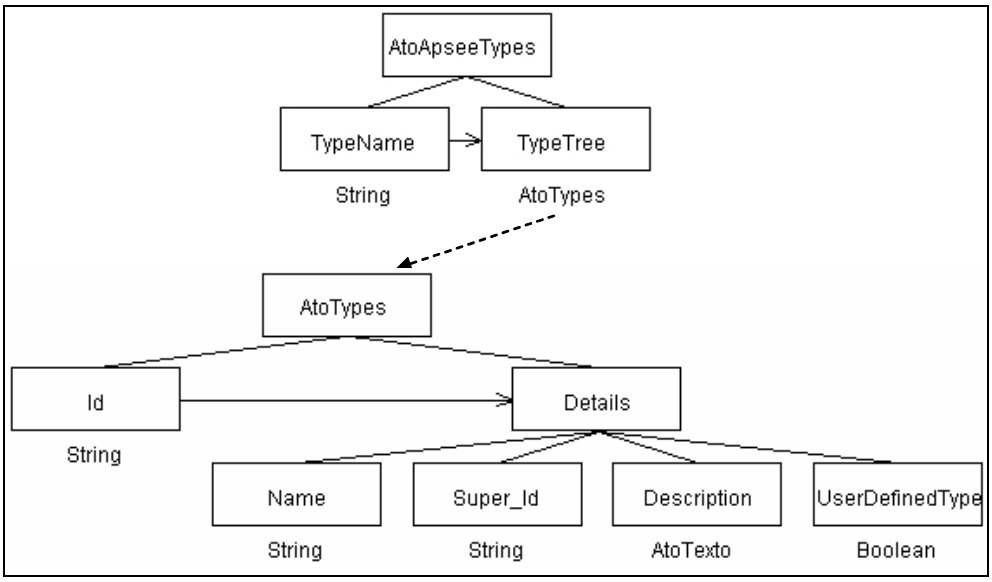

FIGURA 4.5 - As classes *ApseeTypes* e *Types*

4.2.1.3 O tipo *ProcessReuse*

O tipo *ProcessReuse* é o principal da especificação aqui apresentada, sendo descrito pela classe na figura 4.6. Os componentes do registro *ProcessReuse* são:

- *PolMethods*: descreve os métodos disponíveis para a especificação de Políticas de Processos. A classe PolMethods está descrita na seção 4.4.2);
- *ApseeTemplates*: contém todas as instâncias de *templates* descritas;
- *Implementations*: relaciona os *templates* às suas implementações (i.e., instâncias de processos);
- *Origin*: descreve o relacionamento entre *templates* e processos de origem (para *templates* obtidos a partir de generalização);
- *AbstractArtifacts*: descreve os artefatos de software abstratos usados na definição das atividades de um *template*;
- *DefaultAdaptation*: define o tipo de restrição corrente para os *templates*, conforme definido pela classe *TemplateAdaptation* na figura 4.7.

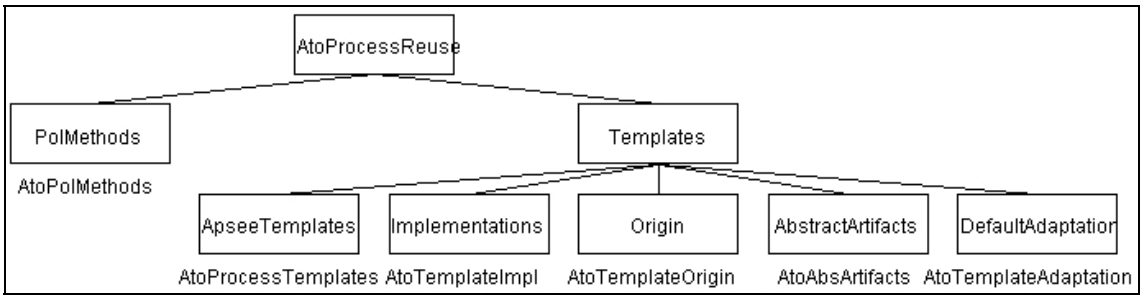

FIGURA 4.6 - A classe *ProcessReuse*

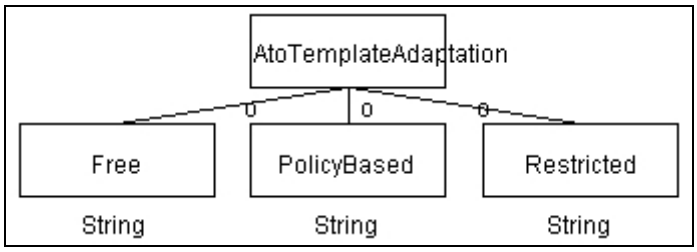

FIGURA 4.7 - A classe *TemplateAdaptation*

4.2.1.4 Os tipos *ProcessTemplates* e *AbsProcessModel*

Esta seção descreve os tipos de dados envolvidos com a definição do pacote *ProcessModel* da dimensão Processo do modelo APSEE-Reuse (seção 3.5.1), descrevendo os tipos *ProcessTemplates* e *AbsProcessModel*.

a) O tipo *ProcessTemplates*

O tipo *ProcessTemplates* está descrito pela figura 4.8. O *template* é a unidade básica utilizada no modelo APSEE-Reuse. Assim, um *template* possui uma identificação única (atributo *ID*), um nome (atributo *Name*), uma descrição textual (*Description*), um tipo (atributo *TypeID*, que se refere a um tipo da hierarquia de atividades), o conjunto de domínios de aplicação (atributo *AppDomains*) e o modelo de processo associados (*ProcessModel*).

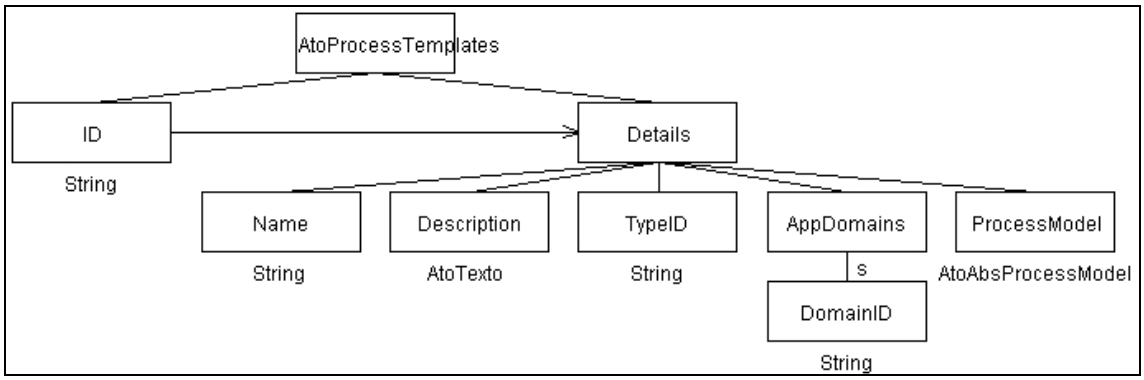

FIGURA 4.8 - A classe *ProcessTemplates*

b) O tipo *AbsProcessModel*

Um modelo de processo abstrato de software é descrito pelo tipo *AbsProcessModel*. O tipo *AbsProcessModel* (figura 4.9) é descrito por um conjunto de atividades abstratas (atributo *Activities*) e um conjunto de conexões (atributo *Connections*). O registro *Policies* define o conjunto de Políticas habilitadas no modelo (atributo *PolicySets*) e um *flag* que descreve se as Políticas Estáticas habilitadas são satisfeitas (atributo *StaticOK*). Finalmente, o atributo *Requirements* descreve os requisitos textuais para o modelo de processo.

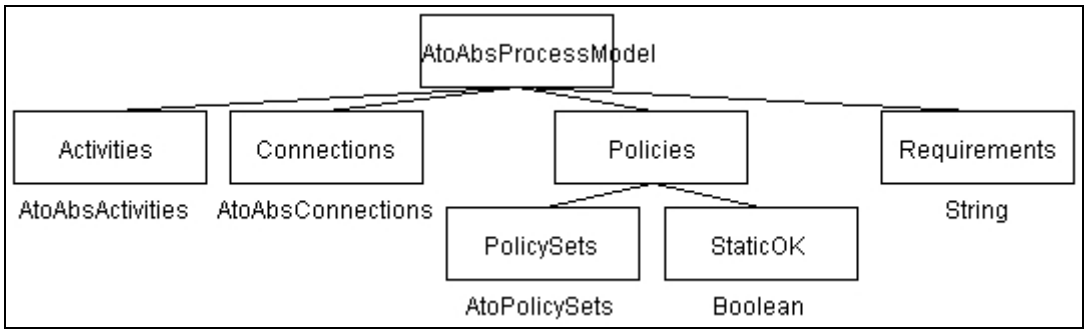

FIGURA 4.9 - A classe *AbsProcessModel*

4.2.1.5 Os tipos *AbsActivities*, *ActDescription*, *NormalAbsDesc* e *AutomaticAbsDesc*

Esta seção descreve os tipos PROSOFT definidos para lidar com as atividades de um *template*, correspondendo ao pacote *Activities* previamente descrito na seção 3.5.1 desse texto. Assim, são descritas em seguida as classes: *AbsActivities*, *ActDescription*, *NormalAbsDesc* e *AutomaticAbsDesc*.

a) O tipo *AbsActivities*

No modelo APSEE, as atividades constituem a unidade básica de execução de um processo. Um *template* é composto por atividades ditas abstratas, ou seja, atividades descritas em alto nível de abstração, sem se preocupar com os detalhes específicos de instanciação do processo. Portanto, uma atividade abstrata é descrita na figura 4.10 e possui o seguintes atributos:

- Um identificador único (atributo *ID*), fornecido pelo usuário (projetista do *template*);
- O nome da atividade (atributo *Name*);
- O tipo da atividade (atributo *TypeID*);
- A descrição da atividade (atributo *Description*);
- As Políticas Habilitadas (atributo *PolicySets*);
- Um sinalizador indicando se as políticas estáticas habilitadas são satisfeitas pela atividade (atributo *StaticOK*);
- O atributo *Position*, descrevendo a coordenada que indica a posição geométrica da atividade em um modelo.

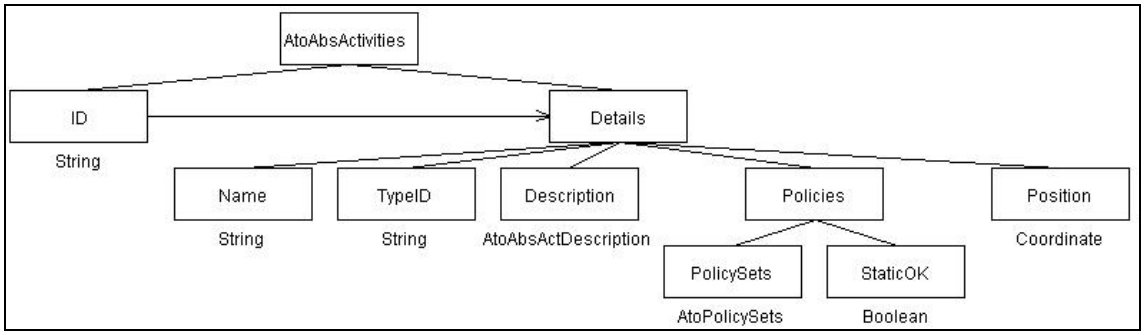

FIGURA 4.10 - A classe *AbsActivities*

b) O tipo *ActDescription*

A descrição detalhada para uma atividade é definida através do tipo *ActDescription*. Como ilustrado pela figura 4.11, uma atividade abstrata é descrita como uma atividade folha (*plain*), um fragmento (alternativa *fragment*) ou indefinida (*undefined*, quando nenhuma das alternativas é selecionada). A figura 4.11 ainda inclui, à direita da classe mencionada, um exemplo de um processo arbitrário composto por um fragmento (rotulado com *a*), e atividades-folha (todas as demais).

A definição de uma atividade folha exige que o usuário forneça os requisitos (atributo *requirements*) e a descrição, a qual define uma atividade-folha como Normal ou Automática (atributo *Model*). O nodo *Model* é opcional pois, em tempo de modelagem é possível que o usuário primeiramente defina somente os requisitos da atividade para posteriormente apresentar sua descrição detalhada como Normal ou Automática.

A referência aos fragmentos, por sua vez, é definida por uma alternativa entre um outro *template* (cujo ID deve ser fornecido para o nodo *TemplateID*) ou na forma recursiva de um novo modelo de processo (nodo *ProcessModel*, com referência ao tipo *AbsProcessModel*).

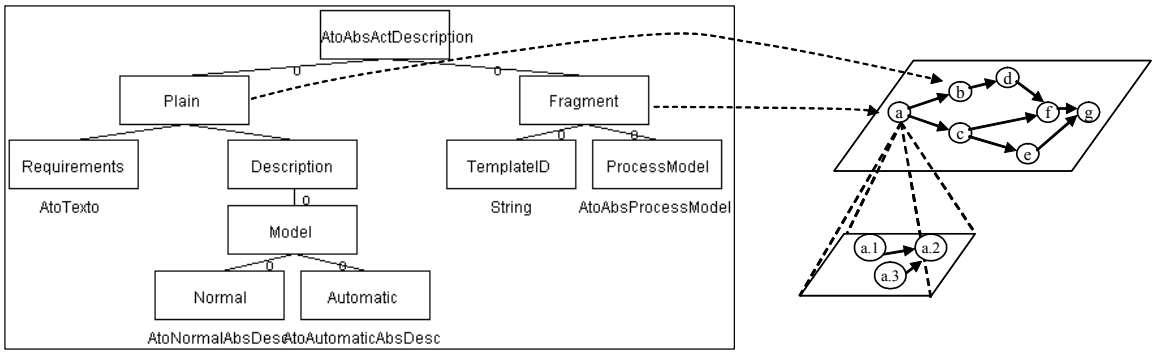

FIGURA 4.11 - A classe *AbsActDescription*

c) O tipo *NormalAbsDesc*

As atividades normais são aquelas projetadas para envolver pessoas desempenhando atividades criativas no desenvolvimento de software. Assim, o tipo *NormalAbsDesc* (figura 4.12) é caracterizado por definir o aspecto humano no item *People*, e o conjunto de tipos de recursos a serem alocados (atributo *Resources*). Os artefatos consumidos e produzidos pela atividade são descritos no atributo *Artifacts*. Por fim, o *Script* descreve uma sugestão de texto a ser apresentado aos agentes durante a realização da atividade.

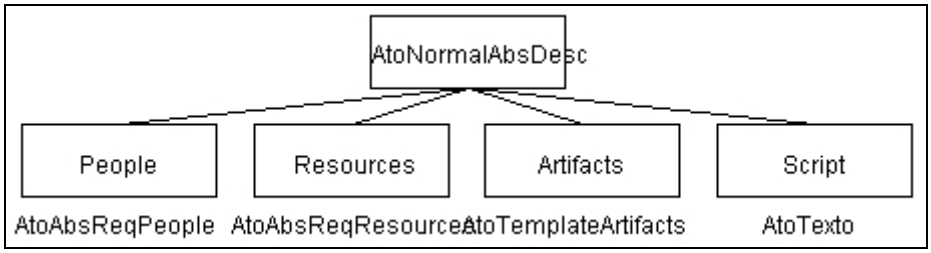

FIGURA 4.12 - A classe *NormalAbsDesc*

d) O tipo *AutomaticAbsDesc*

As atividades automáticas são prescritas em um *template* através do tipo *AutomaticAbsDesc*. Assim, como descrito pelo diagrama da figura 4.13, o projetista de atividades automáticas deve fornecer:

- O tipo de ferramenta a ser invocado (identificado pelo atributo *ToolTypeID*);
- A lista de parâmetros a serem fornecidos para a ferramenta (relacionados com artefatos concretos ou abstratos fornecidos de entrada para a atividade e descritos pelo nodo *InputArtifact*);
- O resultado obtido, descrevendo um artefato abstrato produzido pela atividade automática (nodo *OutputArtifact*).

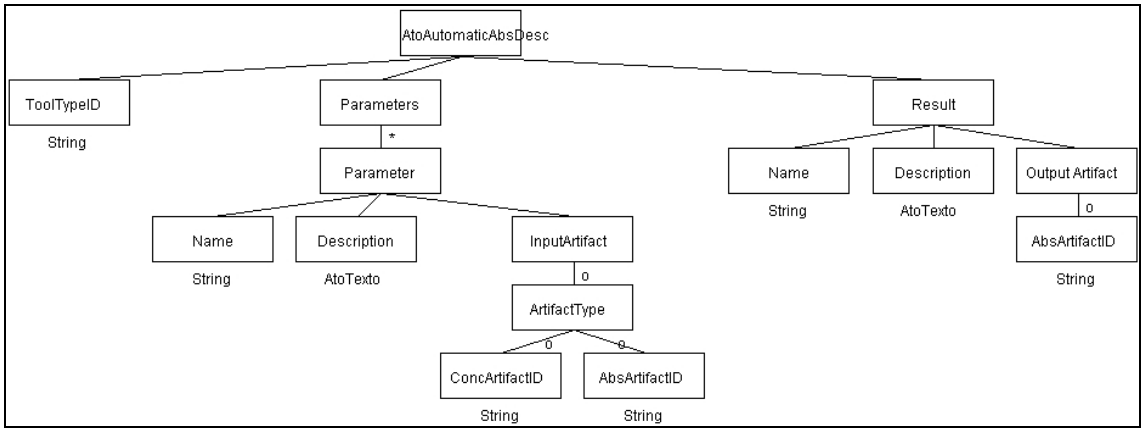

FIGURA 4.13 - A classe *AutomaticAbsDesc*

4.2.1.6 Conexões de Atividades

Conforme descrito anteriormente pelo pacote *Connections* da seção 3.5.1, as conexões possuem grande importância na definição da dimensão Processo de um *template*. As conexões estabelecem associações temporais ou artefato para duas (para conexões simples) ou mais (para conexões múltiplas) atividades definidas na composição de um *template*.

As conexões definidas para um *template* são derivadas das estruturas existentes para a definição de processos executáveis. Assim, quando necessário, o texto a seguir evidencia os componentes reutilizados a partir de outros trabalhos desenvolvidos no projeto *APSEE*.

a) O tipo *AbsConnections*

O tipo *AbsConnections* - descrito pelo diagrama da figura 4.14 - é associado a um modelo de processo abstrato de software (*AbsProcessModel*). As conexões para *templates* são descritas conforme abaixo:

- Um identificador único - fornecido automaticamente pelo sistema - servindo para individualizar o acesso às diferentes conexões contidas no mapeamento (atributo *ID*);
- O tipo de conexão (atributo *TypeID*);
- A definição da conexão em simples, múltipla ou de artefato (atributo *Kind*);

- A coordenada que indica o posicionamento da conexão no diagrama de *template* (atributo *Position*).

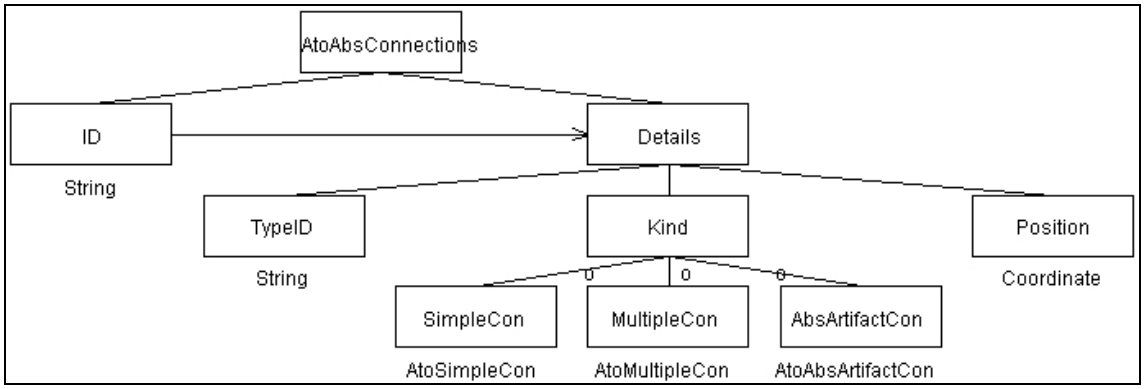

FIGURA 4.14 - A classe *AbsConnections*

b) Os tipos *SimpleCon* e *Dependency*

O tipo *SimpleCon* (contração de *SimpleConnection*) descreve as conexões que envolvem especificamente duas atividades: uma única atividade origem e outra de destino. O tipo *SimpleCon* foi desenvolvido por Lima Reis em [LIM2002d], e sua classe foi reproduzida na figura 4.15. A conexão simples é, portanto, um registro com os seguintes atributos:

- *From_Act_Id*: associa o identificador da atividade de origem;
- *To_Act_Id*: associa o identificador da atividade de destino;
- *Type*: que descreve a conexão como de feedback ou de seqüência. No caso de uma conexão simples de seqüência, esta é relacionada com uma instância do tipo *Dependency* (descrevendo o tipo de dependência temporal como *end-start*, *start-start* ou *end-end*).

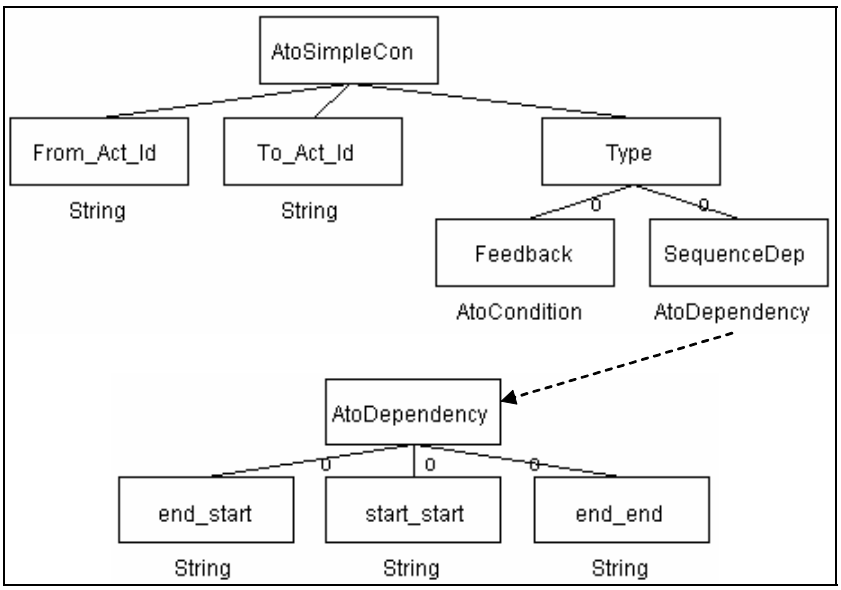

FIGURA 4.15 - A classe *SimpleCon* [LIM2002d]

b) O tipo *MultipleCon*

As conexões múltiplas estão disponíveis em dois tipos:

- *Branch*, possuindo uma origem (uma atividade ou uma conexão múltipla que é conectada à múltiplos destinos (atividades ou conexões múltiplas);
- *Join*, definindo vários pontos de origem (atividades ou conexões múltiplas) e um único destino (uma atividade ou uma conexão múltipla).

A classe MultipleCon é apresentada na figura 4.16. Em virtude do tamanho das classes, os tipos *Join* e *Branch* são descritos no anexo 2 (figura A1).

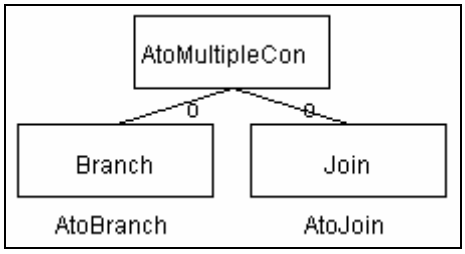

FIGURA 4.16 - A classe *MultipleCon* [LIM2002d]

4.2.1.7 Os tipos *TemplateArtifacts, AbsArtifacts, SwArtifacts e AbsArtifactCon*

Em uma atividade abstrata de *template*, a dimensão Software (previamente descrita na seção 3.5.2) é definida pelos tipos *TemplateArtifacts*, *AbsArtifacts*, *SwArtifacts* e pela conexão de artefato (*AbsArtifactCon*), descritos a seguir.

a) O tipo *TemplateArtifacts*

Uma atividade normal define artefatos de entrada e de saída, os quais são armazenados como objetos da classe *TemplateArtifacts*. Os artefatos de entrada podem ser concretos (i.e., artefatos previamente instanciados que não são modificados pelos projetos) e/ou abstratos (aqueles que são instanciados durante a execução dos processos). Por sua vez, os artefatos de saída para uma atividade somente podem ser definidos como abstratos.

A figura 4.17 apresenta a classe *TemplateArtifacts*, aonde o nodo *InputArtifacts* relaciona um conjunto de identificadores de artefatos de entrada, e *OutputArtifacts* define o conjunto de artefatos de saída.

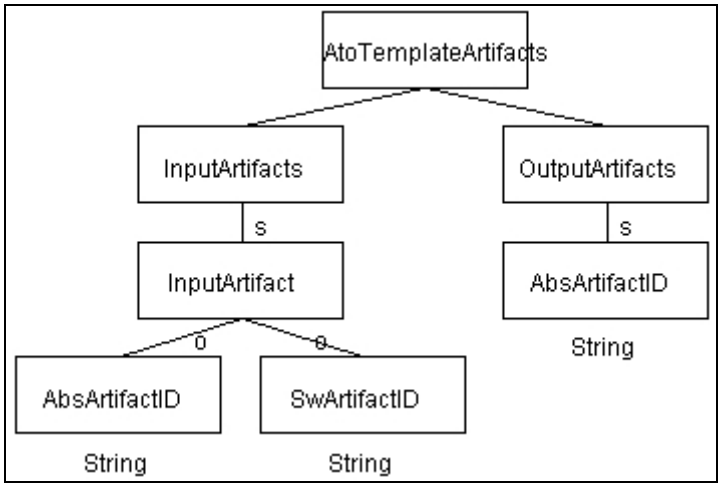

FIGURA 4.17 - A classe *TemplateArtifacts*

b) *AbsArtifacts* e *SwArtifacts*

Os artefatos abstratos e concretos descritos em um *template* são armazenados nos tipos *AbsArtifacts* e *SwArtifacts*, respectivamente.

O tipo *AbsArtifacts* aparece exclusivamente na definição de *templates*. Conforme descrito pela figura 4.18, a definição de um artefato abstrato - um artefato que é instanciado durante a execução dos processos adaptados a partir do *template* associado - envolve a descrição do seu identificador único (*ID*), do tipo de artefato envolvido (*TypeID*), do seu nome e uma descrição textual.

Em um modelo de processo de software podem coexistir artefatos abstratos e concretos. Assim, em alguns casos o projetista de processos pode achar necessário definir um relacionamento que indique que um artefato abstrato deve ser produzido por derivação a partir de outro existente. Isto é útil, por exemplo, para denotar que o código-fonte de um programa deve ser definido por derivação da especificação formal dos seus algoritmos. Assim, o atributo *DerivedFrom* é definido para explicitar o relacionamento entre artefatos.

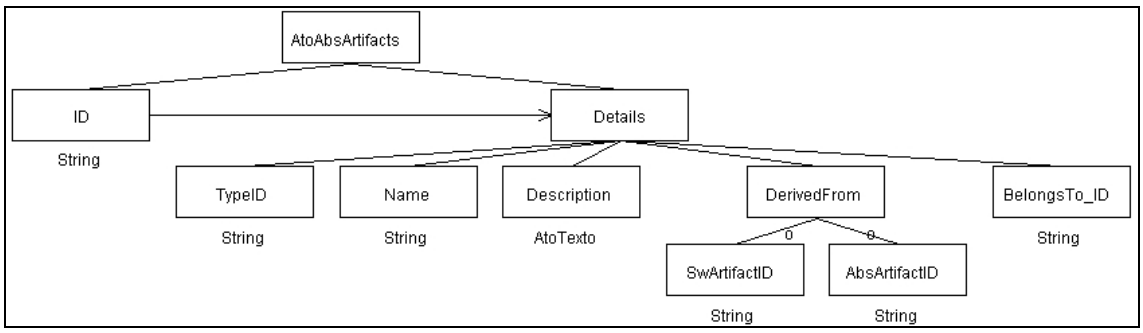

FIGURA 4.18 - A classe *AbsArtifacts*

A classe *AbsArtifacts* foi nitidamente inspirada a partir da classe *SwArtifacts*, descrita por Lima Reis em [LIM2002d] e reproduzida na figura 4.19. A principal diferença na descrição de artefatos concretos consiste na referência ao tipo *CoopObject*, o qual descreve os detalhes de persistência do objeto no repositório distribuído do PROSOFT-Cooperativo [REI98].

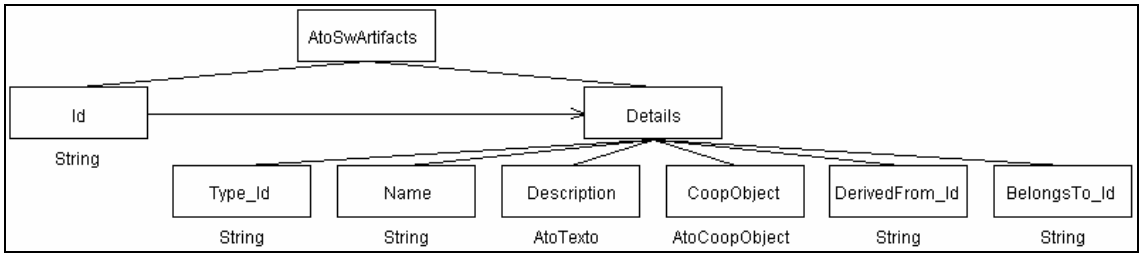

FIGURA 4.19 - A classe *SwArtifacts* [LIM2002d]

c) *AbsArtifactCon*

Uma conexão de artefato - descrita pela figura 4.20 - define a produção e consumo de artefatos em um *template*. Assim, o tipo *AbsArtifactCon* é definido a seguir:

- O nodo *From* descreve as atividades que produzem o artefato descrito;
- O nodo *To* descreve o destino, ou seja, as atividades ou conexões múltiplas que recebem o artefato em questão;

- O nodo *Artifact* descreve o tipo do artefato (*ArtifactTypeID*), e o seu *ID* no repositório de artefatos concretos ou abstratos.

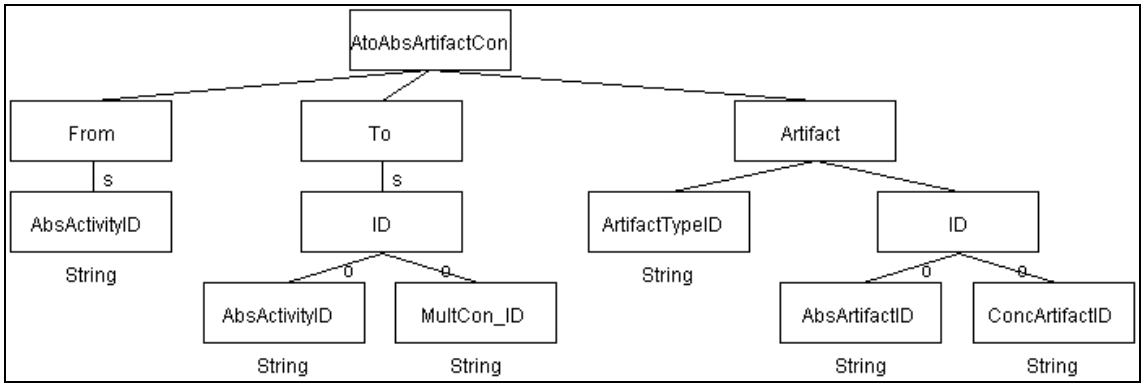

FIGURA 4.20 - A classe *AbsArtifactCon*

4.2.1.8 O tipo *AbsReqPeople*

A dimensão Pessoal - previamente descrita na seção 3.5.3 - é representada aqui através do tipo *AbsReqPeople*, o qual tem a sua classe apresentada na figura 4.21. O tipo *AbsReqPeople* é associado à atividade *Normal* de um *template*.

O tipo *AbsReqPeople* define duas listas: uma lista de tipos de Cargos (nodo *RoleTypes*) enquanto que a segunda define uma lista de tipos de Grupos (nodo *GroupTypes*). Deve-se ressaltar que o construtor Lista é utilizado ao invés de conjunto com o objetivo de permitir a inclusão de dois ou mais tipos iguais, desde que requerido pelo projetista do *template* (por exemplo, duas instâncias do tipo de Cargo "Programador").

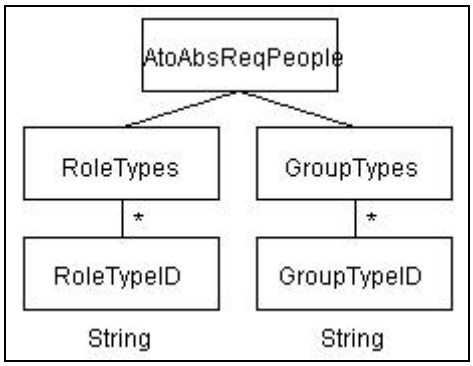

FIGURA 4.21 - A classe *AbsReqPeople*

4.2.1.9 O tipo *AbsReqResources*

A dimensão Recursos descreve a associação entre as atividades de um processo e os recursos de apoio, tal como discutido anteriormente na seção 3.5.5.

Os recursos requeridos por uma atividade *Normal* de um *template* são descritos através da lista definida pela figura 4.22. Um recurso abstrato, portanto, é definido a partir de um tipo de recurso (nodo *ResourceTypeID*) e pelo seu conjunto de Políticas habilitadas (nodo *PolicySets*).

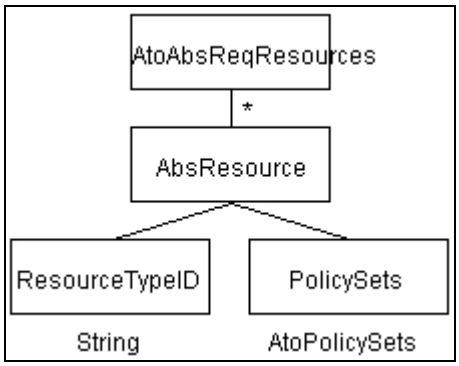

FIGURA 4.22 - A classe *AbsReqResources*

4.2.1.10 O tipo *PolicySets* e *EnabledPolicies*

No modelo de *templates* proposto, Políticas são relacionadas com os modelos de processos, as atividades e outros componentes *APSEE* (ver a seção 3.5.6 para uma descrição detalhada sobre o tema). Desse modo, a classe *PolicySets* (figura 4.23) define o conjunto de Políticas habilitadas para um determinado componente. O tipo *EnabledPolicies* - figura 4.25 - define detalhes sobre cada Política habilitada, definindo além disso uma lista com os parâmetros necessários para a sua execução (atributo *Parameters*).

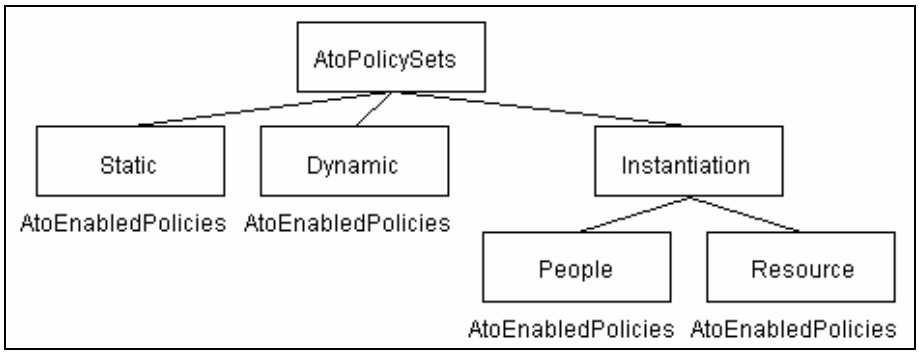

FIGURA 4.23 - A classe *PolicySets*

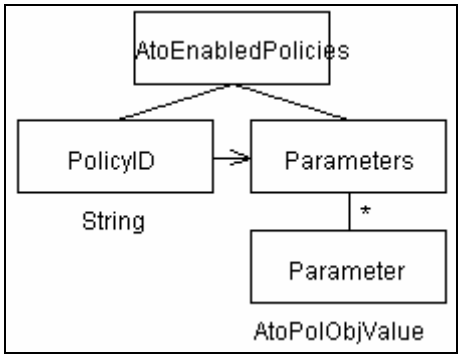

FIGURA 4.24 - A classe *EnabledPolicies*

4.2.2 Generalização de Processos e Adaptação Automática de *Templates*

Esta seção descreve o algoritmo para generalização de processos em *templates*, removendo os detalhes específicos de execução de um processo encerrado, e generalizando as instâncias descritas para tipos genéricos.

A generalização de processos em *templates* é registrada especificamente pelo mapeamento *TemplateOrigin*, descrito na figura 4.25. Tanto o tipo *TemplateOrigin* quanto *TemplateImpl* (responsável por descrever o uso de *templates* na adaptação para modelos de processos concretos de software) são componentes do tipo *ProcessReuse* o qual, por sua vez, está associado ao tipo *APSEE* principal. A classe *PolApseeObj* foi descrita originalmente para a definição de Políticas Estáticas, sendo apresentada na figura 4.57.

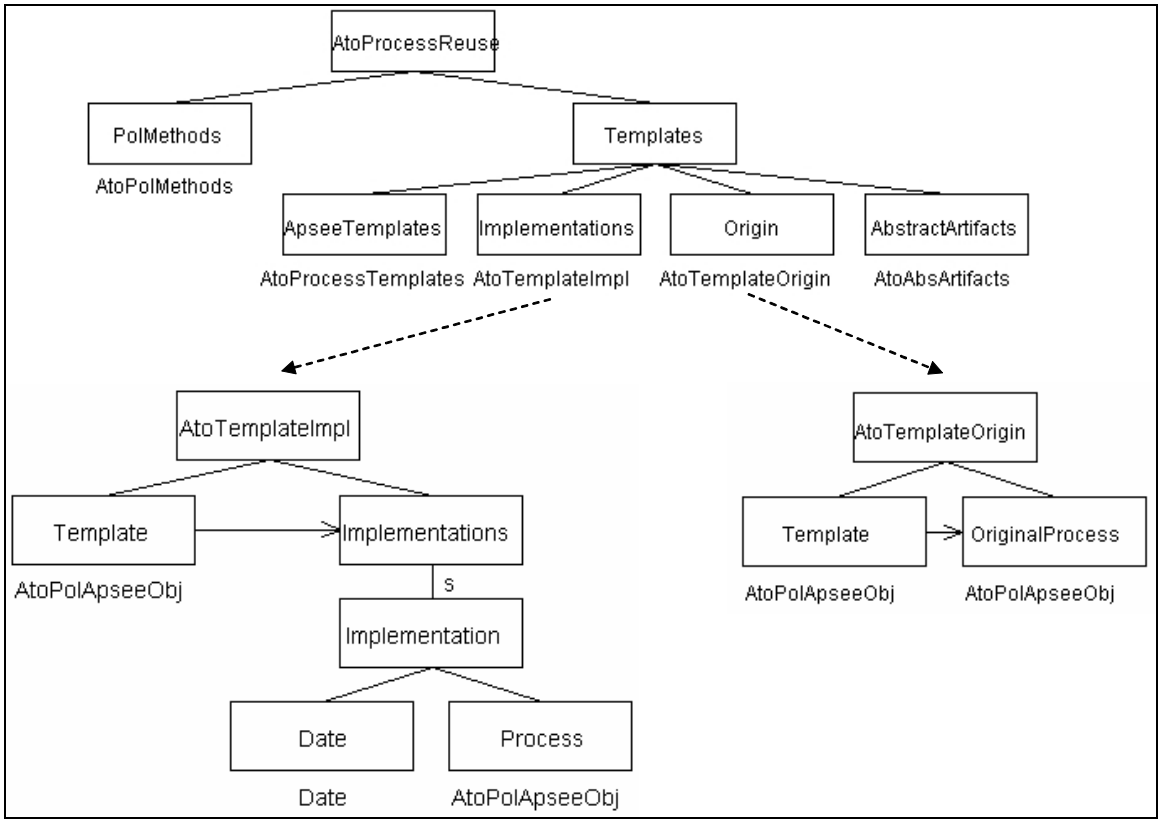

FIGURA 4.25 - As classes *ProcessReuse*, *TemplateImpl* e *TemplateOrigin*

Do ponto de vista dos tipos de dados manipulados, a generalização consiste na extração dos dados existentes em um processo específico (identificado por um *ID* na classe *Processes*) para a criação de um novo elemento no mapeamento *Templates*. A figura 4.26 ilustra o papel desempenhado pelas funções de generalização e adaptação: ambas fornecem instâncias que, obtidas automaticamente a partir do tipo de dados origem, são candidatas a serem ainda aperfeiçoadas em função das necessidades do projetista envolvido.

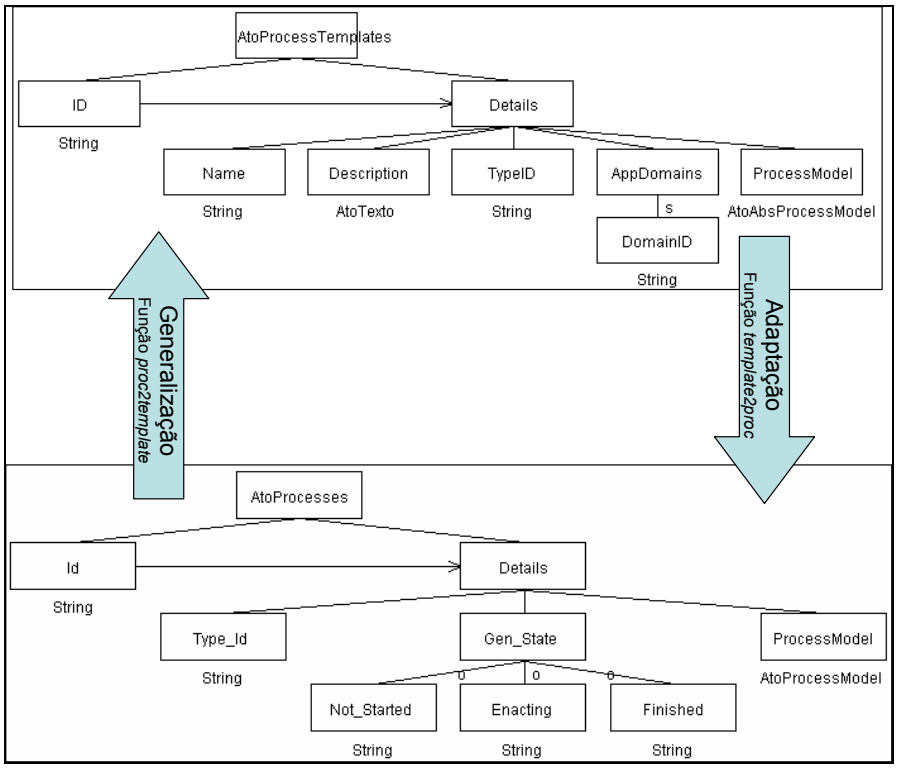

FIGURA 4.26 - Ilustração para as funções de Adaptação e Generalização envolvendo *Templates* e Processos

Esta seção prossegue apresentando a especificação algébrica para a função de generalização de processos encerrados em *templates*, denominada "*proc2template*"[33]. A especificação aqui apresentada usa a notação do PROSOFT-Algébrico descrita por Nunes em [NUN94], e cujos construtores de tipos são fornecidos no Anexo 1. Alguns comentários são incluídos na especificação, usando delimitadores especiais. As classes utilizadas por esta especificação que não pertençam ao pacote APSEE-Reuse aqui proposto estão incluídas no Anexo 5 (que também inclui as classes definidas em [LIM2002d] para a definição de processos executáveis).

Cada figura a seguir apresenta funções relacionadas especificamente com um ATO do sistema. As figuras são ainda apresentadas na ordem em que as funções são chamadas.

A função principal para generalização de *templates* está apresentada na figura 4.27. A função *proc2template* do *AtoAPSEE* recebe como argumentos:

- A instância do ambiente APSEE, contendo todas as informações acerca dos processos modelados e executados;
- O identificador para um processo encerrado *proc-id*;
- O identificador para o *template* a ser criado *template-id*;
- A descrição textual associada ao *template* a ser criado (*desc*);
- O conjunto de domínios de aplicação para o *template* (*appDomains*).

[33] Por simplificação, a função de adaptação "*template2proc*" não é aqui apresentada, resumindo-se à descrição das regras que controlam a adaptação de *templates* para processos na seção 4.3.3.

A semântica descrita para a função envolve a verificação se o *template-id* fornecido é válido - item {1}, se existe um processo com o identificador fornecido - item {2} e se o processo está encerrado - item {3}. Se tais pré-condições forem satisfeitas, é chamada a função *addDerivedTemplate* do ATO *ProcessReuse* para incluir o novo *template* levando em consideração os parâmetros fornecidos no objeto de *ProcessReuse* que já contém os artefatos abstratos necessários (item {4}).

```
AtoAPSEE
Def: proc2template: APSEE, String, String, SetOfString, Text → APSEE
/*
aonde     apsee: ambiente APSEE;
          proc-id : identificador do processo a ser generalizado
          template-id : identificador do template a ser gerado
          desc: descrição
          appDomains: domínios de aplicação
*/
proc2template(apsee, proc-id, template-id, appDomains, desc)
=  if     template-id ∉ domain(ICS(ProcessReuse, getApseeTemplates,                {1}
                    ICS(APSEE, getProcessReuse, apsee))) and
          proc-id ∈ domain(ICS(APSEE, getProcesses, apsee))  and                   {2}
   {3}    ICS(Processes, getGen_State, ICS(APSEE, getProcesses, apsee), <proc-id>) = "Finished"
   then   ICS(ProcessReuse, addDerivedTemplate,
                    ICS(ProcessReuse, addDerivedAbsArtifacts, ICS(APSEE, getProcessReuse, apsee),
                              <apsee, template-id, proc-id>),                      {4}
                    <ICS(Apsee, getProcesses, apsee), template-id, proc-id, appDomains, desc>)
   else   error
```

FIGURA 4.27 - A função *proc2template* do *AtoAPSEE*

A função *addDerivedTemplate* é descrita na figura 4.28. Esta função é responsável por devolver um novo objeto do tipo *ProcessReuse* que inclua o novo *template* obtido por generalização - item {1}. Além disso, uma referência para o novo *template* criado é inserida no objeto *TemplateOrigin* (item {2}). A função *addDerivedAbsArtifacts* - item {3} - é responsável por varrer todos os artefatos existentes no processo original e inserir os artefatos abstratos necessários na construção de um novo objeto do tipo *ProcessReuse*.

```
AtoProcessReuse
Def:
addDerivedTemplate: ProcessReuse, Processes, String, String, SetOfString, Text → ProcessReuse

addDerivedTemplate(
   (_, Templates (ApseeTemplates templates, _, Origin origin, AbstractArtifacts art)),
                              processes, template-id, proc-id, appDomains, desc)
=  (_,
   Templates
   (ApseeTemplates ICS(ProcessTemplates, addDerivedTemplate, templates,
                              <template-id, processes, proc-id, appDomains, desc>),   {1}
   _,
   Origin ICS(TemplateOrigin,addTemplateOrigin, origin, <template-id, current_datetime, process-id>),{2}
   AbstractArtifacts art))


Def: addDerivedAbsArtifacts: ProcessReuse, APSEE, String, String → ProcessReuse      {3}

addDerivedAbsArtifacts((_, Templates (_, _, _, AbstractArtifacts artifacts), _, _), apsee,template-id, proc-id)
=  (_, Templates (_, _, _, AbstractArtifacts ICS(AbsArtifacts, addSetAbsArtifacts, artifacts,
         <ICS(Processes, getInvolvedArtifacts, ICS(APSEE, getProcesses, apsee), <proc-id>),
         ICS(APSEE, getSwArtifacts, apsee), template-id>))
```

FIGURA 4.28 - A função *addDerivedTemplate* do *AtoProcessReuse*

Artefatos abstratos são criados a partir da existência de artefatos descritos no processo original. A função *addSetAbsArtifacts* é responsável por adicionar os novos artefatos abstratos, devolvendo uma nova versão para o objeto *AbsArtifacts* recebido. Os demais artefatos recebidos são: o conjunto de identificadores para os artefatos, a instância de *SwArtifacts* (contendo os detalhes dos artefatos originais) e o *ID* do *template* associado.

```
AbsArtifacts
Def: addSetAbsArtifacts: AbsArtifacts, SetOfString, SwArtifacts, String → AbsArtifacts

addSetAbsArtifacts(absArtifacts, empty-set, _, _) = absArtifacts

addSetAbsArtifacts(absArtifacts, add(set, artifact-id), swArtifacts, template-id)
=   if    ICS(SwArtifacts, getType_Id, swArtifacts, <artifact-id>) ≠ "ConcreteArtifact"
    then  modify(   getMappedArtifactID(template-id, artifact-id),
                    (
                    TypeID ICS(SwArtifacts, getType_Id, swArtifacts, <artifact-id>),
                    Name ICS(SwArtifacts, getName, swArtifacts, <artifact-id>),
                    Description ICS(SwArtifacts, getDescr, swArtifacts, <artifact-id>),
                    DerivedFrom AbsArtifactID getMappedArtifactID(template-id,
                                ICS(SwArtifacts, getDerivedFrom_Id, swArtifacts, <artifact-id>)
                    BelongsTo_ID getMappedArtifactID(template-id,
                                            ICS(SwArtifacts, getBelongsTo_Id, swArtifacts, <artifact-id>)
                    ),
                    addSetAbsArtifacts(absArtifacts, set, swArtifacts, template-id)    )
          ∪ absArtifacts)
    else  addSetAbsArtifacts(absArtifacts, set, swArtifacts, template-id)   ∪ absArtifacts

Função Auxiliar:
getMappedArtifactID: String, String → String
getMappedArtifactID(template-id, artifact-id) = template-id + artifact-id
```

FIGURA 4.29 - As funções *addSetAbsArtifacts* e *getMappedArtifactID* do *AtoAbsArtifacts*

A função *addDerivedTemplate* - cuja especificação é apresentada na figura 4.30 - é responsável por incluir um novo elemento no mapeamento *ProcessTemplates* recebido: assim, é utilizado o construtor *modify* do tipo mapeamento para definir uma nova instância, cujo domínio é fornecido pelo atributo *template-id* {1}, e a imagem pelos outros parâmetros fornecidos (descrição, tipo e domínios de aplicação). No caso do *template* o seu modelo de processo (i.e., o atributo *ProcessModel*) é determinado pelo resultado da função *proc2template* na instância de *ProcessModel* fornecida (item {2}). O mapeamento resultante da função *addDerivedTemplate*, portanto, é composto pela composição do novo *template* com as instâncias existentes (objeto *templates* no item {3} da figura).

```
AtoProcessTemplates
Def: addDerivedTemplate: ProcessTemplates, String, Processes, String, SetOfString, Text →
ProcessTemplates
addDerivedTemplate(templates, template-id, processes, proc-id, desc, appDomains)
=   modify(template-id,                                                          {1}
        (Name template-id,
        Description desc,
        TypeID ICS(Processes, getType_Id, processes, <proc-id>),
        AppDomains appDomains,
        ProcessModel ICS(AbsProcessModel, proc2template,                         {2}
                        ICS(Processes, getProcessModel, processes, <proc-id>), <template-id>),
        templates)                                                               {3}
```

FIGURA 4.30 - A função *addDerivedTemplate* do *AtoProcessTemplates*

A generalização de processo para *template* envolve a generalização das instâncias da classe *ProcessModel* para o seu correspondente abstrato, isto é, *AbsProcessModel*. Tal como descrito na figura 4.31, a generalização de *ProcessModel* envolve a generalização das atividades (item {1}, na figura) e das conexões componentes (item {2}). Enfim, as políticas habilitadas no modelo original são copiadas (item {3}).

```
AtoAbsProcessModel
proc2template: ProcessModel, String → AbsProcessModel
Proc2template(processModel, template-id)
=   (Activities ICS(AbsActivities, proc2template, ICS(ProcessModel, getActivities, processModel),
                        <ICS(Activities, keys, activities), template-id>),                    {1}
    Connections ICS(AbsConnections, proc2template,ICS(ProcessModel, getConnections,processModel),
                        <ICS(Connections, keys, connections>)),                               {2}
    Policies (PolicySets ICS(ProcessModel, getPolicySets, processModel), StaticOK true)),     {3}
    Requirements "")
```

FIGURA 4.31 - A função *proc2template* do *AtoAbsProcessModel*

As atividades consistem de elementos importantes que devem ser seus atributos avaliados na generalização de um processo. A abordagem automática proposta nesta seção, generaliza as atividades folha (atividades *Plain*), divididas em atividades normais e automáticas, e os fragmentos (atividades decompostas).

A função *proc2template* descrita no ATO AbsActivities é responsável por produzir um novo mapeamento das atividades abstratas a partir de três parâmetros:

- o objeto contendo as atividades concretas do modelo original (objeto rotulado como *activities* no item {1});
- o conjunto de identificadores de atividades (conjunto identificado como *add(act-set, act-id)* no item {1} da figura). Tal conjunto é necessário para facilitar a construtor de ;
- o identificador do *template-id* sendo criado.

Com os objetos recebidos como parâmetros de entrada, é criado um mapeamento tendo como elementos as atividades generalizadas (item {2}). A função é baseada em chamadas recursivas (item {3}) para incluir todos os elementos do conjunto de atividades fornecido.

```
AtoAbsActivities
proc2template: Activities, SetOfString, String → AbsActivities

proc2template(empty-mapping, _, _) = empty-mapping
proc2template(_, empty-set, _) = empty-mapping

proc2template(activities, add(act-set, act-id), template-id)                                  {1}
=   modify(                                                                                   {2}
        act-id,
        (
        Name ICS(Activities, getName, activities, <act-id>),
        TypeID ICS(Activities, getType_Id, activities, <act-id>),
        Description ICS(AbsActDescription, proc2template,
                        ICS(Activities, getDescription, activities, <act-id>), <template-id>),
        Policies (PolicySets ICS(Activities, getPolicySets, activities, <act-id>), StaticOK true)
        Position ICS(Activities, getPosition, activities, <act-id>)
        ),
        proc2template(activities, act-set))                                                   {3}
```

FIGURA 4.32 - A função *proc2template* do *AtoAbsActivities*

Como anteriormente descrito na seção 4.2.1.5, toda atividade de um *template* é associada a uma descrição dos seus detalhes, definida pela classe *AbsActDescription*. A função *proc2template* do ATO *AbsActDescription*, portanto, fornece um novo objeto a partir de uma instância do *ActDescription* concreto fornecido (parâmetro formal da função definida no item {1} da figura 4.33).

Os três casos possíveis para um objeto de *ActDescription* são tratados: uma atividade refinada em fragmento (em {2}), atividade-folha normal (em {3}) e atividade-folha automática (em {4}). Portanto, as funções *proc2template* correspondentes são chamadas em resposta ao tipo de objeto *ActDescription* fornecido.

```
AtoAbsActDescription
proc2template: ActDescription, String → AbsActDescription                                  {1}
proc2template(actDescription, template-id)
=   if      ICS(ActDescription, isFragment, actDescription)                                 {2}
    then    (Fragment ProcessModel ICS(AbsProcessModel, proc2template, actDescription))
    else    if      ICS(ActDescription, isNormal, actDescription)                           {3}
            then    (Plain
                        (Requirements ICS(ActDescription, getRequirements, actDescription),
                        Description Model Normal ICS(NormalAbsDesc, proc2template,
                                ICS(ActDescription, getNormal, actDescription), <template-id>)
                    )
            else    // é folha e automática                                                {4}
                    (Plain
                        (Requirements ICS(ActDescription, getRequirements, actDescription),
                        Description Model Automatic ICS(AutomaticAbsDesc, proc2template,
                                ICS(ActDescription, getAutomatic, actDescription))
                    )
```

FIGURA 4.33 - A função *proc2template* do *AtoAbsActDescription*

A figura 4.34 apresenta a função *proc2template* para o ATO *NormalAbsDesc*. Esta função é responsável por generalizar os aspectos pessoais (item {1}), os recursos (em {2}), os artefatos (em {3}), copiando o *script* do objeto *Normal* fornecido (em {4}).

```
AtoNormalAbsDesc
proc2template: Normal, String → NormalAbsDesc
proc2template(normal, template-id)
=   (People ICS(AbsReqPeople, proc2template, ICS(Normal, getPeople, normal)),              {1}
    Resources ICS(ReqResources, getResourceTypes, ICS(Normal, getResources, normal))       {2}
    Artifacts ICS(TemplateArtifacts, proc2template, ICS(Normal, getArtifacts, normal), <template-id>),  {3}
    Script ICS(Normal, getScript, normal)))                                                {4}
```

FIGURA 4.34 - A função *proc2template* do *AtoNormalAbsDesc*

A generalização do objeto *ReqPeople* de uma atividade-folha normal é descrita pelas funções apresentadas na figura 4.35. A função principal *proc2template* do *AtoAbsReqPeople* é responsável por descrever os atributos *RoleTypes* e *GroupTypes*. Para tanto, são fornecidas as funções *agent* e *groupGeneralization* que varrem, respectivamente, as listas de agentes e grupos requeridos por uma atividade concreta, extraindo os cargos e tipos de grupos associados.

```
AtoAbsReqPeople
proc2template: ReqPeople → AbsReqPeople

proc2template(reqPeople)
=   (RoleTypes agentGeneralization(reqPeople, ICS(ReqPeople, sizeofAgents, reqPeople)),
     GroupTypes groupGeneralization(reqPeople, ICS(ReqPeopel, sizeofGroups, reqPeople)))

Def: agentGeneralization: ReqPeople, Integer → SetOfString
agentGeneralization(_, 0) = empty-set
agentGeneralization(reqPeople, n)
=   add(agentGeneralization(reqPeople, n-1), ICS(ReqPeople, getRoleID, reqPeople, <n>))

Def: groupGeneralization: ReqPeople, Integer → SetOfString
groupGeneralization(_, 0) = empty-set
groupGeneralization(reqPeople, n)
=   add(groupGeneralization(reqPeople, n-1), ICS(ReqPeople, getGroupTypeID, reqPeople, <n>))
```

FIGURA 4.35 - As funções *proc2template*, *agentGeneralization* e *groupGeneralization* do *AtoAbsReqPeople*

A generalização dos artefatos envolvidos na descrição de um processo encerrado é fornecida pelas funções especificadas na figura 4.36. A função *proc2template* principal do ATO *TemplateArtifacts* é responsável por obter os artefatos de entrada (item {1}, na figura), e de saída (em {2}). No caso de artefatos de entrada, a função responsável verifica se o artefato fornecido é Concreto (em {3}), a fim de associá-lo corretamente aos tipos existentes no *template*. Tal verificação não é necessária para artefatos de saída, que obrigatoriamente são generalizados para artefatos abstratos (item {4}), visto um artefato concreta não pode ser produzido ou modificado por qualquer atividade do modelo.

```
AtoTemplateArtifacts
Def: proc2template: Artifacts → TemplateArtifacts

proc2template(artifacts)
=   (InputArtifacts  inputArtifactGeneralization(ICS(Artifacts, getInputSet, artifacts), template-id)        {1}
     OutputArtifacts outputArtifactGeneralization(ICS(Artifacts, getOutputSet, artifacts), template-id))  {2}

Def: inputArtifactGeneralization: SetOfString, String → SetOfString

inputArtifactGeneralization(empty-set, _) = empty-set
inputArtifactGeneralization(add(set, artifact), template-id)
=   if    ICS(Artifacts, getInputReqTypeID, artifact) ≠ "ConcreteArtifact"                                {3}
    then  add(inputArtifactGeneralization(set, template-id),
                  AbsArtifactID ICS(AbsArtifacts, getMappedArtifactID,
                        ICS(Artifacts, getInputArtifactID, artifact), <template-id>)))
    else  add(inputArtifactGeneralization(set,
                  SwArtifactID ICS(AbsArtifacts, getMappedArtifactID,
                        ICS(Artifacts, getInputArtifactID, artifact), <template-id>)))

Def: outputArtifactGeneralization: SetOfString, String → SetOfString

outputArtifactGeneralization(empty-set, _) = empty-set
outputArtifactGeneralization(add(set, artifact), template-id)                                              {4}
=   add(outputArtifactGeneralization(set, template-id),
                  AbsArtifactID ICS(AbsArtifacts, getMappedArtifactID,
                        ICS(Artifacts, getOutputArtifactID, artifact), <template-id>)))
```

FIGURA 4.36 - As funções *proc2template*, *inputArtifactGeneralization* e *outputArtifactGeneralization* do *AtoTemplateArtifacts*

A generalização de Conexões (objetos da classe *Connections*) é definida tal como especificado pela função *proc2template* do ATO *AbsConnections* (figura 4.37). A função recebe dois argumentos: uma instância de *Connections* (instância esta que contém os detalhes das conexões existentes no processo original encerrado), e os identificadores associados (instância de *SetOfString*). A função trata de um elemento arbitrário do conjunto fornecido (*connection-id*, em {1}) e depois recursivamente inclui os elementos restantes no conjunto *set* (em {2}).

```
AtoAbsConnections

Def: proc2template: Connections, SetOfString → AbsConnections

proc2template(empty-mapping, empty-set) = empty-mapping

proc2template(connections, add(set, connection-id))
=   modify(connection-id, (     TypeID ICS(Connections, getType_Id, connections, <connection-id>),  {1}
                                Kind ICS(Connections, getKind, connections, <connection-id>),
                                Position ICS(Connections, getPosition, connections, <connection-id>)
    ),
        proc2template(ICS(Connections, keys, restrict_with (connections, connection-id)), set)      {2}
```

FIGURA 4.37 - A função *proc2template* do *AtoAbsConnections*

Finalmente, é apresentada a função para adicionar um novo elemento no mapeamento *TemplateOrigin* (figura 4.38).

```
AtoTemplateOrigin
Def: addTemplateOrigin: TemplateOrigin, String, Date, String → TemplateOrigin

addTemplateOrigin(origin, template-id, date, proc-id)
=   modify(template-id, (Date date, ProcessID proc-id), origin)
```

FIGURA 4.38 - A função *addTemplateOrigin* do *AtoTemplateOrigin*

A generalização de processos é composta ainda por funções auxiliares dos ATOs aqui apresentados e dos ATOs do pacote *APSEE*. Tais funções são especificadas no Anexo 4 desse texto.

4.3 A linguagem para definição de *ProcessTemplates*

A definição de uma linguagem visual apoiada por computador, envolve a definição da estrutura de dados de suporte, do alfabeto gráfico e das regras para composição sintática dos elementos.

Como descrito por Bardohl em [BAR2000], a descrição de linguagens visuais é dividida em dois níveis: enquanto o nível abstrato define o significado lógico da linguagem, o nível de sintaxe concreta é usado para fornecer o *layout*, o que é de extrema utilidade na definição de editores gráficos que implementem a linguagem proposta. A combinação da sintaxe abstrata e concreta é denominada Sintaxe Visual (*Visual Syntax*, em [BAR2000]).

No modelo aqui proposto para definição de *templates* de processos de software, a definição em PROSOFT-Algébrico para os tipos de dados (seção 4.2) é complementada pelo Grafo Tipo e pelas regras descritas nas seções a seguir na definição completa da Sintaxe Visual da linguagem.

Esta seção é dividida em quatro seções, como segue. Na seção 4.3.1, em seguida, é apresentado o Grafo Tipo (que norteia a definição das regras). A seção 4.3.2 apresenta as regras básicas que atuam na modelagem de *templates*. Finalmente, a seção

4.3.3 define as regras que restringem a adaptação dos *templates* para processos executáveis.

4.3.1 O Grafo Tipo

A inerente complexidade da linguagem fornecida para modelagem de processos influenciou o desenvolvimento do Grafo Tipo[34], fazendo com que esse seja um diagrama de grandes proporções que interconecte uma grande quantidade de componentes. Assim, esta seção apresenta o Grafo Tipo que descreve os componentes usados na edição de *templates*.

Os símbolos visuais usados na construção do Grafo Tipo estão de acordo com a notação (sintaxe concreta) previamente apresentada no capítulo 3. É importante observar, todavia, que alguns tipos de dados que não possuem representação gráfica também estão incluídos no Grafo Tipo, pois são fundamentais para a descrição dos *templates*.

Em virtude das suas grandes dimensões, o Grafo Tipo para *templates* completo é apresentado na figura A 11 (Anexo 5). A figura 4.39 mostra uma parte do Grafo Tipo contendo os elementos sintáticos associados à definição de atividades. É importante observar que, da mesma forma que proposto por Lima Reis em [LIM2002a], foram incluídos nesse Grafo Tipo relacionamentos do tipo É-um (*Is_a*): esse tipo de arco representa o relacionamento de herança, com semântica similar ao construtor homônimo usado em linguagens de programação orientadas a objetos. Esse relacionamento *Is-a* simplifica bastante o Grafo Tipo e as regras de transformação correspondentes, uma vez que os atributos e relacionamentos são herdados para os subtipos de um nodo. Assim, por exemplo, quando o lado esquerdo de uma regra contém referências ao tipo principal de atividade abstrata (i.e., o nodo rotulado com *AbsActivity*), então ele pode ser substituído por qualquer uma de suas especializações (*Fragment*, *PlainAbsActivity*, Automatic e *NormalAbsDesc*).

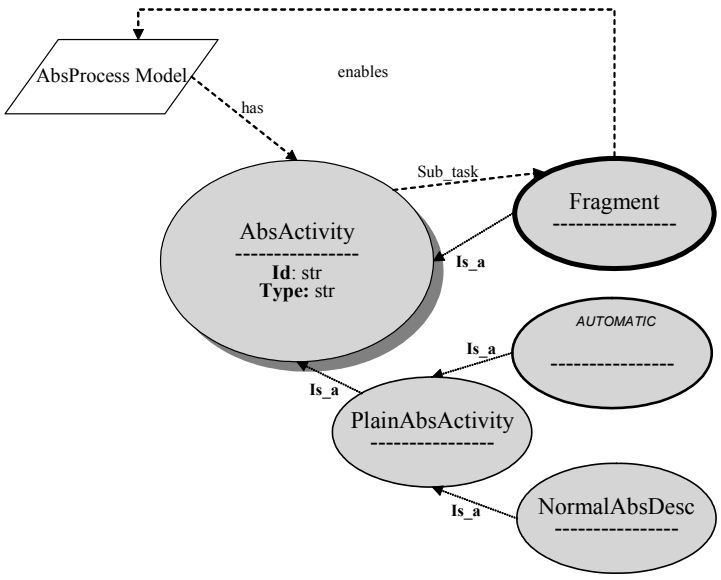

FIGURA 4.39 - Elementos do Grafo Tipo relacionados com a definição de atividades

[34] No contexto de Gramática de Grafos, este texto usa a terminologia em Português adotada por Ribeiro [RIB00], adotando Grafo Tipo para descrever o que a literatura internacional citada por Bardohl em [BAR00] comumente denomina *graph-schema*.

Vale ressaltar ainda que em GGs, os arcos do grafo tipo denotam possibilidades de conexão entre os nodos, sendo que as restrições de cardinalidade e de dependência são verificadas e garantidas pelas regras da sintaxe e pela especificação algébrica do sistema.

A seção a seguir discute as regras que definem a construção de *templates*.

4.3.2 Regras para edição de *Templates*

Usando-se GG, um sistema de software é especificado em termos de estados (que são modelados por grafos), e mudanças de estados (modeladas por regras ou derivações). A aplicação de uma regra a um grafo G é chamada passo de derivação, e isso só é possível se há uma coincidência do lado esquerdo (L) da regra no grafo atual G. O lado direito (R) da regra define o grafo resultante da aplicação desta regra. Segundo Ribeiro [RIB2000], a interpretação de uma regra r: L → R é feita da seguinte forma:

- Itens em L que não têm imagem em R são *eliminados*;
- Itens em L que são mapeados para R são *preservados*;
- Itens em R que não existem em L são *criados*.

Ainda, se um nodo *N* existente em L não é mapeado para R, os arcos que são conectados com *N* também são removidos. Em algumas regras são utilizadas condições negativas (*Negative Application Conditions*-NAC). A interpretação nesse caso é feita da seguinte forma: a regra é ativada se o lado esquerdo da regra ocorre no grafo e as condições negativas **não estão presentes**. As NACs são representadas por elementos cancelados com uma grande marca em forma de "X".

As funções básicas para edição de *templates* são numerosas, sendo apresentadas no Anexo 5. A figura 4.40 apresenta um exemplo de regra extraído do Anexo 5, com o objetivo de ilustrar a composição dos elementos usados na descrição de uma regra. No exemplo, o nome da função envolvida (*NewAbsActivity*), assim como seus parâmetros, são incluídos acima do lado esquerdo da regra. O lado esquerdo da regra descreve a condição esperada para instanciar a função descrita: no caso, um objeto de *ProcessTemplate* deve possuir uma conexão *has* com um objeto de *AbsProcessModel*, o qual - pela NAC inserida - não pode estar conectado por *has* com um objeto *AbsActivity* qualquer que possua o identificador fornecido pelo usuário (*new_id*). O lado direito da regra apresenta o resultado obtido a partir da função: uma atividade *Normal* é inserida e conectada ao *AbsProcessModel* correspondente, sendo definida com o identificador *new_id* fornecido e com um *script* vazio.

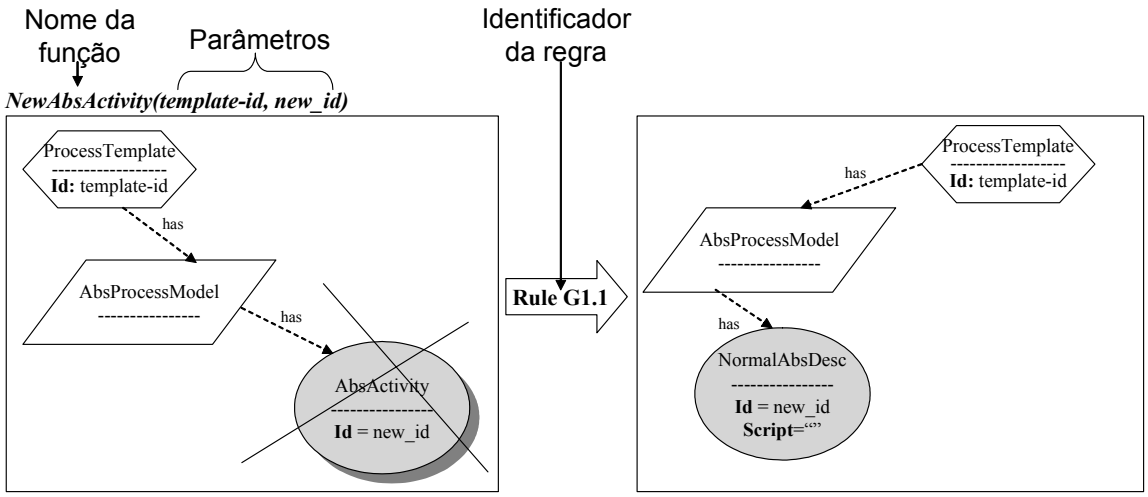

FIGURA 4.40 - Exemplo de regra para inclusão de atividade em um *template*

As regras apresentadas no Anexo 6 descrevem a linguagem proposta, sendo responsáveis por permitir desde a simples inclusão (regra G1.1) e remoção de atividades de um *template* (regras G1.10, G1.11 e G1.12) até por prevenir a ocorrência de ciclos, através de restrições na ordenação das atividades (regras de teste do fluxo de controle especificadas no Anexo 7). Regras ainda foram definidas para restringir a utilização de artefatos (i.e., artefatos concretos somente podem ser consumidos pelas atividades, segundo aquilo especificado pelas regras G2.10, G2.11, G2.12, G2.23, G2.24, G2.25 e G2.26), além da associação de atividades normais aos tipos de recursos, cargos e grupos (regras do grupo G6).

4.3.3 Regras que restringem a adaptação de *Templates*

A adaptação de *templates* para processos é uma atividade manual realizada por um projetista de processos na evolução de um processo obtido a partir de um *template*. No modelo aqui proposto, uma instância inicial do *AtoProcesses* é fornecida a partir da função *template2proc* (discutida na seção 4.2.2), a qual cria correspondentes para os elementos sintáticos descritos originalmente em um *template*. Assim, esta seção complementa o papel desempenhado pela função *template2proc*, descrevendo a semântica das funções que restringem a adaptação de processos executáveis.

Como apresentado na seção 3.7, três tipos de adaptação são definidos (livre, guiada por políticas, ou restrita pelos elementos sintáticos originais). Assim, dependendo do tipo de adaptação escolhido pela organização[35], o conjunto das modificações permitidas no editor de processos é restrito. Como as funções descritas são intimamente relacionadas com a edição interativa de modelos de processos de software executáveis, foi seguida uma abordagem semelhante àquela usada na seção anterior (na descrição de regras sintáticas para modelagem de *templates*): regras foram especificadas a partir de um grafo tipo existente. Como o editor de modelos executáveis foi especificado em outro trabalho do grupo PROSOFT-*APSEE* [LIM2002d], o trabalho foi conduzido da seguinte forma:

[35] Em uma instalação do ambiente APSEE, o tipo de adaptação adotado pela organização é expresso pelo valor do atributo *DefaultAdaption* para o *AtoProcessReuse*.

- As regras propostas em [LIM2002d] foram avaliadas, identificando aquelas que modificam os processos. Em virtude das características das restrições necessárias, são de especial interesse as funções que excluem ou modificam os elementos de um modelo de processo derivado a partir de um *template*;
- Para cada regra identificada, foram verificadas quais propriedades sintáticas adicionais seriam necessárias. Novas regras ainda foram especificadas em virtude de casos especiais tratados para os tipos de adaptação fornecidos;

Foi levado em consideração que a correspondência de elementos de um processo executável e o *template* original é registrada em objetos do *AtoTemplateImpl* (figura 4.25). Assim, por exemplo, a partir de uma atividade de um processo é possível identificar qual atividade do *template* original está envolvida. Funções foram especificadas com o objetivo de identificar a origem para um componente específico de um processo (se houver).

Esta abordagem é exemplificada a seguir. A figura 4.41 apresenta uma regra - P9.1[36] - originalmente proposta para o editor de processos executáveis. Nesse caso, uma atividade concreta (objeto de *Activity*) está conectada (pelo arco *enables*) a uma política (objeto *Policy*). Desta forma, a única restrição expressa pela função *DisablePolicyInstanceActivity* avalia se o atributo *A-st* da atividade correspondente (que indica o estado atual da atividade na execução de um processo) possui o valor *null*, isto é, a atividade ainda não foi instanciada e/ou executada.

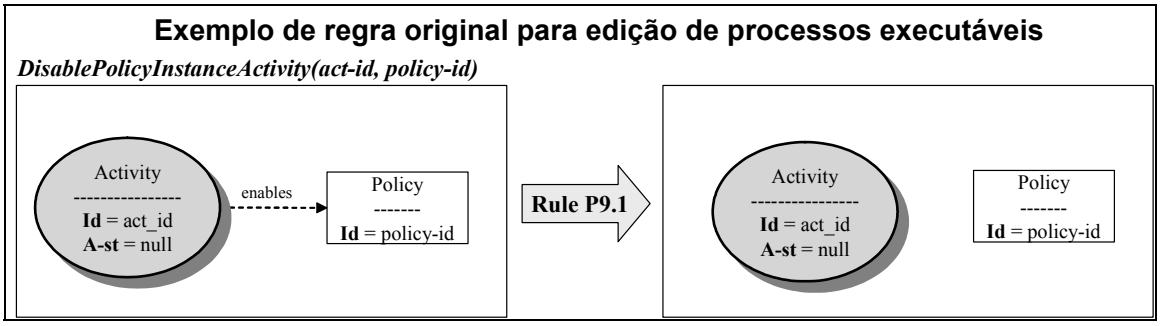

FIGURA 4.41 - Regra P9.1 para desabilitar uma política previamente habilitada em uma atividade concreta (extraída de [LIM2002d])

A adaptação guiada por políticas define que as políticas habilitadas em um *template* não podem ser removidas nos processos derivados. Assim, a função P9.1 apresentada teve de ser reescrita para lidar com esse novo requisito, tal como na figura 4.42: as regras *AdaptRule* 9.1 e 9.2 mantém as restrições expressas na regra P9.1 original e, além disso, tratam explicitamente do tipo de adaptação adotado (na chamada de função *default_adaptation* como condição para habilitar a regra). Na regra *AdaptRule* 9.1, a condição negativa - área circundada marcada com um "X" - ainda restringe que não exista uma atividade abstrata conectada com *act-id* que habilite a política em questão (através da função *is_implemented_by*, a qual é baseada no *AtoTemplateImpl*). No caso em que a adaptação é livre, a definição da regra coincide com a definição original de P9.1.

[36] A fim de distinguir das regras existentes para *templates*, as regras para manipulação de Processos executáveis são identificadas com o rótulo iniciando pela letra P.

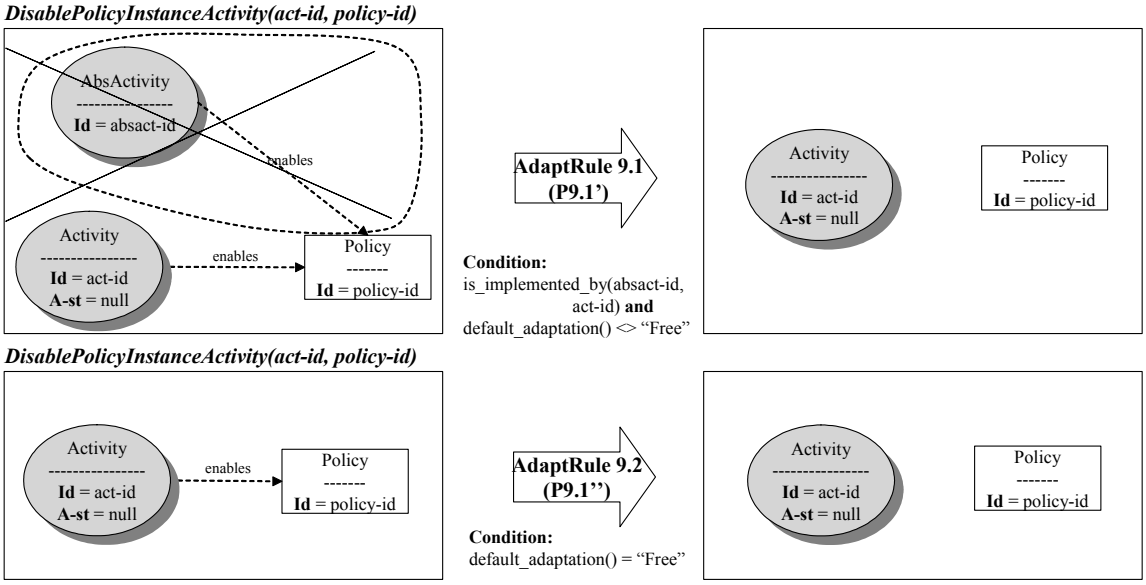

FIGURA 4.42 - Exemplo de regra para adaptação baseada em políticas - função *DisablePolicyInstanceActivity*

A função *DeleteActivity*, especificada pela regra P1.16 na figura 4.43 a seguir, é usada para exemplificar a adaptação restrita aos elementos sintáticos originais. Portanto, a regra P1.16 descreve que se o usuário solicita a remoção de uma atividade concreta rotulada com *act_id* e o elemento sintático existente no processo coincidente seja uma atividade automática, a atividade será removida se o estado da atividade for *null* ou *waiting*[37].

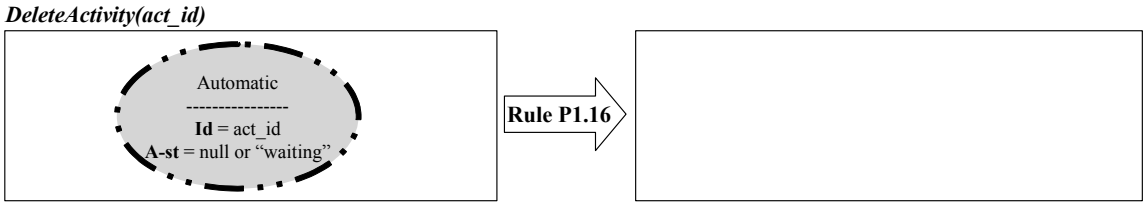

FIGURA 4.43 - Regra P1.16 para remover uma atividade automática de um processo (extraída de [LIM2002d])

Para a adaptação restrita, a regra precisou ser estendida, através de condições adicionais. Assim, se o tipo de adaptação requerido é diferente de *Restricted* (i.e., o atributo *DefaultAdaptation* assume os valores *Free* - para adaptação livre - ou *PolicyBased* - para adaptação guiada por políticas), a condição é satisfeita e a regra se comporta exatamente como a regra P1.16 original. Caso a adaptação seja restrita, é verificado se a atividade não é originada de uma atividade abstrata do *template* envolvido (pela chamada à função *has_abstract_origin*).

[37] O leitor interessado nos estados possíveis para uma atividade de um processo executável deve consultar o trabalho de Lima Reis em [LIM02d].

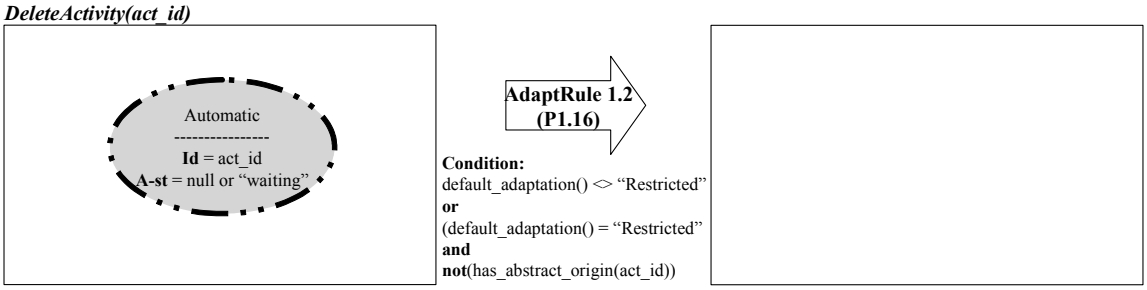

FIGURA 4.44 - Exemplo de regra para adaptação restrita - função *DeleteActivity*

Tanto o grafo tipo (figura A 63) quanto as regras para restringir a adaptação de processos executáveis derivados a partir de *templates* são apresentados no Anexo 7.

4.4 Especificação de Políticas Estáticas

Políticas Estáticas (no modelo de dados descrito a seguir sendo referenciadas como *StaticPolicies*) atuam na modelagem, auxiliando o projetista/programador de processo através da verificação de propriedades sintáticas habilitadas para elementos do processo. Assim, uma política estática descreve uma lista ordenada de propriedades dos processos que, quando avaliadas, devem ser satisfeitas (retornando *True*, na sua avaliação).

Políticas Estáticas são verificadas durante a modelagem de processos, quando o projetista de processo solicita a verificação do modelo (um passo obrigatório no ciclo de vida de processos *APSEE* para habilitar execução de um modelo de processo instanciado).

Os principais componentes de uma política estática são:

- Uma identificação única;
- **Nome** da política;
- **Descrição** textual da política;
- **Tipo da política**, indica o tipo da Política Estática na sua hierarquia de tipos;
- **Tipo de reação**. Determina se a política é obrigatória (*mandatory* - impede o prosseguimento da verificação enquanto a política não for satisfeita), ou desejável (*desirable* - fornecendo somente um aviso para o usuário durante a verificação do modelo quando a política não é satisfeita);
- **Interface da Política.** Especifica o tipo de objeto-*APSEE*[38] que será tratado pela política. Por exemplo, se uma política tem interface definida para o tipo *Activity*, isto significa que a política pode atuar sobre atividades. Ainda, se a política citada for aplicada a um fragmento de processo, isto indica que todas as suas atividades constituintes do processo são tratadas por esta política;

[38] Objetos das classes *Activity, Role, Agent, Artifact, Process* ou *Resource*.

- **Propriedades para habilitação (*Enabling Properties*).** Constituem as pré-condições lógicas que devem ser satisfeitas para que uma política seja executada (relacionadas ao tipo definido na interface da política);
- **Propriedades (*Properties*).** Sempre que as propriedades para habilitação são satisfeitas, as propriedades descritas no corpo de uma política estática são verificadas. Caso alguma propriedade não seja satisfeita, o mecanismo de verificação de processos adota o comportamento especificado pelo tipo de reação atribuído à política. As propriedades correspondem essencialmente às chamadas de métodos disponíveis nos tipos definidos no *APSEE*.

4.4.1 Gramática da Linguagem de Políticas Estáticas

As Políticas Estáticas seguem o formato descrito pela gramática apresentada na figura 4.45.

```
<policy>          → Id <string> ([ Name <string>] [; Description <string>] ;
                                 TypeID <pol_type_id> ;
                                 Mandatory <boolean> ;
                                 Interface <pol_interface> ;
                                 Enabling_Properties <condition> ;
                                 Properties <condition>) ;
<boolean>         → true | false
<pol_interface>   → <interface_label> : <interface_type>
<condition>       → <pol_operand>  [<opt_relation>] [<opt_connection>]
<pol_expression>  → <pol_operand> | <pol_operand> <expression_type> <pol_expression>
<expression_type> → union | intersection
<opt_relation>    → <comparison_type> <pol_expression>
<comparison_type> → > | < | >= | <= | = | <> | contains | not_contains | sub_type_of
<opt_connection>  → <conn_type>  <condition>
<conn_type>       → and | or
<pol_operand>     → <pol_object> . <pol_operator>*
<pol_object>      → <interface_label>
<pol_operator>    → <method_id> ( <list_parameters> ) | <reserved_word>
<list_parameters> → <pol_expression>*
<reserved_word>   → any | all | no
<interface_type>  → Activity | Artifact | Process | Resource | Agent | Role
<interface_label> → <string>
<method_id>       → <string>
<pol_type_id>     → <string>          // nodo da hierarquia de tipos de políticas estáticas
```

FIGURA 4.45 - Gramática da linguagem *StaticPolicies*

4.4.2 Os Tipos de Dados

Esta seção apresenta alguns dos principais tipos de dados do PROSOFT-Algébrico [NUN92] criados para o armazenamento e processamento de Políticas Estáticas. A definição de Política Estática é realizada pelo usuário em um editor de textos do ambiente PROSOFT (ATO Texto). Nesse texto de entrada é realizada análise sintática e léxica que gera como resultado objetos das classes PROSOFT descritas nesta seção, conforme ilustrado pelo esquema da figura 4.46.

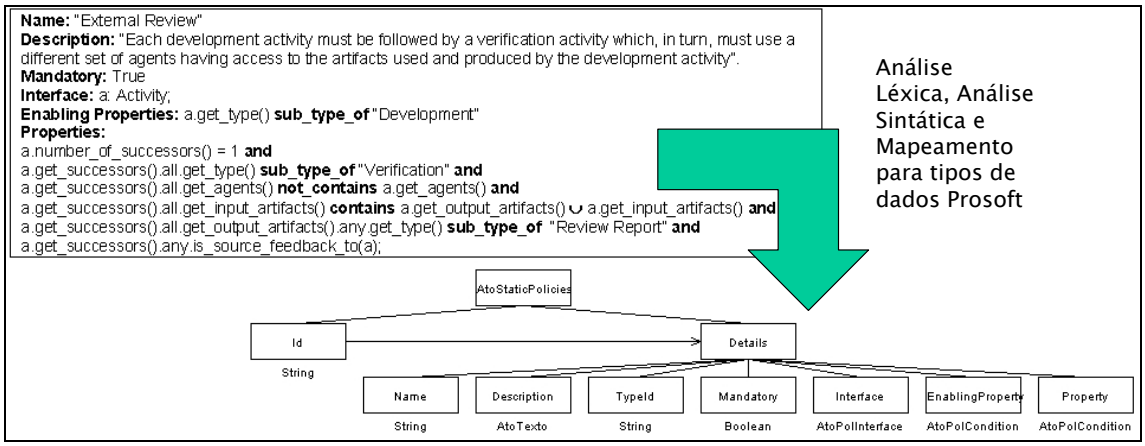

FIGURA 4.46 - Esquema do mapeamento da definição de políticas para tipos de dados internos do ambiente

O tipo *StaticPolicies* está associado ao tipo *Policies* que, por sua vez, é referenciado pelo tipo principal *Apsee*, tal como apresentado na figura 4.47. Políticas Estáticas são individualizadas por um identificador único (*ID*) e consistem dos atributos *Name, Description, Type-id, Mandatory, Interface, EnablingProperty e Property*, correspondentes aos itens descritos na seção 4.4, definidos na linguagem.

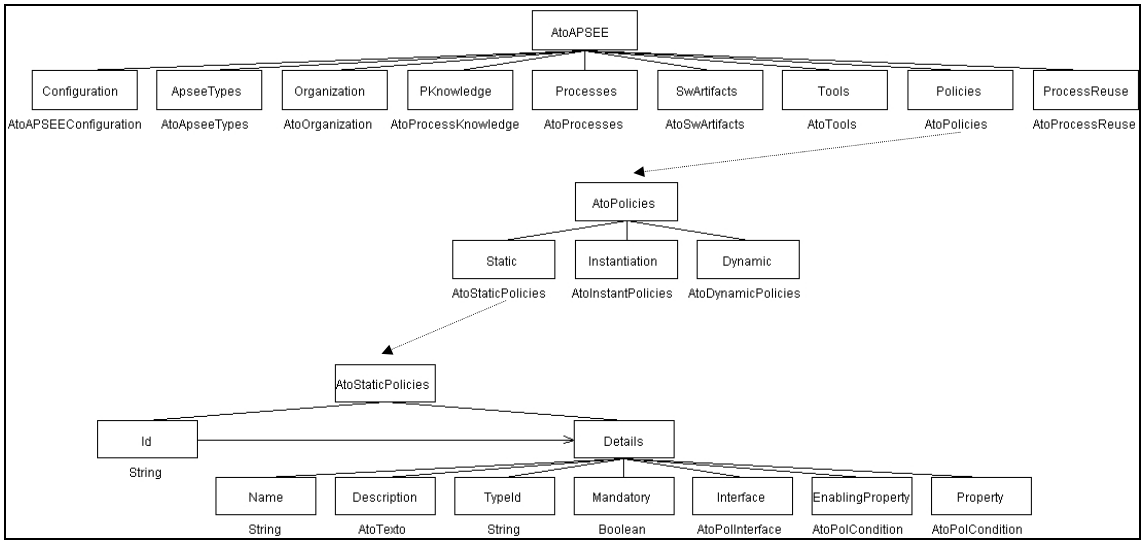

FIGURA 4.47 - Relacionamento entre as classes *APSEE*, *Policies* e *StaticPolicies*

A interface de uma política (*PolInterface*) é definida por uma tupla com o rótulo (*Label*) na figura 4.48 e a identificação do tipo *APSEE* correspondente (*Type*) - figura 4.49.

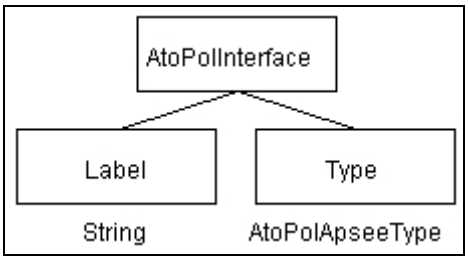

FIGURA 4.48 - Classe para definição da Interface de uma Política Estática (*AtoPolInterface*)

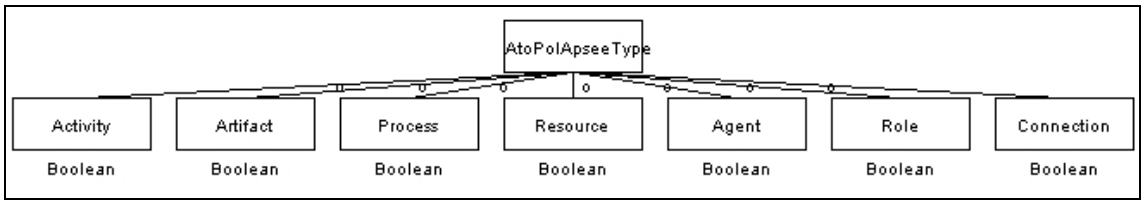

FIGURA 4.49 - Classe para definição de tipos de objeto APSEE (*AtoPolApseeType*)

O tipo *PolCondition* (figura 4.50) é essencialmente a classe mais importante dessa especificação, descrevendo as propriedades de uma política, com uma Expressão (objeto de *AtoPolExpression*), uma relação opcional (*Opt_relation*), e uma conexão opcional (*Opt_conn*, que expressa a conexão *and* ou *or* com outra propriedade). Uma expressão lógica de política é definida pelo tipo *AtoPolExpression* (figura 4.51).

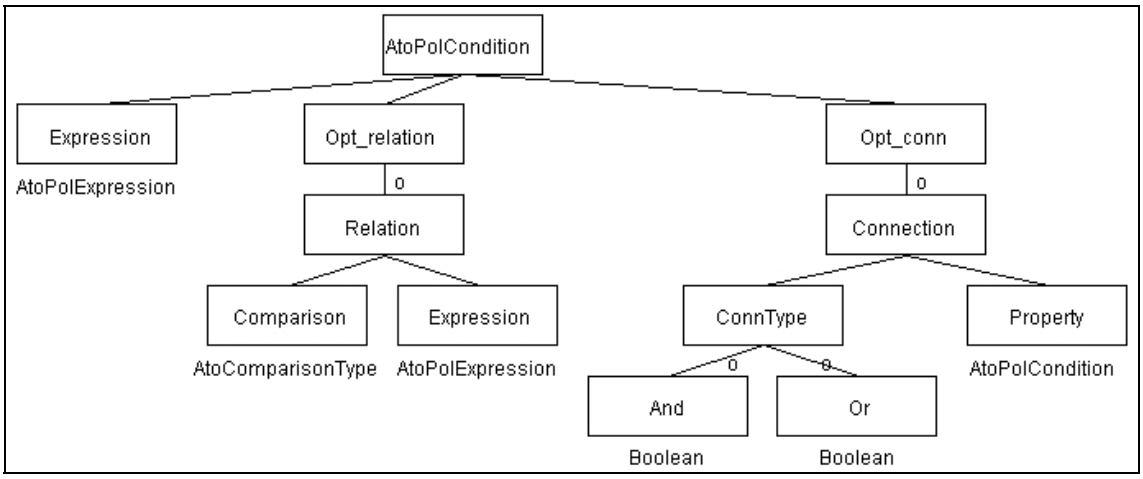

FIGURA 4.50 - ATO para definição de condições lógicas (*AtoPolCondition*)

A classe *AtoPolExpression* (figura 4.51 - Esquerda) é definida como uma alternativa entre um operando (ramo *Operand*) e uma expressão (ramo *Expression*). Os tipos de expressões lógicas disponíveis incluem a União e Interseção de conjuntos (figura 4.51 - Direita).

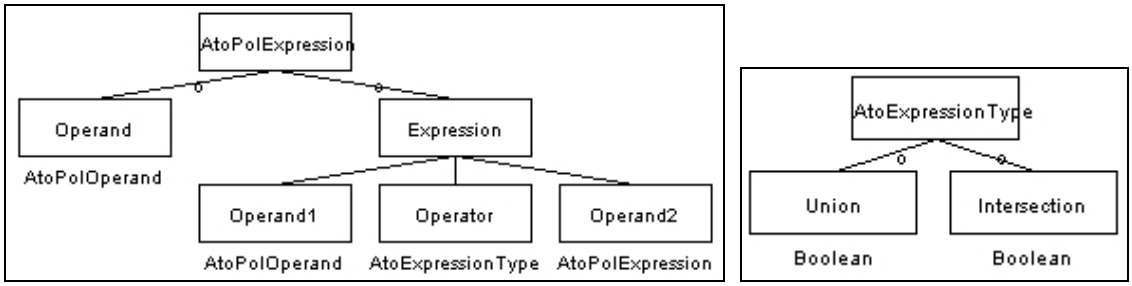

FIGURA 4.51 - Classes para definição de expressões lógicas (*AtoPolExpression* e *AtoExpressionType*)

Operandos são definidos pela classe *AtoPolOperand* (figura 4.52 - esquerda), contendo referência a um objeto - que pode ser a interface definida na política ou objetos primitivos - tal como definido à direita da figura 4.52.

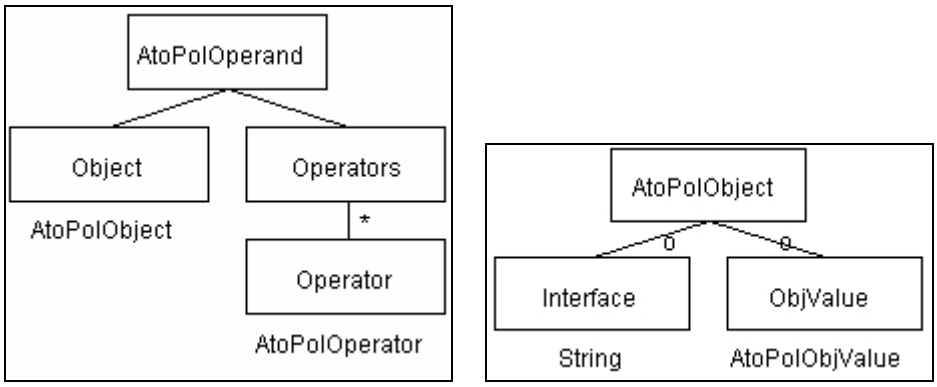

FIGURA 4.52 - Classes para definição de Operandos (*AtoPolOperand*) e objetos manipulados em políticas (*AtoPolObject*)

A classe *AtoComparisonType* (figura 4.53) determina o tipo de comparação usado para uma relação entre dois operandos (igualdade, desigualdade, maior, maior ou igual, menor, menor ou igual, contém e não-contém).

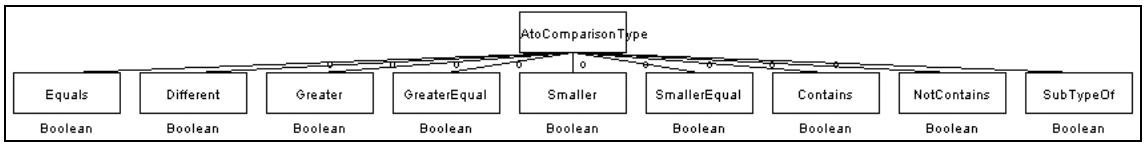

FIGURA 4.53 - *AtoComparisonType*

O *AtoPolOperator* (figura 4.54) é especificado como uma alternativa entre uma chamada de método (*MethodCall*) ou uma palavra reservada (*Any, All* e *No*) da linguagem (*ReservedWord*). Na figura 4.55 é apresentado o tipo *ListOfParameters*, denotando que os parâmetros consistem de uma lista ordenada de instâncias de *PolExpression*.

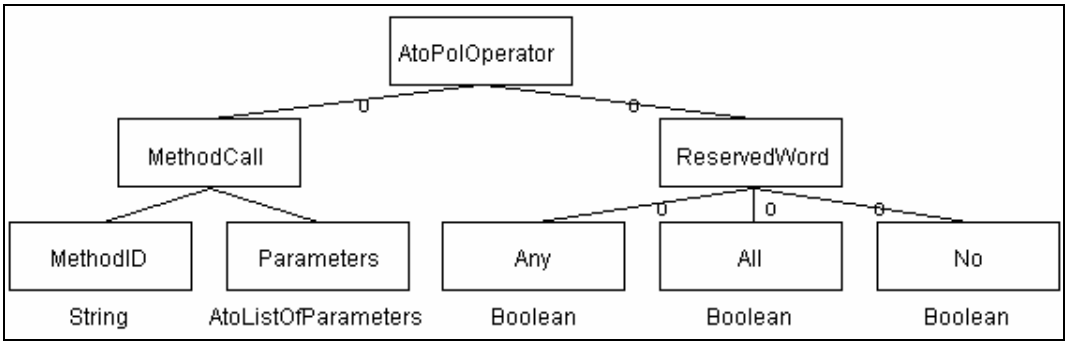

FIGURA 4.54 - Classe para definição de chamada de métodos ou palavras reservadas (*AtoPolOperator*)

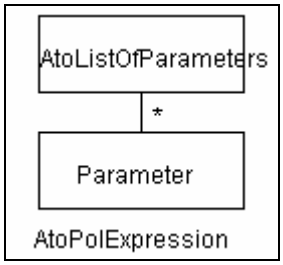

FIGURA 4.55 - Classe para definição de parâmetros de operadores (*AtoListOfParameters*)

As figuras numeradas de 4.56 a 4.58, apresentadas nas páginas a seguir, correspondem a tipos de dados que armazenam informações adicionais utilizadas na descrição de uma Política Estática na linguagem proposta.

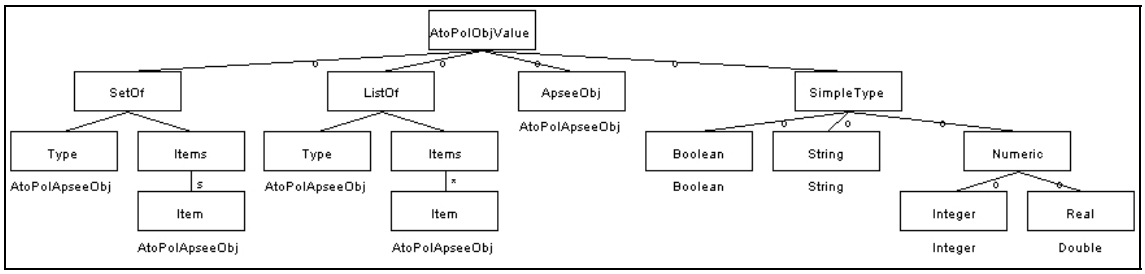

FIGURA 4.56 - Classe que define o valor de um objeto gerado por Políticas Estáticas (*AtoPolObjValue*)

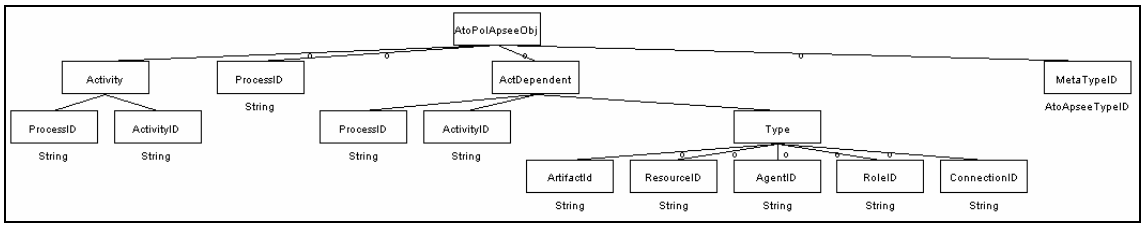

FIGURA 4.57 - Classe que identifica um objeto manipulado por uma Política Estática (*AtoPolApseeObj*)

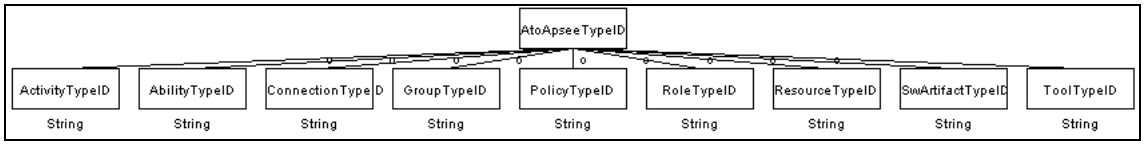

FIGURA 4.58 - Classe que identifica um tipo de objeto manipulado pelo sistema APSEE (*AtoApseeTypeID*)

A figura 4.59 apresenta o relacionamento entre as classes *APSEE*, *ProcessReuse* e *PolMethods*. *PolMethods* descreve os métodos definidos pela linguagem para serem usados pelos usuários na definição de propriedades de políticas.

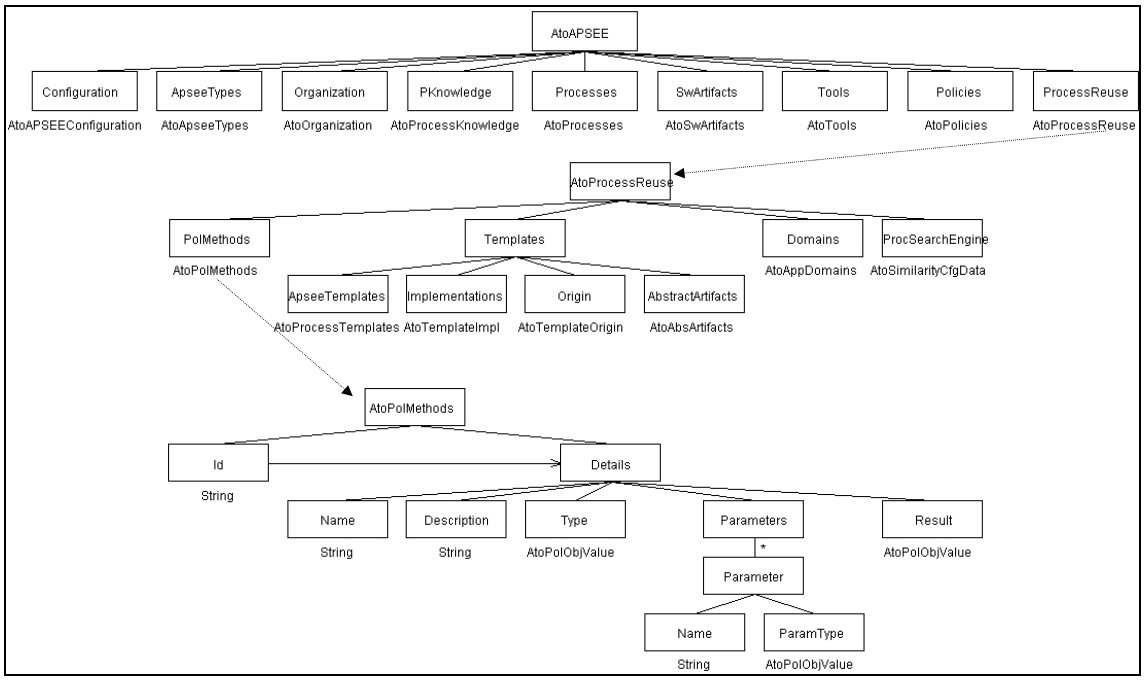

FIGURA 4.59 - Relacionamento entre as classes *APSEE*, *ProcessReuse* e *PolMethods*

4.4.3 Visão geral do processo de verificação de Políticas Estáticas

O algoritmo para interpretação de Políticas Estáticas é informalmente descrito através do Pseudocódigo Estruturado da figura 4.60. Ao final da verificação das políticas para um processo, será gerado um *log* que registra as ocorrências, isto é, situações em que as políticas habilitadas foram violadas. Somente se o *log* não contiver qualquer referência a um erro, o valor do atributo *StaticOK* do processo ou *template* avaliado será alterado para 'verdadeiro'.

Parâmetros recebidos: *apsee* (do tipo APSEE) e *proc-id* (String)
organization_policies ← conjunto de políticas estáticas habilitadas para a organização
process_policies ← conjunto de políticas estáticas habilitadas para o processo *proc-id*
general_evaluation_log ← empty-list
activities ← conjunto de atividades que compõem o fragmento em questão
Enquanto Activities ≠ Empty-Set
    activity_under_evaluation ← arbitrary element of (activities)
    activity-id ←activity_under_evaluation.get_id()
    activity_policies ← conjunto de políticas habilitadas para activity-id
    enabled_policies ← activity_policies ∪ process_policies ∪ organization_policies
    activity_log ← empty-list
    Enquanto Enabled_policies ≠ Empty-Set // verifica para atividade activity-id todas as políticas habil.
        policy-id ← arbitrary element of (enabled_policies).get_id()
        Se *enabling_properties_satisfied*(process-id, activity-id, policy-id)
        Então Activity_log ← *evaluate_policy_activity*(process-id, activity-id, policy-id)
        general_evaluation_log ← Concatenation(general_Evaluation_Log, activity_log)
        enabled_Policies ← other elements of(enabled_Policies)
    Fim-Enquanto
    activities ← other elements of(activities)
Fim-Enquanto
Resultado: general_evaluation_log: tipo StaticPolEval

FIGURA 4.60 - Pseudocódigo Estruturado do procedimento de verificação de políticas estáticas

4.4.4 Semântica Algébrica do Mecanismo de Interpretação

O procedimento de verificação de políticas para um (fragmento de) processo de software gera como resultado um objeto da classe *StaticPolEval* (figura 4.61) registrando, para verificação realizada em um processo (nodo *ProcessId*) e num instante específicos (Date e Time), todas as ocorrências que surgiram, agrupadas por atividades. No campo *Log* da classe *StaticPolEval* são armazenadas ocorrências (*warnings* ou *errors*) para cada uma das atividades que compõem o fragmento de processo em avaliação: objetos da classe *ActEvalLog* (figura 4.62) são criados para registrar especificamente a ocorrência gerada pela avaliação de uma atividade (se existirem).

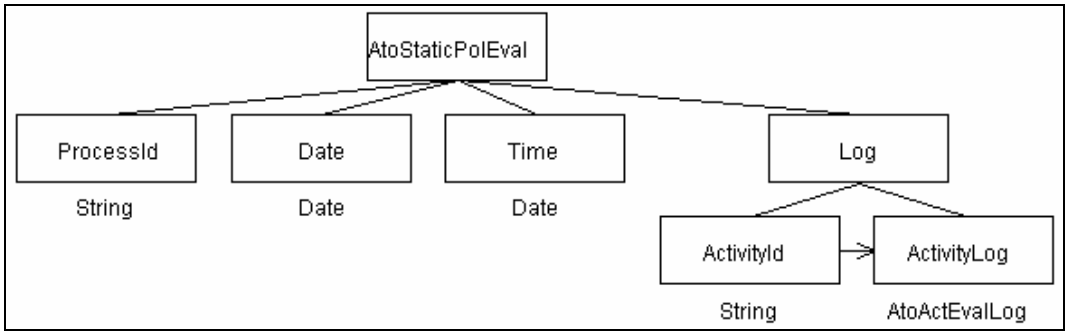

FIGURA 4.61 - Registro da avaliação de políticas estáticas em um processo de software (*AtoStaticPolEval*)

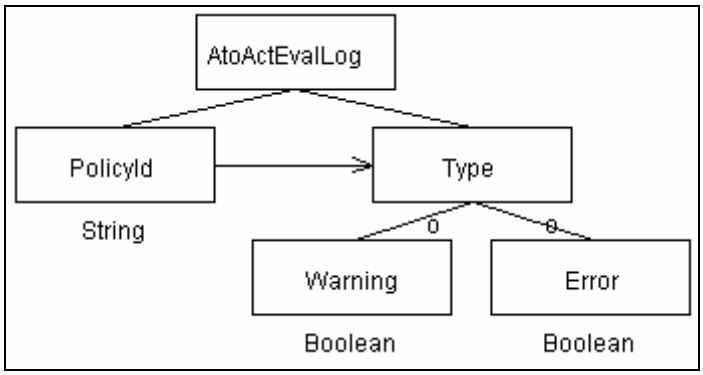

FIGURA 4.62 - Mapeamento que armazena ocorrências na avaliação de políticas para uma atividade (*AtoActEvalLog*)

A função *verify_static_policies* (figura 4.63) é a principal do mecanismo de interpretação fornecido, recebendo como parâmetros um objeto do tipo *AtoAPSEE* (instância do meta-gerenciador de processos, com informações sobre todos os processos de uma organização) e um String (*proc-id*) que identifica um fragmento de processo a ser verificado, gerando como resultado um objeto do tipo *StaticPolEval*. Para tanto, é criado um objeto inicial de *StaticPolEval* (item {2}, ao qual serão incorporados os dados gerados pelo *log* de verificação das atividades), e obtidas as políticas habilitadas em todo o processo {3}, e em toda a organização {4}. Em {5}, são obtidas todas as atividades que compõem o fragmento de processo. Todos esses objetos construídos são passados como argumentos para a função *verify_static_policies_activities* {1}, descrita no parágrafo a seguir.

```
Definição: verify_static_policies: APSEE, String → StaticPolEval

verify_static_policies(apsee, proc-id)
=       verify_static_policies_activities(apsee,                                        {1}
                ICS(StaticPolEval, create, proc-id, <current_date(), current_time()>),  {2}
        {3}     ICS(Processes, get_enabled_static_policies, select-Processes(apsee), <proc-id>) ∪
                ICS(Organization,get_enabled_static_policies,select-Organizaton(apsee)), {4}
                ICS(Processes, get_set_activities, select-Processes(apsee), <proc-id>), {5}
        proc-id))
```

FIGURA 4.63 - A função principal do interpretador de políticas (*verify_static_policies*)

A função *verify_static_policies_activities* (figura 4.64) realiza a avaliação de uma atividade específica do processo, adicionando um novo item de *Log* no objeto *StaticPolEval* se tal avaliação gerar alguma ocorrência (i.e., *warning* ou *error*). Esta função se baseia em chamadas recursivas {1}, tratando cada uma das atividades que compõem o processo em questão (conjunto *setOfActivityIds*). O item {2} mostra a transformação que ocorre no objeto de *StaticPolEval* recebido: a esse será adicionado o resultado da avaliação de uma atividade específica ao *Log* de ocorrências do processo (função *include_activity_log*). No item {3}, o conjunto de políticas que precisam ser verificadas (que já possuía as políticas habilitadas para a organização e para o processo) é complementado com as políticas habilitadas especificamente para a atividade em questão {4}. Finalmente, são determinadas as condições de parada para a recursão: quando não existem mais atividades a serem verificadas {5}.

```
Definição:
verify_static_policies_activities: APSEE, StaticPolEval, SetOfString, SetOfString, String → StaticPolEval

verify_static_policies_activities(apsee, staticPolEval, setOfPolicyIds, add(setOfActivityIds, act-id), proc-id)
=   verify_static_policies_activities(apsee,                                        {1}
        ICS(StaticPolEval, include_activity_log, staticPolEval,                     {2}
            <act-id,
            verify_static_policies_activity(
                apsee, proc-id, act-id,
                setOfPolicyIds ∪                                                    {3}
                ICS(Processes, get_enabled_static_policies,
                                        select-Processes(apsee),                    {4}
                <proc-id, act-id>))>),
            setOfPolicyIds, setOfActivityIds, proc-id))

verify_static_policies_activities(_, staticPolEval, _, empty-set, _) = staticPolEval    {5}
```

FIGURA 4.64 - Função que avalia todas as atividades que compõem o fragmento de processo (*verify_static_policies_activities*)

O mecanismo de verificação de políticas habilitadas para uma atividade específica é determinado pela função *verify_static_policies_activity* apresentado na figura 4.65. Esta função verifica inicialmente se as propriedades de habilitação (pré-condições ou *enabling properties*) da política na atividade em questão são satisfeitas {1}. Se as pré-condições forem atendidas, isto significa que a política em questão será avaliada em função da atividade conectada (função *evaluate_policy_activity* em {2}) e o resultado desta avaliação será agrupado com o resultado da avaliação das políticas restantes (item {3}). A condição de parada de recursão é definida pela inexistência de políticas a serem avaliadas (item {4}).

```
Definição: verify_static_policies_activity: APSEE, String, String, SetOfString → ActEvalLog

verify_static_policies_activity(apsee, proc-id, act-id, add(setOfPolicyIds, pol-id))
=       if      enabling_properties_satisfied(apsee, proc-id, act-id, pol-id)                          {1}
        then    ICS(ActEvalLog, composition,
                        evaluate_policy_activity(apsee, proc-id, act-id, pol-id),                      {2}
                        <verify_static_policies_activity(apsee, proc-id, act-id, setOfPolicyIds)>)     {3}
        else    verify_static_policies_activity(apsee, proc-id, act-id, setOfPolicyIds)

verify_static_policies_activity(_, _, _, empty-set) = empty-mapping                                    {4}
```

FIGURA 4.65 - Função que avalia todas as políticas habilitadas para uma atividade específica (*verify_static_policies_activity*)

O anexo 8 apresenta algumas das funções adicionais mais importantes que definem a semântica de *enabling_properties_satisfied* e *evaluate_policy_activity*.

5 Protótipo do modelo APSEE Reuse em PROSOFT-Java

A especificação algébrica - usando o PROSOFT-Algébrico e Gramáticas de Grafos - descrita no capítulo anterior serviu de base para a implementação de protótipos no ambiente PROSOFT-Java. Tal implementação teve como principal objetivo fornecer uma base computacional que permitisse a experimentação prática de alguns dos componentes críticos do modelo *APSEE*-Reuse proposto.

Embora uma descrição e avaliação crítica do ambiente PROSOFT-Java estejam fora do escopo desse texto, as seções a seguir apresentam uma visão geral da infraestrutura de apoio disponível, enfatizando as características que influenciaram na implementação experimental do modelo *APSEE*-Reuse proposto. Portanto, as seções 5.1 e 5.2 descrevem, respectivamente, o ambiente PROSOFT-Java e o sistema *APSEE* que forneceram a infraestrutura de software para a implementação do meta-modelo aqui proposto. A seção 5.3 descreve o editor de *templates*, enfatizando os aspectos relacionados com a especificação descrita no capítulo 4. A seção 5.4 descreve as modificações introduzidas no editor de processos executáveis para restringir a adaptação de processos obtidos a partir de *templates*. Finalmente, a seção 5.5 descreve a implementação fornecida para o interpretador de Políticas Estáticas.

5.1 O ambiente PROSOFT-Java

PROSOFT-Java é a denominação da mais recente[39] implementação do paradigma PROSOFT, na forma de um ambiente homogêneo e integrado de desenvolvimento de software escrito na linguagem Java (Schlebbe, [SCH97]). O desenvolvimento atual do PROSOFT-Java é resultado do esforço cooperativo de estudantes e pesquisadores do PPGC-UFRGS e da *Fakultät Informatik* da *Universität Stuttgart* (Alemanha), sob orientação do Prof. Dr. Daltro José Nunes.

O núcleo do PROSOFT-Java consiste de um conjunto de classes e interfaces Java que cooperam entre si para fornecer a funcionalidade requerida pelo paradigma PROSOFT (descrito em [NUN94]). O principal objetivo do ambiente é estabelecer uma infraestrutura que apóie o desenvolvimento de software de alta complexidade através da integração de ferramentas escritas em um paradigma próprio. Atualmente, o PROSOFT-

[39] A primeira versão monousuária do PROSOFT foi desenvolvida em Solaris-Pascal, como descrito por Nunes [NUN92]. Posteriormente, o PROSOFT-Distribuído foi desenvolvido em Pascal e C, segundo descrito por Granville [GRA96] e Schlebbe [SCH95]. Finalmente, segundo Schlebbe [SCH97], em 1997 foi selecionado Java como nova linguagem hospedeira para o ambiente.

Java é um ambiente portável[40], distribuído[41] e cooperativo[42] que integra ferramentas CASE para auxiliar diferentes fases do processo de software.

É importante observar que há uma correspondência entre ATOs Algébricos e ATOs Java, conforme ilustrado pelo exemplo da figura 5.1. Como indicado pela figura, o conceito de classe do PROSOFT-Algébrico é diretamente mapeado para implementação em PROSOFT-Java, enquanto que métodos precisam ser implementados a partir de derivação das funções algébricas (axiomas) correspondentes.

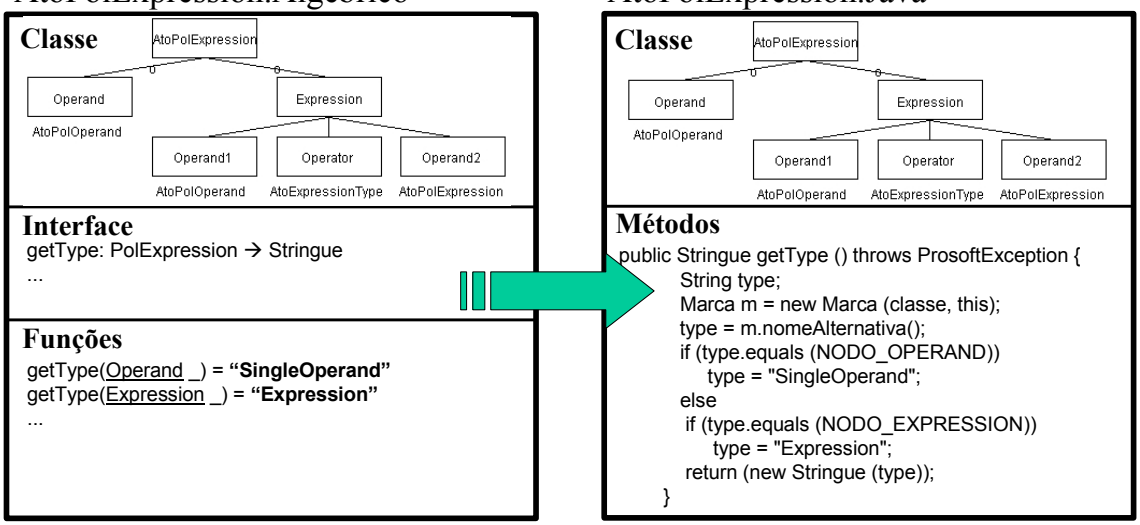

FIGURA 5.1 - Exemplo esquemático de derivação manual de ATOs Java a partir de ATOs Algébricos

A ferramenta ATO-Classe está disponível para edição de classes PROSOFT, conforme ilustrado na figura 5.2, servindo para gerar automaticamente o código-fonte do ATO-Java correspondente, o qual inclui funções para criação e remoção de objetos e para obtenção (funções rotuladas com o prefixo *get*) e alteração (funções rotuladas com o prefixo *set*) dos valores armazenados nos nodos-folha do tipo definido.

A definição de um ATO interativo em Java é dividida em dois arquivos. O primeiro é rotulado com o mesmo nome da classe fornecida, com a extensão *.java*. As funções de interação com o usuário são incluídas em um arquivo separado, cujo rótulo inclui o sufixo SI no nome do ATO, e inclui métodos que são relacionados com as opções de menus (denominada de "parte semântica" do ATO). O esqueleto dos dois arquivos são gerados automaticamente pela operação "Gera Ato+Sem" disponível no ATO-Classe.

Na implementação atual do sistema PROSOFT, as funções adicionais especificadas algebricamente devem ser traduzidas manualmente para métodos escritos de acordo com a sintaxe da linguagem Java (Gosling et al em [GOS96]), podendo utilizar os construtores disponíveis pelo pacote *prosoft.kernel* (uma descrição completa

---

[40] Portável no sentido em que o sistema é executável nas plataformas para as quais estão disponíveis o *Java Runtime Environment* [SUN02a].

[41] A distribuição entre ATOs é fornecida atualmente através do mecanismo *ad-hoc* de comunicação denominado ICS-Distribuído que está implementado sobre o protocolo Java-RMI [SUN02b].

[42] Funcionalidade para trabalho cooperativo está disponível a partir da extensão denominada PROSOFT-Cooperativo, descrita por Reis [REI98], Nunes [NUN01] e Alves [ALV2002].

acerca do desenvolvimento de ATOs-Java foi disponibilizada por Schlebbe em [SCH97] e [SCH2002]).

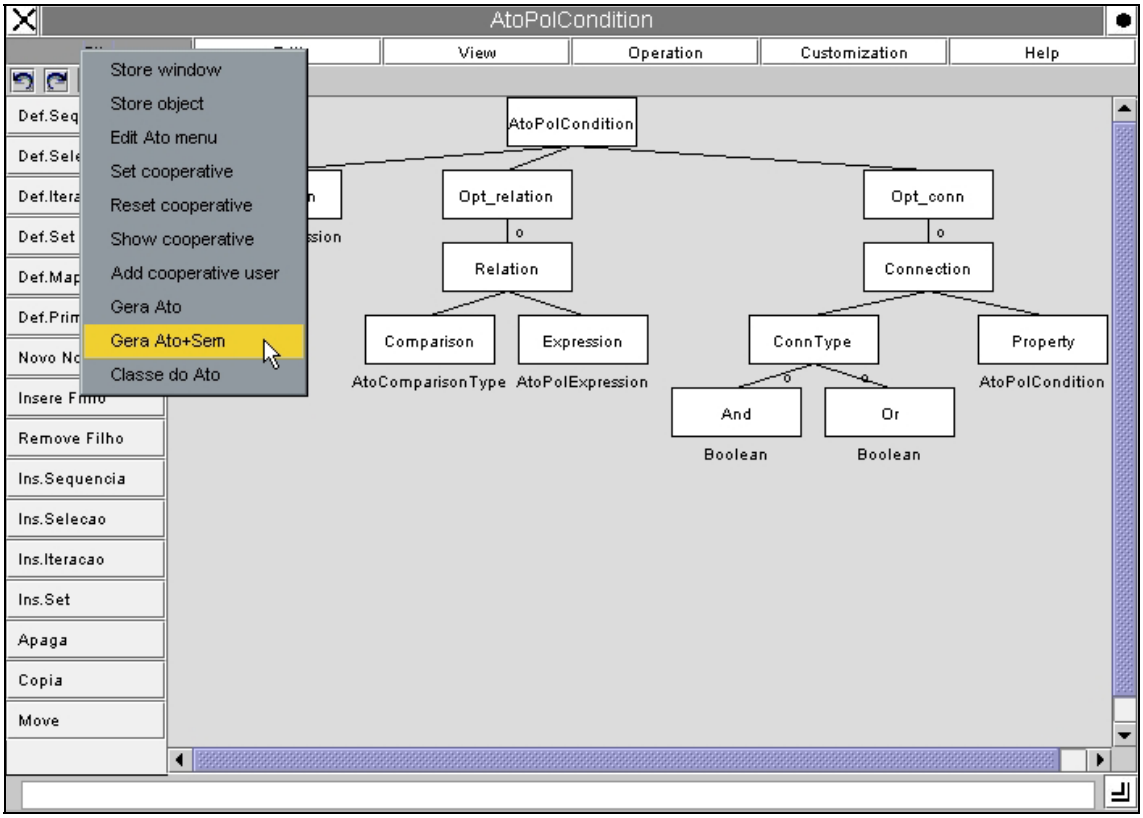

FIGURA 5.2 - Tela do Editor ATO Classe

5.2 O ambiente APSEE

A implementação atual do sistema *APSEE* é composta por mais de uma centena de ATOs-Java, como resultado do trabalho de diferentes membros do grupo de pesquisa. Os ATOs estão relacionados entre si como definido pela arquitetura apresentada no capítulo 3, no qual o gerenciador de processos (*APSEE*-Manager) interconecta os diferentes serviços para definição, visualização e execução de processos. O *APSEE* se vale dos serviços de apoio ao trabalho distribuído e cooperativo fornecidos pelo PROSOFT-Java permitindo, por exemplo, que o gerenciador de processos execute em um servidor, enquanto que instalações remotas do PROSOFT podem executar as agendas de tarefas.

5.3 O editor de *Templates*

Um editor para *Templates* foi desenvolvido em PROSOFT-Java e integrado ao sistema *APSEE*. O editor implementa os tipos de dados, funções e regras descritas no capítulo 4 (nas seções 4.2.1, 4.2.2 e 4.3, respectivamente). A implementação do editor de *Templates* foi realizada no período de outubro de 2001 a abril de 2002, contando com o auxílio dos Srs. Heribert Schlebbe e Marcelo Abreu (pesquisadores associados ao projeto).

A figura 5.3 apresenta uma tela com o editor de *templates*, apresentando no destaque a classe *AbsProcessModel* relacionada com o ATO. O PROSOFT define que a representação (gráfica ou textual) de um objeto para o usuário deva ser implementada

por um método Java denominado “exibe”, o qual é automaticamente invocado pelo sistema quando necessário [SCH97]. Assim, a tela na figura 5.3 apresenta o resultado da função “exibe” para o objeto armazenado no sistema no momento. A figura ainda inclui no retângulo interior a classe relacionada com o objeto sendo exibido no momento (tipo *AtoAbsProcessModel*).

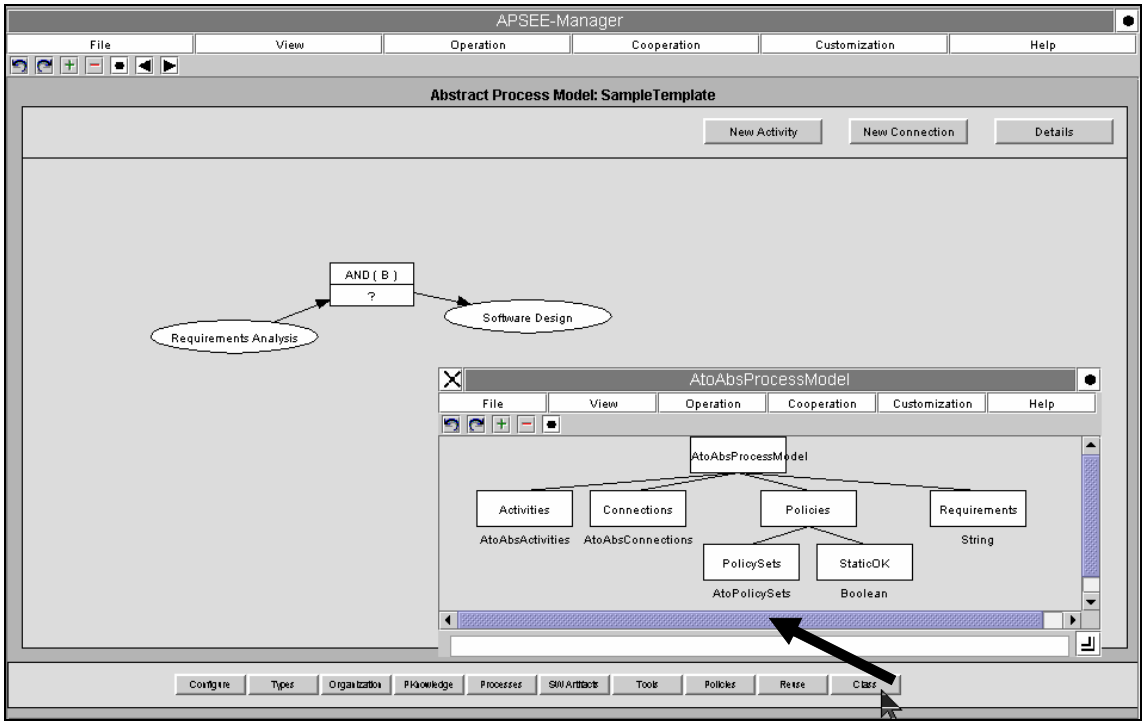

FIGURA 5.3 - Tela do editor de *templates*, apresentando no destaque a classe *AbsProcessModel* correspondente

Alguns atributos dos ATOs especificados para o meta-modelo ~~APSEE~~-Reuse armazenam informações essencialmente textuais, o que implica no fato que a implementação atual do método “exibe” está relacionada com os formulários a serem manipulados pelo usuário. A figura 5.4 apresenta dois exemplos de formulários, exibidos para descrever os detalhes de uma atividade abstrata (i.e., objeto da classe *AbsActivities*). No canto superior esquerdo da figura é apresentado o formulário para a descrição dos atributos *ID*, *Name*, *TypeID* e *Description* do objeto, tal como ilustrado pelas setas inseridas no diagrama. Na parte inferior da figura, está o formulário que define as instâncias de Políticas habilitadas para uma atividade, correspondendo ao resultado do *exibe* para um objeto de *PolicySets*.

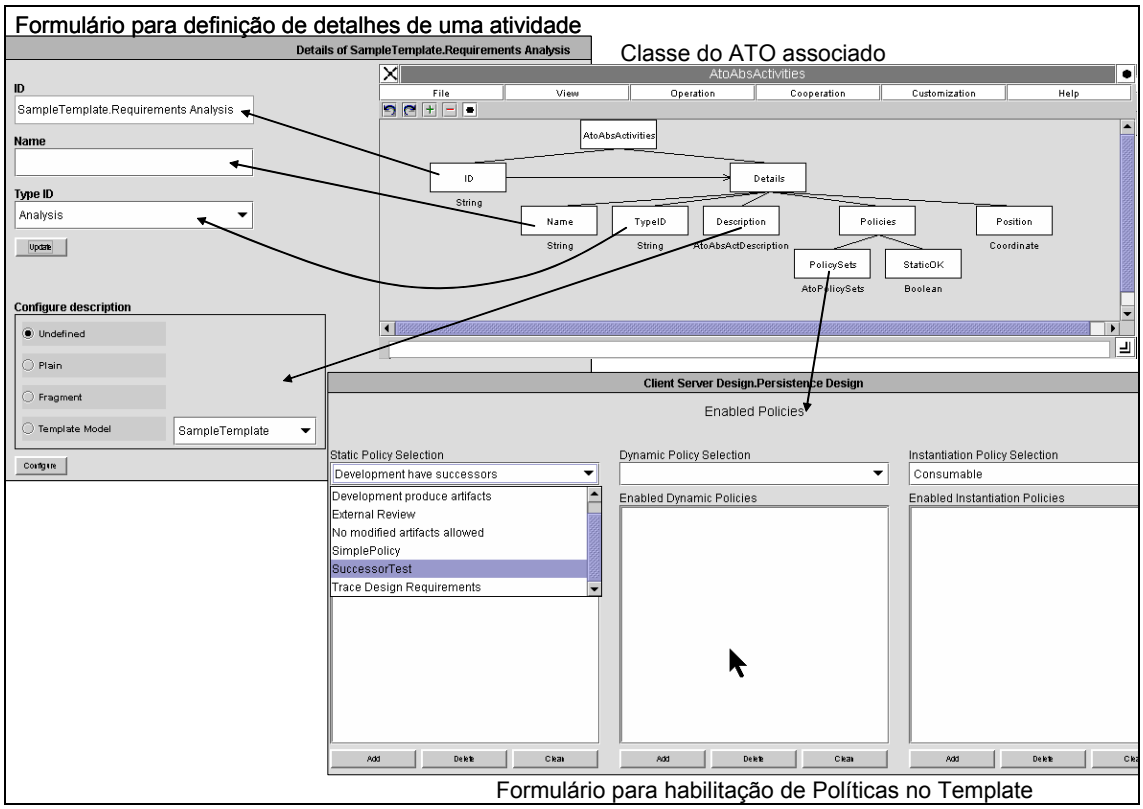

FIGURA 5.4 - Formulários para a definição de detalhes para uma atividade

O editor ainda fornece serviços para edição, visualização e composição dos diferentes detalhes envolvidos na modelagem de processos, tal como no exemplo da figura 5.5.

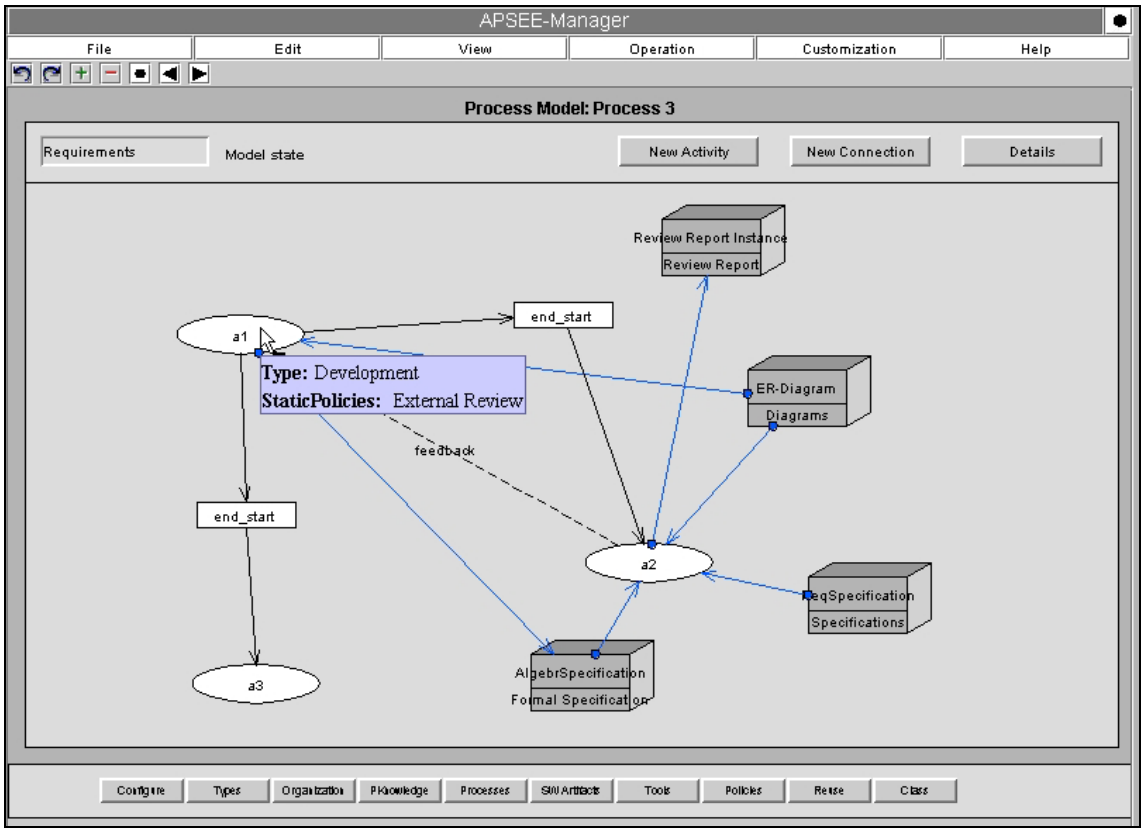

FIGURA 5.5 - Exemplo de uso do editor de *Templates* disponível com uma instância de processo

O editor de *templates* também implementa as regras descritas nos anexos desse trabalho. No caso das regras sintáticas para *templates*, foram implementados métodos em PROSOFT-Java tendo por base as regras definidas, mantendo uma correspondência de nomes entre as regras fornecidas e os métodos derivados. A figura 5.6 apresenta como ilustração o exemplo da regra G1.1, a qual inspirou a implementação do método *semNewAbsActivity* que foi incorporado ao código do *AtoAbsProcessModelSI*. Por fim, na porção mais inferior da figura foi incluída uma tela que descreve o comportamento expresso pela função implementada.

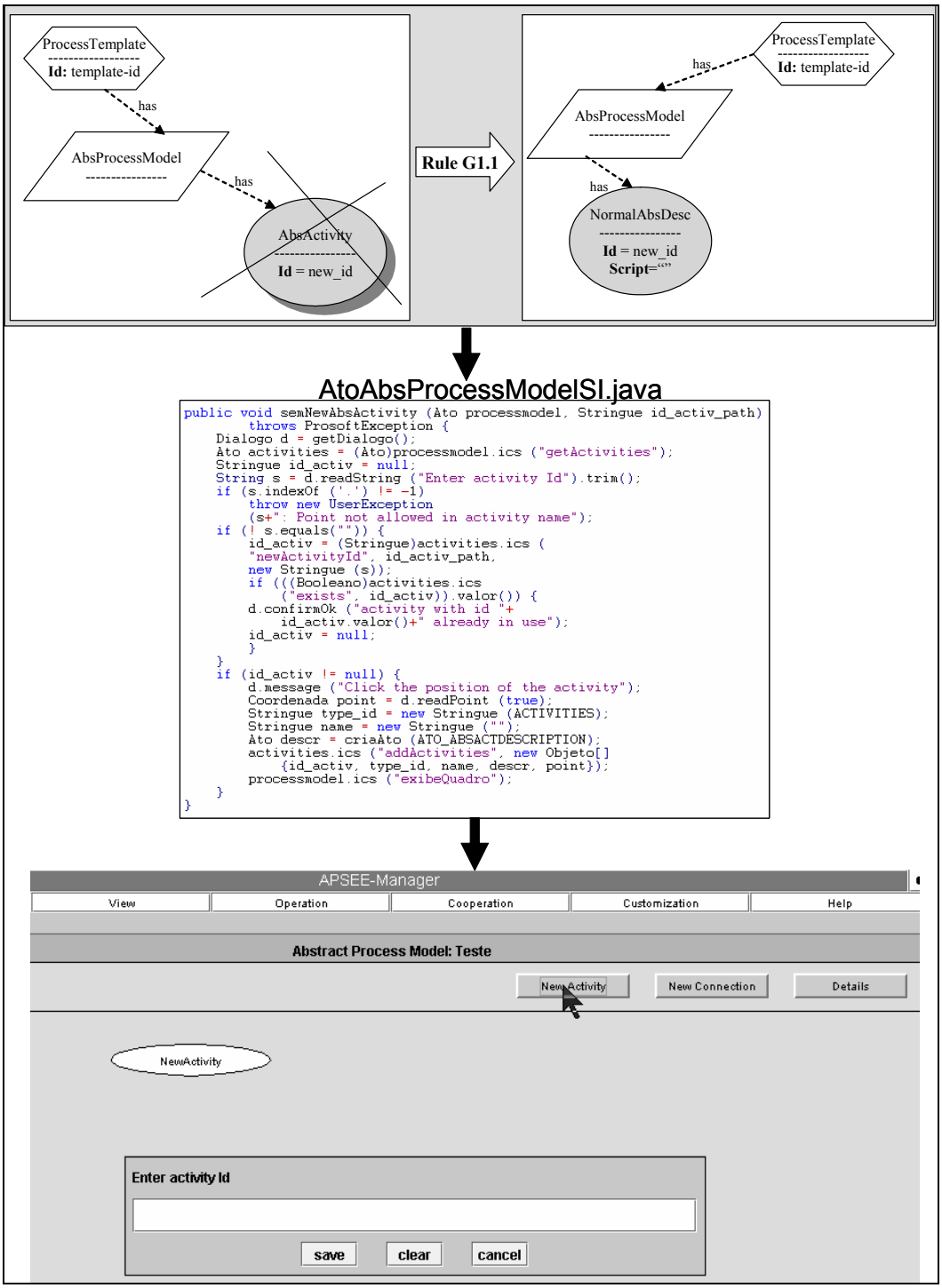

FIGURA 5.6 - Implementação para a regra G1.1 - *NewAbsActivity*

5.4 Modificações no editor de Processos Executáveis

As regras para adaptação de *template*, descritas na seção 4.3.3 e no Anexo 7 desse texto, foram implementadas como métodos no editor de processos executáveis. A figura 5.7 apresenta uma tela do editor de processos executáveis com a solicitação do usuário em remover a atividade *Case Base Development* em um processo derivado do

template INRECA. Como nesse exemplo a adaptação é **restrita** e a atividade consta da definição do *template* original, é exibida a mensagem de erro à direita da figura.

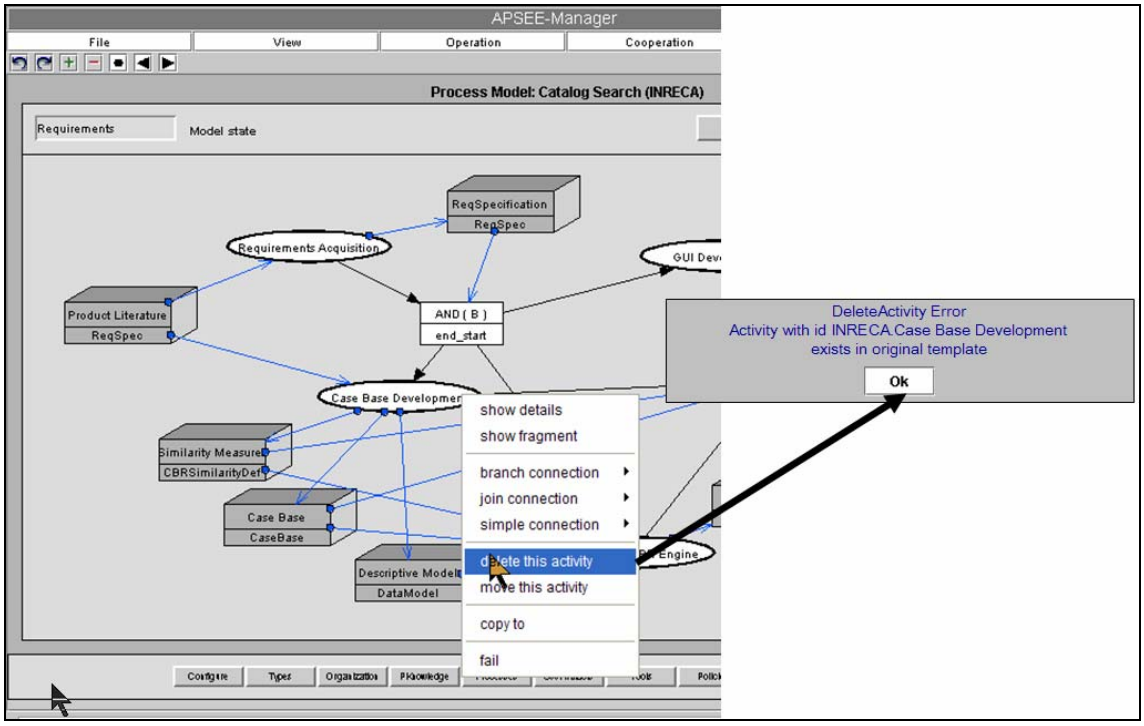

FIGURA 5.7 - Implementação da regra AdaptRule 1.4

5.5 O interpretador de Políticas Estáticas

Um protótipo do interpretador de Políticas Estáticas foi derivado a partir da especificação algébrica apresentada na seção 5.4. A implementação desse mecanismo contou com o apoio do Sr. Heribert Schlebbe e foi realizada no período de junho a outubro de 2001.

A figura 5.8 ilustra o mapeamento da função algébrica principal *verify_static_policies* para método Java correspondente no *AtoStaticPoliciesSI*. Na parte superior da figura é colocada a especificação algébrica para a função *verify_static_policies* (previamente apresentada na seção 4.4.4), enquanto que o retângulo situado na parte inferior descreve o método Java obtido a partir de derivação manual. Como pode ser percebido através do exemplo, manteve-se uma correspondência entre os nomes de funções e identificadores de objetos usados na especificação e a implementação dos métodos correspondentes.

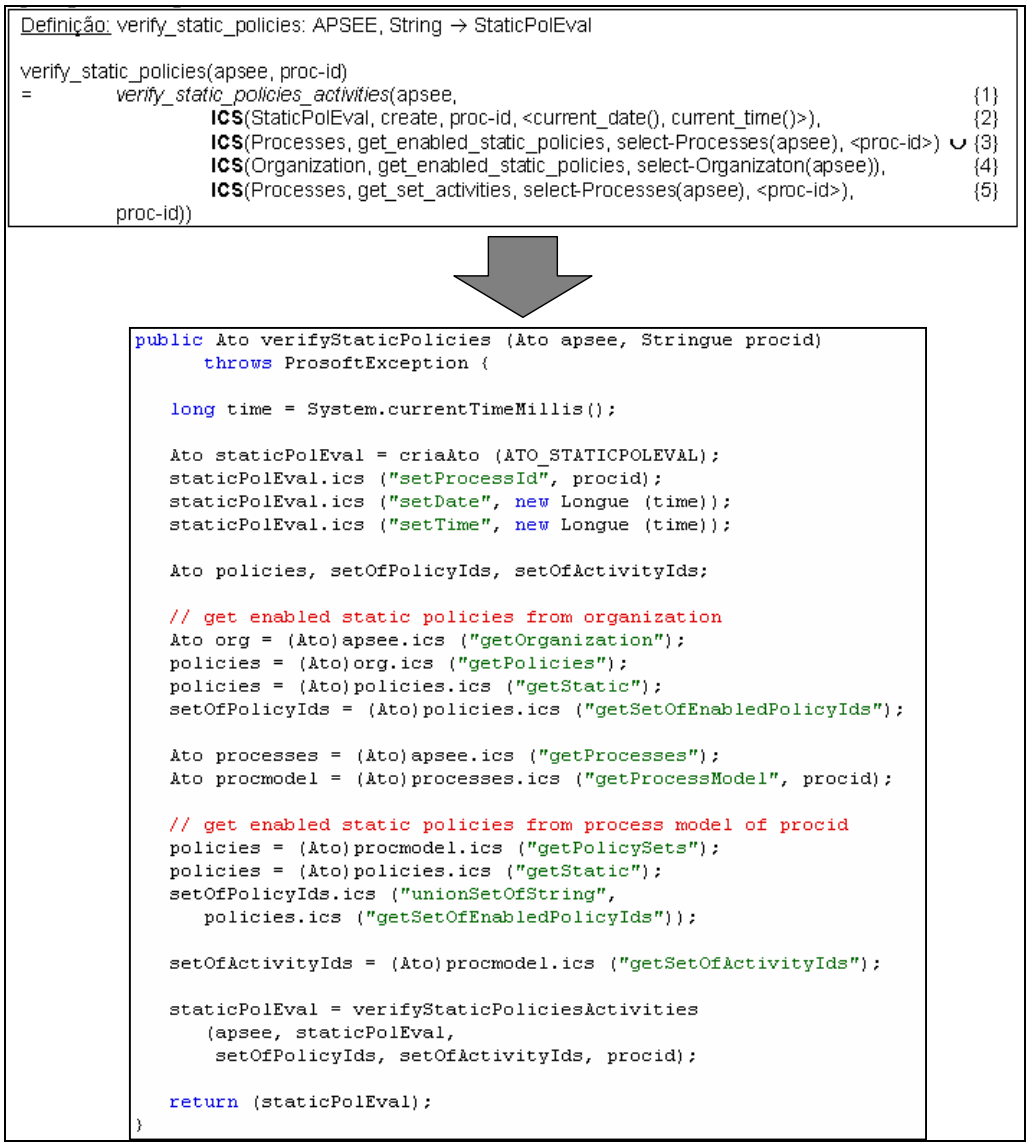

FIGURA 5.8 - A função *verify_static_policies*: especificação algébrica e a sua derivação para PROSOFT-Java

A figura 5.9 apresenta o editor disponível para Políticas Estáticas. Nessa figura é incluído uma Política denominada *External Review*, a qual restringe que atividades do tipo *Development* para que possuam exatamente um único sucessor (função *number_of_successors*). A atividade sucessora ainda deve ser do tipo *Verification*, devendo existir uma conexão de *feedback* desta com a primeira atividade. Finalmente, as atividades devem envolver grupos diferentes de pessoas na sua realização.

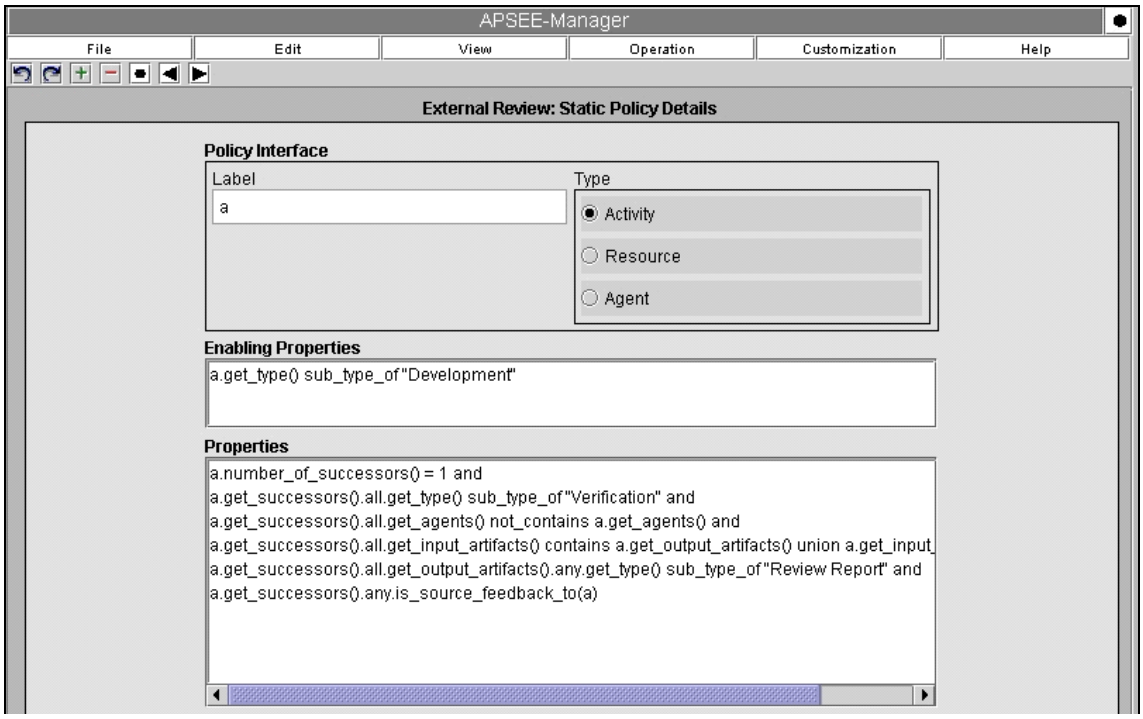

FIGURA 5.9 - Formulário para edição dos detalhes de uma Política Estática

A figura 5.10 apresenta o resultado da avaliação das políticas habilitadas em *Process 3* (previamente apresentado na figura 5.5), como exemplo de resultado para a função "exibe" no *AtoStaticPolEval*, cujas classes estão reproduzidas na parte superior da figura. O resultado da etapa de verificação de políticas estáticas em processos APSEE é composto por uma lista ordenada de avisos (*warnings*), gerados por políticas não obrigatórias. No exemplo dado, a avaliação falhou na verificação da política *External Review* para a atividade *a1*, visto que a avaliação da primeira propriedade da política (a.number_of_successors() = 1) não foi satisfeita. Portanto, um erro é obtido no caso de *External Review* ser definida como uma política obrigatória.

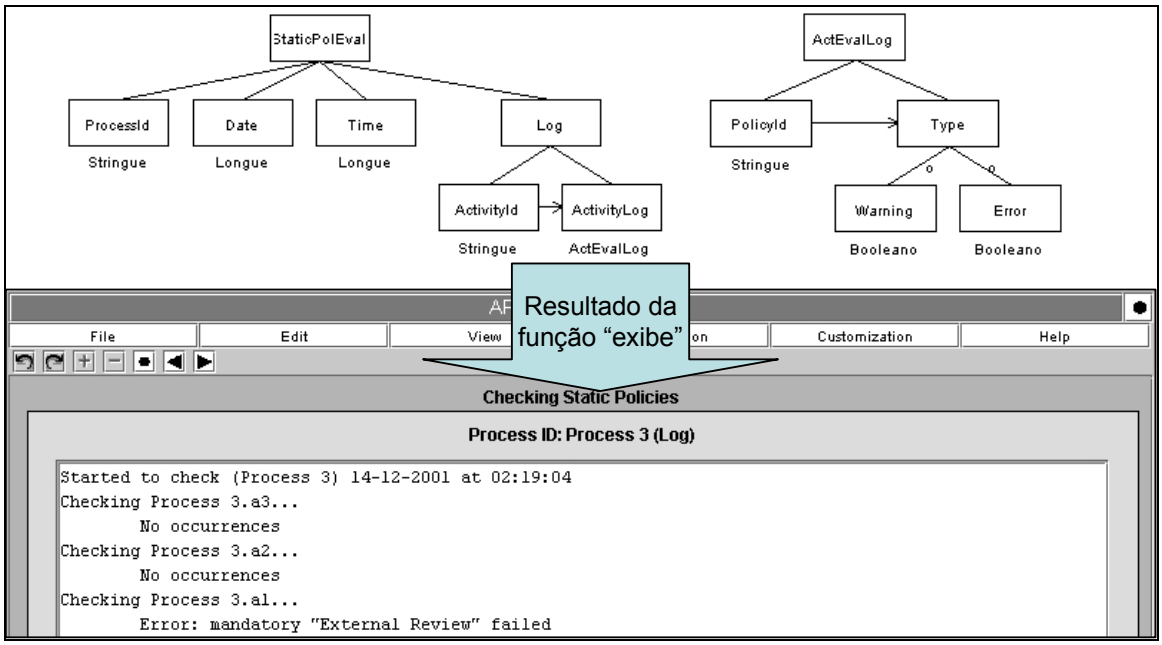

FIGURA 5.10 - *Log* da verificação das políticas estáticas habilitadas em um processo exemplo

6 Estudos de caso

Esse capítulo descreve o uso prático do modelo proposto em situações que envolvem a construção e reutilização de *templates* para problemas reais. Esses estudos de casos foram conduzidos com o objetivo de avaliar o meta-modelo proposto, os construtores sintáticos disponíveis e as ferramentas de apoio implementadas em diferentes situações relacionadas com a modelagem e reutilização de processos, variando de exemplos extraídos da literatura até novos modelos projetados para atender necessidades específicas.

A experiência descrita nesse capítulo indica que a construção de *templates* invariavelmente envolve a utilização da grande maioria dos tipos de dados descritos nas seções anteriores. Isto ocorre em conseqüência da necessidade de se representar os diferentes aspectos nas definições fornecidas para modelagem de *templates*.

Em virtude da documentação completa gerada por um *template* real ocupar um número considerável de páginas, esse capítulo descreve resumidamente algumas das principais experiências, fornecendo referências bibliográficas em que o leitor interessado pode obter as descrições completas correspondentes.

6.1 O *template* PROSOFT-Java

O projeto de metodologias de desenvolvimento de software para o paradigma PROSOFT é objeto de investigação realizada no âmbito do projeto [NUN94]. Embora metodologias Orientadas a Objetos possam ser usadas - com alguma adaptação - em diferentes etapas do desenvolvimento de software para o PROSOFT, os aspectos específicos do paradigma e do seu ambiente precisavam ser documentados de uma forma mais precisa.

Com o surgimento do PROSOFT-Java, implementação do PROSOFT para uma plataforma de software muito mais popular e disponível que a versão anterior (Solaris-Pascal), constatou-se a necessidade de documentar o desenvolvimento de ATOs usando as novas ferramentas disponibilizadas. Um primeiro esforço nesse sentido foi o desenvolvimento do Manual do PROSOFT-Java (*Java-PROSOFT Guide* [SCH2002]), desenvolvido pela equipe alemã do projeto. Assim, a partir da constatação da necessidade de um detalhamento maior para esse processo, foi proposta a especificação de um *template* denominado *JavaAtoImplementation*, com o objetivo de fornecer uma descrição genérica que pudesse ser reutilizada nas diversas implementações.

A figura 6.1 apresenta uma descrição parcial para o *template* proposto (somente as atividades e suas conexões estão incluídas na figura). Assim, de acordo com as ferramentas disponibilizadas pelo PROSOFT-Java, o desenvolvimento de ATOs Java é realizado por uma seqüência de atividades listadas abaixo:

- A atividade normal *Prosoft Class Definition* é responsável por definir uma classe PROSOFT usando o editor fornecido no ATO Classe;

- A atividade automática *Basic Ato Generation* gera o esqueleto de código-fonte correspondente à classe PROSOFT fornecida, utilizando uma função disponibilizada pelo ATO Classe;
- O fragmento *Add functionality* é responsável por definir os métodos adicionais necessários para que o ATO proposto atenda os requisitos especificados. Tal como expresso pelas conexões *XOR Branch* e *Join* fornecidas na parte inferior do diagrama, esse fragmento é opcional, podendo ser suprimido para ATOs simples que não requeiram métodos adicionais;
- A compilação do ATO (atividade automática *Ato Compiling*) é fornecida para invocar automaticamente o compilador Java para a geração do *bytecode* correspondente;
- A integração do novo ATO ao sistema PROSOFT constitui uma atividade rotulada como *Ato Integration*. Segundo [SCH2002], a integração é importante por definir os recursos do ATO que estarão disponíveis para acesso remoto por outros ATOs;
- Finalmente, a atividade *Ato Testing* descreve o teste final realizado no ATO recém implementado confrontando-o com os requisitos da aplicação desenvolvida.

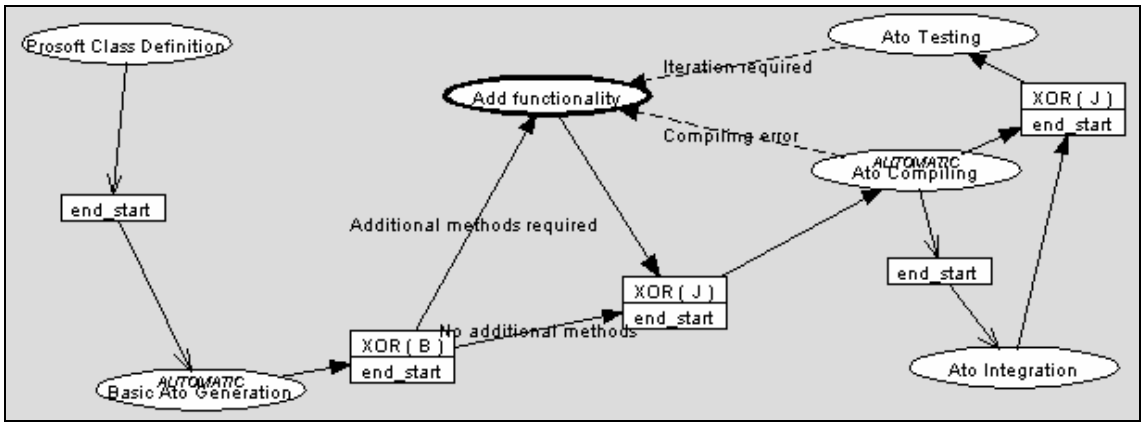

FIGURA 6.1 - Descrição parcial do *template* que descreve o desenvolvimento de software no PROSOFT-Java

Para cada atividade definida, foram especificados detalhes adicionais os quais são fornecidos na versão *online* do manual do PROSOFT (em [SCH2002]), e que acompanha a distribuição pública do pacote PROSOFT-Java atual. A figura 6.2 mostra, como exemplo, o *script* fornecido para a atividade *Prosoft Class Definition*.

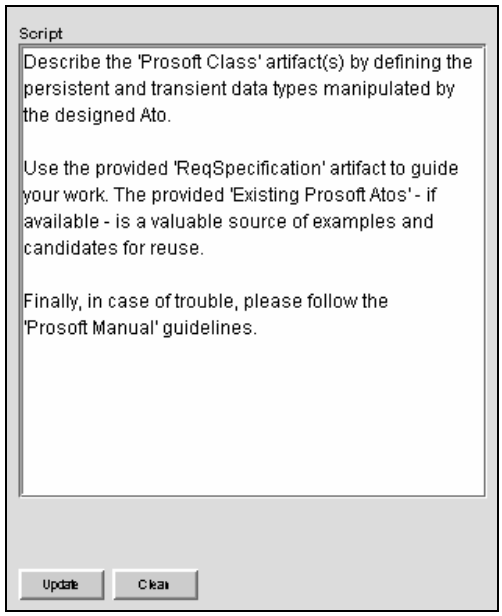

FIGURA 6.2 - Script da atividade *Prosoft Class Definition*

O *template JavaAtoImplementation* vem sendo utilizado no treinamento de novos programadores para o PROSOFT, assim como para guiar a construção de diversos ATOs do projeto. De fato, muitos dos componentes envolvidos no desenvolvimento do sistema *APSEE* - incluindo os tipos relacionados com *ProcessTemplates* - foram desenvolvidos com versões preliminares desse *template*. Assim, a evolução desse *template* contou com o apoio e *feedback* fornecidos pelos bolsistas e pesquisadores do projeto.

A experiência prática com a utilização do template proposto demonstrou que a maioria das adaptações necessárias envolve especializações do fragmento *Add functionality*. Tal constatação é natural, visto a diversidade de formalismos que podem ser utilizados no projeto dos métodos de um ATO. Assim, a figura 6.3 apresenta a adaptação livre usada na implementação do editor de *ProcessTemplates*, representando na parte superior o *template* original e na parte inferior o processo adaptado obtido. Nesta adaptação, contrastando o modelo adaptado com o original foram removidos alguns elementos, enquanto outros foram adicionados em função das especificidades do problema tratado, as quais são ressaltadas a seguir:

- Os artefatos foram instanciados, isto é, foram rotulados com nomes que possuem significado específico para o problema tratado;
- Dois artefatos adicionais foram fornecidos como entrada da atividade *Add functionality*, sendo utilizados para especificar o projeto dos métodos necessários para o ATO envolvido. O artefato *Template Editor GUI Design* possui um projeto dos formulários e editores gráficos usados na manipulação dos componentes de um *template*, enquanto que *EditorModelingRules* corresponde às regras fornecidas no Anexo 5 desse texto;
- Como o processo derivado envolvia a implementação de métodos adicionais, a atividade *Add functionality* foi tornada obrigatória (pela remoção das conexões XOR localizadas na parte inferior do diagrama do *template* original).

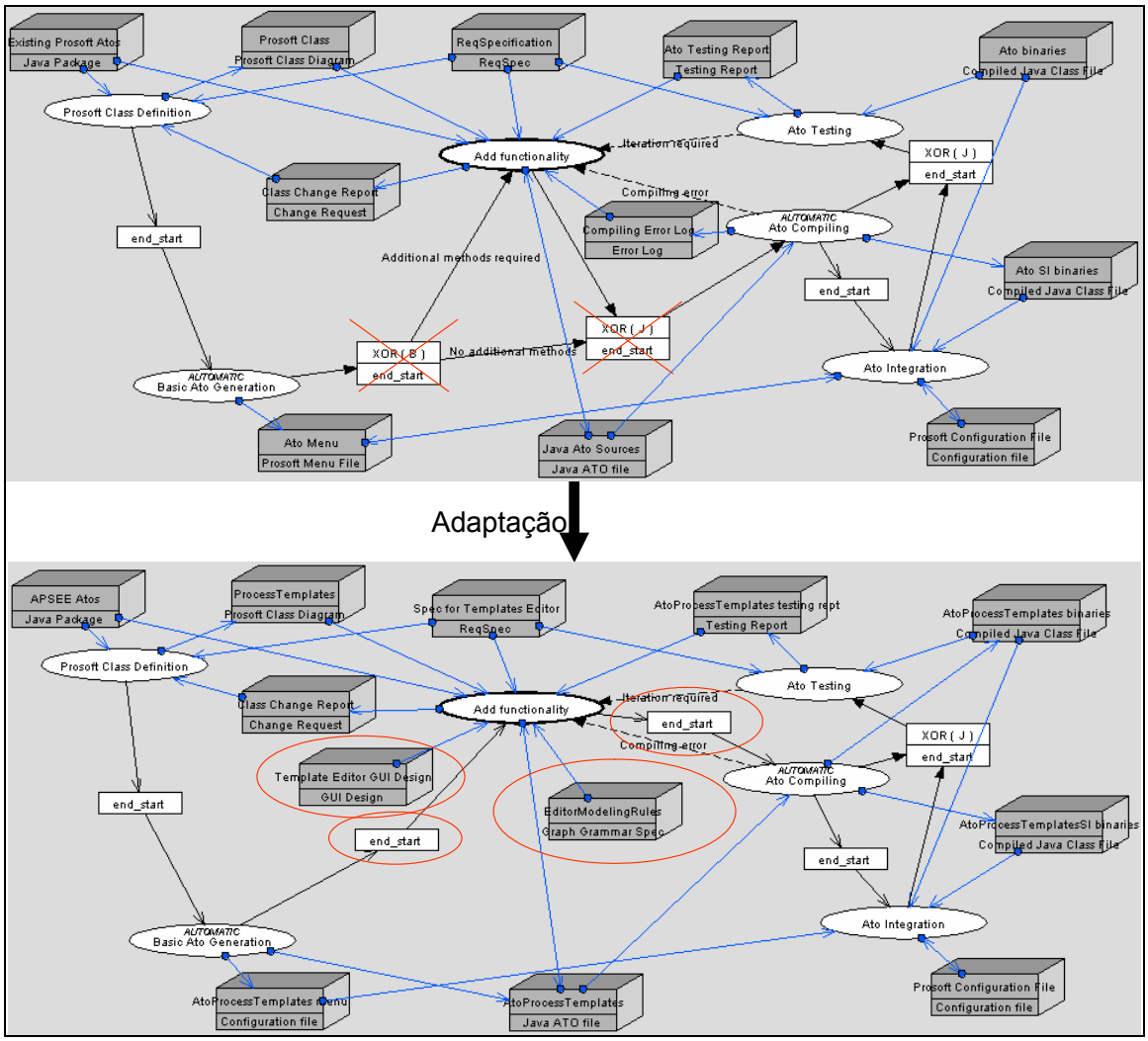

FIGURA 6.3 - Adaptação do *template JavaAtoImplementation* usado no desenvolvimento do editor de *templates*

6.2 Os *templates* para a metodologia CBR-INRECA

A definição de processos de software para apoiar o desenvolvimento de aplicações para contextos ou tecnologias específicos é uma tendência recente na literatura [BER99]. Tais modelos de processos têm o potencial de reduzir os riscos de adoção e o tempo de entrega dos produtos, fornecendo uma estrutura consistente que orienta os novatos na realização dos passos necessários no desenvolvimento dos produtos requeridos.

A metodologia INRECA, proposta por Bergmann et al em [BER99] para orientar o desenvolvimento de aplicações da tecnologia de Raciocínio Baseado em Casos (*Case-based Reasoning* - CBR), é um modelo de processo de software semi-formal - descrito através de uma notação proprietária e de narrativas em inglês - organizadas na forma de hiperdocumentos *HTML* e em diagramas para o editor Microsoft Visio [MIC2002]. A metodologia INRECA foi desenvolvida a partir da experiência do grupo de pesquisa da Universidade de Kaiserslautern (Alemanha) e seus diversos parceiros industriais na construção de sistemas que estão baseados em uma premissa básica: a busca de soluções para novos problemas deve estar relacionada às

soluções para problemas similares registradas em uma base de casos disponível no sistema.

A metodologia descreve um modelo genérico - denominado *Generic CBR Development* - o qual descreve o desenvolvimento de aplicações CBR de propósito geral. Processos para domínios de aplicações específicos também estão descritos, incluindo: sistemas de busca "inteligente" em catálogos (processo *Catalog Search*), sistemas de apoio à decisão (processo *Case-based Help Desk*) e sistemas de manutenção de equipamentos técnicos (processo *Development of maintenance applications for technical equipment*).

Templates correspondentes aos processos INRECA foram construídos de forma bem sucedida, estando descritos em um relatório de pesquisa [REI2001e]. A metodologia INRECA foi escolhida como estudo de caso para o modelo *APSEE*-Reuse por duas razões principais:

- A descrição original dos processos INRECA em [BER99], embora não seja formal, é bastante detalhada, o que permitiria a avaliação prática acerca do poder de expressão da linguagem de modelagem de *templates* aqui proposta;
- A metodologia INRECA - em especial, o processo *Catalog Search* - seria útil para guiar o desenvolvimento do componente *APSEE-SearchEngine*, proposto para realizar buscas de processos e seus componentes utilizando a tecnologia CBR. Esta experiência é resumidamente descrita na seção 6.5 a seguir.

A figura 6.4 apresenta o *template* de mais alto nível de para o processo *Generic CBR Development* (rotulado como *INRECA.Generic CBR*). No detalhe à esquerda da figura é apresentada a hierarquia de sub-processos envolvidos (visão fornecida pela ferramenta *APSEE*-Monitor, descrita por [SOU2002]).

A construção de *templates* a partir da metologia INRECA demonstrou alguns dos benefícios da utilização de um modelo formal: alguns erros no modelo original foram detectados em tempo de modelagem[43], enquanto que estratégias gerenciais descritas informalmente (através de instruções em linguagem natural) puderam ser especificadas precisamente através de Políticas Estáticas.

[43] Conforme descrito em [REI2001e], a documentação original de INRECA possui inconsistências em diversos pontos, incluindo a inexistência de artefatos citados na descrição dos *scripts* de atividades. Por exemplo, tomando-se o *template* exibido na figura 6.4, na documentação original o artefato *CBRApplicationPrototype* não estava associado à atividade *Feasibility Study*, o que evidentemente constitui um erro no modelo.

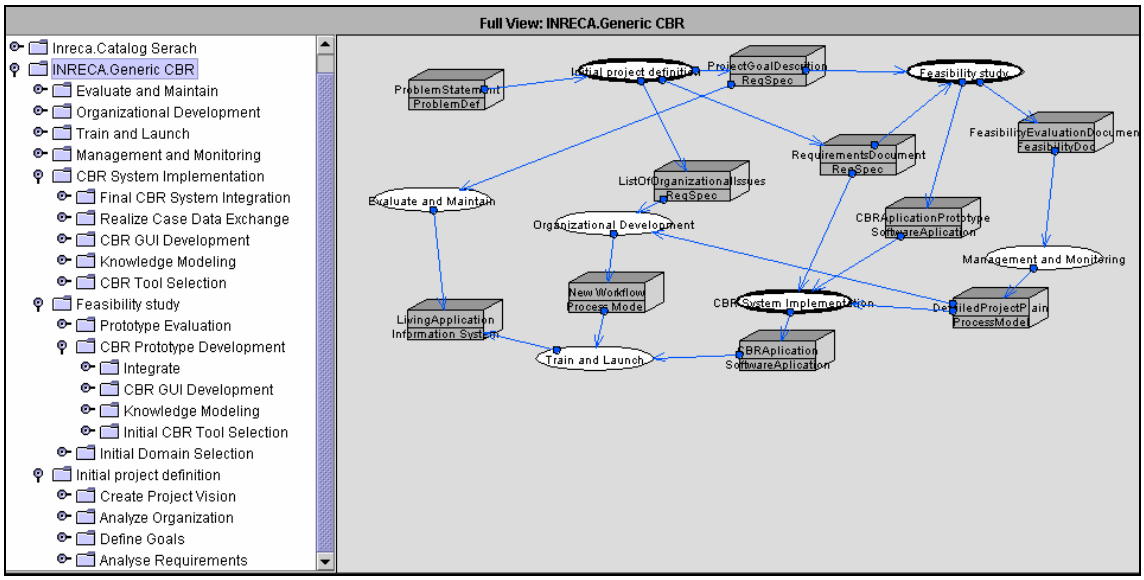

FIGURA 6.4 - Visão hierárquica para o *Template INRECA.Generic CBR*

Um exemplo para processo *Catalog Search* é fornecido pelas figuras 6.5 e 6.6 a seguir. A figura 6.5 apresenta duas telas capturadas a partir da documentação original INRECA. A janela localizada na parte superior esquerda da figura representa a atvidade *Test the Integrated System*, componente da atividade principal *System Integration*. Na janela localizada na parte inferior da figura, é apresentada a descrição textual fornecida para a atividade, sendo destacado o trecho que explicitamente define o comportamento análogo ao expresso pela Política Estática *External Review* (mostrada anteriormente pela figura 5.9). Assim, na tela capturada do *template INRECA.Catalog Search* expresso com a linguagem aqui proposta (figura 6.6) é mostrado no detalhe a Política Estática correspondente habilitada, Política esta construída manualmente a partir da documentação original.

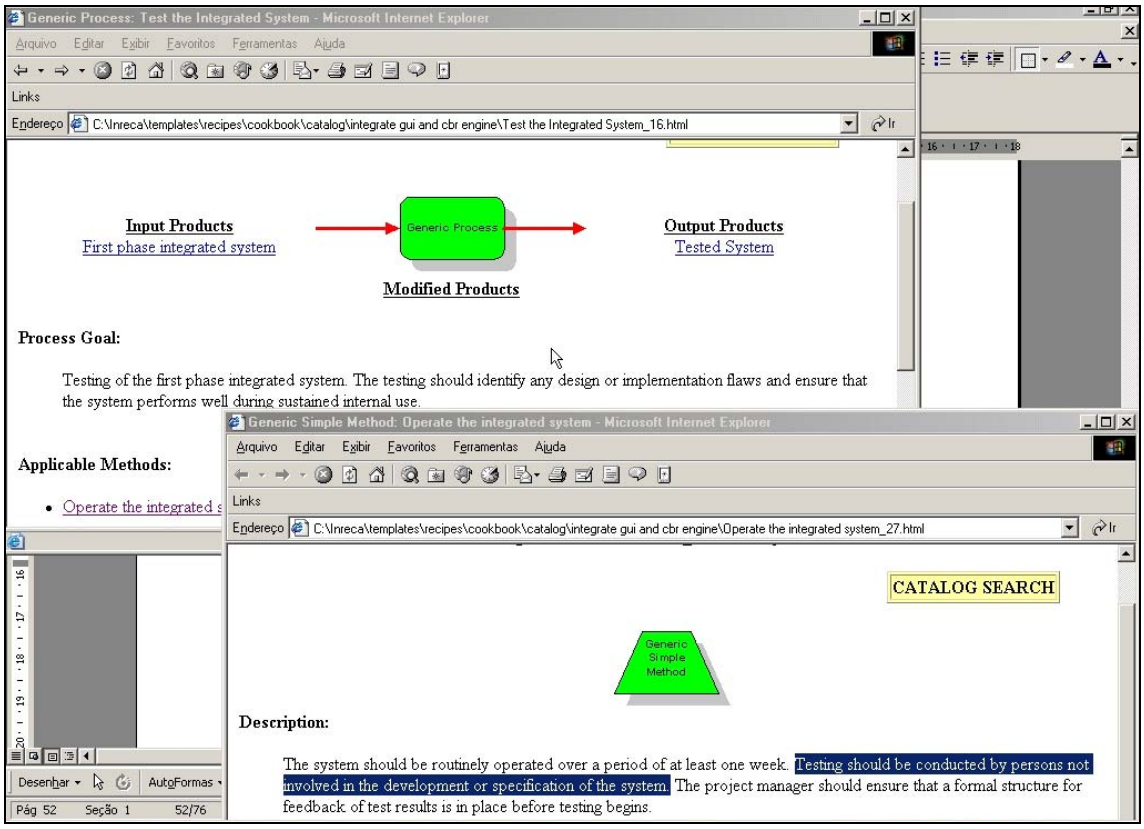

FIGURA 6.5 - Telas capturadas da documentação original INRECA (processo *Catalog Search*)

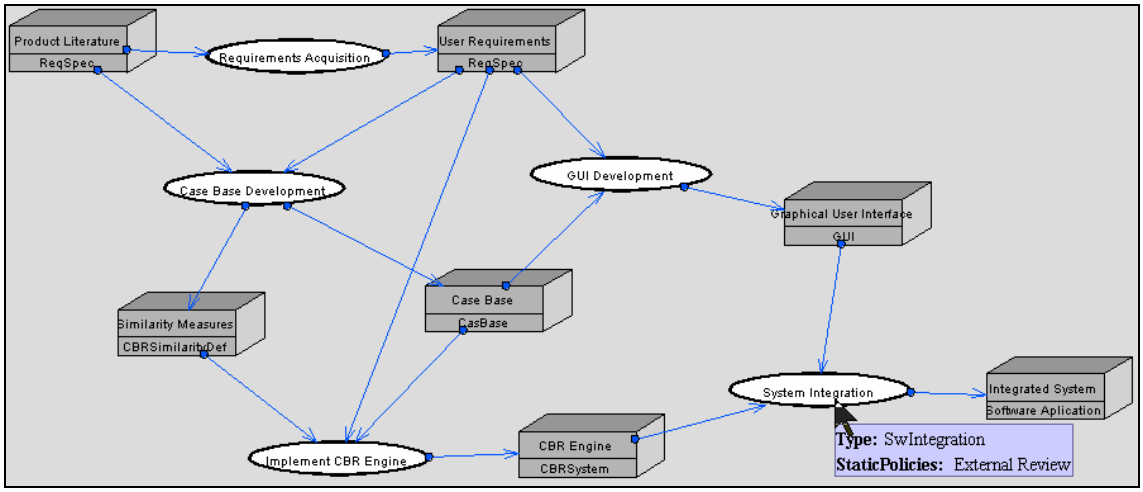

FIGURA 6.6 - *Template INRECA.Catalog Search*

É interessante notar que os *templates* da metologia INRECA compartilham tipos comuns entre si para as atividades, artefatos, cargos e recursos requeridos. Assim, a figura 6.7 apresenta a hierarquia de tipos para Artefatos de Software construída a partir das informações fornecidas em [BER99].

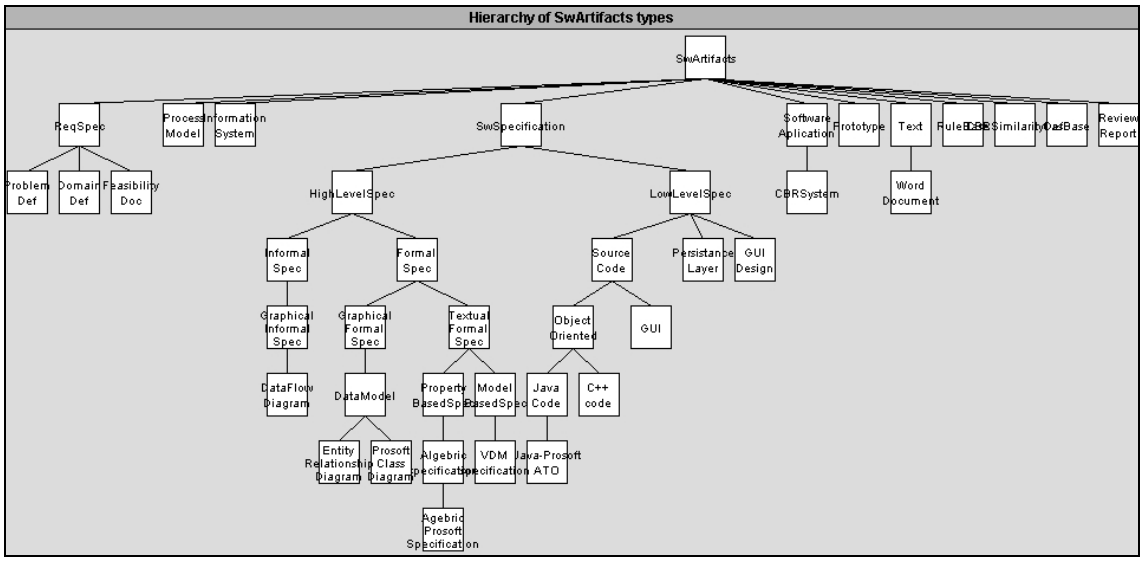

FIGURA 6.7 - Hierarquia de Tipos de Artefatos para os *templates* INRECA

6.3 O *template* para o *Rational Unified Process*

O sucesso da linguagem UML no desenvolvimento de software Orientado a Objetos motivou o surgimento de metologias que orientem a utilização das notações fornecidas. Assim, o *Rational Unified Process* (RUP) [KRU2000] é um modelo de processo de propósito geral proposto para apoiar UML.

A experiência na conversão de RUP para o formato de um *template* foi similar ao caso da metodologia INRECA (na seção 6.2): como a documentação original do processo é apresentada na forma narrativa e com diagramas da própria UML, algumas situações ambigüas foram detectadas. Portanto, a figura 6.8 apresenta um fragmento do *template* RUP voltado à descrição do processo de definição de requisitos de software (fragmento *Requirements Workflow*).

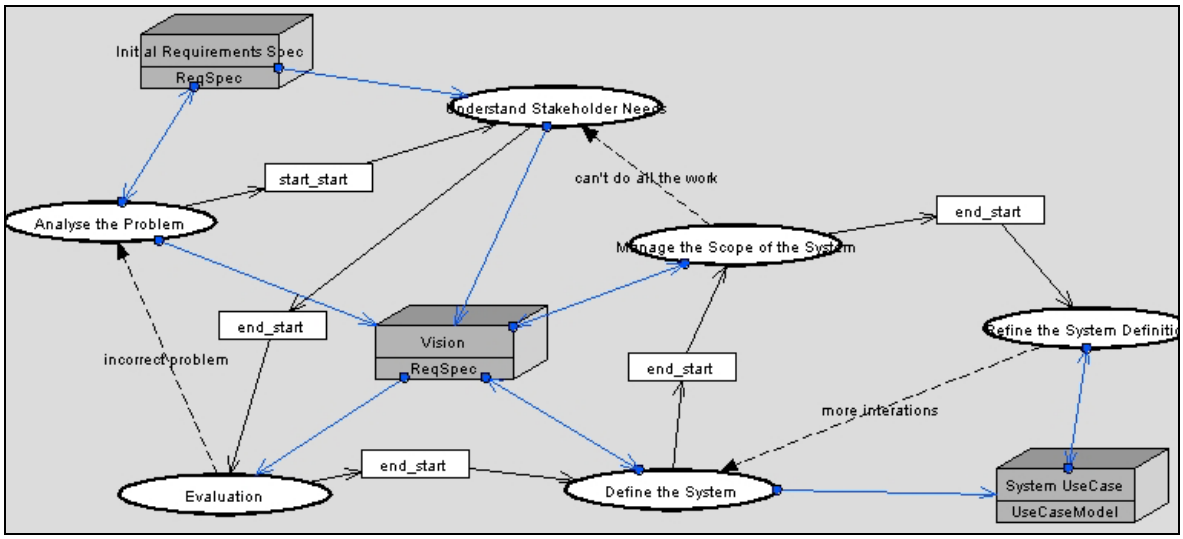

FIGURA 6.8 - O fragmento *RUP Requirements Workflow* expresso como um *template* APSEE

6.4 O *template* para Sistemas Móveis

Esta seção descreve um *template* desenvolvido como resultado de atividades de pesquisa realizadas cooperativamente entre os grupos PROSOFT e o GPPD (Grupo de Processamento Paralelo e Distribuído) do PPGC-UFRGS, envolvendo o autor desse trabalho e os Srs. Adenauer Yamin, Carla Alessandra Lima Reis e Iara Augustin (doutorandos do PPGC-UFRGS), sob a orientação dos Professores Cláudio Resin Geyer (GPPD) e Daltro Nunes (PROSOFT). Em linhas gerais, a pesquisa realizada tem o objetivo de fornecer paradigmas e práticas de Engenharia de Software para disciplinar a produção de software que possua características de exploração de paralelismo e/ou distribuição de seus componentes.

O trabalho descrito por esta seção foi motivado pelo desenvolvimento recente no GPPD de uma infraestrutura que apóia o desenvolvimento de Sistemas Móveis - denominado ISAM - que permite a mobilidade de agentes para a resolução de problemas que demandam mobilidade física e lógica de software [YAM202001]. Assim, no período de julho de 2001 a janeiro de 2002, a pesquisa foi conduzida com os seguintes objetivos principais:

- Investigar as características específicas do desenvolvimento de sistemas móveis;
- Avaliar e investigar os elementos específicos da plataforma de software desenvolvida, a fim de explicitar atividades não-usuais necessárias e que influenciam o desenvolvimento de software de tal natureza;
- Averiguar a viabilidade de se utilizar os recursos disponíveis pelo sistema APSEE e suas linguagens na descrição formal de um processo que atenda os requisitos levantados.

A computação móvel promete modificar a maneira pela qual as pessoas interajem e, por conseqüência, o processo para guiar o desenvolvimento de software de código móvel lida com aspectos de um domínio de programação específico que implica, por exemplo, na noção de uma arquitetura de software dividida em Cliente, Agente e Servidor. Esse novo paradigma computacional emergiu a partir da evolução das tecnologias de sistemas distribuídos e redes sem fio, permitindo a existência de diferentes cenários (variando desde a computação nômade até a pervasiva). A mobilidade física e lógica requer novos tipos de aplicações que possuam intrinsecamente recursos para mobilidade, flexibilidade e adaptabilidade. Segundo Satyanarayanan [SAT202001], esse ambiente dinâmico introduz assim novos desafios na pesquisa de sistemas computacionais.

O ponto de partida para o trabalho foi a metodologia proposta por Booch em [BOO94] para disciplinar o desenvolvimento de sistemas Cliente/Servidor. A partir desse processo - modelado como *template* - foram realizadas diversas simulações envolvendo diferentes cenários para aplicações móveis, em busca de aspectos do desenvolvimento de sistemas móveis que não fossem contemplados por metodologias tradicionais. O resultado obtido foi um *template* que detalha o processo de projeto de sistemas móveis, levando em consideração aspectos de adaptação dinâmica [REI2002a]. O *template* proposto foi projetado para apoiar o desenvolvedor nas decisões típicas de projeto de sistemas móveis, envolvendo artefatos descritos com a notação UML padrão.

A figura 6.9 apresenta de forma compacta a hierarquia que descreve a composição de atividades do template *Mobile System Development*. A figura 6.10, por

sua vez, descreve em detalhe o ordenamento das atividades e os artefatos de software envolvidos. No processo de projeto do sistema móvel (fragmento *Mobile System Design*), são previstas três atividades principais: Projeto da Arquitetura do Sistema (*Architectural Design*), Planejamento da Implementação (*Implementation Planning*) e Planejamento da Implantação (*Implantation Planning*).

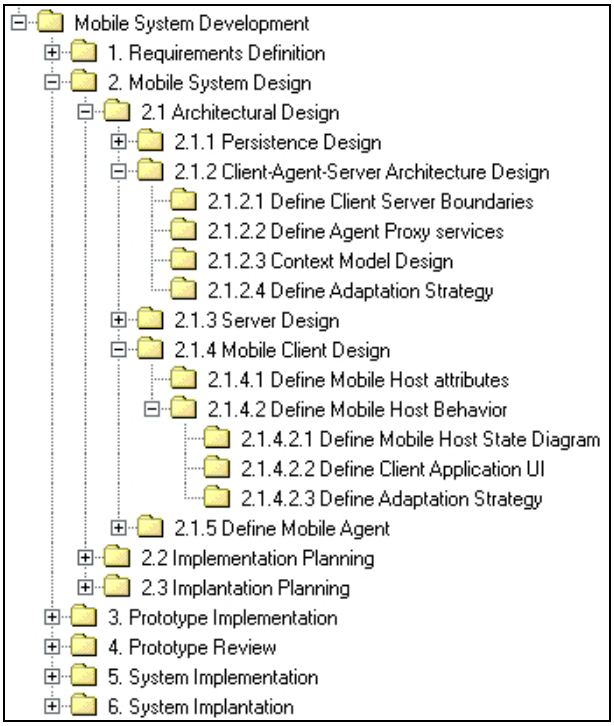

FIGURA 6.9 - Hierarquia de atividades para o *template Mobile System Development*

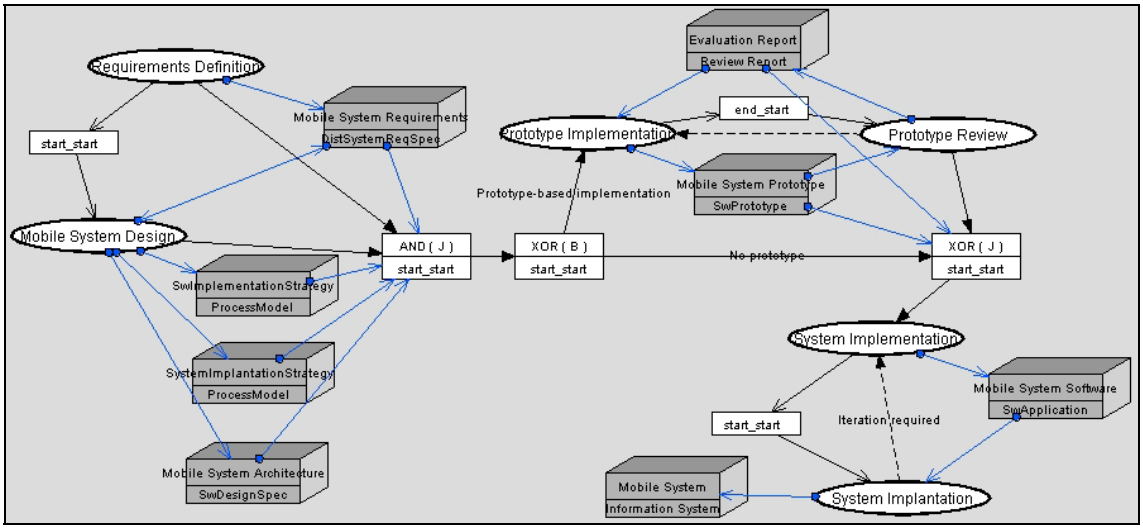

FIGURA 6.10 - O processo de alto nível para o *template Mobile System Development*

A figura 6.11 fornece o detalhamento para o fragmento *Architectural Design* (acima) e a decomposição *Client-Agent-Server Architecture Design*. Para ilustrar o detalhamento do *template*, a figura 6.12 fornece os *scripts* correspondentes a cada uma das atividades-folha que compõem o fragmento *Client-Agent-Server Architecture Design*.

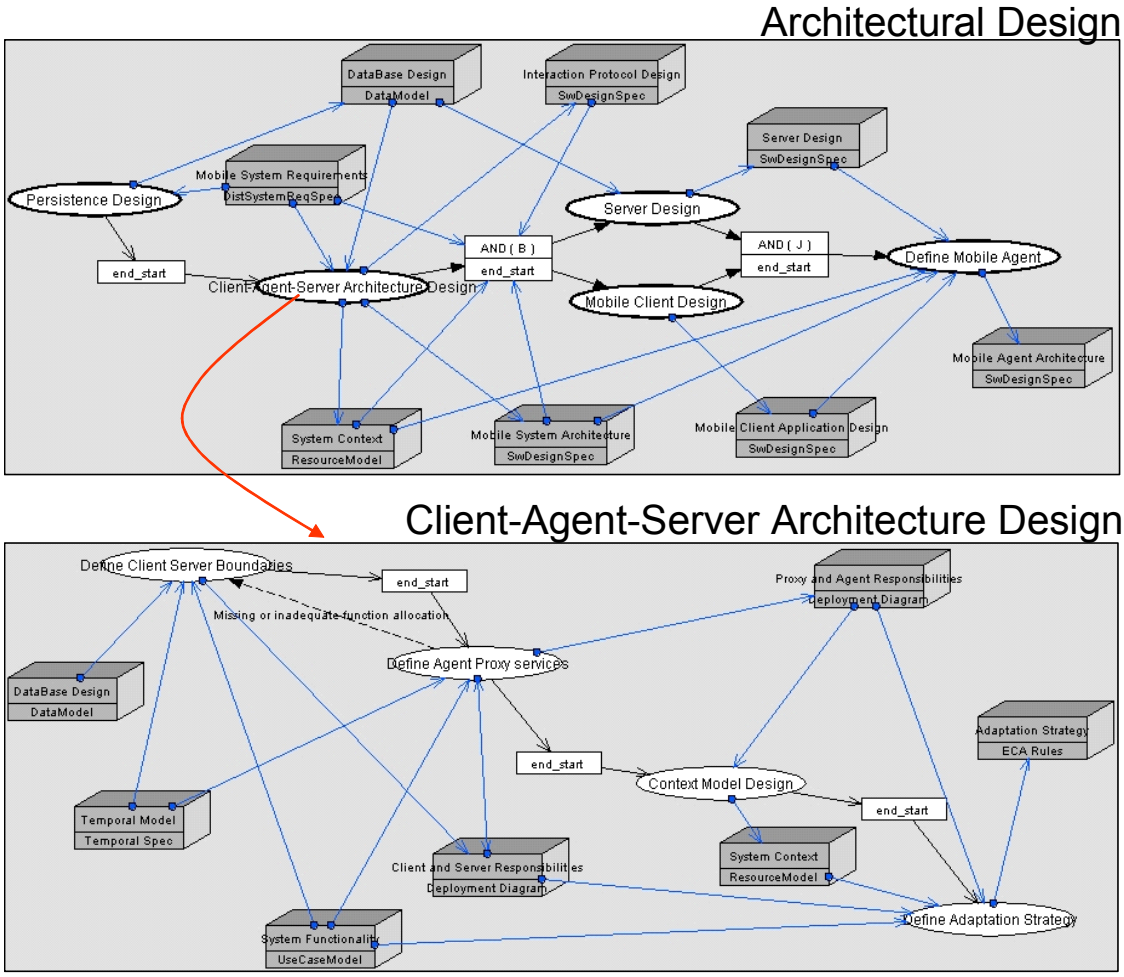

FIGURA 6.11 - Fragmentos *Architectural Design* e *Client-Agent-Server Architecture Design* do *template Mobile System Development*

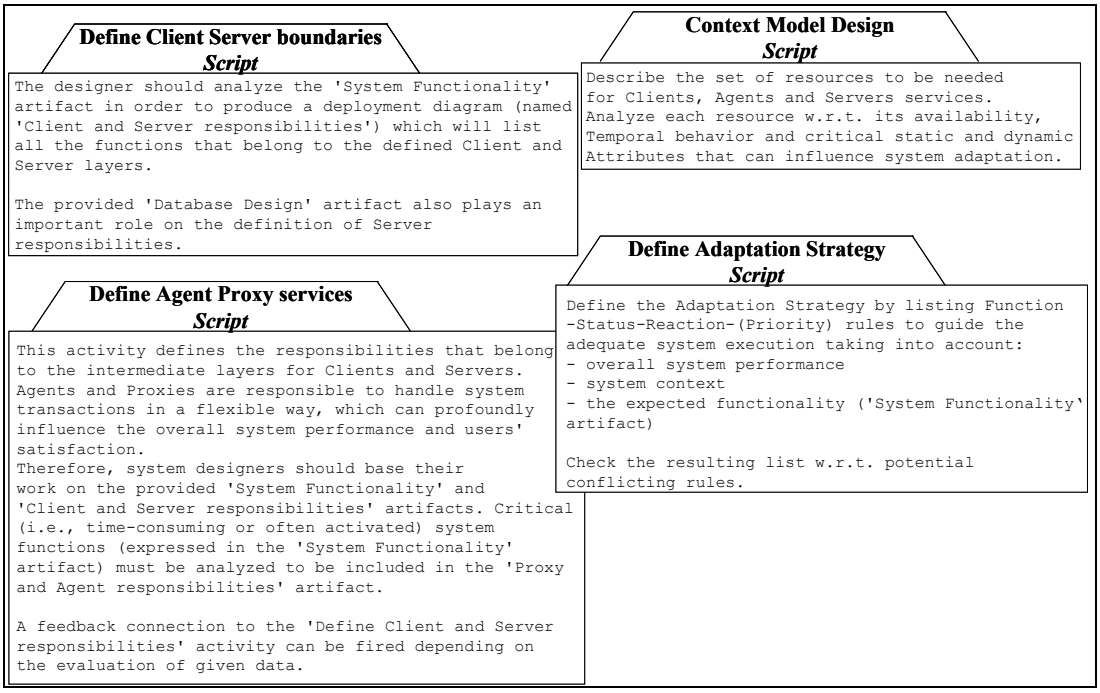

FIGURA 6.12 - Os scripts fornecidos para as atividades componentes do fragmento *Client-Agent-Server Architecture Design*

Finalmente, vale ressaltar que o trabalho com esse *template* prossegue à medida que novas ferramentas vem sendo desenvolvidas pelo GPPD para automatizar as atividades de projeto, a programação e os testes de sistemas móveis. Assim, um exemplo da aplicação do *template* proposto no desenvolvimento de um sistema de apoio a reuniões virtuais - que se vale dos recursos de mobilidade de agentes e leva em consideração a heterogeneidade de dispositivos móveis envolvidos - é apresentado em [REI2002a].

6.5 Combinando os templates INRECA e PROSOFT-Java no desenvolvimento do APSEE-*SearchEngine*

Esta seção descreve um exemplo prático da reutilização de dois dos *templates* apresentados nas seções anteriores. O desenvolvimento do *APSEE-SearchEngine* envolveu o projeto e implementação de um mecanismo que utiliza a tecnologia *CBR* para facilitar a recuperação de *templates*, permitindo que os resultados sejam apresentados aos usuários de forma ordenada segundo o grau de similaridade.

O processo para o desenvolvimento do *SearchEngine* foi gerado a partir de dois *templates*, relacionados abaixo:

- O *template INRECA.Catalog Search* foi usado para descrever a estrutura de mais alto nível do processo;
- O *template PROSOFT-Java* foi adaptado para guiar o processo de implementação da ferramenta no PROSOFT.

A figura 6.13 apresenta dois dos fragmentos do processo, destacando os elementos que foram adaptados em relação ao *template* original. Na parte superior da figura está o processo de mais alto nível: os círculos introduzidos no diagrama representam conexões temporais que foram adicionadas ao processo adaptado, restringindo a ordem da realização das atividades.

A parte inferior da figura 6.13 apresenta o processo adaptado a partir do *template* PROSOFT-Java. Nesse caso, os círculos introduzidos na figura evidenciam as conexões de artefatos que foram renomeadas (tais como *UserRequirements*, *CBREngine* e *CBR Engine Sources*) e incluídas no modelo (artefatos *CaseBase* e *SimilarityMeasures*).

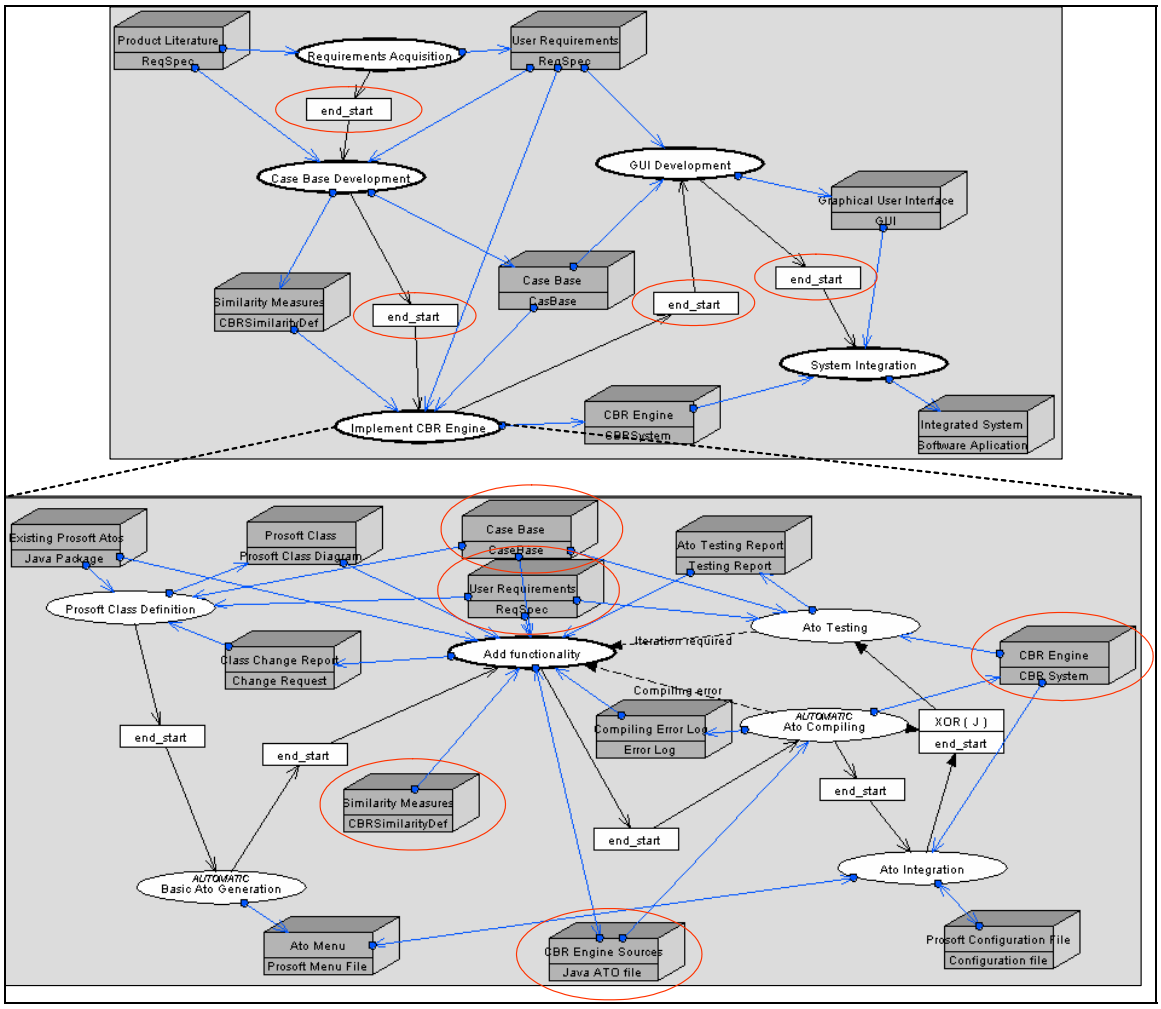

FIGURA 6.13 - O processo de desenvolvimento do APSEE-*SearchEngine*

A figura 6.14 mostra uma tela que contém um dos formulários existentes na ferramenta desenvolvida como resultado da execução do processo descrito. O formulário descrito apresenta campos que devem ser preenchidos pelo usuário (*Process Template Type* e *Activity Type*) para descrever um critério que restrinja o espaço de busca. Atributos diferentes podem ser selecionados pelo usuário, sendo que tais atributos são agrupados em função das dimensões de modelagem de *templates* propostas por este trabalho.

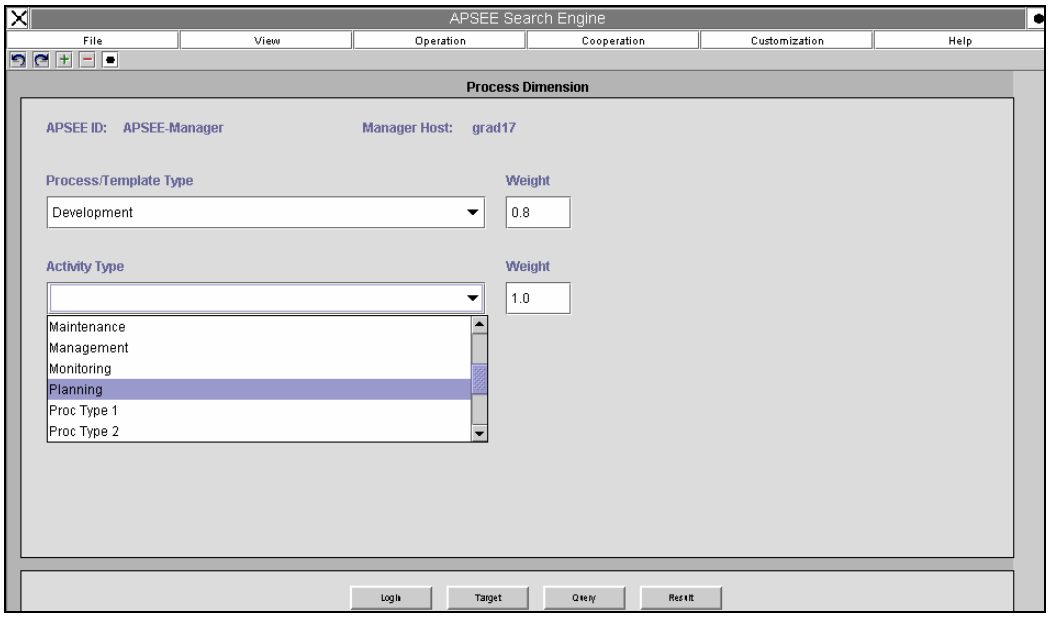

FIGURA 6.14 - Tela do mecanismo de consultas de processos (APSEE-*SearchEngine*)

7 Análise da Pesquisa e Conclusões

O trabalho aqui apresentado trata de um tópico interessante e complexo, estando relacionado com diferentes aspectos tecnológicos e organizacionais que influenciam decisivamente a forma como o software é desenvolvido e mantido. Dessa forma, apesar dos objetivos traçados originalmente restringirem o escopo da Tese em fornecer apoio automatizado à reutilização de modelos de processos de software, o caráter inerentemente multi-disciplinar do contexto estudado levou à investigação de uma grande variedade de tópicos.

A pesquisa descrita nesse texto resultou no desenvolvimento de um meta-modelo integrado que apóia a modelagem de processos reutilizáveis. Assim, embora o trabalho aqui apresentado não forneça uma solução completa para o problema investigado - e nem tampouco é esperado o surgimento de tal solução no futuro próximo em virtude da complexidade do tema - acredita-se que o meta-modelo proposto constitui uma contribuição importante para o estado da arte tecnológico atual em virtude dos formalismos fornecidos e da experiência na especificação, implementação e experimentação prática da proposta.

As seções a seguir aprofundam a discussão dos resultados da pesquisa segundo critérios específicos.

7.1 Resumo das contribuições

O trabalho de pesquisa desenvolvido produziu um conjunto de resultados estabelecidos na Proposta de Tese[44]. A apresentação dos tópicos investigados foi organizada seguindo sugestões da banca de avaliação, enfatizando os aspectos centrais do meta-modelo proposto para definição de processos reutilizáveis.

Em virtude do grande volume de informações produzidas direta e indiretamente por esse trabalho, decidiu-se segmentar a apresentação na forma de relatórios de pesquisa que descrevem os detalhes específicos das especificações, implementações, contatos com pesquisadores do tema e os relatos de experiências práticas. Tais documentos são referenciados quando necessário no texto, estando listados nas referências bibliográficas.

Resumidamente, as principais contribuições introduzidas pela presente Tese são listadas a seguir:

- O texto propõe um conjunto de requisitos para o apoio automatizado à reutilização de modelos de processos de software. Acredita-se que tais

[44] A Proposta de Tese é uma etapa obrigatória no curso de Doutorado em Ciência da Computação do PPGC-UFRGS que, no caso do presente trabalho, foi apresentada no dia 27 de agosto de 2001, sendo avaliada pelos Professores Doutores Ana Maria Alencar Price, Maria Lúcia Blanck Lisbôa, Itana Gimenes (Depto. de Informática – Universidade Estadual de Maringá), além do orientador do trabalho, Prof. Dr. Daltro José Nunes.

requisitos fornecem uma base importante para avaliação do trabalho aqui apresentado, permitindo a comparação com abordagens similares, levando em consideração os recentes avanços da Tecnologia de Processo de Software, Sistemas de *Workflow* e a Reutilização de Software. Uma versão preliminar destes requisitos foi publicada na forma de um artigo de trabalho em andamento no *7th International Workshop on Groupware* [REI2001a];

- O presente trabalho avança no estado-da-arte por fornecer uma solução que está integrada a um ambiente de execução de processos. Assim, a reutilização é tratada como uma atividade de primeira ordem do meta-processo de software, permitindo que modelos abstratos sejam recuperados e utilizados na composição de processos em execução. Isto permitirá ainda, em um futuro próximo, a investigação acerca de mecanismos que automatizem a adaptação de *templates* para processos executáveis, tal como discutido na seção 7.4 a seguir.
- O meta-modelo proposto por esse trabalho fornece uma série de construtores sintáticos que permitem que os diferentes aspectos envolvidos na modelagem de processos de software sejam descritos segundo múltiplas perspectivas, complementares entre si. Os diferentes componentes envolvidos na definição do meta-modelo foram especificados algebricamente, constituindo uma base semântica de alto nível de abstração, que deu origem a um conjunto de protótipos implementados no ambiente PROSOFT-Java. Estes assuntos foram tratados em diferentes níveis de detalhe em artigos publicados em conferências internacionais, a saber: *V Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS)* [REI2002c] e o *26th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2002)* [REI2002e];
- Um conjunto de regras foi proposto para restringir a adaptação dos processos produzidos a partir da reutilização de *templates*. Tais restrições foram especificadas algebricamente através de regras em Gramáticas de Grafos, as quais foram implementadas no editor de processos fornecido no ambiente *APSEE*;
- Uma linguagem para definição de Políticas Estáticas foi apresentada como um formalismo que completa a linguagem de modelagem proposta, descrevendo propriedades sintáticas que são ortogonais aos modelos de processos. Esse assunto deu origem a um relatório de pesquisa do PPGC-UFRGS (RP-311) [REI2001c], um artigo no XV Simpósio Brasileiro de Engenharia de Software [REI2001d], e a um artigo no periódico *Annals of Software Engineering*, na edição especial sobre *Process-based Software Engineering* [REI2002b];
- A tese produziu como resultado derivado um mecanismo de busca - denominado *APSEE-SearchEngine* - baseado em raciocínio baseado em casos. Tal mecanismo resultou da aplicação prática dos *templates INRECA.Catalog Search* e PROSOFT para gerenciar o desenvolvimento, envolvendo os bolsistas de Iniciação Científica do grupo para a sua implementação, sendo descrito em detalhe em [REI2001f].

A experimentação prática com o meta-modelo *APSEE*-Reuse resultou no desenvolvimento de *templates* que definem metodologias para o desenvolvimento de software que se mostraram de extrema utilidade no desenvolvimento de software para necessidades específicas em alguns projetos de pesquisa do PPGC-UFRGS, como descrito a seguir:

- Um *template* para documentar o desenvolvimento de **software móvel** foi desenvolvido em cooperação com o Grupo de Processamento Paralelo e Distribuído da PPGC-UFRGS (conforme apresentado na seção 6.4). A experiência na descrição desse *template* foi assunto de um artigo publicado no *6th World Conference on Integrated Design & Process Technology* [REI2002a];
- O *template* **INRECA** foi construído com os construtores sintáticos propostos a partir da metodologia homônima para guiar o desenvolvimento de aplicações de raciocínio baseado em casos [BER99]. A experiência na descrição desse modelo resultou em um relatório de pesquisa (RP-322) [REI2001e]. O *template* INRECA foi ainda utilizado para o desenvolvimento de um mecanismo de recuperação de informações de processos de software do ambiente *APSEE*, conforme descrito na seção 6.5;
- Um *template* - denominado **PROSOFT-Java** - foi definido com o objetivo de documentar o processo de desenvolvimento de software no ambiente PROSOFT-Java. Esse *template* foi construído a partir da documentação que descreve a reengenharia do PROSOFT para Java [SCH97], do *Java-PROSOFT Guide* [SCH2002], e da experiência transmitida informalmente pelos auxiliares de pesquisa Srs. Heribert Schlebbe (*Institut für Informatik - Universität Stuttgart*) e Marcelo Abreu (Engenharia da Computação - UFRGS). Esse *template* foi incorporado à documentação oficial do ambiente, o *Java-PROSOFT Guide* [SCH2002], usado no treinamento de novos desenvolvedores de ferramentas para o ambiente.

7.2 Análise dos Resultados

Esta seção faz uma análise do meta-modelo proposto, enfatizando os aspectos relacionados com o seu desenvolvimento e a experiência na modelagem dos casos apresentados no capítulo 6.

7.2.1 Formalismos utilizados na especificação

O modelo foi aqui apresentado através da combinação de diferentes formalismos. No capítulo 3, foram utilizadas as notações para pacotes e classes fornecidas pela UML para descrever de uma forma geral os tipos de dados definidos, tendo como objetivo auxiliar no entendimento da estrutura principal do meta-modelo, usando uma notação que é universalmente aceita. O capítulo 4, por outro lado, descreveu os detalhes de projeto do modelo proposto usando o PROSOFT-Algébrico para descrever os tipos de dados e suas funções principais, enquanto que Gramáticas de Grafos foram usadas para descrever regras de transformação usadas na especificação do editor de *templates* proposto. Essa abordagem para apresentação do trabalho foi adotada a partir da experiência nas avaliações dos artigos submetidos: constatou-se que a notação UML poderia ser utilizada em textos que explorassem a descrição da arquitetura do modelo em alto nível, quando não houvesse preocupação com detalhes

específicos das especificações algébricas ou acerca da implementação dos protótipos desenvolvidos[45].

Do ponto de vista das especificações algébricas, constatou-se que o uso de múltiplas notações especializadas - a saber, a notação de classes PROSOFT, os axiomas em PROSOFT-Algébrico e as regras em Gramáticas de Grafos - auxiliaram no desenvolvimento e entendimento da proposta, visto que aspectos diferentes do modelo são descritos com notações apropriadas.

A escolha dos formalismos citados mostrou-se adequada, pois permitiu o particionamento de detalhes dos diferentes componentes do modelo. Algumas das regras de transformação foram submetidas ao simulador AGG [AGG2002], o que permitiu avaliar características acerca do comportamento do sistema logo no início do processo. Além disso, a implementação de software correspondente se demonstrou viável, sendo que tanto as especificações em PROSOFT-Algébrico quanto as regras de transformação foram usadas para derivar os protótipos funcionais descritos nesse texto.

7.2.2 Escalabilidade

Uma das principais vantagens proclamadas pelos modernos sistemas PSEEs está relacionada à idéia de fornecer assistência automatizada para gerenciar grandes organizações de desenvolvimento de software. Embora o uso da Tecnologia de Processos de Software em organizações de pequeno e médio porte também tenha o potencial de aumentar a qualidade dos processos adotados, a probabilidade de se obter benefícios econômicos significativos aumenta quando o alvo é constituído por grandes organizações, visto que estas envolvem um grande número de recursos humanos e de suporte que cooperam e competem entre si em múltiplos processos simultâneos de desenvolvimento e manutenção de software. A escalabilidade constitui, portanto, um elemento crítico na avaliação das soluções propostas nesse contexto.

O conjunto de dimensões definido para a modelagem de *templates* foi influenciado pela tendência recente no campo de linguagens de programação, e é orientado no sentido de fornecer apoio à separação explícita de detalhes. Assim, o projeto de *templates* pelos usuários finais é simplificado, pois os detalhes são separados em dimensões bem definidas, permitindo a composição de um modelo a partir de componentes independentemente definidos.

Um aspecto chave na proposta aqui apresentada consiste na associação dos componentes essenciais de processos com árvores hierárquicas de tipos. Cada nodo em uma hierarquia de tipos possui o potencial de ser reutilizado em diferentes processos e *templates*, assim como suas especializações. Algumas hierarquias de tipos estabelecem ainda semântica especial para nodos específicos (como a hierarquia de tipos de recursos proposta por Lima Reis e exemplificada na figura 3.17). Entretanto, conforme observado por Malone et al [MAL99], Botha [BOT202001] e confirmado pela nossa experiência prática, tais hierarquias de tipos podem ser de manutenção complexa para organizações e/ou processos grandes: a profundidade e a largura das árvores de tipos

[45] Os diagramas de classes em UML também foram utilizados nos artigos escritos a partir desse trabalho que apresentavam a visão geral do modelo, com o objetivo de facilitar o entendimento por parte de leitores não familiarizados com as notações algébricas aqui utilizadas (em específico, [REI2001a], [REI2002c], [REI2002d] e [REI2002e]). Por outro lado, classes PROSOFT e especificações algébricas foram adotadas em artigos que tratavam de detalhes específicos, como por exemplo a especificação do algoritmo de verificação de Políticas Estáticas (em [REI2001d] e [REI2002b]).

tendem a aumentar à medida que o tamanho e quantidade de processos modelados aumentem. A área de gerência de hierarquias de especialização de tipos é um campo de pesquisa em aberto que atualmente conta com eventos específicos para a sua discussão[46]. Assim, a experiência em outros trabalhos deve ser investigada no sentido de auxiliar a composição de processos e seus elementos.

7.2.3 Adaptabilidade

A adaptabilidade é um conceito abstrato que, para o modelo proposto, pode ser avaliado segundo várias perspectivas. Essa seção discorre essencialmente acerca da adaptabilidade dos construtores do meta-modelo proposto para diferentes PSEEs ou PMLs. À primeira vista, isto é possível uma vez que o meta-modelo de suporte possui semântica formal, a descrição é precisa e em alto nível de abstração, o que possivelmente pode guiar diferentes implementações. Em geral, acredita-se ser possível adaptar o meta-modelo proposto para ambientes e linguagens que adotem um paradigma orientado a atividades. Todavia, investigação adicional sobre esse tópico ainda é necessária.

7.2.4 Facilidade de entendimento

Um modelo de processo contém informações de uma grande variedade de fontes, lidando com múltiplos contextos organizacionais e tecnológicos. Assim, um desafio importante no projeto de uma PML é facilitar o entendimento de modelos de processos, tendo o potencial de incrementar a sua eficiência [FIN94] e a posterior reutilização [PER96] [ISO92] [FRA99]. Como destacado por Jørgensen [JØR2001], o entendimento é um aspecto crítico que influencia decisivamente a reutilização de processos, podendo aumentar a qualidade e a eficiência dos modelos gerados por adaptação.

A experiência geral da comunidade de Engenharia de Software demonstra que notações gráficas tendem a constituir alternativas que facilitam o entendimento de modelos complexos de software. Conforme justificado na seção 2.4.5, notações gráficas também são requeridas no projeto de PMLs modernas. Essa seção, portanto, discute alguns dos aspectos principais relacionados à representação gráfica dos modelos construídos, levando em consideração a implementação atual do meta-modelo.

O modelo *APSEE*-Reuse aqui proposto é baseado na hipótese que, para promover a sua reusabilidade, um modelo de processo deva ser composto por componentes independentemente definidos. Assim, o protótipo atual do sistema fornece notações especializadas para auxiliar o projetista de processos no entendimento dos diferentes aspectos que constituem um modelo completo. Como mostrado pelos exemplos apresentados no capítulo 6, o usuário lida com uma grande variedade de notações gráficas alternativas. Não existe uma notação que seja universalmente aceita em todos os casos. Além disso, embora o uso de representações gráficas tenha o potencial de simplificar o entendimento de tais modelos, tais notações requerem grandes áreas no *display* gráfico do usuário.

---

[46] É de conhecimento do autor o Workshop MASPEGHI - *MAnagement of SPEcialization/Generalization HIerarchies*, a ser realizado em setembro de 2002 em Montpellier (França), como parte integrante do *8th International Conference on Object-Oriented Information Systems* [MAS2002].

A experiência prática na modelagem e reutilização do *template* para Sistemas Móveis (previamente descrito na seção 6.4) demonstrou que as notações propostas foram bem aceitas mesmo por uma comunidade de usuários não habituada ao contexto de modelagem de processos. O uso do construtor de Políticas Estáticas desmonstrou ser um desafio para novatos, visto que treinamento ou experiência numa linguagem específica são requisitos para construir novas instâncias. Por outro lado, a experiência prática demonstrou que os usuários foram aptos a reutilizar (habilitar) as Políticas existentes (pré-instaladas no sistema) de forma bem sucedida em diversos fragmentos do *template* proposto.

Enfim, vale ressaltar que a apresentação gráfica de modelos de processos de software é um tópico de pesquisa próprio, e um projeto específico[47] no contexto do PROSOFT-APSEE está em andamento para fornecer linguagens e ferramentas para extração de visões especializadas de processos executáveis e de *templates*.

7.3 Trabalhos relacionados

No momento da conclusão desse trabalho, a literatura especializada apresenta poucas soluções concretas no que tange especificamente a reutilização de processos. De fato, os trabalhos encontrados na literatura atual são investigações recentes, e a grande maioria são trabalhos em andamento. Assim, essa seção faz um levantamento geral de como o trabalho está relacionado com soluções importantes encontradas no contexto de modelagem de processos de software para, em seguida, comparar a abordagem aqui proposta com soluções recentes da área de *workflow* e demais soluções especificamente propostas para apoiar a reutilização de processos de software.

7.3.1 A reutilização de processo tratada do ponto de vista das PMLs

A necessidade de se apoiar múltiplas perspectivas na modelagem de processos vem sendo investigada pela literatura desde meados da década de 1990. Essa seção compara, de forma geral, o trabalho aqui exposto com alguns exemplos significativos existentes na literatura. Vale ressaltar, entretanto, que comparações detalhadas acerca de aspectos específicos de componentes do meta-modelo proposto são apresentadas nas referências do autor.

O ambiente EPOS [NGU97] foi um dos primeiros ambientes a fornecer notações gráficas específicas para a descrição de artefatos e processos. EPOS utiliza uma linguagem de programação de processos denominada SPELL. SPELL é uma linguagem concorrente e reflexiva que foi baseada em Prolog, fornecendo recursos do paradigma Orientado a Objetos, tendo sido desenvolvida como uma extensão das linguagens de manipulação e definição de dados (DDL/DML) disponíveis para o banco de dados utilizado pelo ambiente EPOSDB. Entretanto, conforme Sutton [SUT97], a linguagem EPOS/SPELL é considerada de primeira geração, sendo tipicamente considerada uma linguagem de programação de processos (de baixo nível). A programação de processos é feita com bastante detalhe, não havendo suporte para descrição de processos abstratos. Já o modelo APSEE-Reuse aqui proposto é mais rico, fornecendo maior flexibilidade na modelagem de aspectos relacionados com a dimensão

[47] Projeto "APSEE-MONITOR de Visualização e Representação de Modelos de Processos de Software", apoiado pelo Programa de Apoio ao Desenvolvimento Científico-Tecnológico-Artístico-Cultural do Estado do Rio Grande do Sul (PROADE) sob o número de processo 0115453 de 2002.

Recursos e Pessoal. Além disso, distingue-se por definir Políticas como componentes de primeira ordem na modelagem de processos.

As linguagens de modelagem surgidas na segunda metade da década de 1990 começaram a se preocupar com a reutilização de modelos de uma forma um pouco mais consistente. Assim, por exemplo, as linguages E^3 [JAC98], OPSIS [AVR96a] [AVR96b] e PYNODE [ENG97] fornecem capacidades de reutilização limitadas estando essencialmente baseadas em construtores sintáticos que fornecem o conceito de extração e composição estática (no caso de E^3) e dinâmica (OPSIS e PYNODE) de visões de processos, assim como estão baseadas na noção de generalização/especialização de tipos (E^3). A extração e composição dinâmica de componentes de processos, em função de "características bem definidas", constitui um importante recurso que não é tratado pelo modelo aqui proposto.

7.3.2 Soluções da área de *Workflow*

Workflow é uma disciplina que evoluiu de forma significativa na última década principalmente devido a disponibilização de produtos comerciais por várias empresas. Embora existam divergências na literatura especializada acerca da fronteira entre PSEEs e sistemas de gerência de *workflow*, estes são caracterizados por fornecer apoio à gerência de processos administrativos em geral. Apesar da tendência atual em concentrar esforços na uniformização dos conceitos e experiências nas áreas de gerência de processo de software e *workflow*, constata-se que as soluções tomaram rumos distintos.

Embora as soluções no contexto de workflow avançarem em áreas importantes, tais como na uniformidade das soluções apresentadas e na modelagem da estrutura organizacional de apoio, pouca ênfase é dada à modelagem de processos (quando comparado com suporte fornecido pelos PSEEs). A maioria das linguagens de programação/modelagem de processos, adotados por sistemas *workflow*, são monoparadigma, isto é, representam os diferentes aspectos dos processos utilizando somente um paradigma de linguagem (por exemplo, orientação a objetos, procedimental, baseado em regras) o que restringue, sobremaneira, a aplicabilidade de tais ambientes no projeto de processos de software [SOT2001].

Do ponto de vista de reutilização de processos, os trabalhos científicos e produtos comerciais, no contexto de *workflow*, fornecem soluções que são limitadas à estratégia de "copiar e colar" [RIB2002]. Destaca-se o modelo *Action Port Model* (APM) [KRU96], que fornece uma notação para modelagem de *templates* e um construtor de substituição, permitindo que as atividades decompostas possam ser substituídas por outros *templates*. O modelo APM, entretanto, possui uma linguagem de modelagem bastante limitada e impõe uma abordagem *top-down* para a composição de processos.

7.3.3 Soluções específicas para a reutilização de processos de software

Segundo Ribó (em [RIB2002]), as soluções atuais para automatizar a reutilização de processos de software podem ser classificadas em dois tipos principais: os *frameworks* de reutilização (que fornecem uma infraestrutura de software que trata a reutilização como uma das atividades de um processo maior, isto é, o meta-processo de software), e as PMLs que fornecem construtores sintáticos para a modelagem de processos reutilizáveis. O modelo *APSEE*-Reuse aqui proposto consiste, portanto, de um

framework de reutilização que também fornece uma linguagem de modelagem que está intimamente ligada a uma linguagem executável, o que facilita a adaptação dos *templates* recuperados para a reutilização.

Os *frameworks* de reutilização de processos representam um importante passo no rumo do desenvolvimento de uma infraestrutura integrada para automação de processos de software. Grande parte das abordagens existentes - incluindo os trabalhos de Vasconcelos [VAS97] e Gnatz [GNA2001] - está baseada em extensões do conceito de Padrões de Projeto (*Desing Patterns* [GAM94]) para a descrição de processos abstratos. Portanto, tais meta-modelos descrevem os diferentes atributos de um padrão (processo abstrato) e seus componentes, privilegiando a descrição dos seus interrelacionamentos e aplicabilidade. Esses meta-modelos, invariavelmente, adotam notações textuais baseadas na sintaxe da linguagem para definição de padrões proposta por Gamma et al [GAM94]. A abordagem APSEE-Reuse aqui proposta diferencia-se por fornecer uma linguagem de modelagem muito mais rica, que ainda assim permite que os processos sejam descritos de forma abstrata, mas com um nível de detalhe que evidencia o comprometimento com a executabilidade dos modelos. Outra diferença a ser destacada é a separação explícita de detalhes de modelagem de processos, o que permite que a complexidade do modelo completo seja particionada em componentes independentes e também reutilizáveis.

A linguagem PROMENADE [RIB2002] é uma PML recente, desenvolvida com o objetivo de apoiar a reutilização de processos através da composição de objetos, focalizando um nível de detalhe que não foi tratado pelo modelo aqui proposto. PROMENADE possui construtores sintáticos sofisticados, que permitem a composição, projeção, parametrização, substituição, renomeação, e inclusão estática e dinâmica de componentes. Além disso, o trabalho em PROMENADE pretende fornecer no futuro próximo uma representação gráfica baseada em UML para os componentes e suas associações. Como destacado a seguir (seção 7.5) espera-se em um futuro próximo a investigação acerca da integração das soluções pontuais fornecidas pela PML de PROMENADE à infraestrutura de modelagem, reutilização e execução de processos fornecidos pelo sistema APSEE.

7.3.4 Políticas Estáticas

Poucos trabalhos no campo de processos de software estão relacionados com a utilização de Políticas como componentes de primeira ordem para a modelagem de processos. Muitas das abordagens propostas na literatura descrevem Políticas Dinâmicas, que estabelecem estratégias de reação em função de eventos e condições pré-definidas pelos usuários.

Perry propõe uma abordagem mais próxima às Políticas Estáticas, aqui apresentadas: em [PER97] são propostos construtores sintáticos estão definidos para definir Políticas programáveis pelo usuário que, por sua vez, definem regras sintáticas e dinâmicas para a PML *Interact*. Na abordagem de Perry, as Políticas descrevem propriedades de um modelo de processo e estão embutidas na definição global de um processo. Assim, a abordagem de Políticas Estáticas proposta por esse texto diferencia-se pelos seguintes fatores:

- Nesse trabalho, as Políticas foram propostas com o objetivo exclusivo de fornecer um mecanismo adicional e independente para modelagem de

processos, encapsulando conhecimento gerencial expresso através de um formalismo compacto;

- Políticas Estáticas descrevem especificamente restrições sintáticas em um modelo de processos;
- O meta-modelo de Políticas é ortogonal à definição de Processos, *Templates* e seus componentes;
- No modelo proposto, as Políticas encapsulam estratégias gerenciais em construções independentemente definidas, estabelecendo uma interface bem definida que permite a reutilização de suas instâncias em diferentes processos e seus componentes.

7.4 Questões em aberto e perspectivas para o futuro

O trabalho aqui apresentado representa um primeiro passo no sentido de fornecer uma solução automatizada para reutilização de processos de software. Muitas questões ainda estão em aberto, definindo pontos que devem ser explorados em outros trabalhos que sigam a mesma linha. Desse modo, as seções a seguir destacam elementos de pesquisa para serem investigados no futuro próximo.

7.4.1 Avaliação da influência do modelo proposto na produtividade e custo da modelagem de processos de software

A realização desse trabalho, por si só, não garante um aumento da qualidade, produtividade ou melhoria dos custos para a modelagem de processos ou para o software resultante. Assim, vislumbra-se como uma importante atividade a ser realizada a condução de uma avaliação empírica que determine, através de estudos de casos aplicados na indústria, os benefícios reais obtidos quando a infraestrutura proposta é utilizada, comparando com situações similares quando nenhum apoio automatizado está disponível. Além disso, elementos específicos do modelo - como por exemplo os tipos de restrição para adaptação de *templates* - ainda precisam ser investigados com um maior número de casos para determinar com precisão a sua adequabilidade.

7.4.2 Experimentação com diferentes ambientes, linguagens e ferramentas

A heterogeneidade na terminologia, notações e soluções adotadas no contexto de Tecnologia de Processos de Software constitui um grande obstáculo para a adoção e amadurecimento das ferramentas propostas. Assim, uma extensão imediata para o trabalho aqui exposto corresponde à investigação acerca da viabilidade de se utilizar os contrutores propostos em sistemas heterogêneos de execução e modelagem de processos - i.e., diferentes PMLs e PSEEs - a fim de tomar vantagem da experiência atual de projetistas de processos na modelagem com paradigmas e linguagens específicas.

Em particular, o autor gostaria de avaliar a evolução futura do trabalho de Franch e Ribó [FRA99], que pretende estender a notação UML para representar modelos executáveis de processos de software, visto que a unificação em torno de uma notação amplamente aceita pela comunidade internacional parece ser uma alternativa atraente e viável para os próximos anos.

7.4.3 Investigar a adaptabilidade do meta-modelo proposto no contexto de processos de negócios

Embora o presente trabalho tenha sido decisivamente influenciado pela recente evolução da tecnologia de *workflow* e gerência de processos de negócios, os objetivos perseguidos foram estritamente delimitados no contexto de desenvolvimento de software. O intercâmbio de ferramentas para *workflow* e PSEEs vem sendo perseguido e deve ser estimulado, tal como no provocativo texto de Conradi, Fuggeta e Jaccheri [CON98], que afirma que os problemas trabalhados por ambas as tecnologias são essencialmente os mesmos. Embora, aparentemente, o modelo aqui descrito possa ser aplicado em processos adminitrativos, investigação adicional ainda é necessária para avaliar a sua exeqüibilidade nesse novo contexto.

7.4.4 Fornecer uma metodologia para desenvolvimento de processos de software reutilizáveis

A experiência no uso prático das linguagens e ferramentas aqui apresentadas levou a constatação que alguma assistência é necessária para guiar os usuários novatos na modelagem de *templates* e processos de software. Essa é uma constatação natural, visto que, assim como processos de software são propostos para disciplinar e orientar a produção de software, abordagens metodológicas também seriam necessárias para orientar a modelagem dos processos de software, registrando experiências bem sucedidas para uso futuro.

7.4.5 Investigar semelhanças com novos paradigmas para o desenvolvimento de software

O processo de desenvolvimento do meta-modelo aqui proposto presenciou o surgimento e evolução de novos paradigmas de desenvolvimento de software. Nesse ínterim, a abordagem orientada a aspectos [LOP97] ganhou atenção da comunidade internacional, propondo uma solução arrojada para a separação explícita de detalhes no projeto e programação de sistemas de software.

O presente trabalho e, em especial, o modelo de Políticas de Processos possui semelhanças nos objetivos e na forma da solução com o paradigma orientado a aspectos sendo, entretanto, voltados para o contexto específico de modelagem de processos de software. Uma discussão preliminar sobre o tema, que traça um paralelo entre os requisitos tratados por Políticas (na modelagem de processos) e aspectos (na definição de elementos arquiteturais de software), foi tópico de um artigo do autor publicado no *International Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design* [REI2002d]. Tal discussão, contudo, ainda precisa ser amadurecida, se valendo da necessária unificação da terminologia adotada por esse paradigma.

7.4.6 Investigar modelos para avaliação das soluções disponíveis

A reutilização de processos de software é um tópico recente que está atraindo crescente atenção da comunidade internacional. Assim, espera-se que no futuro próximo muitas soluções sejam propostas para os mais diversos problemas existentes nesse campo. Desse modo, é necessário algum modelo que auxilie a avaliação das soluções propostas, permitindo a comparação e uniformização dos conceitos tratados.

O conjunto de requisitos apresentados no capítulo 2 desse texto constitui uma contribuição importante para o tema, categorizando e definindo as especificidades das ferramentas desenvolvidas. Entretanto, deve-se dar ainda maior ênfase à avaliação prática das soluções. A disponibilização de exemplos-padrão para área, tal como o modelo ISPW [HEI91] para avaliação de ambientes de modelagem e execução de processos de software, constituiria um elemento importante para auxiliar a avaliação prática da adequabilidade e eficiência das soluções propostas.

7.4.7 Investigar mecanismos para prevenção e detecção automática de Políticas Estáticas conflitantes

No modelo *APSEE*-Reuse proposto, as Políticas Estáticas habilitadas em um processo ou *template* podem incorrer em conflitos. Uma Política é dita conflitante para um processo se, para o conjunto de Políticas habilitadas (na organização, no processo mencionado, e nos componentes do processo), tal Política conflitante somente é satisfeita se pelo menos uma das Políticas habilitadas falhar. Embora tal situação de conflito não tenha conseqüências críticas para o modelo geral, alguma assistência para detectar e prevenir automaticamente a habilitação de Políticas conflitantes seria de grande valia para facilitar a modelagem de processos.

A ocorrência de conflitos é um aspecto comum em muitas das abordagens de Políticas propostas na literatura para as diversas áreas da Ciência da Computação [MOF94][COL2001]. Uma das principais razões que justificam a utilização de métodos de especificação formal na definição de um modelo de meta-políticas é a possibilidade de se definir mecanismos para detecção conflitos e análise de consistência (Steem apud Cole et al [COL2001]). Em virtude da base semântica algébrica aqui fornecida para a definição e interpretação de Políticas Estáticas, vislumbra-se como trabalho futuro a realização de investigações adicionais acerca da análise e prevenção de conflitos.

7.5 Interação com pesquisadores do tema

No período de janeiro a março de 2001, o autor e uma doutoranda do grupo PROSOFT (Carla Reis) estiveram participando de uma missão científica no Programa de Cooperação Internacional em Tecnologia da Informação entre Brasil-Alemanha (apoiada pelo CNPq e DLR). A missão - descrita em [REI2001b] - envolveu um estágio na *Universität Stuttgart* com o objetivo geral de discutir e propor aperfeiçoamentos que àquela altura eram necessários do ambiente PROSOFT-Java, contando com o apoio do Sr. Heribert Schlebbe (pesquisador associado ao projeto PROSOFT em Stuttgart). A estada em Stuttgart foi de fundamental importância para o trabalho aqui apresentado, pois contribuiu para o início da especificação e implementação do protótipo do sistema *APSEE*.

No âmbito do programa de cooperação internacional, a estada da Alemanha também foi útil para realizar visitas a importantes centros de pesquisa alemães que tratam do tema de automação de processo de software. Em Stuttgart, aconteceram várias reuniões com membros do grupo de Engenharia de Software liderado pelo Prof. Jochen Ludewig, que trabalha em um ambiente de simulação de processos de software chamado SESAM [LUD2001].

No dia 13 de fevereiro de 2001 foi realizada uma visita ao grupo de Engenharia de Software da *Universität Kaiserslautern* e do Fraunhofer IESE (*Institut für Experimentale Software Engineering*), liderados pelo Prof. Dieter Rombach. A visita a

Kaiserslautern foi particularmente útil por detectar semelhanças entre objetivos das pesquisas realizadas no contexto de reutilização de processos de software e o trabalho realizado pelo Sr. Jürgen Münch (que, naquela época, estava realizando Doutorado sob orientação do Prof. Rombach). Visitas e contatos posteriores com o Sr. Münch - pessoalmente em setembro de 2001 e através da Internet - estabeleceram iniciativas de trabalho futuro em conjunto no contexto do projetos APSEE e MVP-L [LOT95], vislumbrando, em especial, a possibilidade de descrever uma evolução da ferramenta *ProTail* [MÜN97] para adaptar *templates* APSEE.

No dia 5 de março de 2001 foi realizada uma visita científica ao grupo de Métodos Formais da *Techniche Universität Berlin*, liderado pelo Prof. Hartmut Ehrig, propiciando discussão acerca dos métodos usados na especificação formal do sistema APSEE. A visita foi importante por propiciar uma frutífera discussão sobre a experiência do grupo de Berlim no uso de técnicas baseadas em Gramáticas de Grafos, as quais foram utilizadas de forma bem sucedida na especificação do trabalho aqui apresentado.

Em 15 de março foi realizada a última visita, dessa vez ao grupo de Engenharia de Software da *Universität Dortmund*, liderado pelo Prof. Volker Gruhn. Esta visita propiciou discussões acerca do mecanismo de execução proposto e sobre a recente proposta da doutoranda Ursula Wellen na composição de processos abstratos usando o conceito de *Process Landscaping* [WEL99a] [WEL99b].

A participação do autor em eventos internacionais, assim como as críticas obtidas a partir da avaliação de artigos submetidos, foram, também importantes mecanismos de aperfeiçoamento do trabalho e criação de vínculos com personalidades atuantes na área. Em especial, vale destacar o contato com o doutorando Josep Maria Ribó da *Universitat Politècnica de Catalunya* (Barcelona, Espanha), durante o CRIWG'2001, o que vem permitindo o contínuo intercâmbio de informações acerca de pontos de interesse comum em relação aos assuntos investigados.

7.6 Considerações finais

Esse trabalho representa um passo na evolução da Tecnologia de Processo de Software no sentido de proporcionar uma abordagem automatizada para auxiliar a reutilização de processos. O autor acredita que o aprofundamento da pesquisa nesse assunto pode levar a um aumento significativo na qualidade dos processos adotados e, por conseqüência, nos produtos resultantes de organizações de desenvolvimento de software.

Apesar de tudo que a comunidade internacional já desenvolveu, a modelagem de processos ainda é uma tarefa complexa que demanda profissionais altamente especializados. Esforços ainda devem ser realizados no sentido de unificar a terminologia, notações e requisitos no contexto das soluções automatizadas para reutilização de processos de software, a fim de amadurecer a pesquisa na área. A ocorrência da primeira edição do *International Workshop on Software Development Process Patterns*[48], em novembro de 2002, é um indício do amadurecimento da área e da necessidade de discussão dos problemas tratados a partir de uma perspectiva global.

---

[48] Workshop proposto pelo projeto Zen (*Institut für Informatik - Technische Universität München* – Alemanha) a ser realizado como evento satélite do *XVII ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2002)*, como descrito em [GNA2002].

Anexo 1 Construtores de tipos do PROSOFT-Algébrico

TABELA 7.1 - Lista de construtores disponíveis no PROSOFT-Algébrico

Type	Operation	Functionality	Example
Set	Add	Set x Component → Set	add ({x1,x2},x3) = {x1,x2,x3}
	Belongs_to (∈)	Component x Set → Boolean	x1 ∈ {x1,x2} = TRUE
	Cardinality (car)	Set → Nat	car ({x1,x2,x3}) = 3
	Complement (\ )	Set x Set → Set	{x1,x2} \ {x2} = {x1}
	Containtion (⊇)	Set x Set → Boolean	{x1,x2,x3} ⊇ {x1,x3} = TRUE
	Delete	Set x Component → Set	delete ({x1,x2},x2) = {x1}
	Empty-set	→ Set	empty-set = { }
	Equal (=)	Set x Set → Boolean	{x1} = {x2} = FALSE
	Intersection (∩)	Set x Set → Set	{x1,x2} ∩ {x2} = {x2}
	Is_in	Set x Component → Boolean	is_in ({x1,x2},x1) = TRUE
	Union (∪)	Set x Set → Set	{x1,x2} ∪ {x3} = {x1,x2,x3}
Map	Composition (Θ)	Map x Map → Map	[x1 → y1] Θ [x2 → y2] = [x1 → y2]
	Domain (dom)	Map → 2^{Set}	dom ([x1 → y1, x2 → y2]) = {x1, x2}
	Empty-Mapping	→ Map	empty-mapping = { }
	Image_of	Domain, Map → Rng	image_of (x1, [x1 → y1]) = y1
	Merge (⊍)	Map x Map → Map	[x1→y1] ⊍ [x2→y2] = [x1→y1,x2→y2]
	Modify	Domain x Rng x Map → Map	modify (x1, y1, [ ]) = [x1 → y1]
	Override (+)	Map x Map → Map	[x1→y1,x2→y2]+[x2→y3]=[x1→y1, x2→y3]
	Range (rng)	Map → Set	rng ([x1 → y1,x2 → y2]) = {y1, y2}
	Restrict_to (|)	Map x 2^{Set} → Map	[x1 → y1,x2 → y2] | {x1} = [x1 → y1]
List	Concatenation(^)	List x List → List	<x1, x2> ^ <x1> = <x1, x2, x1>
	Cons	Component x List → List	cons (x1, <>) = <x1>
	Elements (elems)	List → 2^{Set}	elems (<x1, x2, x1>) = {x1, x2}
	Empty-list	→ List	empty-list = < >
	Head (hd)	List → Component	hd (<x1, x2>) = x1
	Index (inds)	List → 2^{Nat}	inds (<x1, x2, x1>) = {1, 2, 3}
	Last	List → Component	last (<x1, x2, x1>) = x3
	length (lng)	List → Nat	lng (<x1, x2, x1>) = 3
	Projection (_,[ ])	List x Nat → Component	<x1, x2> [2] = x2
	Replace (_,+,[_])	List x Nat x Component → List	<x1, x2> + [2, x3] = <x1, x3>
	Tail (tl)	List → List	taill (<x1, x2>) = x2
Record	Record construction	(_,_,...,_):C_1,C_2, ...,C_n→ Record	(Address street, City city-name, Zip zipcode)
	Select-_	Record → C_k	select-City (_, City "Rio de Janeiro", _) = "Rio de Janeiro"

Anexo 2 Classes *Branch* e *Join*

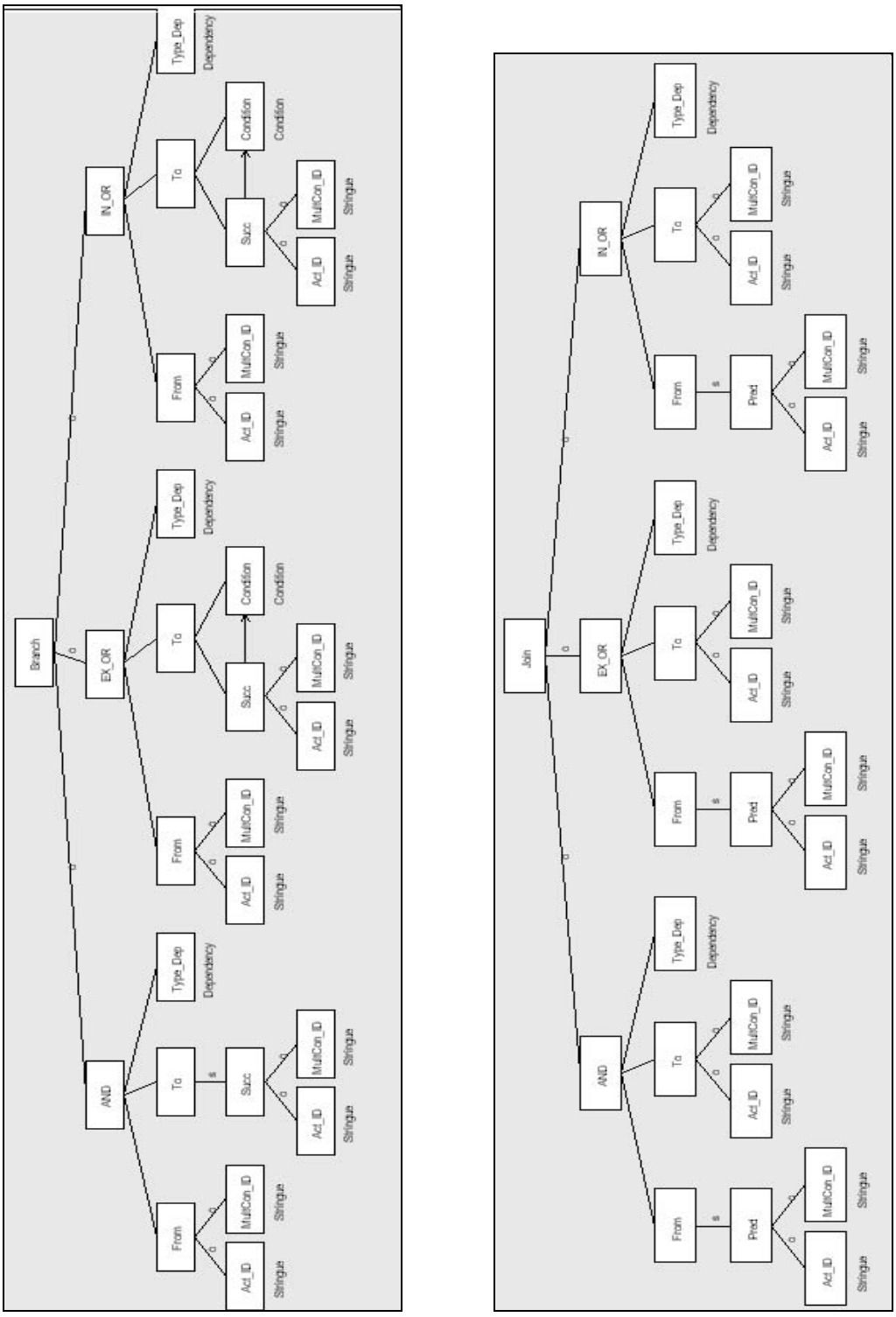

FIGURA A 1 As classes *Branch* e *Join*

Anexo 3 O Pacote StaticPolicy

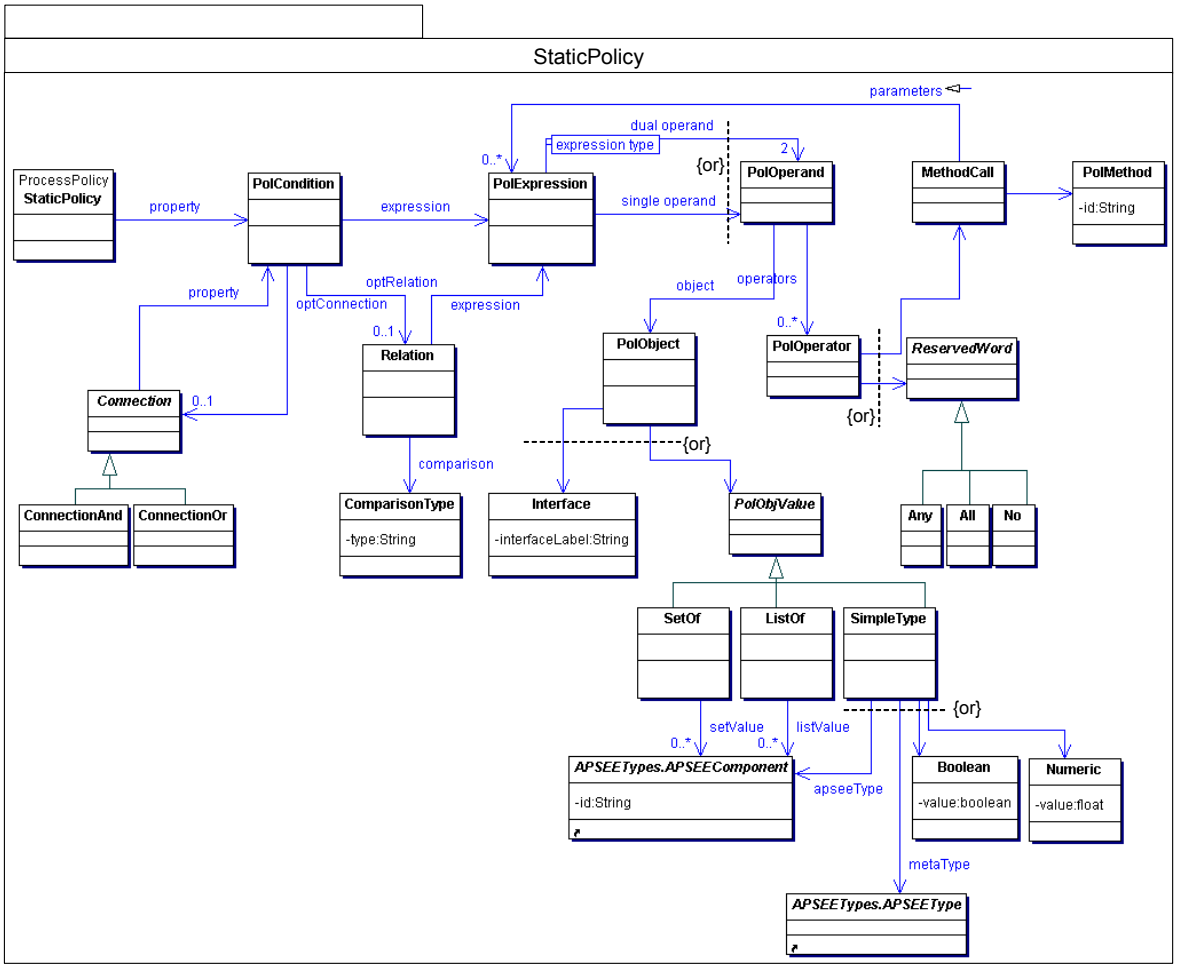

FIGURA A 2 A - definição de tipos para Políticas Estáticas

Anexo 4 As classes do pacote APSEE envolvidas na definição de processos executáveis

Esse anexo apresenta as classes descritas por Lima Reis em [LIM2002d] e utilizadas na especificação dos algoritmos de generalização de processos e adaptação de *templates* (descritos na seção 4.2.2).

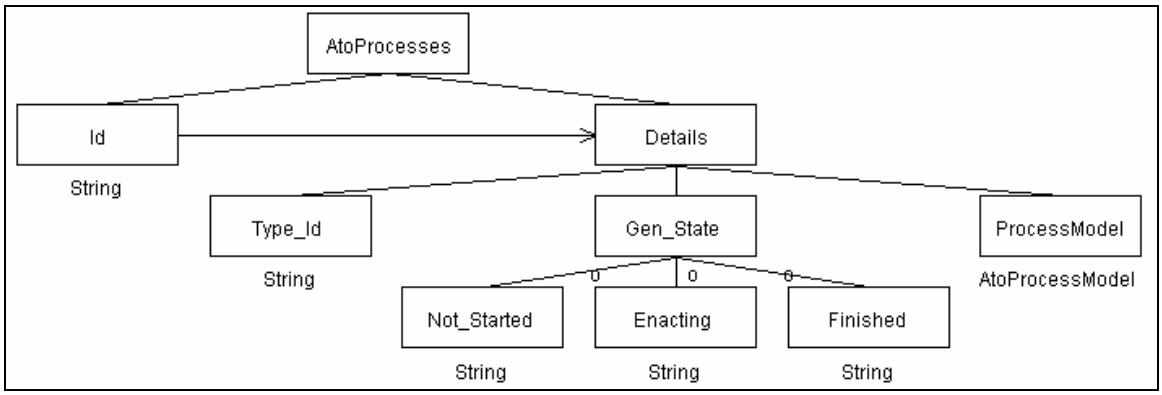

FIGURA A 3 - A classe *Processes*

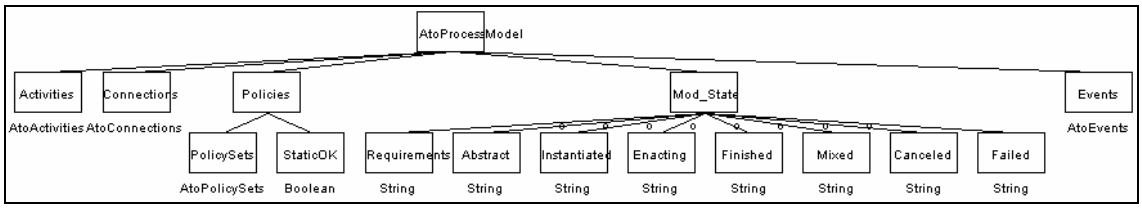

FIGURA A 4 - A classe *ProcessModel*

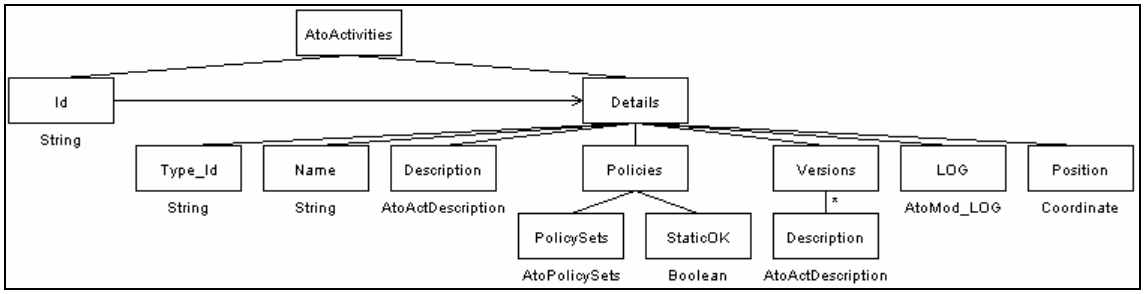

FIGURA A 5 - A classe *Activities*

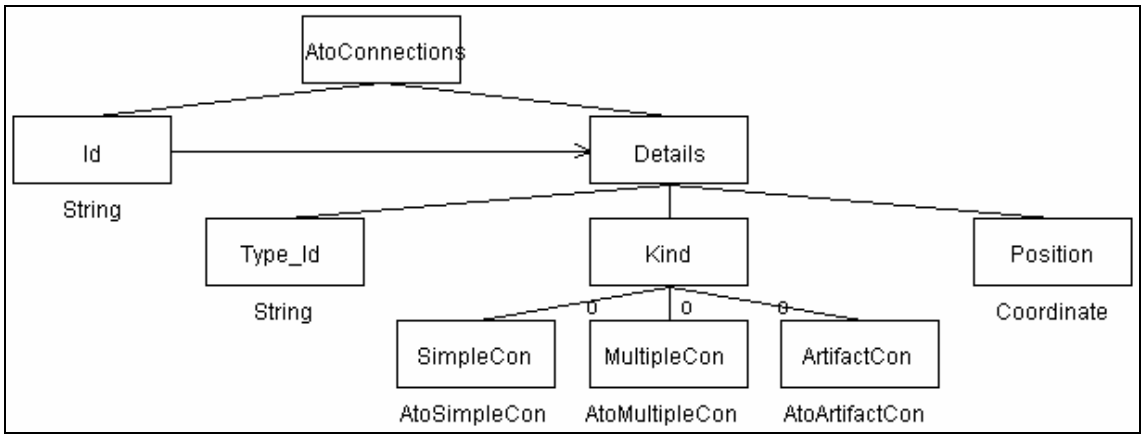

FIGURA A 6 - A classe *Connections*

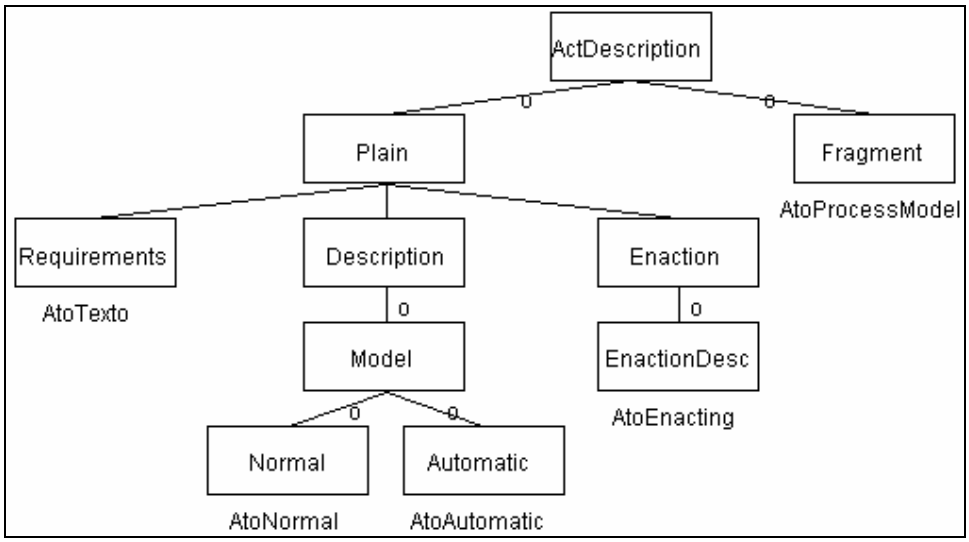

FIGURA A 7 - A classe *ActDescription*

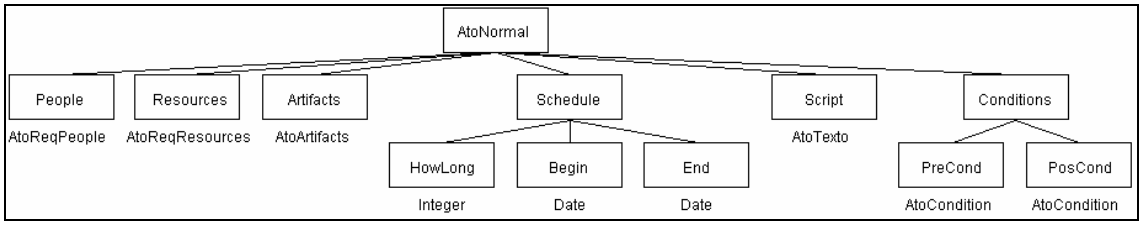

FIGURA A 8 - A classe *Normal*

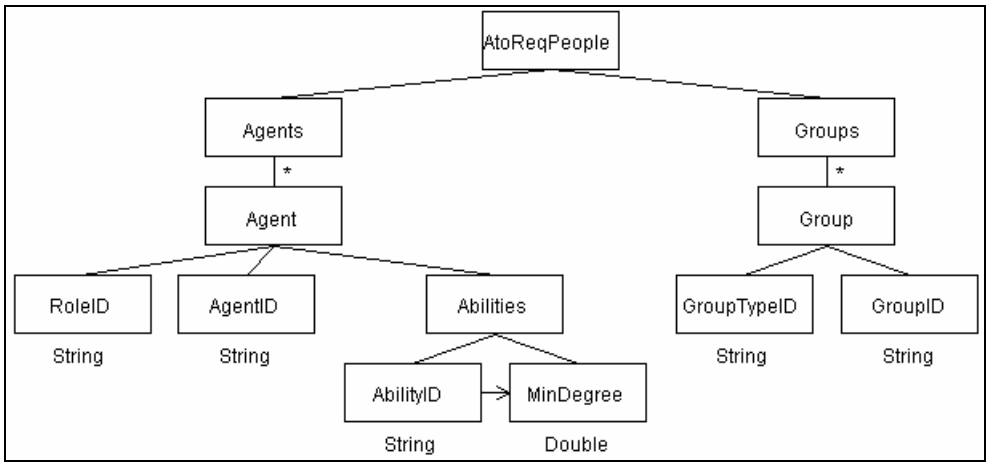

FIGURA A 9 - A classe *ReqPeople*

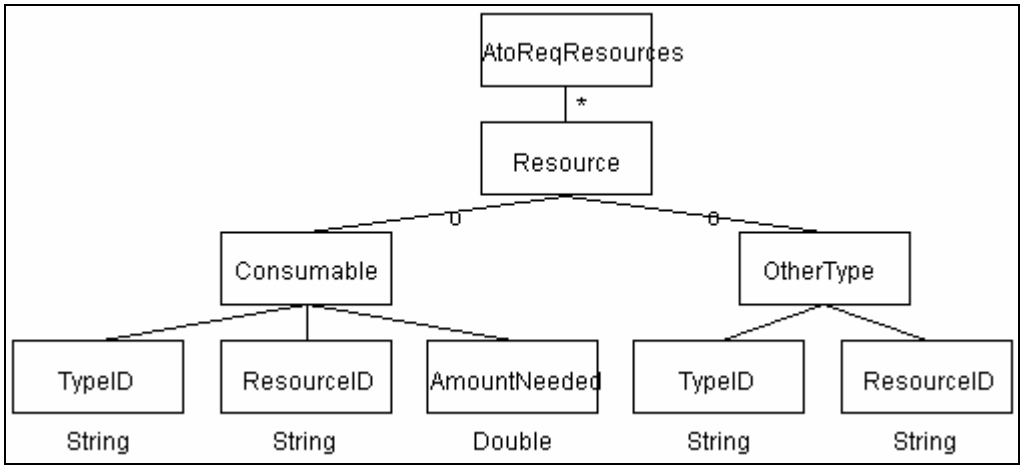

FIGURA A 10 - A classe *ReqResources*

Anexo 5 Regras para definição de *Templates*

A figura A 11 apresenta o grafo tipo para as regras de manipulação de *templates*. O grafo descreve o inter-relacionamento dos componentes usados na descrição nas regras.

O grafo é composto por nodos e arcos, com significados específicos. Os **nodos** são rotulados, e possuem representação gráfica específica, estando relacionados às classes PROSOFT descritas no capítulo 4. Alguns nodos são representados usando a notação concreta da linguagem gráfica para definição de *templates*: assim, os nodos rotulados como *Fragment*, *Automatic*, *NormalAbsDesc*, *AbsArtifactCon*, *SimpleCon* e *MultiCon* são representados graficamente com a mesma notação disponível no editor de *templates* construído. Os demais nodos não são exibidos graficamente para os usuários constituindo, portanto, de estruturas que são fornecidas pelo usuário através do preenchimento de formulários textuais ou seleção de itens de dados em listas. Alguns dos nodos do grafo tipo ainda descrevem - abaixo do rótulo do nodo - os atributos da classe PROSOFT correspondente que são úteis para descrever as regras de transformação que compõem o sistema.

Os **arcos** incluídos são divididos em dois tipos principais: os arcos com tracejado fino representam as associações de generalização-especialização do tipo **É-um** (*Is-a*), enquanto que as demais (com tracejado mais espaçado) descrevem associações diversas do modelo, devidamente evidenciadas pelos rótulos existentes. As associações do tipo é-um são orientadas no sentido da especialização para a generalização, enquanto que as demais associações são orientadas conforme as associações de tipos de dados descritas no capítulo 4.

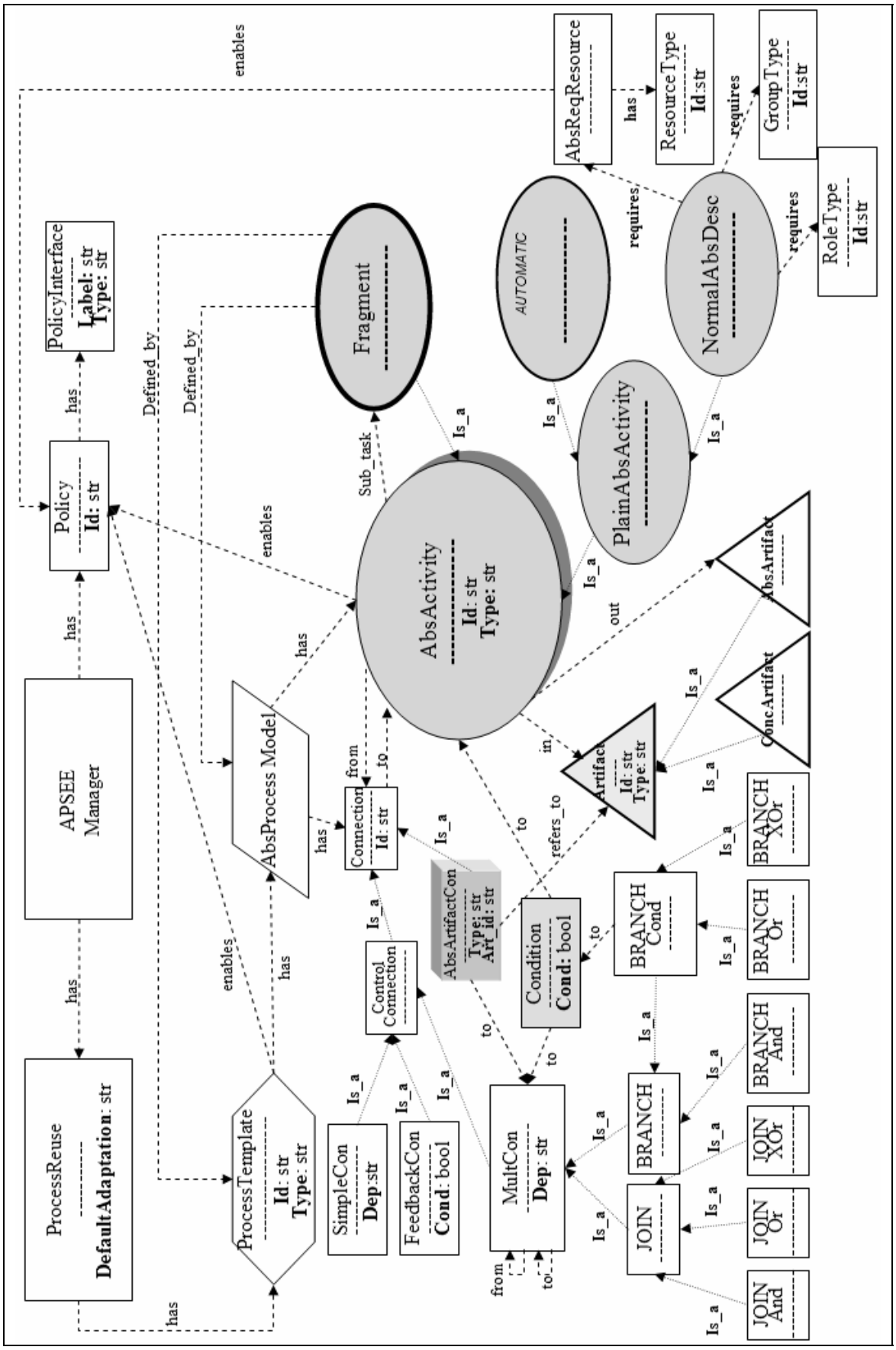

Figura A 11 - Grafo Tipo para *Templates*

As sub-seções a seguir apresentam regras que definem a linguagem visual de *ProcessTemplates*. Todas as regras são rotuladas e numeradas, de acordo com o elemento sintático associado. As regras são agrupadas conforme descrito abaixo:

- As regras do grupo G1 estão envolvidas com a manipulação de atividades em um *template*;
- As regras do grupo G2 tratam as conexões de artefato;
- As regras do grupo G3 descrevem a ordenação temporal de atividades através de conexões simples e múltiplas;
- As regras do grupo G4 descrevem regras relacionadas com *Join*;
- As regras do grupo G5 descrevem regras relacionadas com *Branch*;
- As regras do grupo G6 descrevem regras para a associação de tipos de cargos, grupos e recursos em atividades normais.

A5.1 Regras para manipulação de Atividades

As seções a seguir apresentam as regras envolvidas com a definição de atividades em um *template.*

A5.1.1 Regra G1.1 - incluindo uma nova atividade em um *template* existente

A figura A 12 apresenta a regra para inclusão de uma atividade em resposta a uma solicitação do usuário. Nesse texto, as regras são graficamente dispostas de uma maneira uniforme: o grafo origem é colocado à esquerda, enquanto que o grafo destino está à direita da figura. Acima do grafo origem é colocada a função correspondente (com o seu nome e parâmetros reais). A seta, colocada entre os dois grafos, é rotulada com o nome da regra descrita.

Na regra *NewAbsActivity*, o usuário deve fornecer identificadores para o *template* e para atividade a ser criada (*template-id* e *new-id*, respectivamente). O lado esquerdo da regra restringe que não pode existir uma atividade com mesmo identificador para o *template* associado, enquanto que o lado direito cria a atividade abstrata Normal[49] com *script* vazio.

[49] O comportamento *default* do editor de *templates* aqui especificado implica sempre na criação de atividades do tipo *Normal*, as quais podem ser posteriormente alteradas pelo usuário para fragmentos ou atividades automáticas.

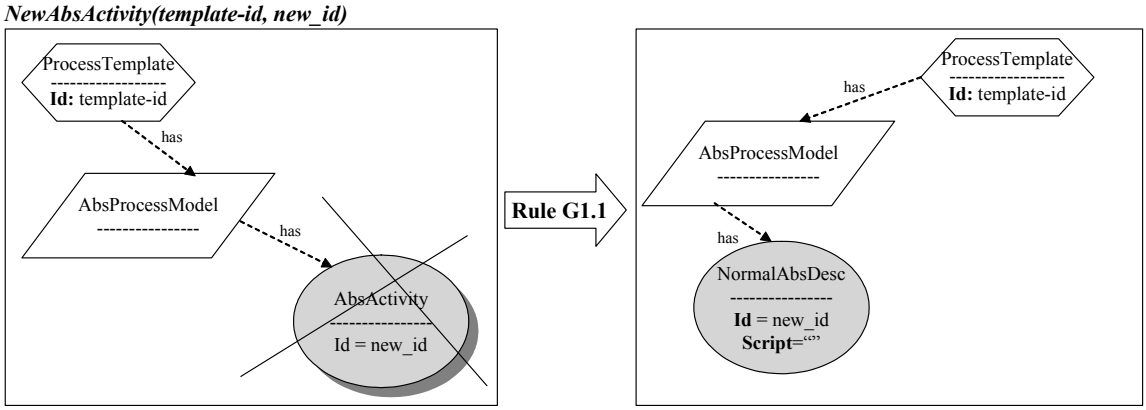

FIGURA A 12 - A regra G1.1 - função *NewAbsActivity*

A5.1.2 Regras G1.2 e G1.3 - refinando uma atividade em fragmento

A partir de uma atividade existente no *template*, o usuário deve ter a opção de transformá-la em um fragmento. A figura A 13 apresenta a regra G1.2, a qual é aplicável para atividades normais (i.e., instâncias de *NormalAbsActivity*). Se uma atividade normal que corresponda àquela indicada como parâmetro da função (com seu identificador coincidindo com *act-id*, pertencente à *template-id*), então esta é transformada em um fragmento, o qual está associada uma nova instância de *AbsProcessModel*, permanecendo com o mesmo identificador (*act-id*).

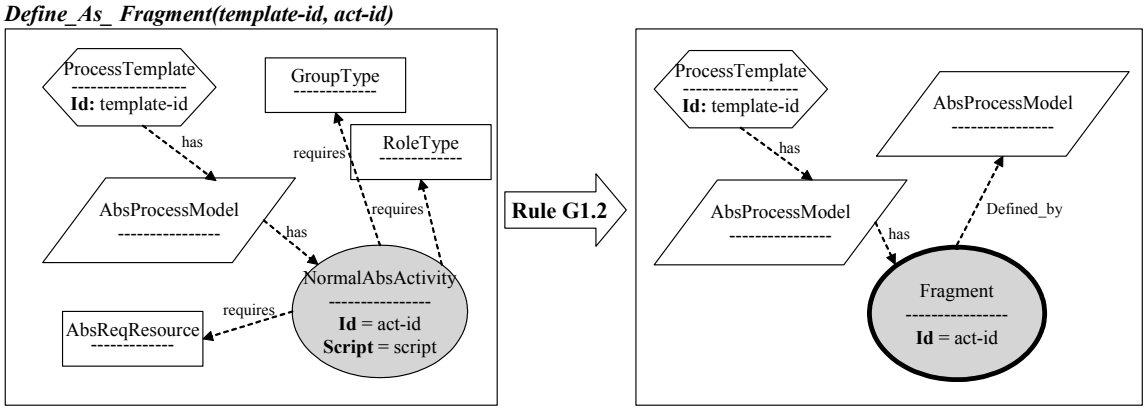

FIGURA A 13 - A regra G1.2 - *Define_As_Fragment* para atividade normal.

A transformação de uma atividade automática em fragmento é definida pela regra G1.3, apresentada na figura A 14, sendo análoga à regra anterior.

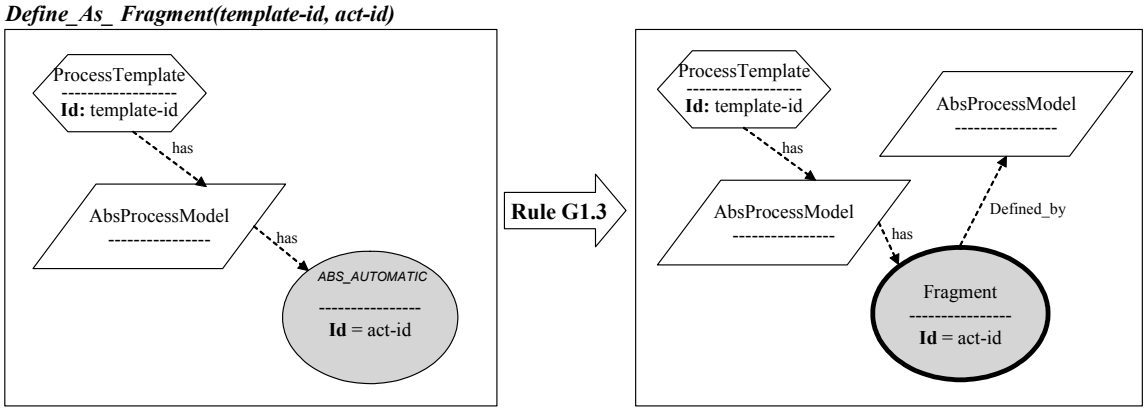

FIGURA A 14 - A regra G1.3 - *Define_As_Fragment* para atividade automática

A5.1.3 Regras G1.4 e G1.5 - transformando em atividade automática

Durante a edição de um *template* é possível que o usuário deseje transformar uma atividade existente em automática. Esta seção apresenta os dois casos possíveis.

A figura A 15 apresenta a regra G1.4 que trata da transformação de uma atividade normal em automática. No caso, todos os elementos associados à atividade (associação *requires*) são removidos, e a atividade nova criada mantém o mesmo identificador.

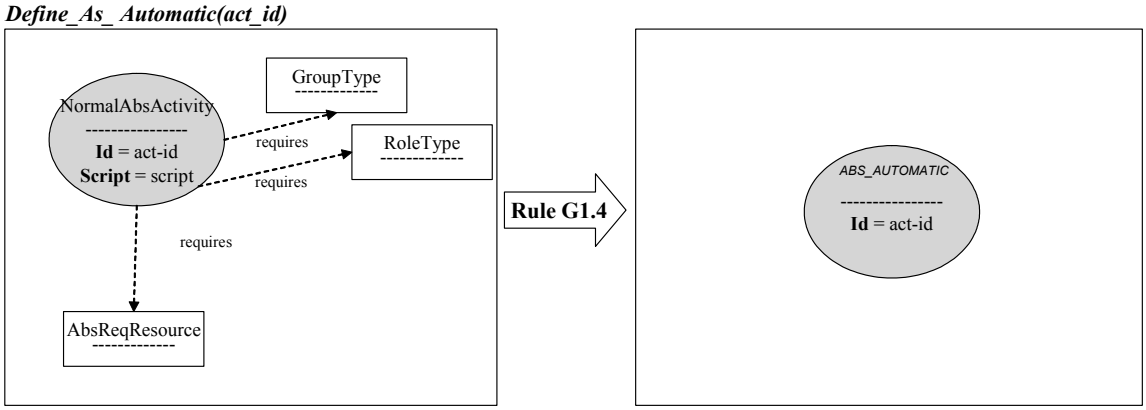

FIGURA A 15 - A regra G1.4 - *Define_As_Automatic* para atividade Normal

A transformação de um fragmento em uma atividade folha automática é definida na figura A 16. A restrição expressa no lado esquerdo da regra exige que o fragmento esteja vazio, isto é, não exista atividade-componente.

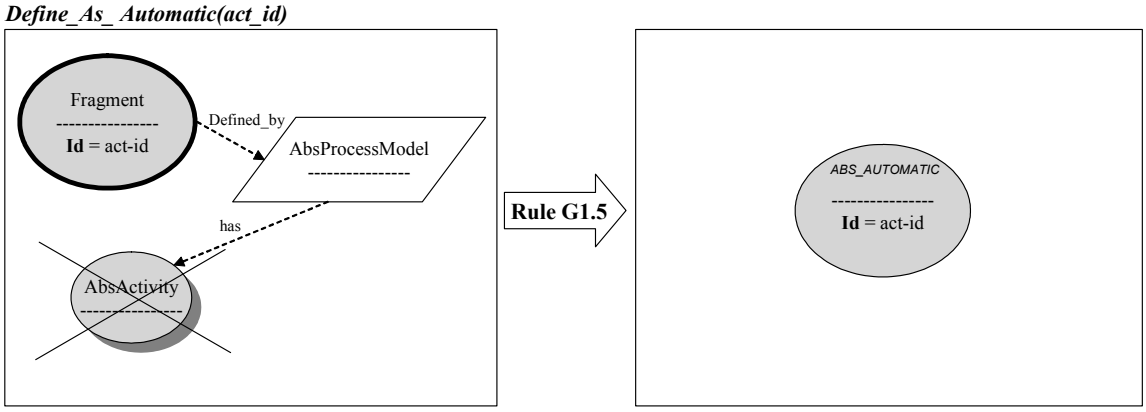

FIGURA A 16 - A regra G1.5 - *Define_As_Automatic* para fragmento

A5.1.4 Regras G1.6 e G1.7 - transformando em atividade folha normal

A criação de uma atividade folha automática a partir de uma atividade normal ou de um fragmento é descrita pela função *Define_As_Normal*, a qual é especificada através das regras G1.6 e G1.7 na figura A 17. Nas duas regras o identificador da atividade original é mantido na nova atividade automática criada. No caso de um fragmento, é imprescindível que esteja vazio.

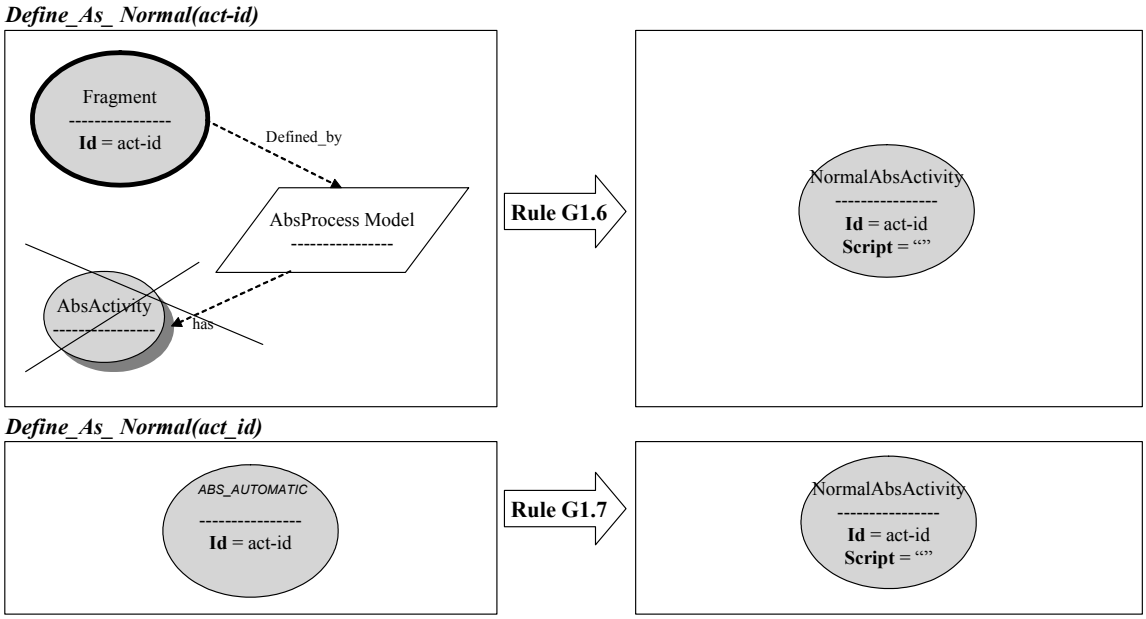

FIGURA A 17 - As regras G1.6 e G1.7 - *Define_As_Normal* para fragmento e atividade automática

A5.1.5 Regras G1.8 e G1.9 - definindo artefatos de entrada e saída

Os artefatos de software previamente definidos (i.e., que possuam um identificador válido) são associados à entrada ou saída de uma atividade. A regra G1.8 define a inclusão de um artefato qualquer (super-tipo *Artifact*) pela porta *in* de uma atividade qualquer (figura A 18). A função *Define_Output_Artifact* (regra G1.9) é restrita para artefatos abstratos.

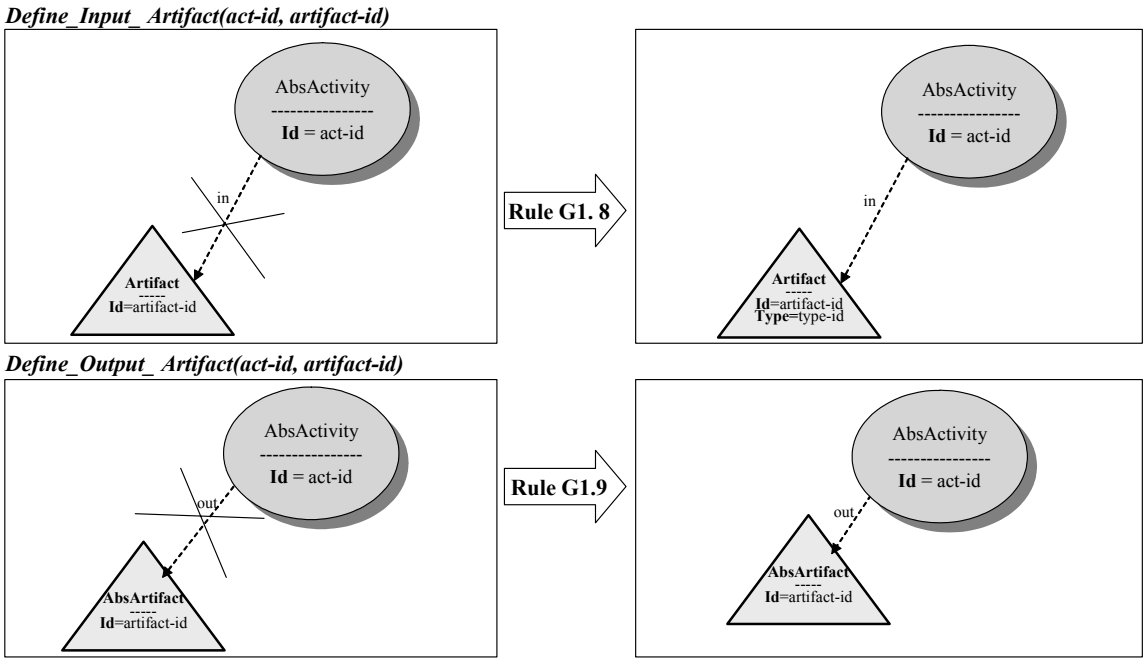

FIGURA A 18 - As regras G1.8 e G1.9 - funções *Define_input_artifact* e *Define_output_artifact*

A5.1.6 Regras G1.10, G1.11 e G1.12 - removendo atividade do *template*

A remoção de uma atividade de um *template* é definida pela função *DeleteActivity*, a qual é definida pelas regras especificadas na figura A 19. São três os casos tratados:

- a regra G1.10 trata da remoção de Fragmentos (os quais devem estar obrigatoriamente sem componentes);
- a regra G1.11 define a remoção de atividades automáticas;
- a regra G1.12 trata da remoção de atividades normais (removendo também o item *AbsReqResource* e mantendo os tipos de Grupo e Cargo que estiverem associados).

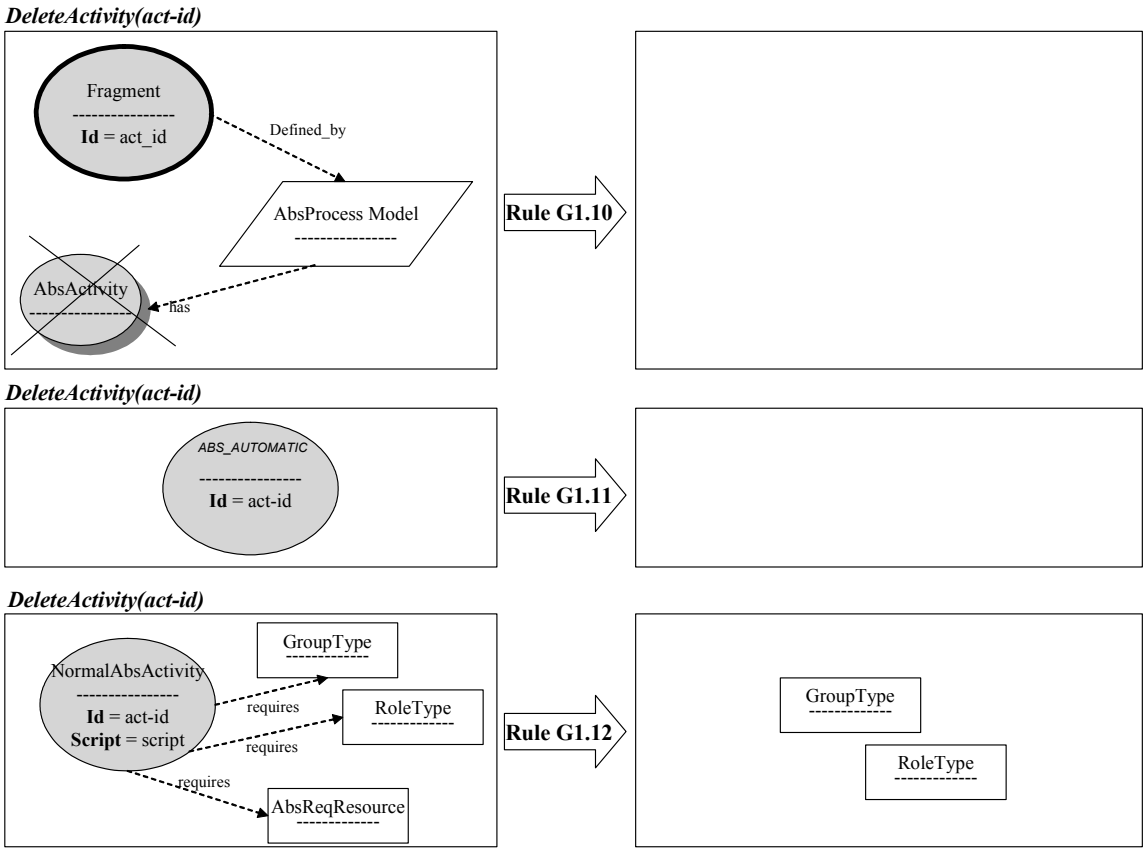

FIGURA A 19 - As regras G1.10, G1.11 e G1.12 - a função *DeleteActivity*

A5.2 Regras para manipulação de Conexões de Artefato

Esta seção descreve as regras de manipulação de conexões de artefato em um *template* de processo de software.

A5.2.1 Regra G2.1 - Criação de Conexões de Artefato

Uma conexão de artefato pode ser criada livremente em qualquer área livre de um modelo de processo de software. As conexões possuem identificadores usados internamente pelo sistema para guardar informações sobre a origem e destino das conexões, além do tipo associado. Esses identificadores não são visíveis para o usuário, visto que na implementação do editor assume-se a realização de operações sobre conexões de artefato são identificadas a partir da coordenada clicada pelo usuário. Portanto, a regra G2.1 descreve a criação de uma conexão de artefato, sendo *com-id* determinado internamente pelo sistema e o atributo *Type* indefinido.

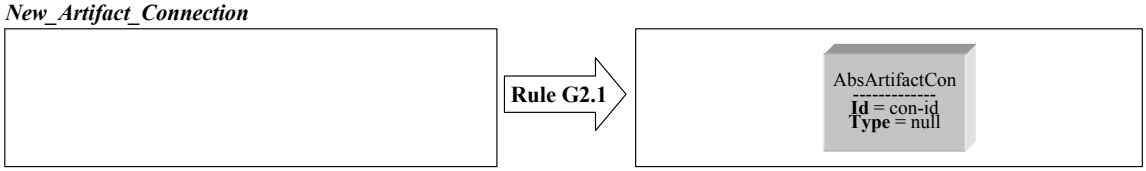

FIGURA A 20 - A regra G2.1 - função *New_Artifact_Connection*

A5.2.2 Regras G2.2 a G2.8 - Definindo e alterando o tipo de uma conexão de artefato

Uma conexão de artefato pode ser definida primeiramente a partir do tipo de artefato de software. Esse tipo pode ser definido pelo usuário para restringir as instâncias de artefato que podem ser associadas a conexão tratada.

A figura A 21 apresenta a regra G2.2 para a função que define um tipo de artefato em uma conexão recentemente criada (i.e., com o atributo *Type* possuindo o valor *null*).

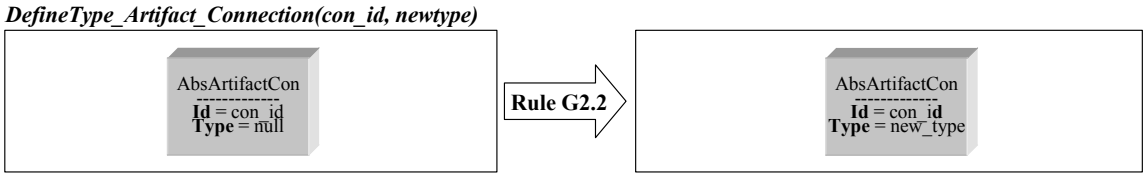

FIGURA A 21 - A regra G2.2 - função *DefineType_Artifact_Connection*

A função para alteração do tipo de uma conexão já definida é descrita pelas regras G2.3 e G2.4. Em G2.3 o caso descreve uma conexão de artefato já associada um artefato identificado por *artifact-id*. Nesse caso, a conexão *refers_to* é removida, o identificador do artefato associado é removido do atributo *Art_id*, e o novo tipo fornecido pelo usuário (*newtype*) é associado ao atributo *Type* (conforme descrito pelo lado direito da regra G2.3). A regra G2.4, por sua vez, é aplicável para uma conexão de artefato ainda não associada a um artefato.

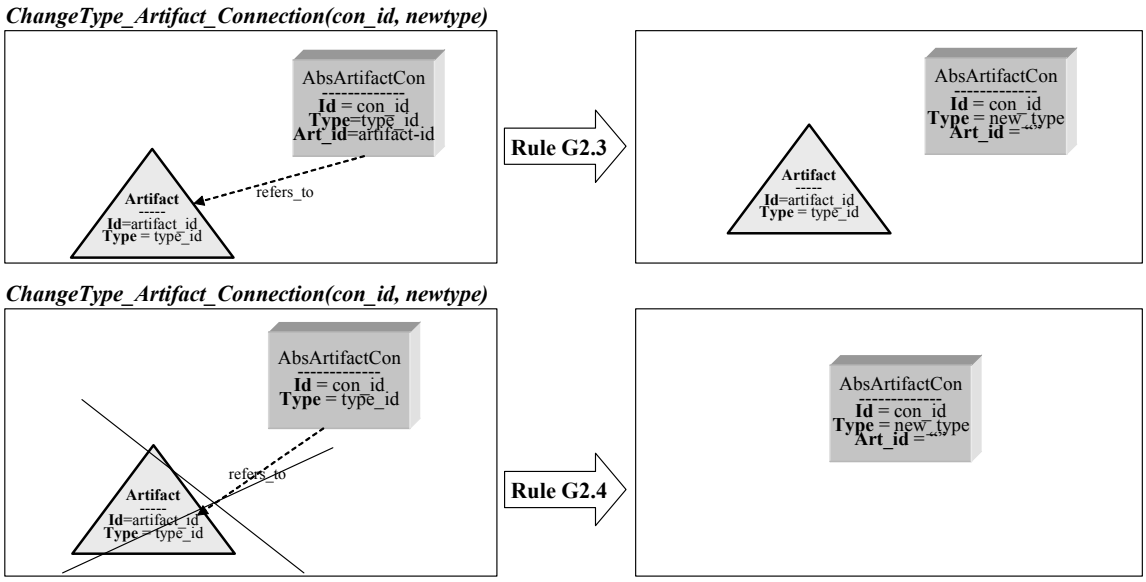

FIGURA A 22 - As regras G2.3 e G2.4 - função *ChangeType_Artifact_Connection*

A função fornecida para definir a instância de artefato de software associada a uma conexão de artefato (função *DefineInstance_Artifact_Connection*) é fornecida pelas regras das figuras abaixo:

- Regra G2.5 - A condição expressa no lado esquerda da regra exige que a conexão de artefato não esteja associada a nenhum outro artefato (identificado por *other_art_id*) e que a conexão tenha sido recentemente criada (i.e., o atributo *Type* assume o valor *Null*);

- Regra G2.6 - Esta regra é aplicada se a conexão de artefato alvo (com identificador *con-id*) está associada a um artefato existente (com identificador *other_art_id*). A transformação remove a associação *refers_to* para o artefato antigo, associando o objeto de *AbsArtifactCon* para o artefato identificado por *artifact-id*.
- Regra G2.7 - A regra é aplicável quando o tipo expresso na condição de artefato (*typedef*) já está definido, e a conexão de artefato não está associada a qualquer artefato. Daí, a associação de artefato fica restrita ao tipo definido. Portanto, a condição expressa abaixo da seta "Rule G2.7" exige que o tipo do artefato fornecido pelo usuário (*type_id*) seja um subtipo válido do tipo expresso na conexão (*typedef*);
- Regra G2.8 - Estende a regra G2.7, tratando para o caso em que um artefato já esteja associado à conexão.

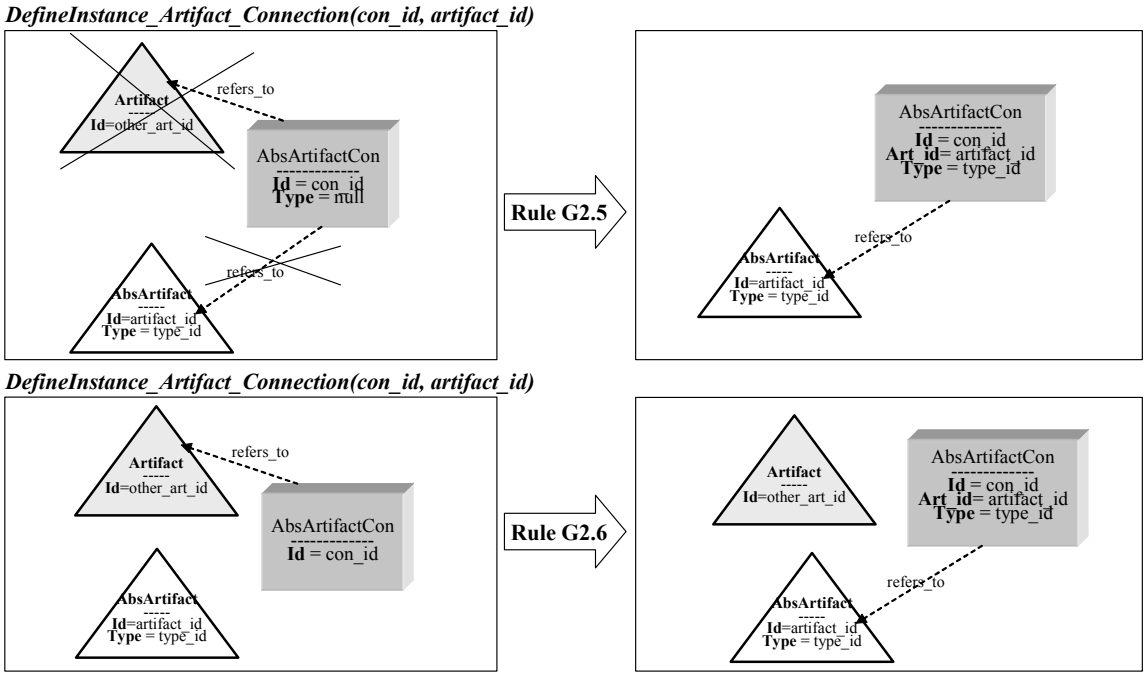

FIGURA A 23 - As regras G2.5 e G2.6 - função *DefineInstance_Artifact_Connection*

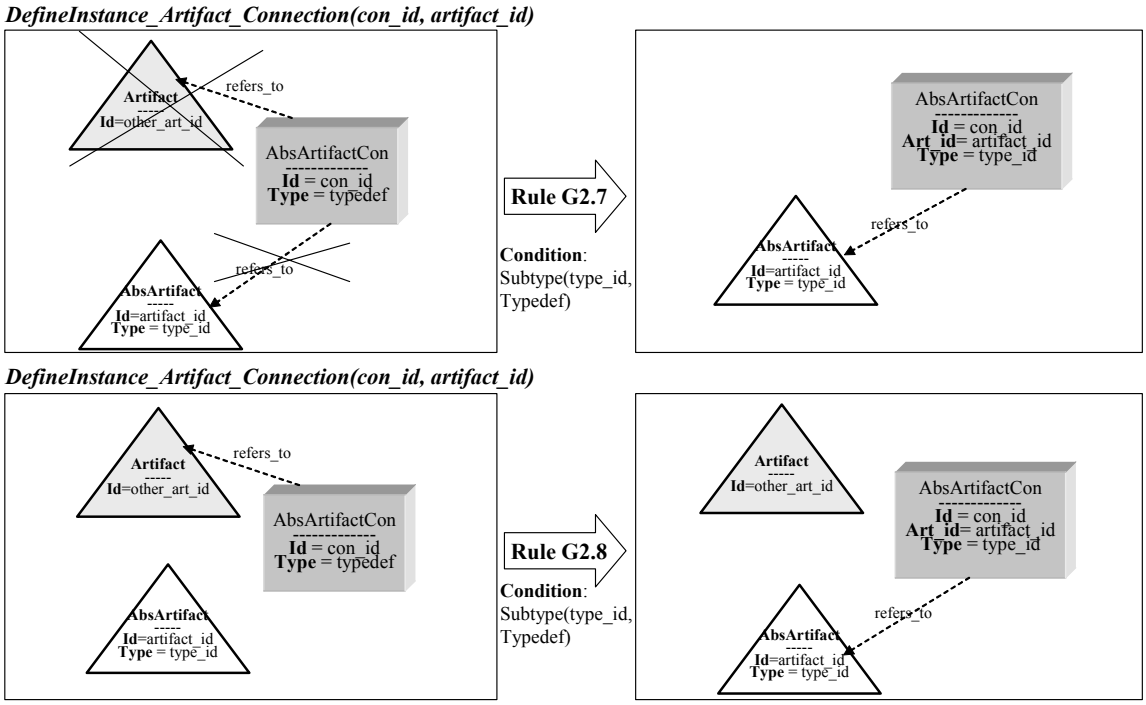

FIGURA A 24 - As regras G2.7 e G2.8 - função *DefineInstance_Artifact_Connection*

A5.2.3 Regra G2.9 - Eliminando uma Conexão de Artefato

A regra G2.9 descrita na figura A 25 abaixo define a semântica para a função *RemoveInstance_Artifact_Connection*. A partir do parâmetro fornecido (o identificador de conexão *con_id*), a associação *refers_to* da conexão é removida.

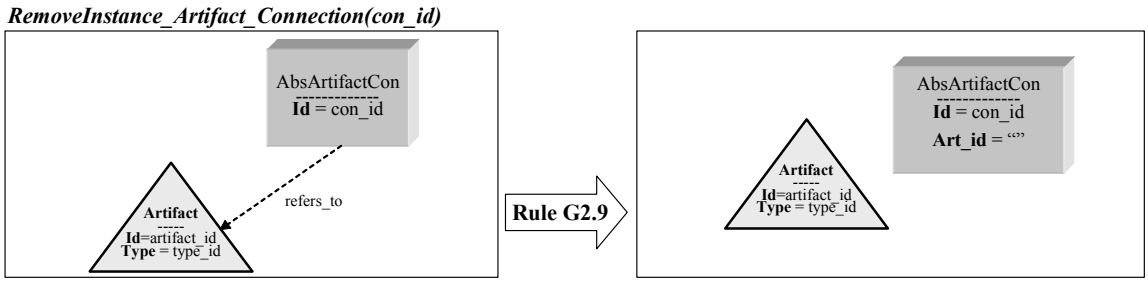

FIGURA A 25 - Regra G2.9 - função *RemoveInstance_Artifact_Connection*

A5.2.4 Regras G2.10 a G2.22 - Definindo as entradas e saídas de uma Conexão de Artefato

Esta seção descreve como conexões de artefatos são associadas às atividades e conexões múltiplas. O primeiro grupo de regras (de G2.10 a G2.13) descreve as funções relacionadas com a definição de artefatos de saída (*output*), as quais lidam com o atributo *out* das atividades. Enquanto as regras G2.10 e G2.11 lidam com conexões para atividades folha (objeto *PlainAbsActivity*), a regra G2.12 lida com atividades decompostas (fragmentos).

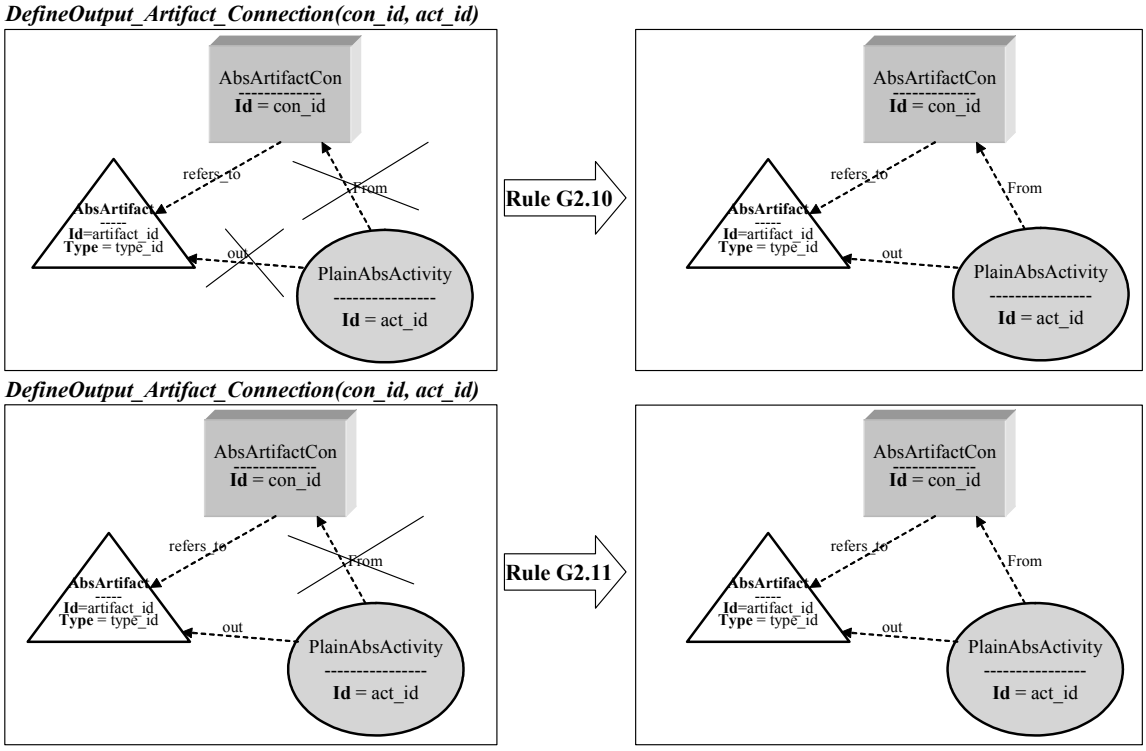

FIGURA A 26 - Regras G2.10 e G2.11 - função *DefineOutput_Artifact_Connection*

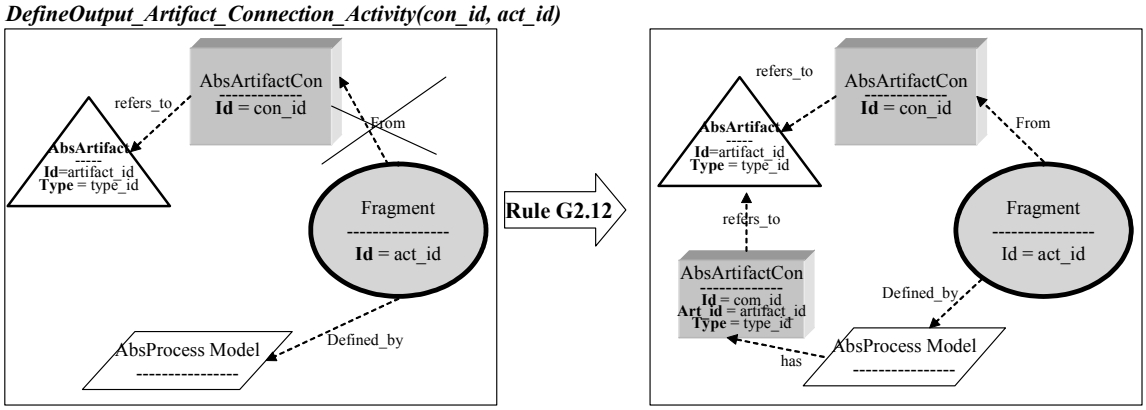

FIGURA A 27 - Regras G2.12 - função *DefineOutput_Artifact_Connection_Activity*

A figura A 28 apresenta a regra G2.13 que define a remoção de um artefato de saída para uma atividade qualquer.

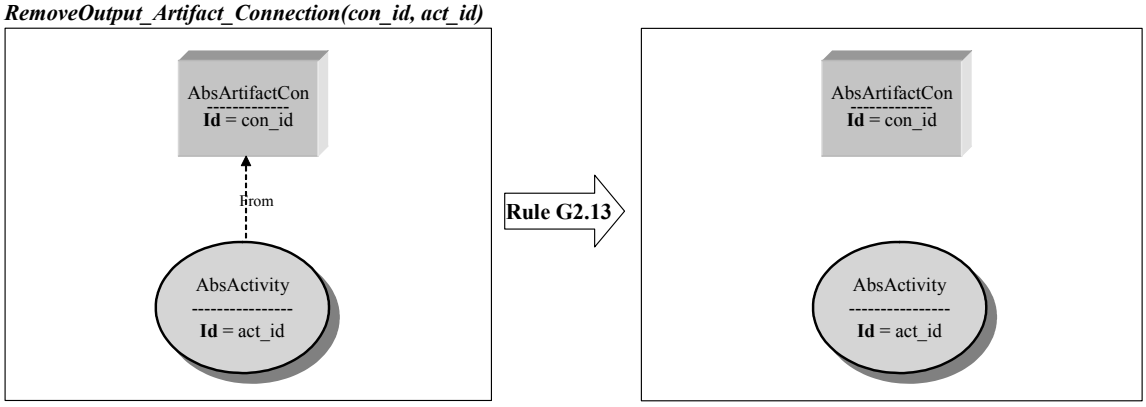

FIGURA A 28 - Regras G2.13- função *RemoveOutput_Artifact_Connection*

O segundo grupo de regras (de G2.14 a G2.16) apresenta as regras envolvidas com os artefatos de entrada para uma atividade folha (regra G2.14), para fragmento (regra G2.15) e a remoção (para qualquer tipo de atividade na regra G2.16).

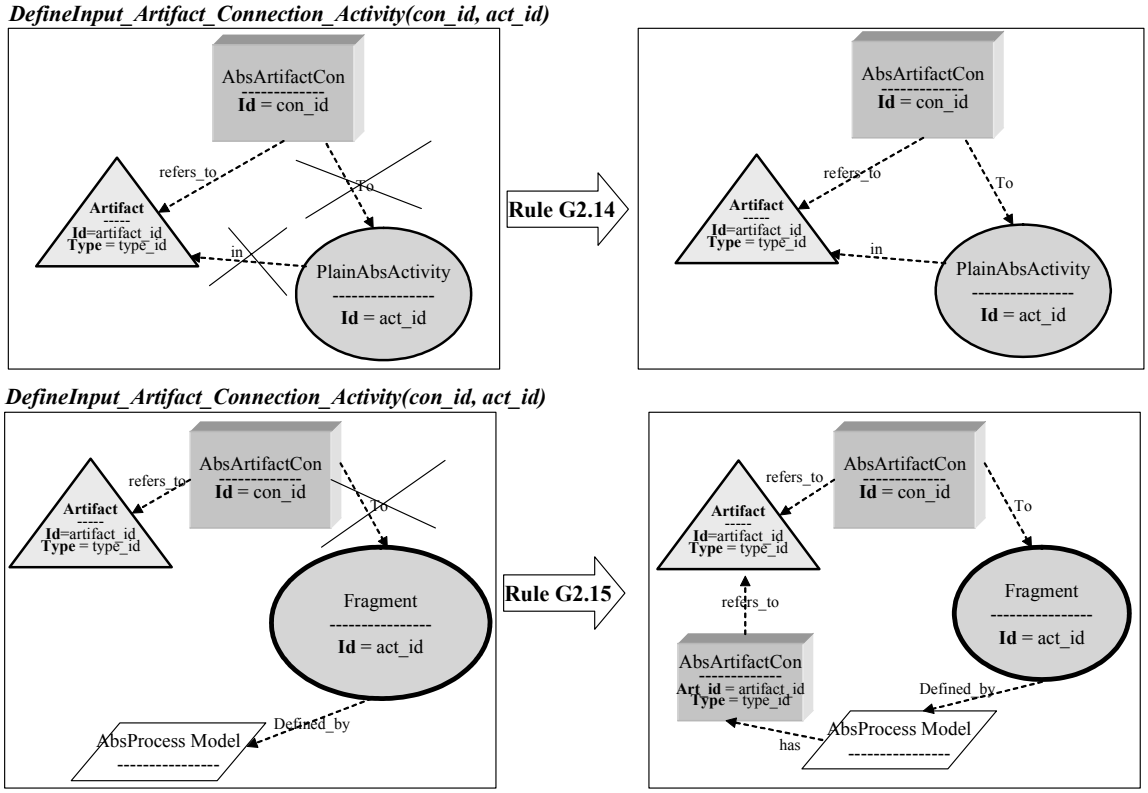

FIGURA A 29 - Regras G2.14 e G2.15 - função *DefineInput_Artifact_Connection_Activity*

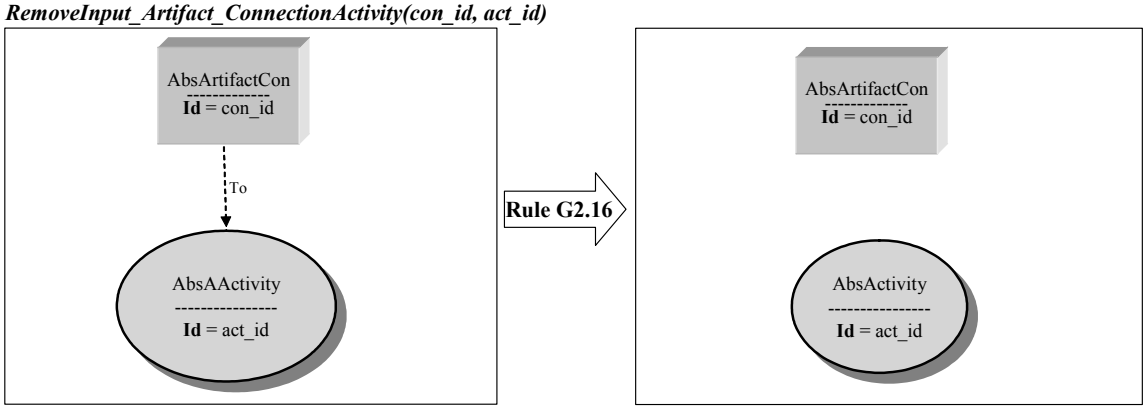

FIGURA A 30 - Regra G2.16 - função *RemoveInput_Artifact_Connection_Activity*

A regra G2.17 descreve a ligação de uma conexão de artefato à entrada de uma conexão múltipla (*Branch* ou *Join*).

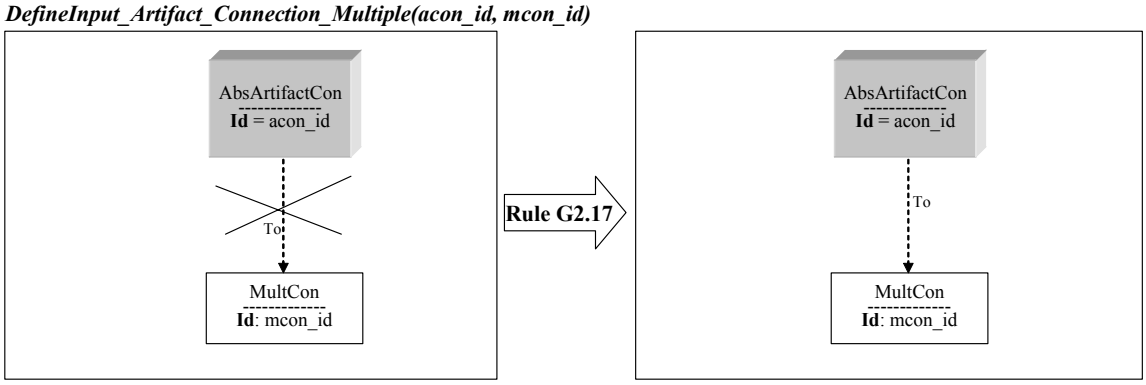

FIGURA A 31 - Regra G2.17 - função *DefineInput_Artifact_Connection_Multiple*

A consistência das conexões múltiplas é garantida pela função *Artifact Connection Consistency*, que não possui parâmetros, sendo ativada sempre que as condições expressas do lado esquerdo são alcançadas.

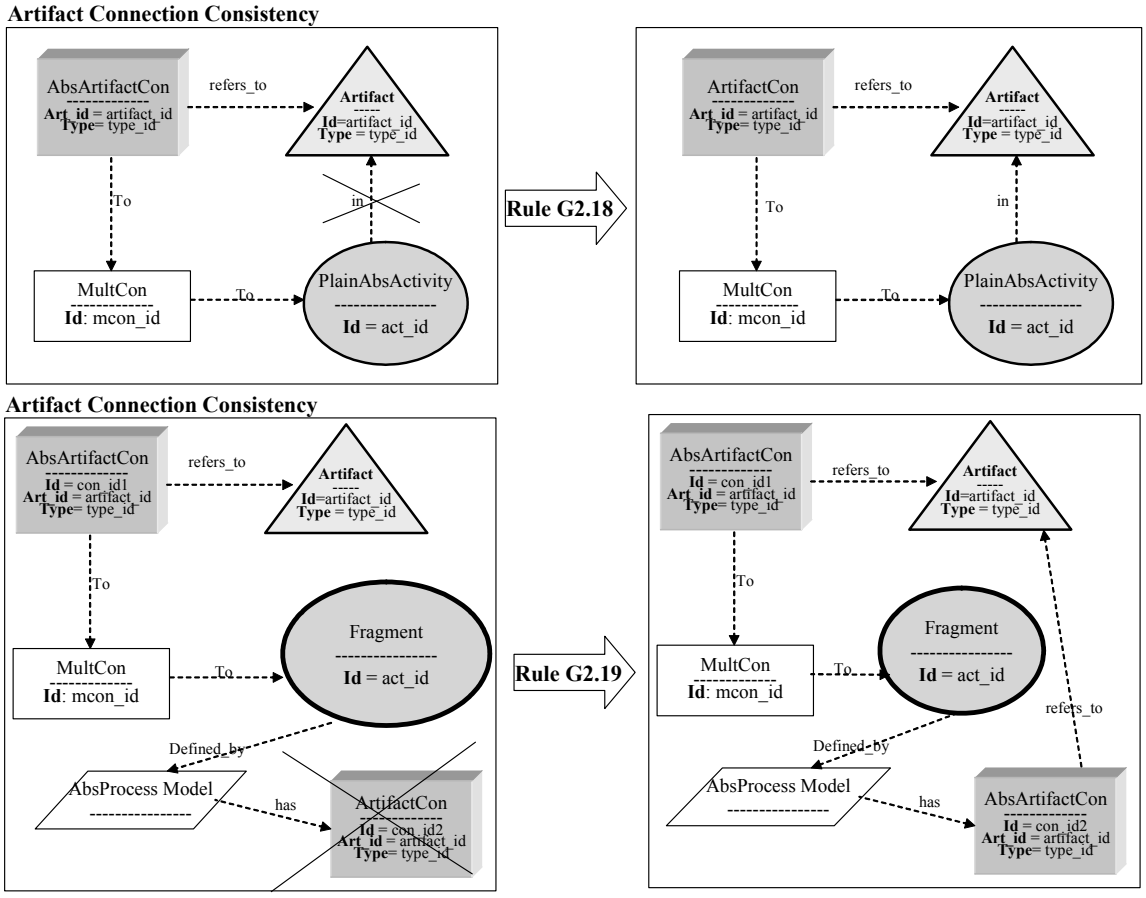

FIGURA A 32 - Regras G2.18 e G2.19 - função *Artifact Connection Consistency*

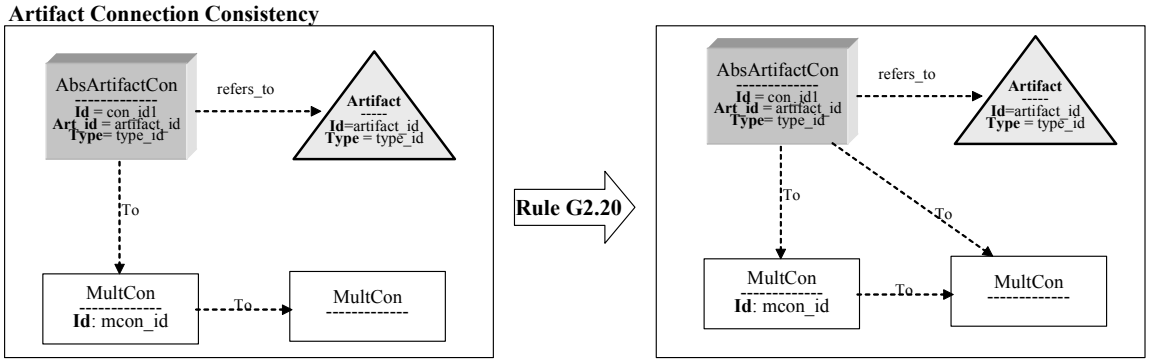

FIGURA A 33 - Regra G2.20 - função *Artifact Connection Consistency*

A regra G2.17 é complementada pela regra G2.21, descrita na figura A 34.

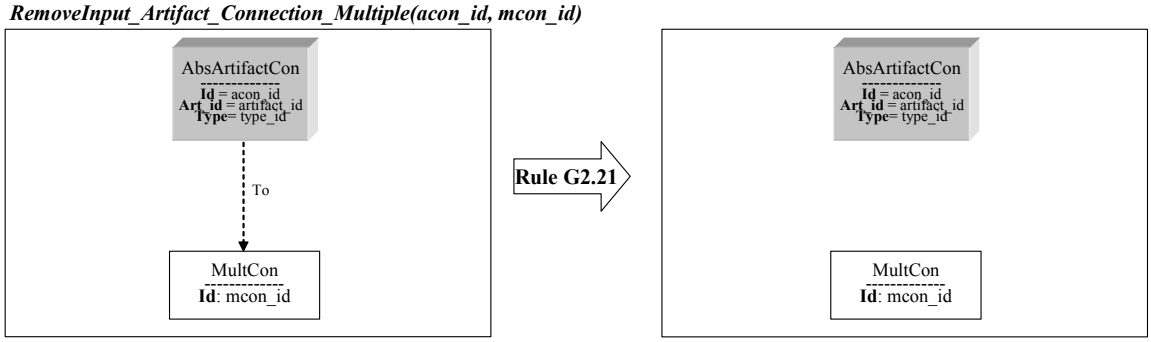

FIGURA A 34 - Regra G2.21 - função *RemoveInput_Artifact_Connection_Multiple*

A regra G2.22 define de forma genérica a remoção de conexões em uma instância de *AbsProcessModel* (figura A 35).

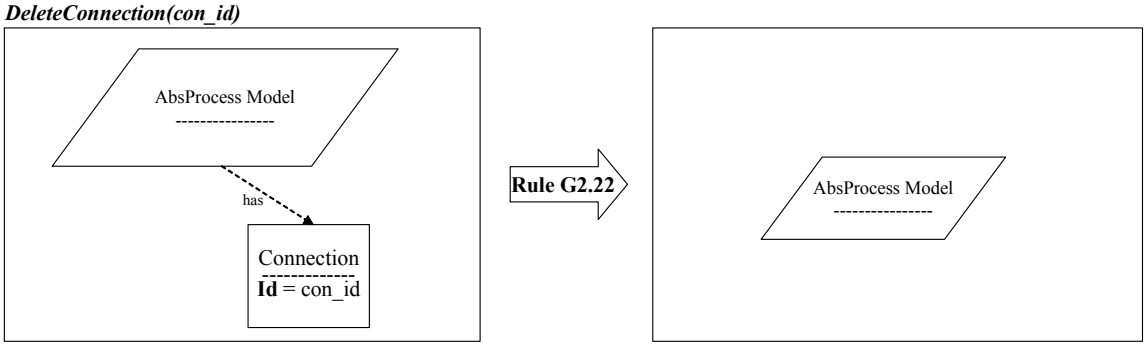

FIGURA A 35 - Regra G2.22 - função *DeleteConnection*

A5.2.5 Regras G2.23 a 2.26 - Lidando com artefatos concretos

O grupo de regras numeradas de G2.5 a 2.8 lidam com a instanciação de artefatos abstratos. Artefatos concretos devem ser tratados de uma forma especial: em virtude de estes constituírem elementos que não podem ser modificados durante a execução de um processo, não deve ser permitido definir como instâncias de conexões de artefato de saída de uma atividade sejam associadas à artefatos concretos.

Assim, as regras numeradas de G2.23 a G2.26, apresentadas nas figuras a seguir descrevem o comportamento requerido. Como pode-se observar pelas figuras, as regras aqui apresentadas incluem como condição negativa do lado esquerdo das regras a

impossibilidade da conexão de artefato estar sendo produzida (arco *from*) por uma atividade qualquer (nodo *AbsActivity*).

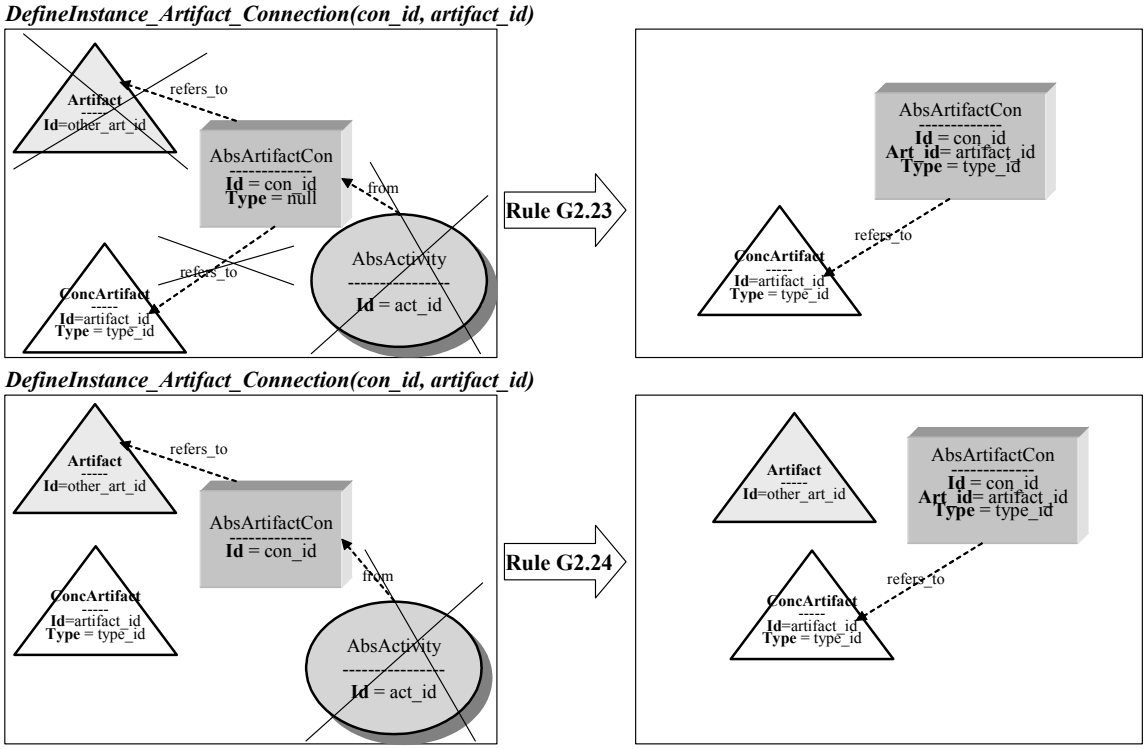

FIGURA A 36 Regras G2.23 e G.24 - função *DefineInstance_Artifact_Connection*

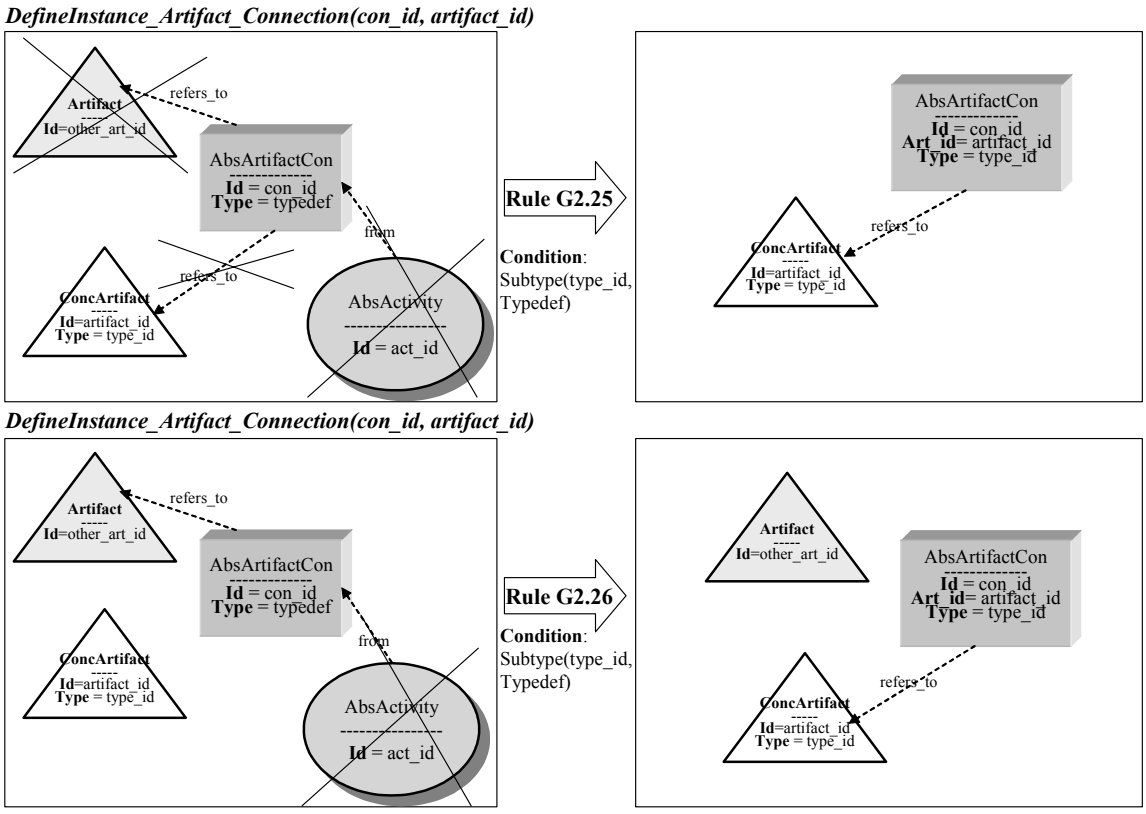

FIGURA A 37 - Regras G2.25 e G2.26 - função *DefineInstance_Artifact_Connection*

A5.3 Regras para Manipulação e Ordenação Temporal de Atividades

A interconexão de atividades descreve um importante elemento usado na modelagem de processos de software. No modelo de *Templates*, é especialmente útil restringir descrições que sejam válidas, isto é, possam ser derivadas para processos executáveis.

A regra G3.1 (figua A 38) descreve a adição de uma nova conexão simples entre duas atividades. Nesse caso, a função *AddSimpleConnection* requer três: dois identificadores de atividades (origem e destino) e um string com a dependência requerida. Como descrito na figura A 38, uma conexão simples só pode ser inserida se não houver qualquer outro fluxo de controle (direto ou indireto) entre as atividades envolvidas (*control flow*).

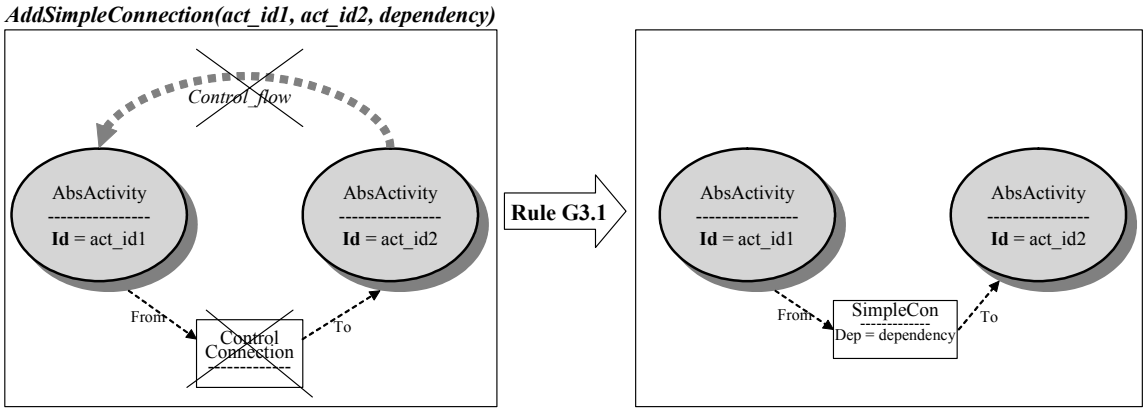

FIGURA A 38 - Regra G3.1 - função *AddSimpleConnection*

A inclusão de uma conexão de *feedback*, tal como descrito pela função *AddFeedBack*, por outro lado, exige que exista um fluxo de controle entre a atividade de origem e a de destino. A regra correspondente está definida na figura A 39.

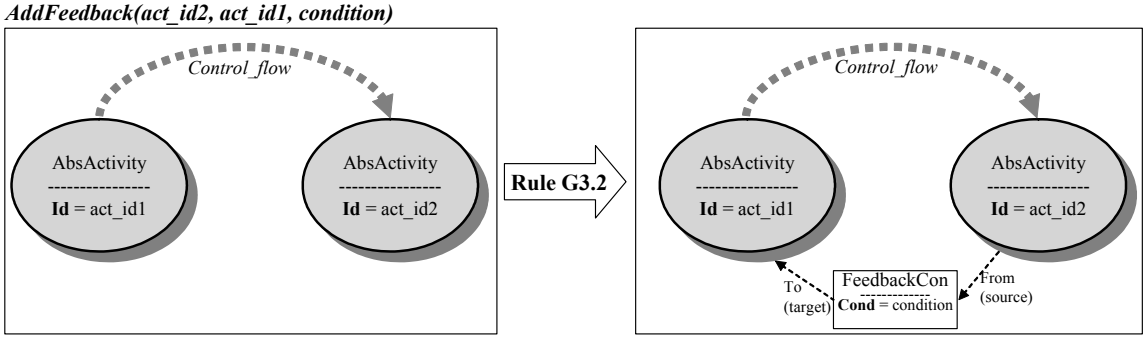

FIGURA A 39 - Regra G3.2 - função *addFeedback*

A figura A 40 descreve seis regras envolvidas com a inclusão dos diferentes tipos de conectores para conexões múltiplas (i.e., *Join And, Join Or, Join XOR, Branch And, Branch Or* e *Branch XOR*).

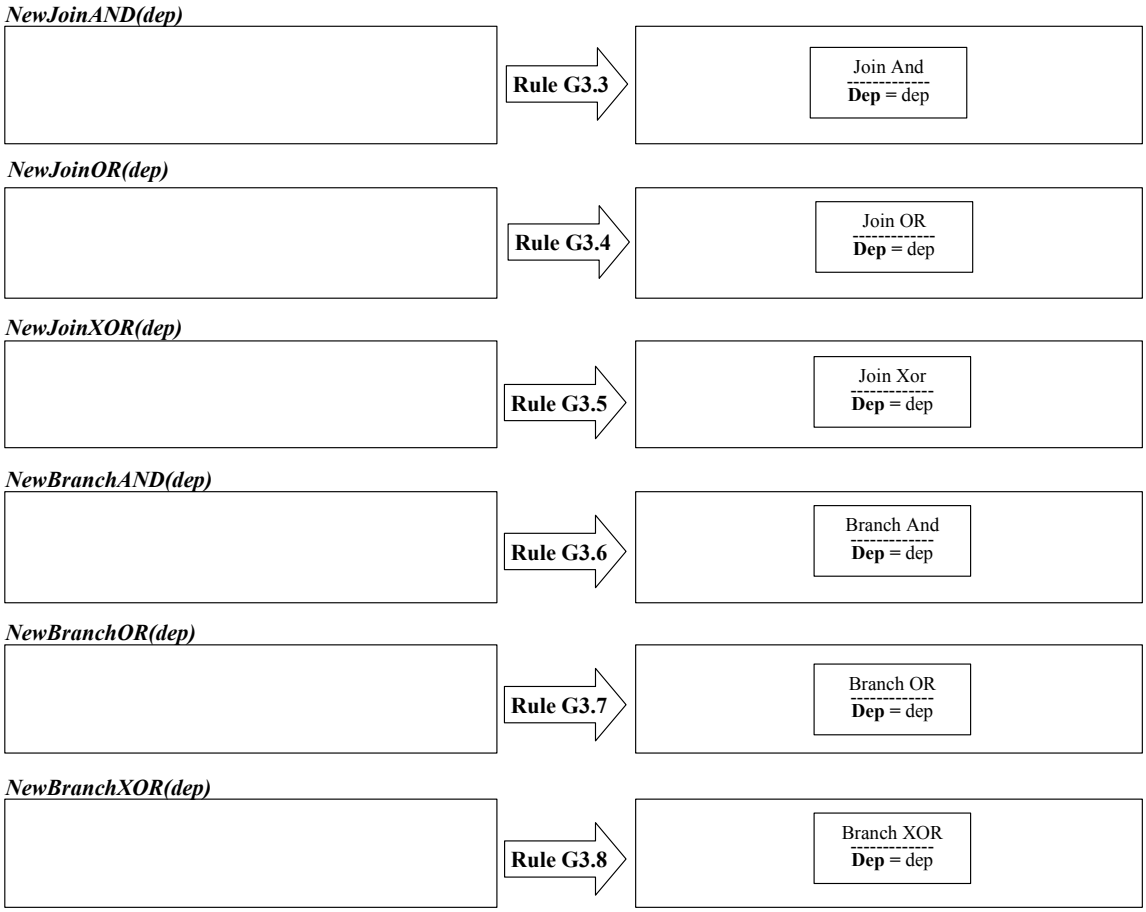

FIGURA A 40 - Regras G3.3 a G3.8 - funções para inclusão de conexões múltiplas

A figura A 41 apresenta as funções responsáveis pela remoção de conectores do tipo *MultiCon* em um *template* de processo.

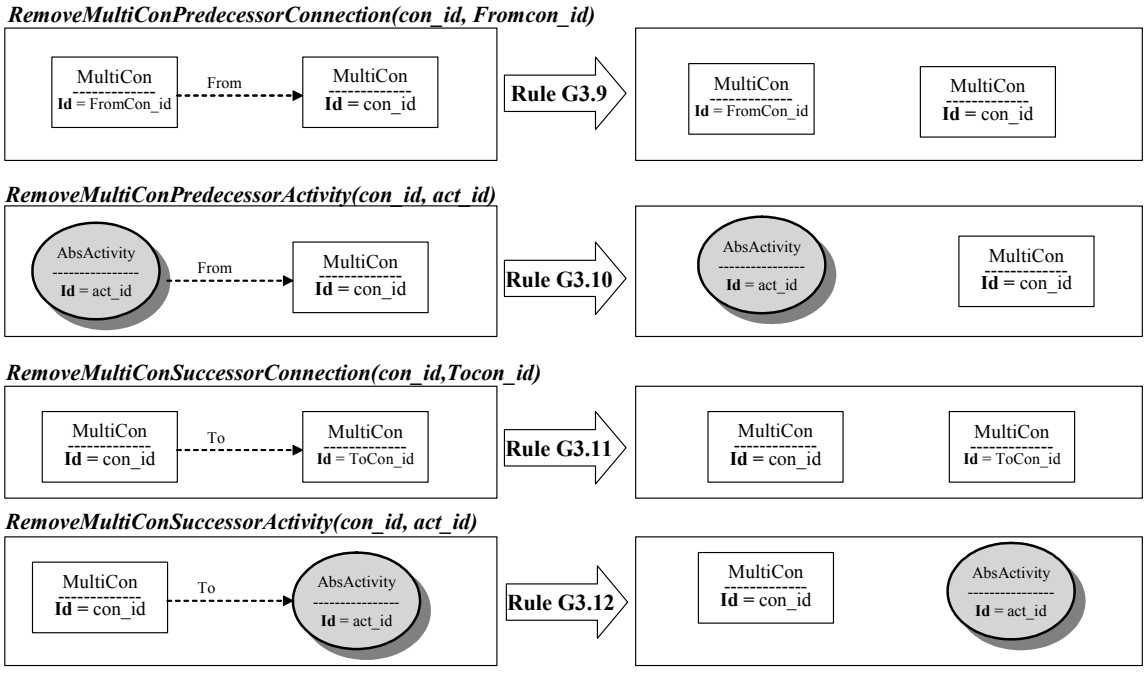

FIGURA A 41 - Regras G3.9 a G3.12 - funções para remoção de conexões múltiplas

A5.4 Regras envolvendo as conexões *Join*

As conexões *Join* estabelecem a relação entre vários elementos de entrada e um de saída, aonde tais elementos podem ser atividades ou conexões. Portanto, as regras desenhadas nesta seção respeitam esta restrição.

A função *DefineJoinTO_Activity* - descrita na figura A 42 - estabelece como destino de um *Join* uma atividade (tipo genérico *AbsActivity*).

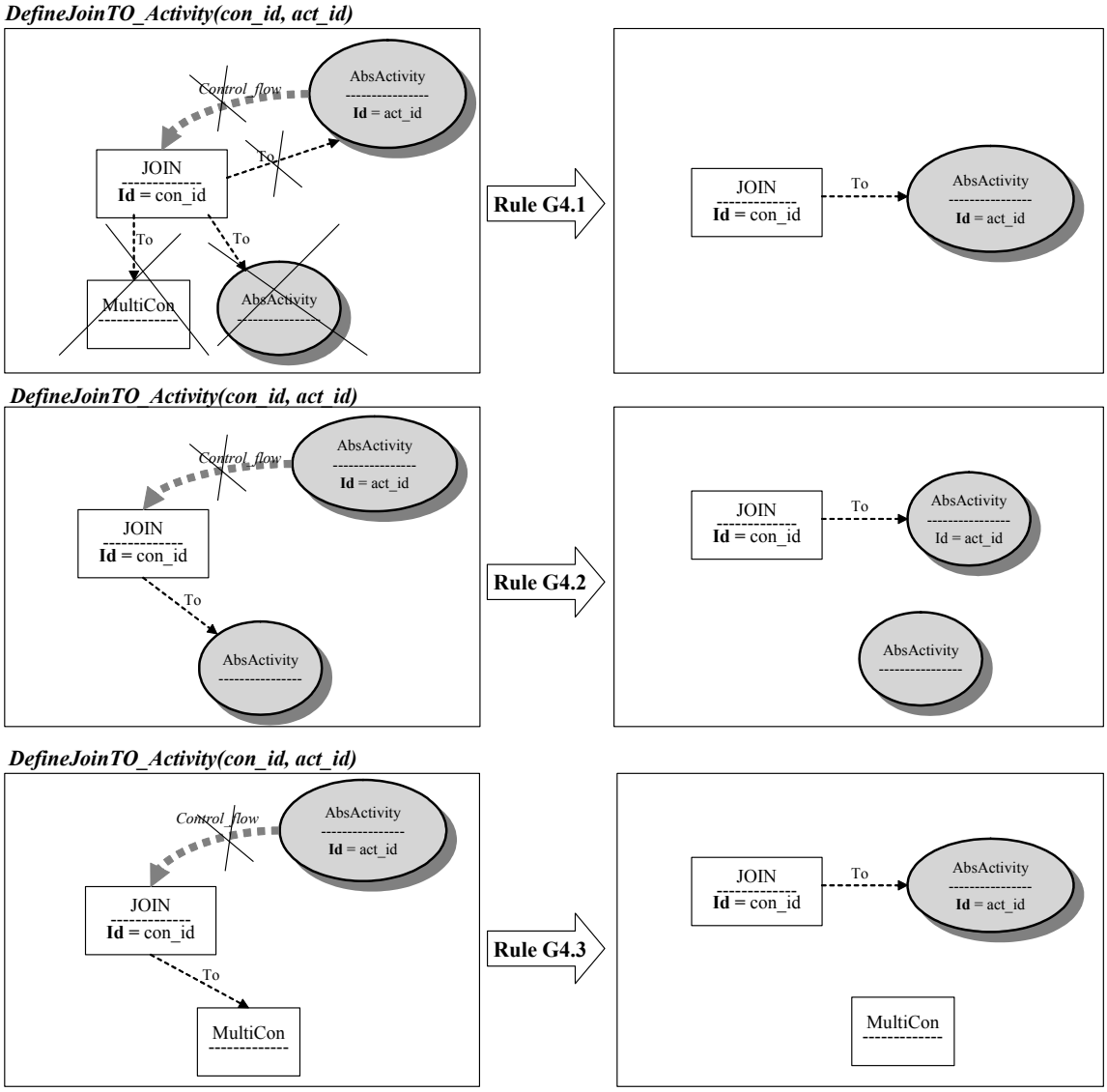

FIGURA A 42 - Regras G4.1, G4.2 e G4.3 - função *DefineJoinTO_Activity*

A regra G4.4 restringe a definição do destino de um *Join* para uma nova conexão (figura A 43).

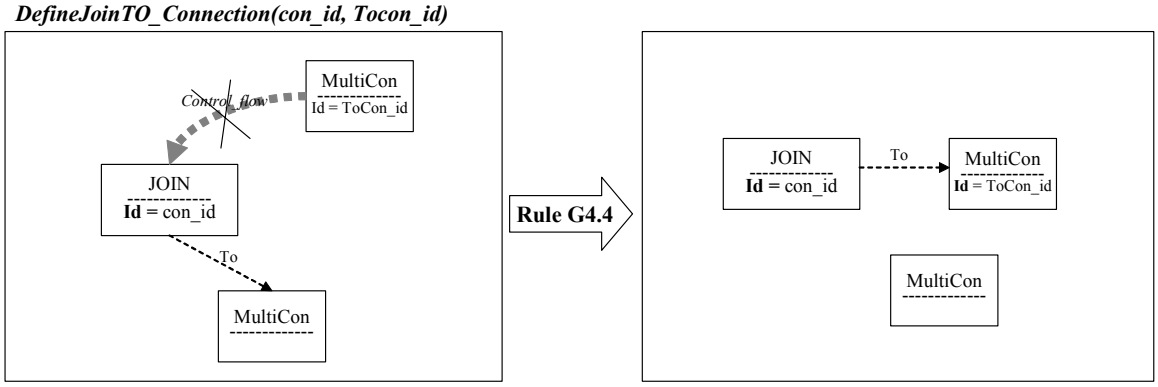

FIGURA A 43 - Regra G4.4 - função *DefineJoinTO_Connection*

As regras G4.5, G4.6, G4.7 e G4.7 descritas nas figuras a seguir estabelecem a adição de elementos origem (*from*) para um *Join*.

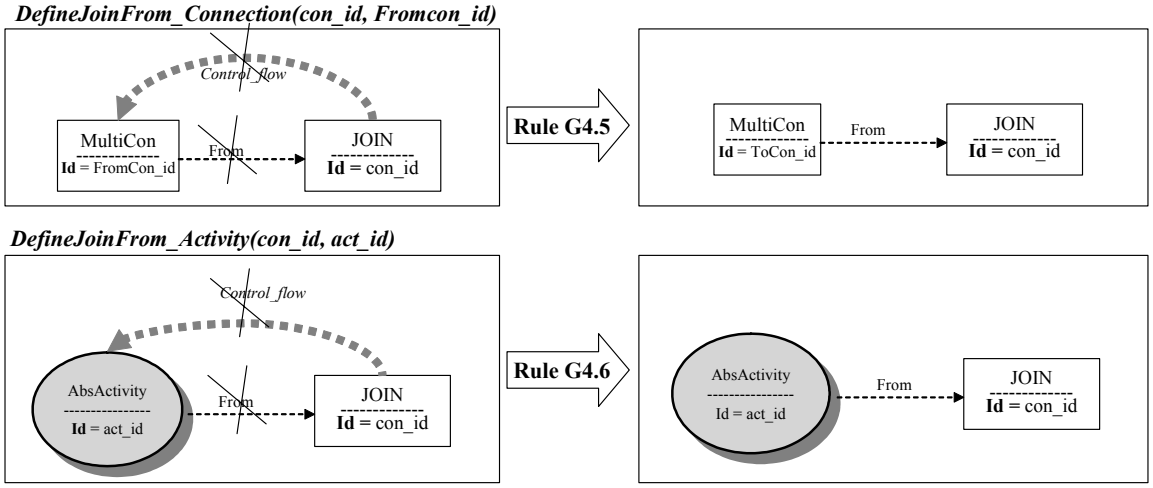

FIGURA A 44 - Regras G4.5 e G4.6 - funções *DefineJoinFrom_Connection* e *DefineJoinFrom_Activity*

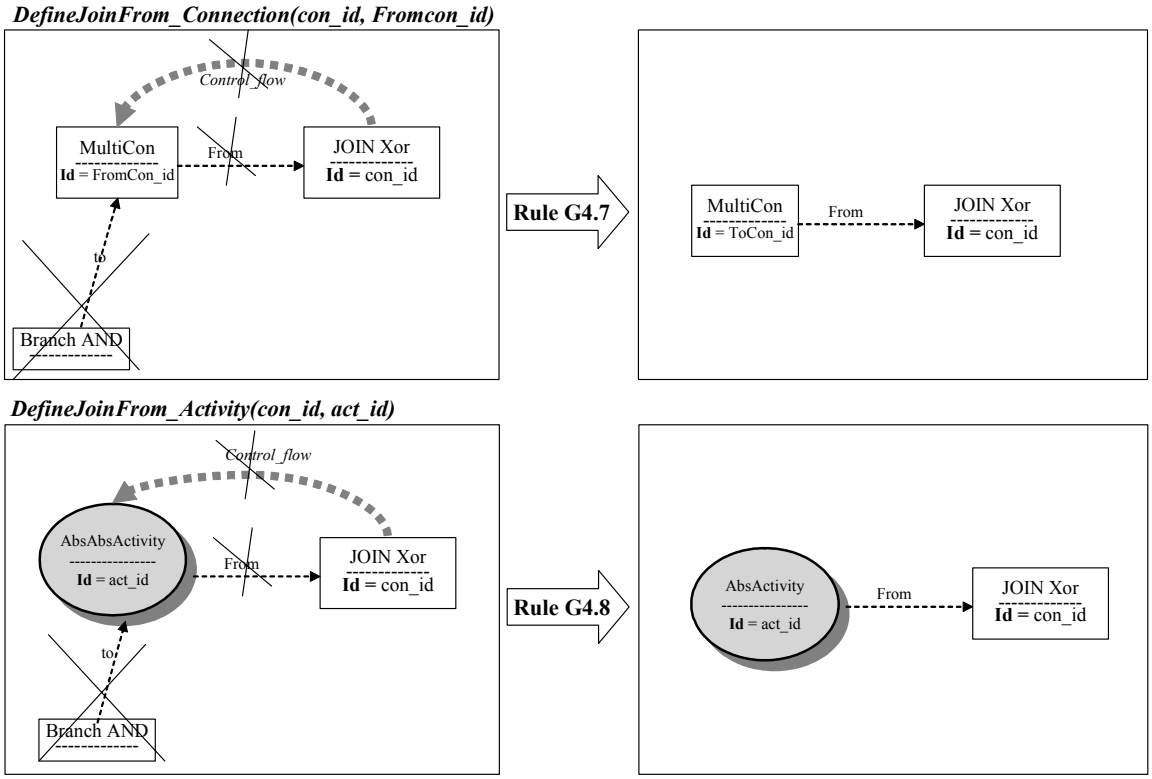

FIGURA A 45 - Regras G4.7 e G4.8 - funções *DefineJoinFrom_Connection* e *DefineJoinFrom_Activity*

A5.5 Regras envolvendo as conexões *Branch*

Assim como as conexões *Join*, cujas regras foram descritas na seção anterior, existe um conjunto de regras para definir a consistência de conexões *Branch*. Uma conexão *Branch* possui uma origem que aponta para um ou mais destinos, aonde tanto a origem quando os destinos podem ser atividades ou conexões múltiplas.

A primeira função a ser apresentada é a *DefineBranchFrom_Activity*, a qual descreve que a conexão *Branch* possuirá como origem uma atividade. Na regra G5.1, tanto a atividade quanto a conexão já estão inseridas no template, sendo que não existem outras atividades ou conexões como origem do *Branch* nem tampouco existe um fluxo de controle entre o *Branch* e a atividade origem candidata (para não formar um ciclo no modelo). Na regra G5.2 uma atividade já está associada como origem do *Branch*. Assim, a associação *from* é movida para a nova atividade identificada por *act_id*. Finalmente, o terceiro caso (regra G5.3) também trata da substituição da origem do *Branch*, sendo que nesse caso a origem previamente definida é uma conexão múltipla.

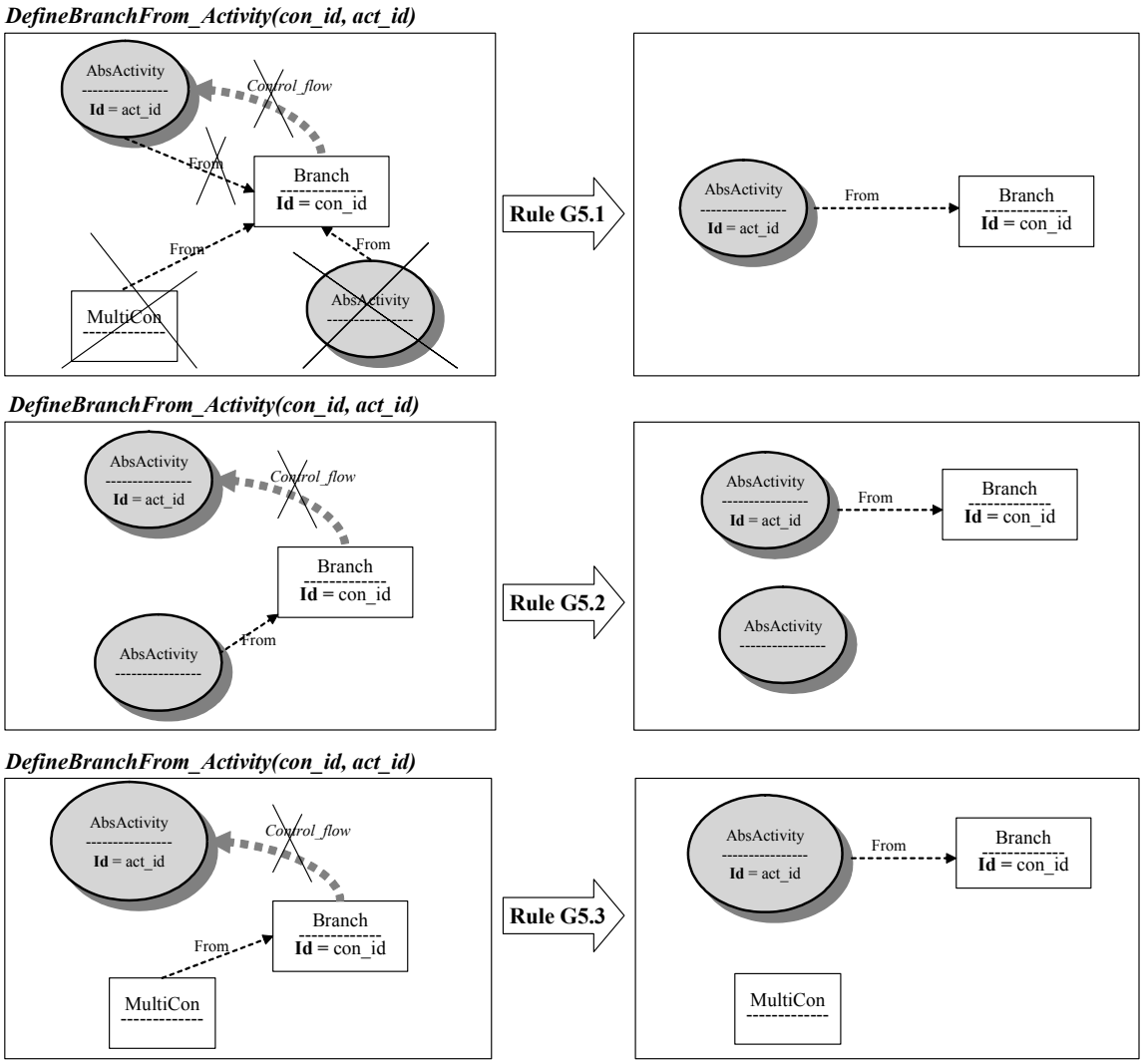

FIGURA A 46 - Regras G5.1, G5.2 e G5.2 - função *DefineBranchFrom_Activity*

A função *DefineBranchFrom_Connection* é análoga à *DefineBranchFrom_Activity*, conforme ilustrado pelas regras G5.4, G5.5 e G5.6 na figura A 47 a seguir.

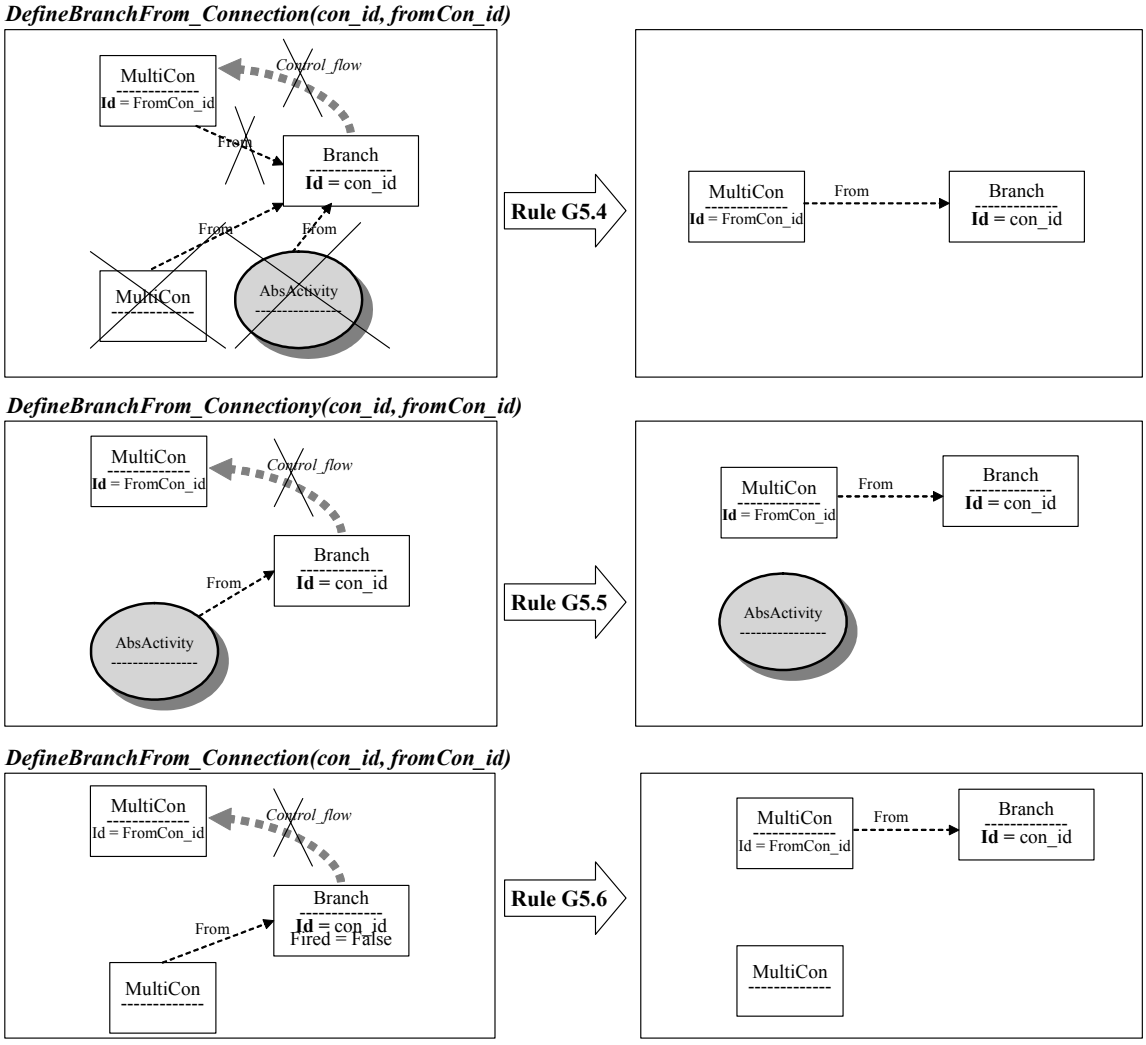

FIGURA A 47 - Regras G5.4, G5.5 e G5.6 - função *DefineBranchFrom_Connection*

A definição do destino de um *Branch* é descrita pelas funções *DefineBranchTo_Activity* e *DefineBranchTo_Connection* a seguir. No caso de um *Branch* condicional (*branch cond*) um objeto *Condition* também é requerido.

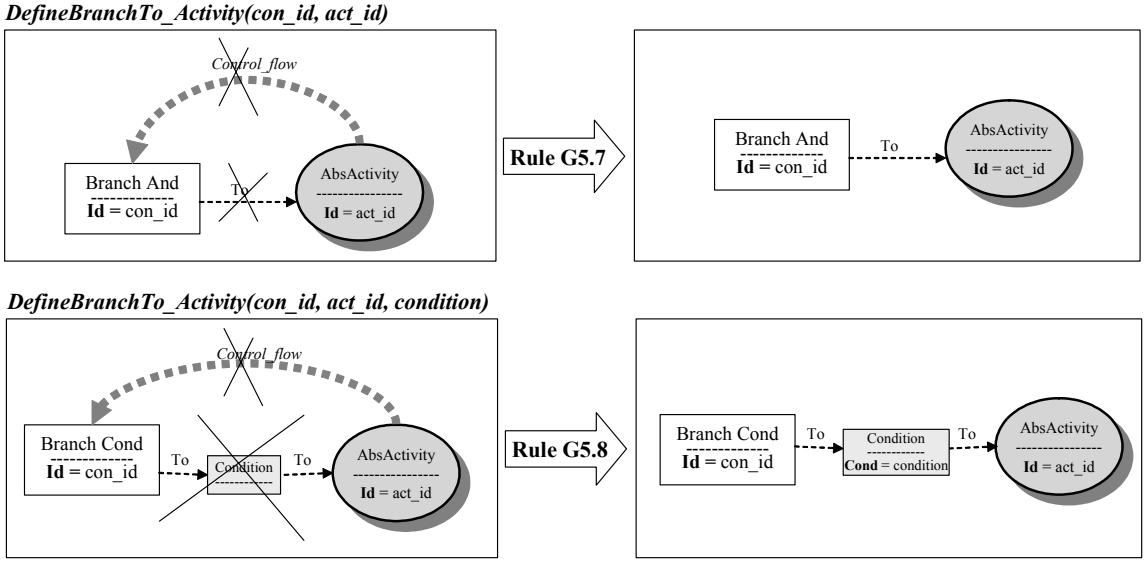

FIGURA A 48 - Regras G5.7 e G5.8 - função *DefineBranchTo_Activity*

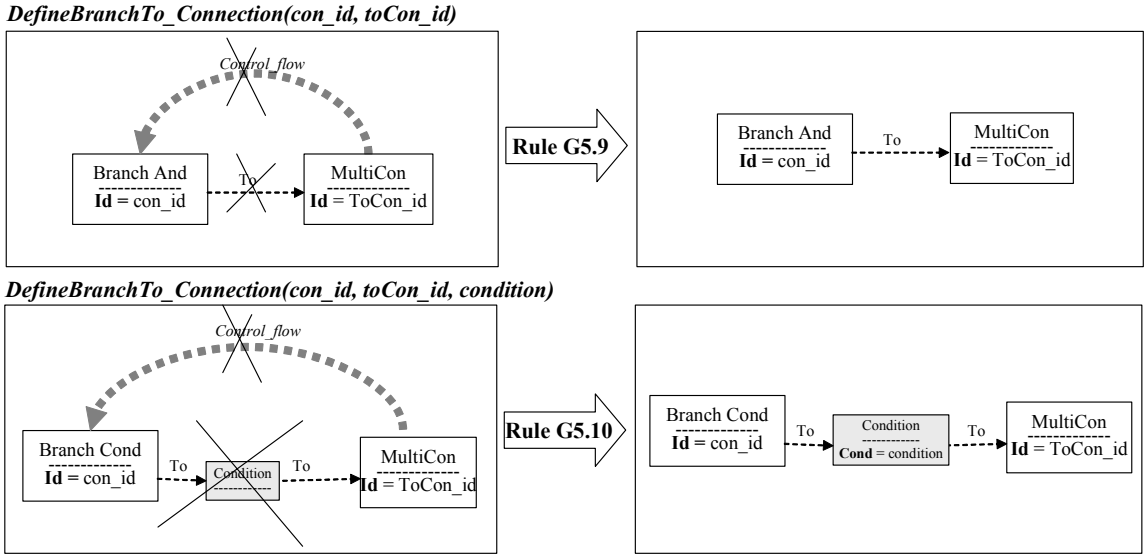

FIGURA A 49 - Regras G5.9 e G5.10 - função *DefineBranchTo_Connection*

A remoção de uma condição na associação entre um *Branch* e uma conexão múltipla e entre um *Branch* e uma atividade é descrita pelas regras G5.11 e G5.12 na figura A 50.

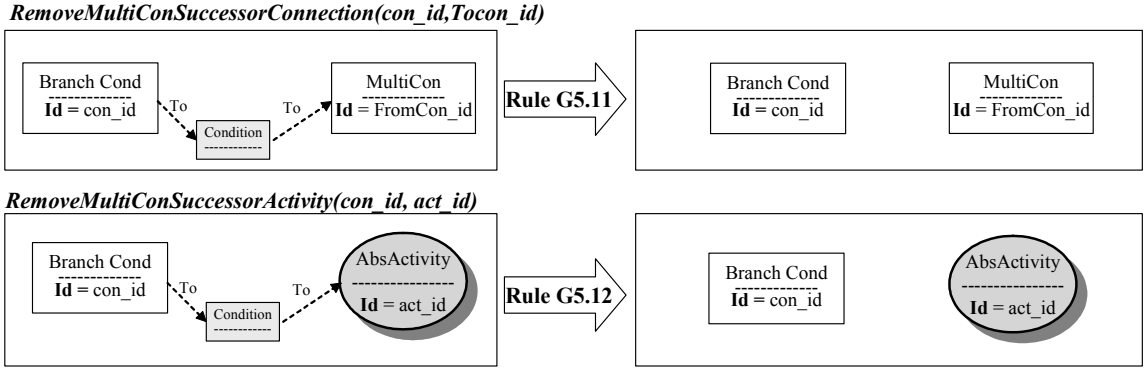

FIGURA A 50 - Regras G5.11 e G5.12 - funções *RemoveMultiConSuccessorConnection* e *RemoveMultiConSucessorActivity*

A5.6 Regras envolvendo recursos, cargos e grupos

As regras que envolvem a inclusão (funções com prefixo *add*) e remoção (*remove*) de recursos, cargos e grupos para atividades normais (*NormalAbsDesc*) são descritas a seguir.

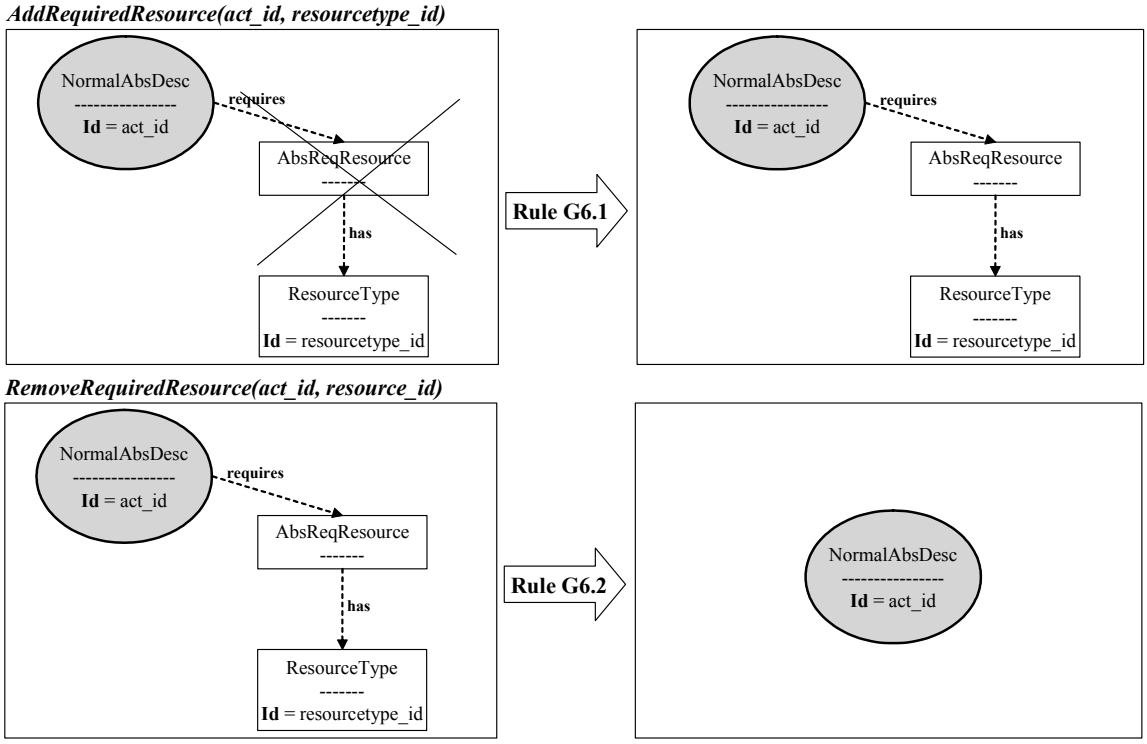

FIGURA A 51 - Regras G6.1 e G6.2 - funções *AddRequiredResource* e *RemoveRequiredResource*

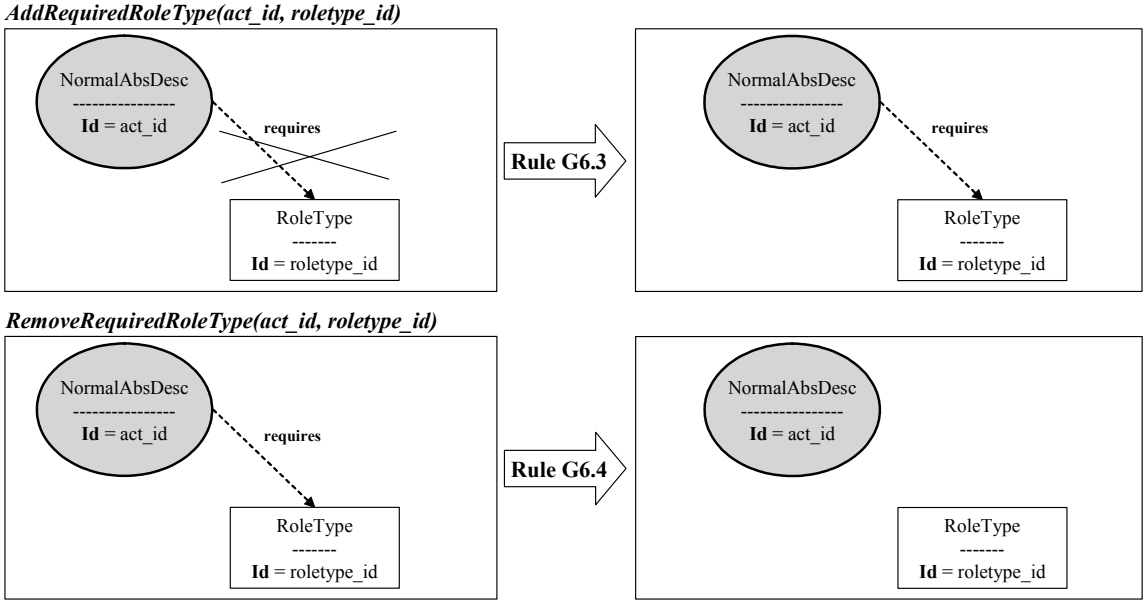

FIGURA A 52 - Regras G6.3 e G6.4 - funções *AddRequiredRoleType* e *RemoveRequiredRoleType*

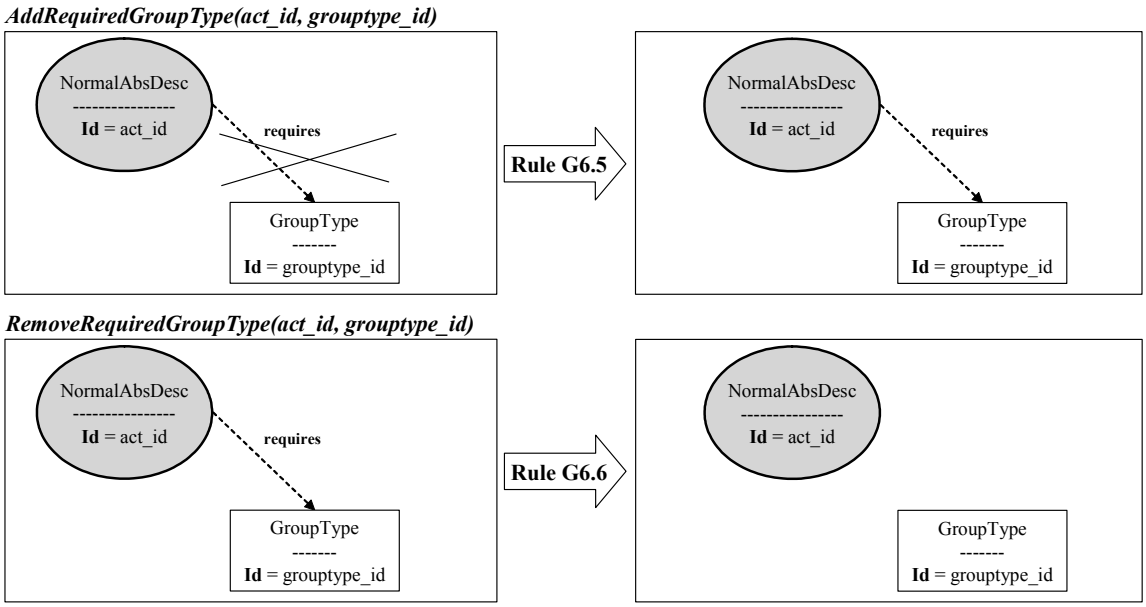

FIGURA A 53 - Regras G6.5 e G6.6 - funções *AddRequiredGroupType* e *RemoveRequiredGroupType*

A5.7 Regras envolvendo Políticas

Durante a modelagem de um *template*, um usuário pode habilitar ou desabilitar políticas nas atividades, nos componentes de uma atividade (i.e., recursos, ferramentas e agentes), ou em um *template* completo. Esta seção descreve algumas das regras necessárias para a edição dos *templates* sob o ponto de vista da habilitação das políticas.

A figura A 54 descreve as funções *Enable/DisablePolicyInstanceActivity*. A regra G7.1 define que uma política para ser habilitada em uma atividade (objeto de *AbsActivity*), a Política identificada por *policy-id* não pode estar habilitada, e deve obrigatoriamente possuir como interface um objeto do tipo *Activity*. As regras apresentadas na figura A 55 fazem um papel análogo para os *templates.*

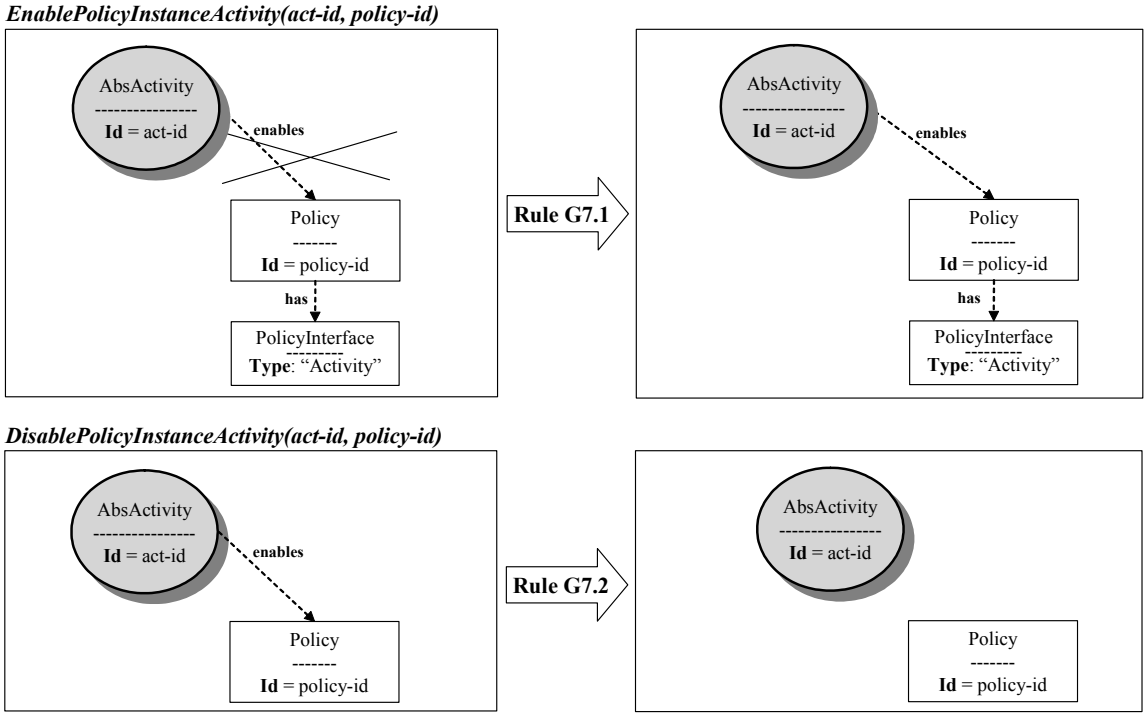

FIGURA A 54 - Regras G7.1 e G7.2 - funções *EnablePolicyInstanceActivity* e *DisablePolicyInstanceActivity*

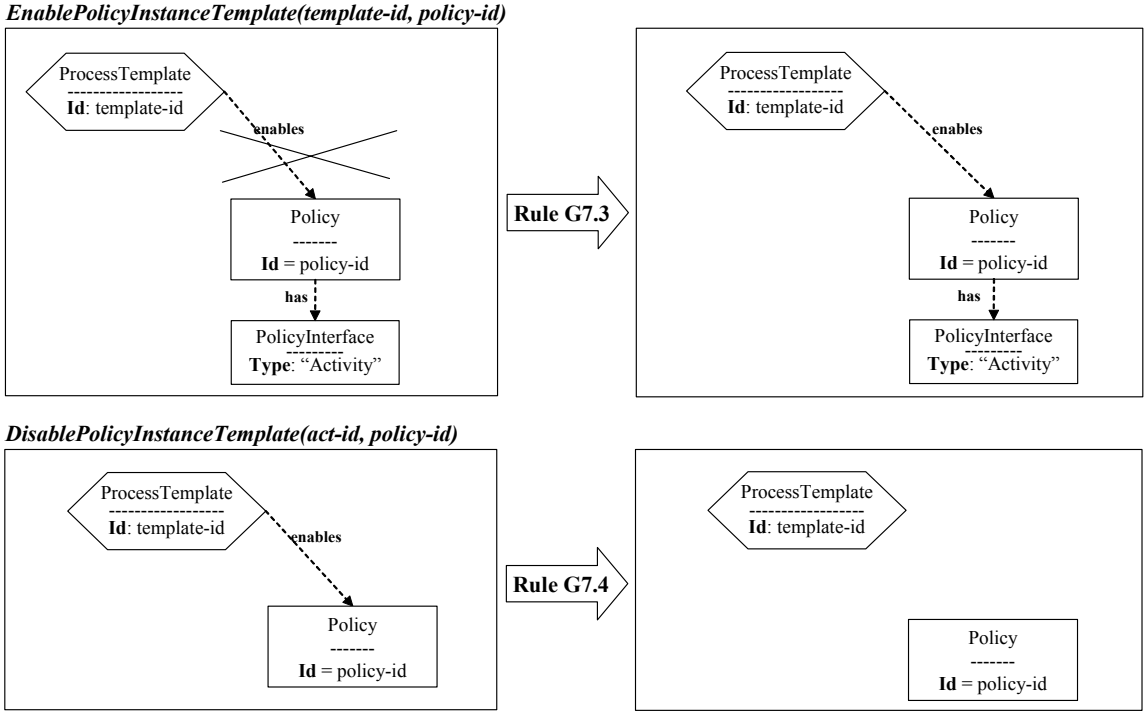

FIGURA A 55 - Regras G7.3 e G7.4 - funções *EnablePolicyInstanceTemplate* e *DisablePolicyInstanceTemplate*

Anexo 6 Regras para detectar o fluxo de controle

Esse anexo inclui as funções utilizadas para detectar o fluxo de controle entre atividades de um processo. Portanto, as regras descritas a seguir foram originalmente desenvolvidas por Lima Reis (em [LIM2002d]) e foram adaptadas para o tipo *AbsActivity* aqui apresentado. No modelo *APSEE*, o teste de fluxo de controle é de extrema utilidade na detecção e prevenção de ciclos em modelos de processos abstratos (*templates*), instanciados ou executáveis.

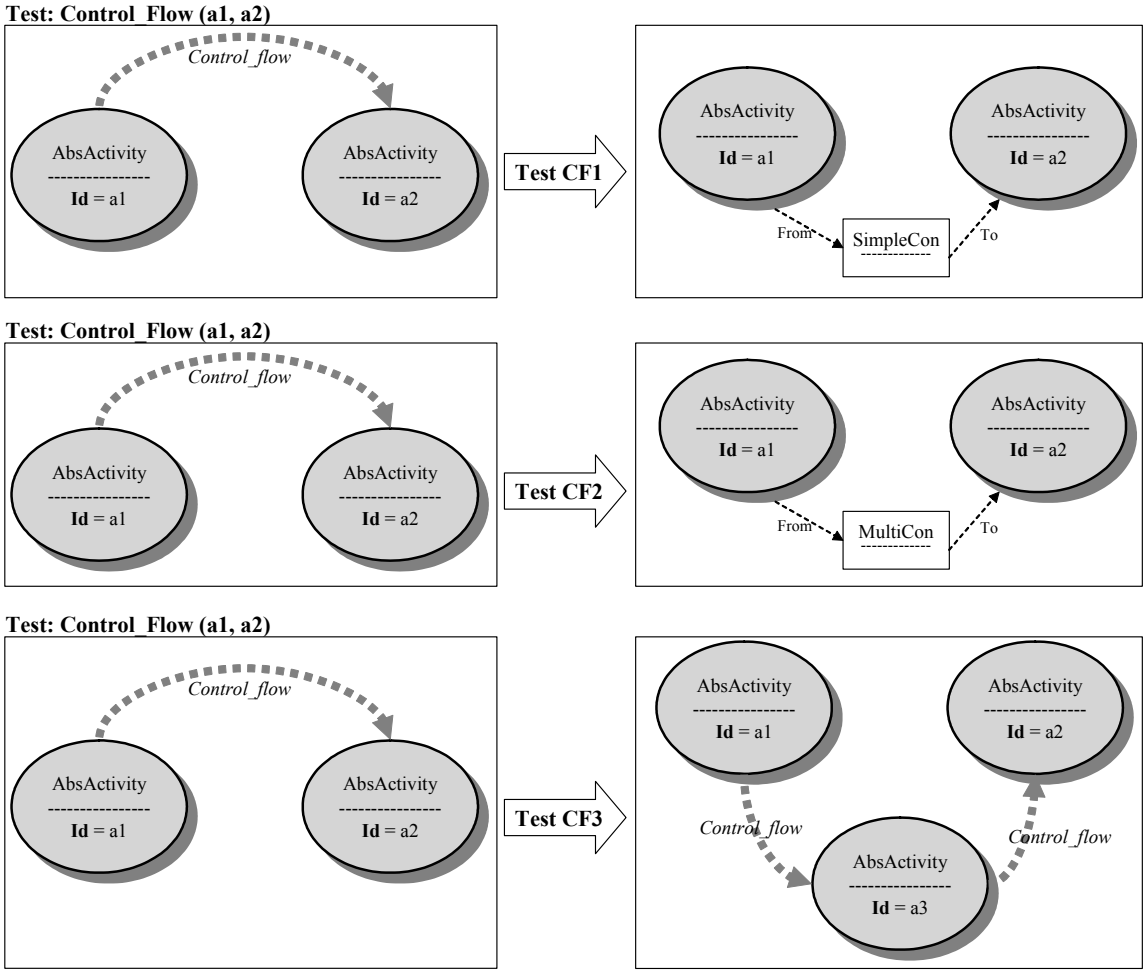

FIGURA A 56 - Regras Test CF1, CF2 e CF3

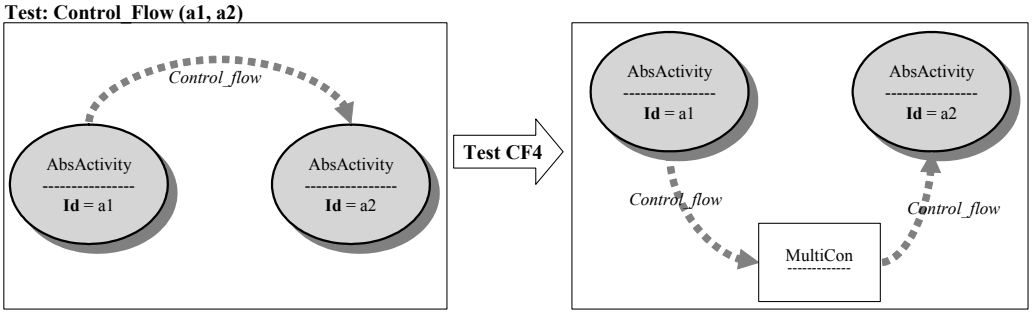

FIGURA A 57 - Regra Test CF4

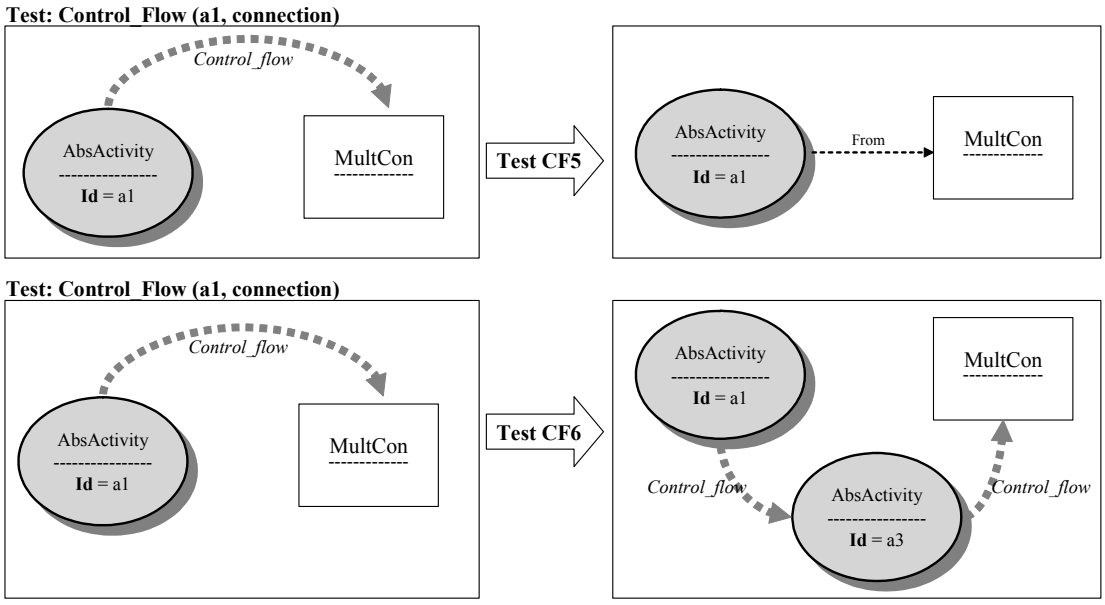

FIGURA A 58 - Regras Test CF5 e CF6

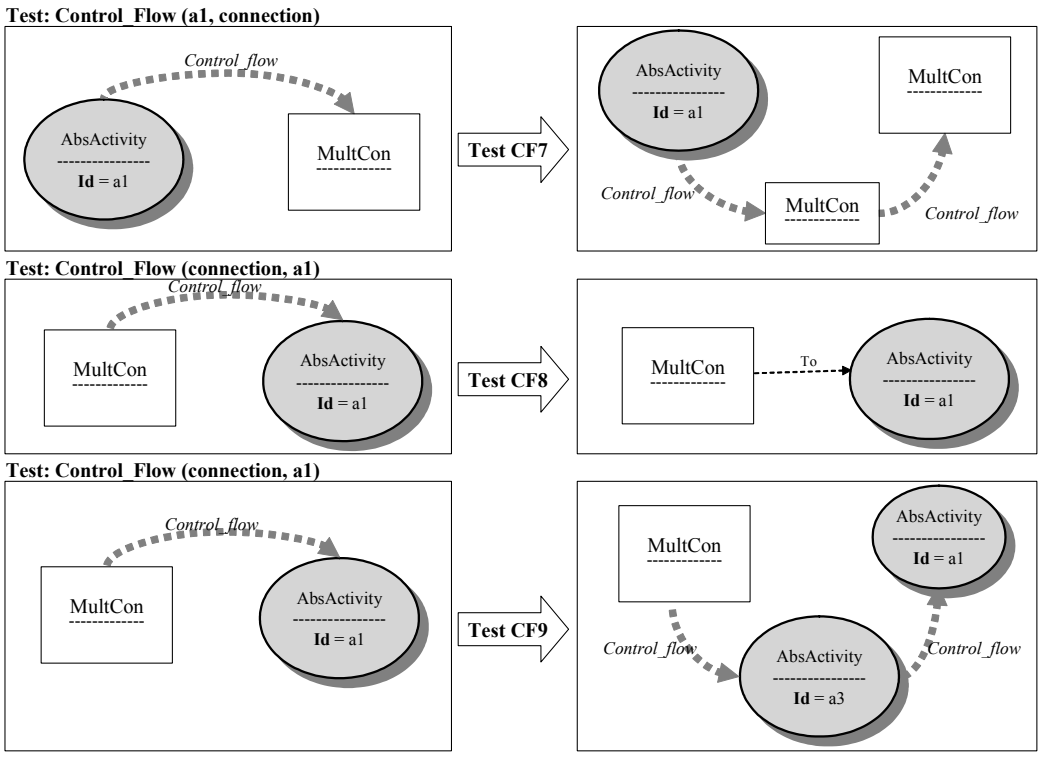

FIGURA A 59 - Regra Test CF7, CF8 e CF9

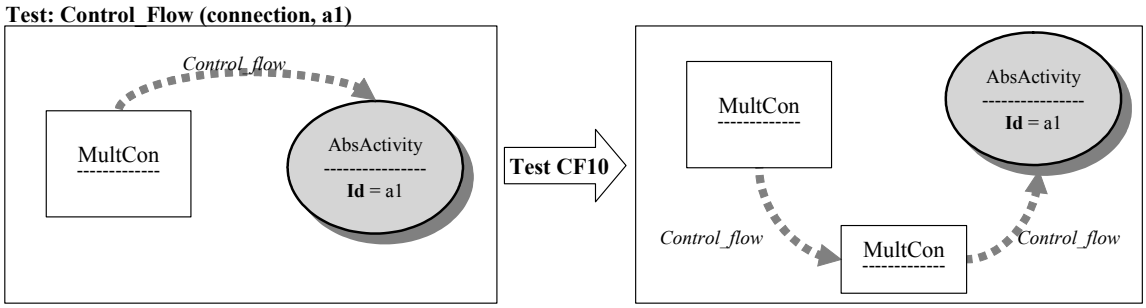

FIGURA A 60 - Regra Test CF10

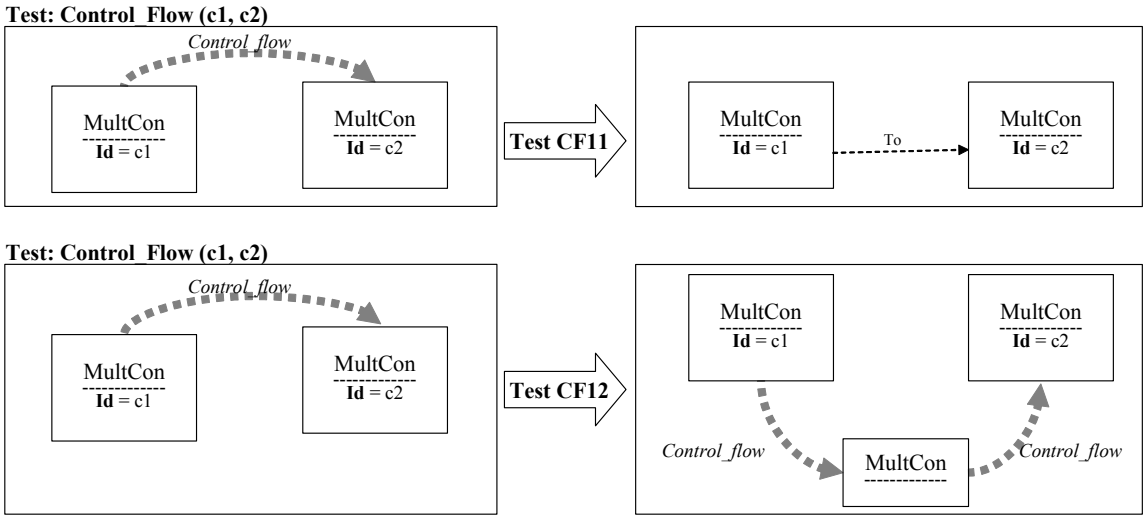

FIGURA A 61 - Regras Test CF11 e CF12

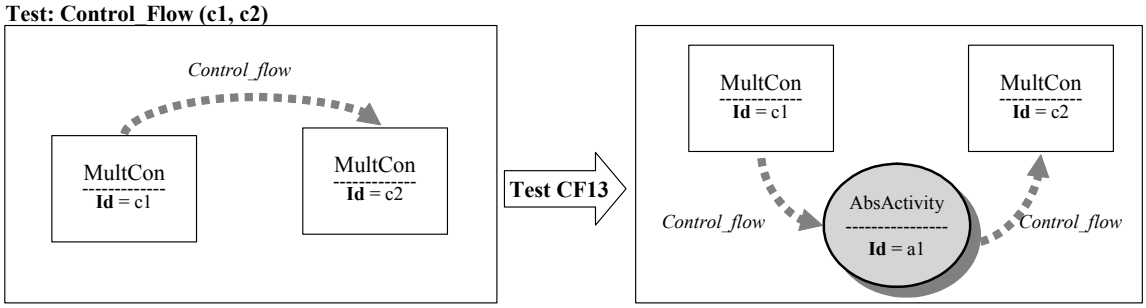

FIGURA A 62 - Regra Test CF13

Anexo 7 Regras para Adaptação de *Templates*

A7.1 Grafo tipo

O grafo tipo que define as regras para a execução de processos foi definido por Lima Reis em [LIM2002d]. Uma versão resumida do grafo tipo, apresentando os principais componentes do meta-modelo envolvidos com a definição de processos executáveis é reproduzido na figura A 63. Esse grafo tipo é bastante similar aquele apresentado no Anexo 5 (figura A 11), possuindo algumas especificidades como descrito a seguir:

- Muitos dos componentes descritos são versões concretas para os componentes abstratos usados na definição de *templates*. Por exemplo, o nodo *Activity* e seus subtipos são correspondentes concretos para o tipo *AbsActivity* descrito no Anexo 5;
- Os componentes concretos possuem atributos adicionais relacionados com aspectos específicos da execução do processo. Nas regras aqui apresentadas, vale ressaltar o atributo *A-st* (que armazena um string com o estado corrente de uma atividade), *Pm-st* (o estado para um modelo de processo) e *P-st* (o estado corrente para o processo de software);

O grafo tipo da figura A 63 inclui diversos componentes que não são usados nas regras aqui descritas, sendo que tais componentes foram mantidos nesse documento apenas para manter consistência com a documentação original em [LIM2002d]. Tais nodos incluem detalhes específicos da execução de processos, o que não é de interesse desse trabalho tais como: *feedback sign*, *redo sign*, *end feedback sign*, *block sign*, *Events*, *ActVersion*, *APSEEManager* e *Manager*;

- Em comparação com o grafo tipo de *templates*, um tipo de arco adicional é incluído: o arco sólido representa o fluxo de mensagens entre componentes do sistema APSEE que ocorre em resposta à eventos do sistema.

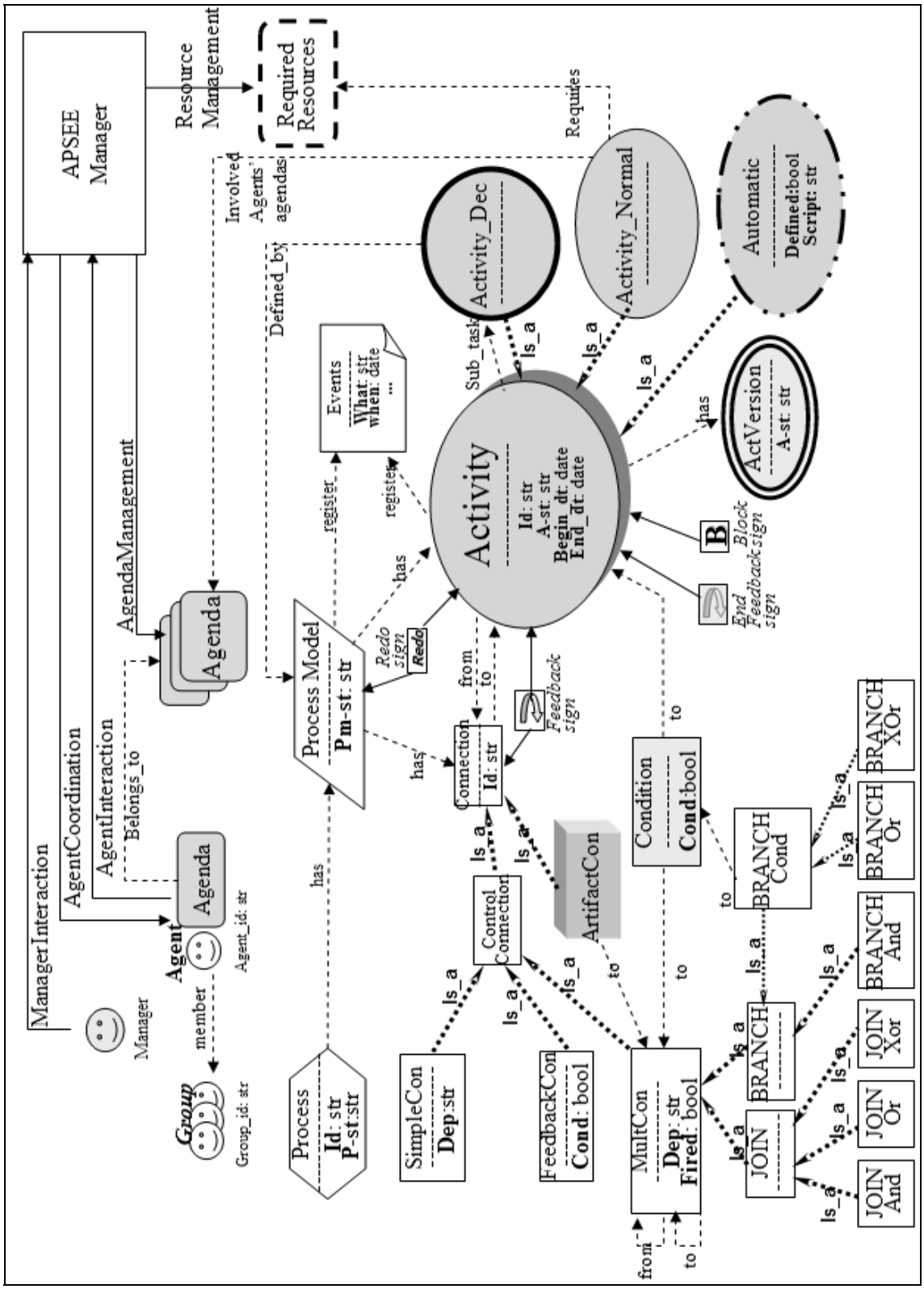

FIGURA A 63 - Grafo tipoo para processos executáveis (adaptado de [LIM2002a])

A7.2 Regras para Adaptação Guiada por Políticas

A adaptação guiada por políticas define que as políticas habilitadas em um *template* não podem ser removidas nos processos derivados. Assim, as funções responsáveis por remover as políticas habilitadas em um processo executável descritas

por Lima Reis em [LIM2002d] tiveram que ser adaptadas para atender esse novo requisito.

Esta seção apresenta exemplos de duas funções que foram modificadas para restringir a adaptação de processos, impedindo a remoção de políticas que estejam habilitadas no *template* original. No exemplo da figura A 64, são mostradas as duas regras desenvolvidas a partir da regra P9.1[50], esta última originalmente proposta em [LIM2002d]. Na regra AdaptRule 9.1, é verificado se o tipo de adaptação adotado é diferente de *Free* (i.e., assume o valor *Restricted* ou *PolicyBased*): se esta condição é verdadeira e a atividade original não habilita a política selecionada para remoção, o objeto é transformado. Na figura A 65 são apresentadas regras adicionais, as quais lidam com políticas habilitadas em processos derivados de *templates*.

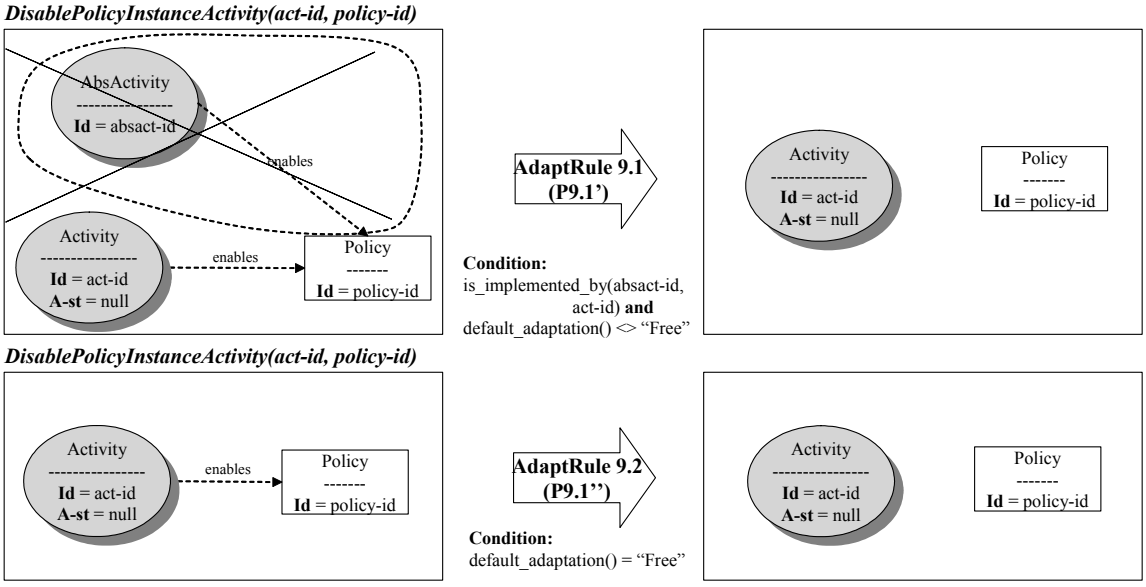

FIGURA A 64 - Regras AdaptRule 9.1 e 9.2 - função *DisablePolicyInstanceActivity*

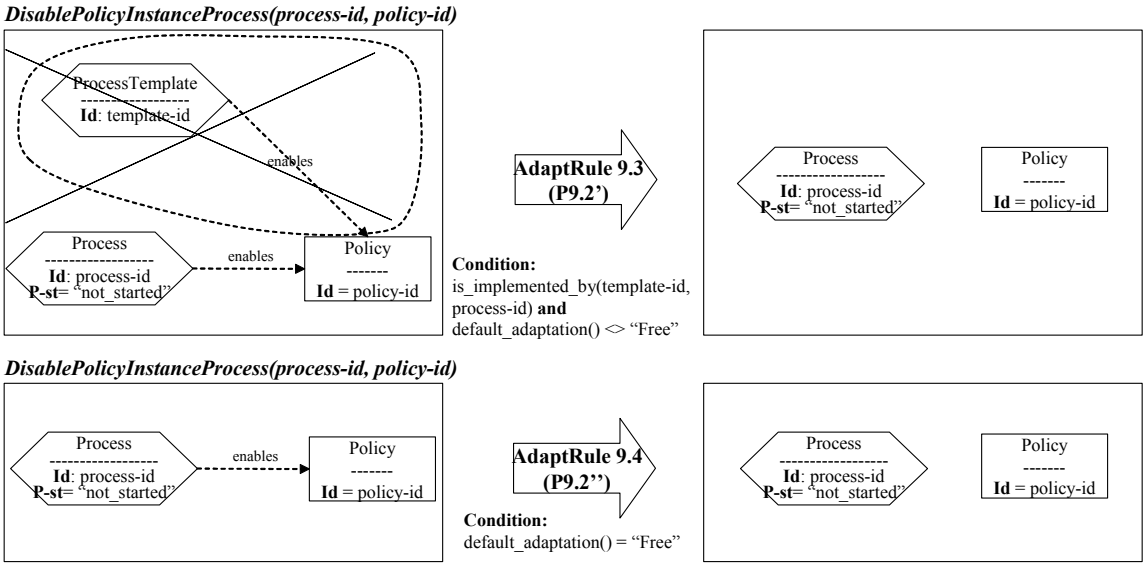

FIGURA A 65 - Regras AdaptRule 9.3 e 9.4 - função *DisablePolicyInstanceProcess*

[50] Neste capítulo, o rótulo da regra original que foi estendida é apresentado entre parênteses abaixo do nome da nova regra.

A7.3 Regras para Adaptação Restrita

A definição algébrica para a semântica da adaptação restrita é fornecida por esta seção. A adaptação restrita engloba as regras definidas na seção anterior (regras para adaptação guiada por políticas) e inclui modificações em diferentes regras que descrevem o funcionamento do editor de processos executáveis, conforme apresentado nas sub-seções a seguir.

A7.3.1 Regras para restringir a remoção de componentes

Segundo a definição da adaptação restrita (seção 3.7.3), os elementos originais existentes em um *template* não podem ser removidos. Assim, esta seção apresenta as funções originalmente propostas em [LIM2002d] para excluir componentes de um modelo de processo executável que foram modificadas para tratar da correspondência com componentes originais descritos no *template* de origem.

As regras foram alteradas através da colocação de novas condições de habilitação (texto abaixo da seta que separa os lados direito e esquerdo de uma regra). Em geral, a condição de habilitação das regras aqui apresentadas é composta pela seguinte expressão lógica:

- Primeiro, é verificado o tipo de adaptação corrente no sistema. Desse modo, se o resultado da chamada da função *default_adaptation* retornar um valor diferente de *Restricted* (i.e., *Free* ou *PolicyBased*), então a regra é habilitada;
- Se *default_adaptation* retornar *Restricted*, então é verificado se o(s) componente(s) de processo selecionado(s) para remoção possuem correspondência com elementos originais de um *template*. Assim, se a chamada a função *has_abstract_origin* retornar falso, a regra é habilitada.

A figura A 66 apresenta duas regras envolvidas com a remoção de atividades em um modelo de processo de software executável. No primeiro caso (regra AdaptRule 1.1), o componente selecionado para remoção e identificado por *act_id* é uma atividade decomposta (tipo *Activity_Dec*). Assim, se o *ProcessModel* correspondente estiver vazio (i.e., sem atividades-componentes, tal como expresso pela condição negativa existente no lado esquerdo da regra), as condições existentes no campo *Condition* serão avaliadas.

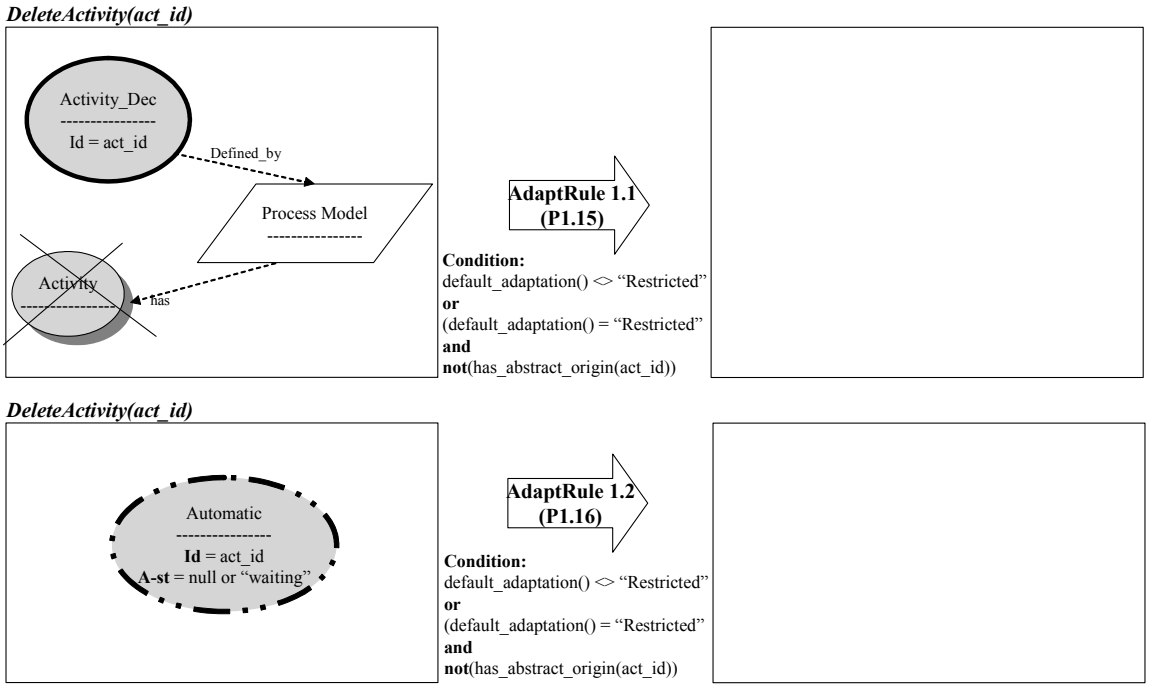

FIGURA A 66 Regras AdaptRule 1.1 e 1.2 - função *DeleteActivity*

As figuras a seguir apresentam casos adicionais para a função *DeleteActivity* expressas através das regras AdaptRule 1.3, 1.4 e 1.5.

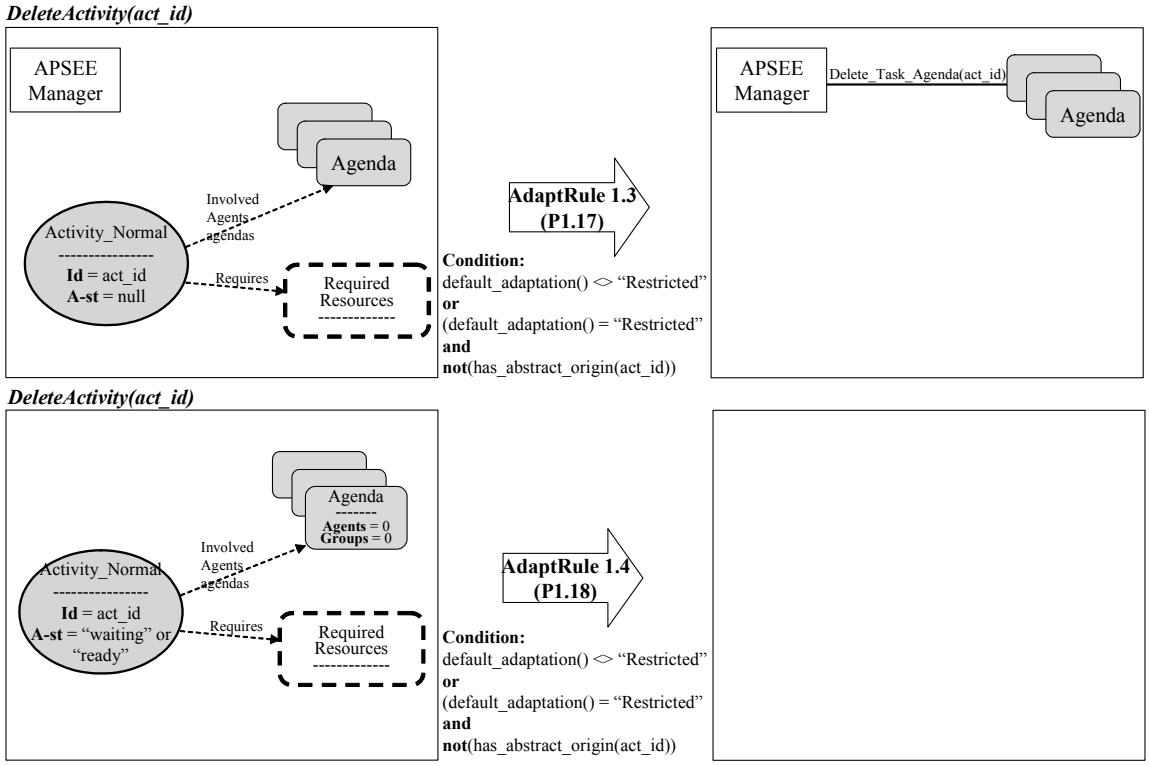

FIGURA A 67 - Regras AdaptRule 1.3 e 1.4 - função *DeleteActivity*

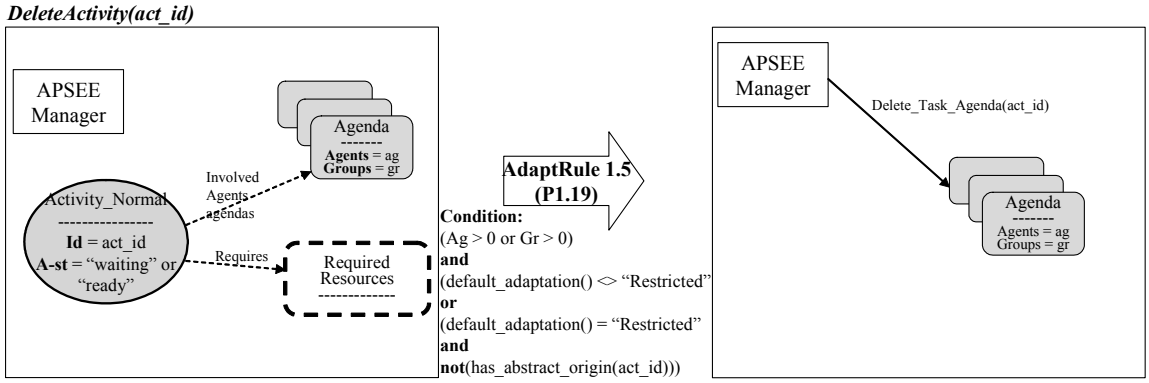

FIGURA A 68 - Regra AdaptRule 1.5 - função *DeleteActivity*

As regras que restringem a remoção de conexões de artefato estão descritas na figura A 69.

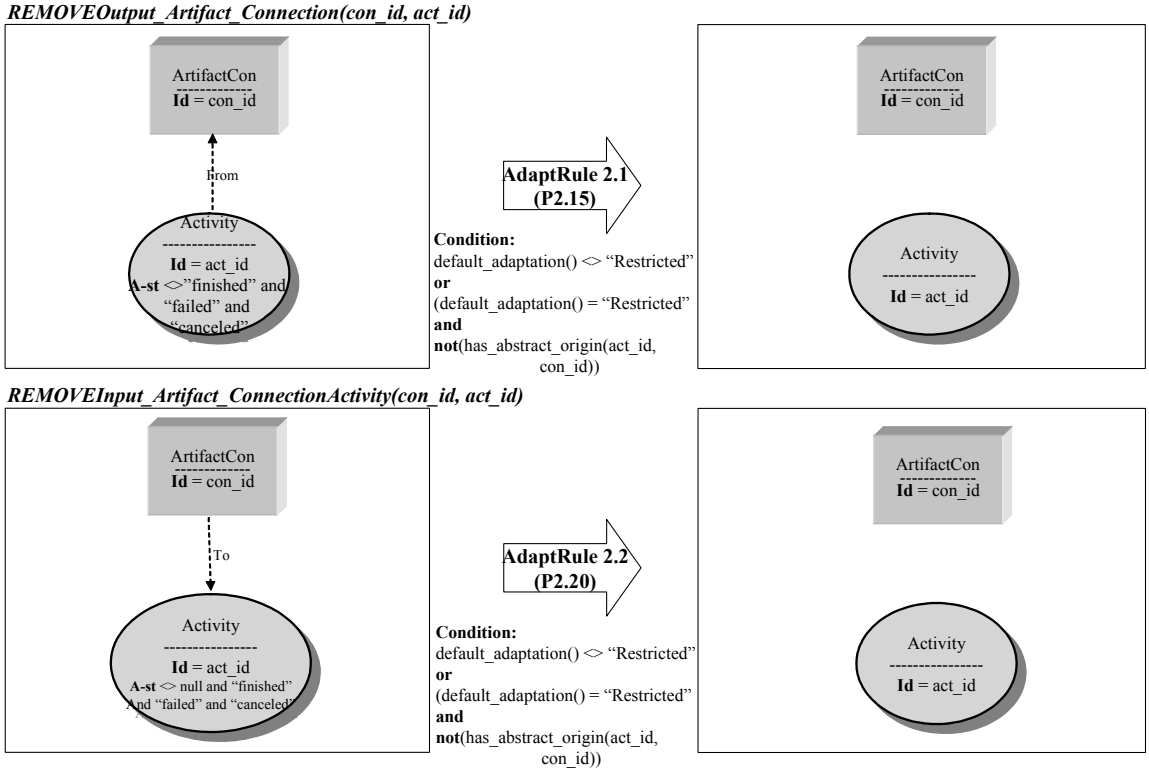

FIGURA A 69 - Regras AdaptRule 2.1 e 2.2 - função *REMOVEOutput_Artifact_Connection* e *REMOVEInput_Artifact_ConnectionActivity*

A remoção da ligação entre uma conexão de artefato e uma conexão múltipla é restrita pela regra AdaptRule 2.3 da figura A 70.

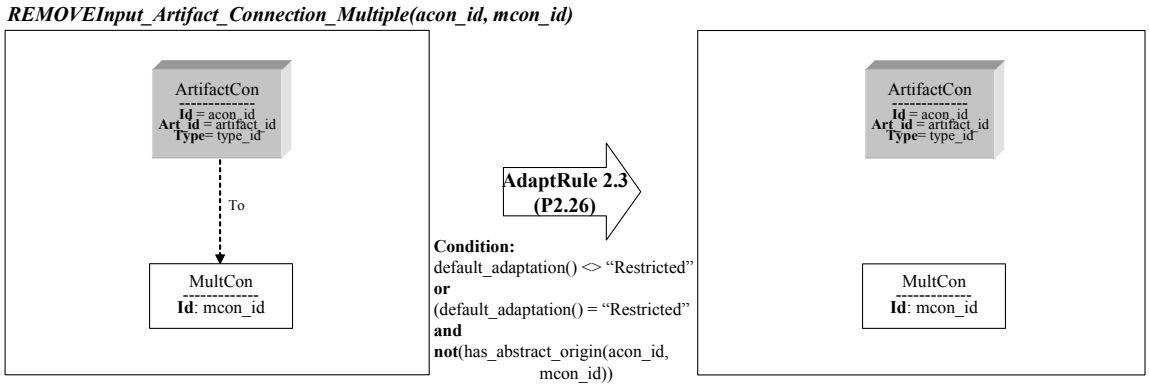

FIGURA A 70 - Regra AdaptRule 2.3 - função *REMOVEInput_Artifact_Connection_Multiple*

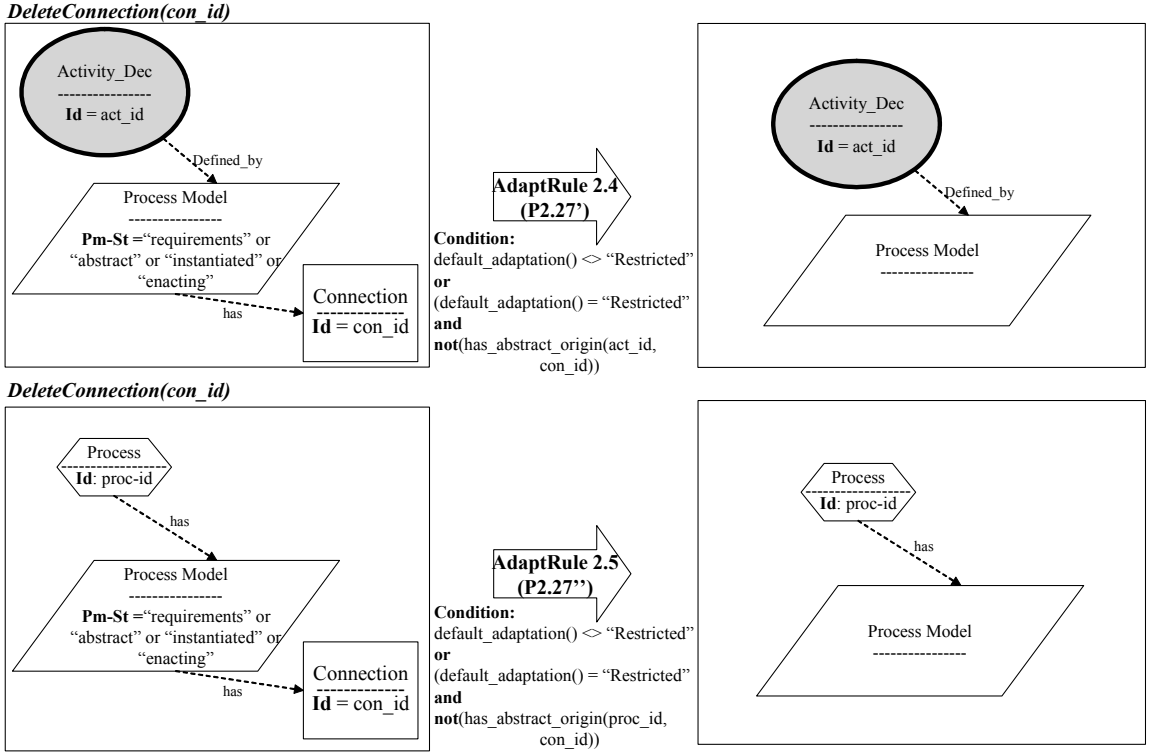

FIGURA A 71 - Regras AdaptRule 2.4 e 2.5 - função *DeleteConnection*

As figuras a seguir descrevem as regras que restringem a remoção dos demais tipos de conexões em um modelo de processo derivado a partir de um *template*.

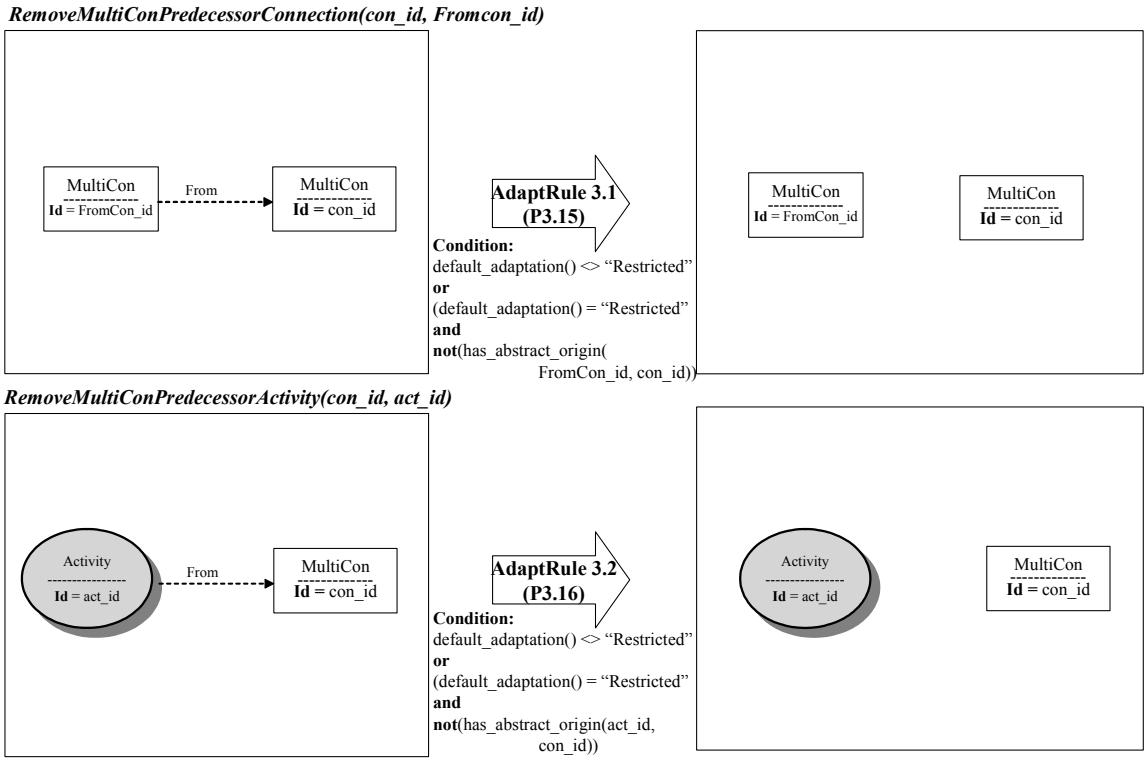

FIGURA A 72 - Regras AdaptRule 3.1 e 3.2 - funções *RemoveMultiConPredecessorConnection* e *RemoveMultiConPredecessorActivity*

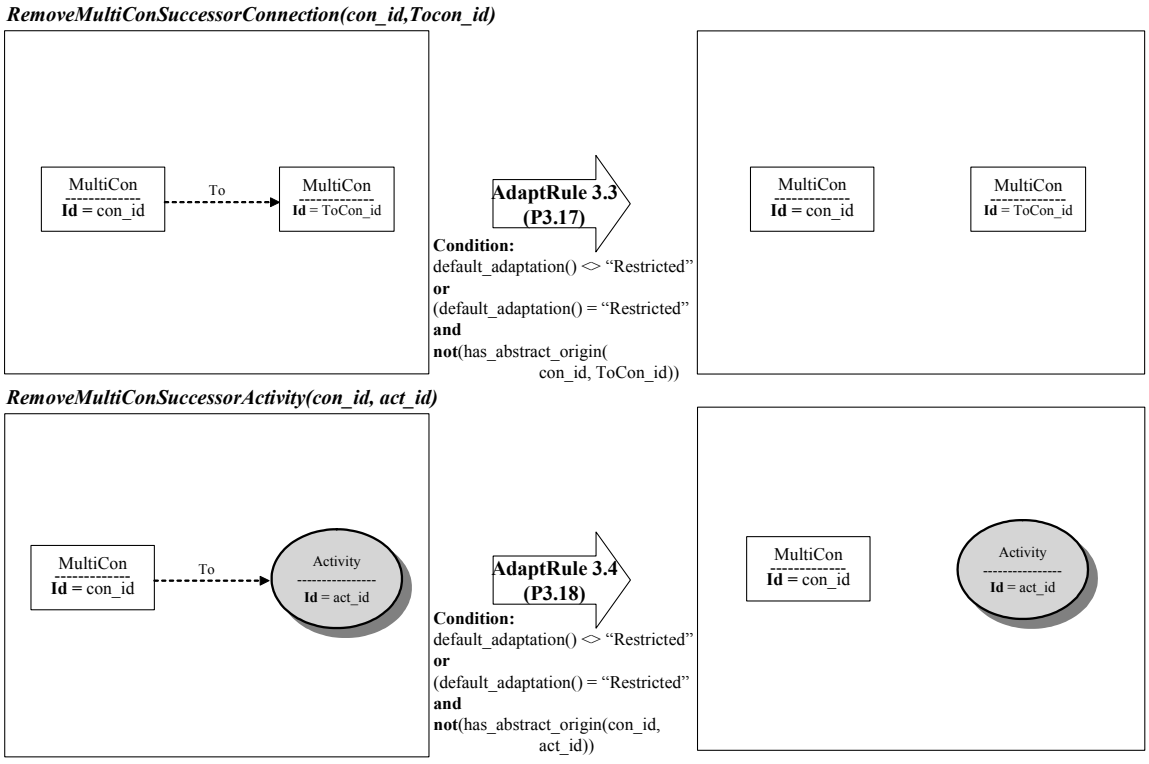

FIGURA A 73 - Regras AdaptRule 3.3 e 3.4 - funções *RemoveMultiConSuccessorConnection* e *RemoveMultiConSuccessorActivity*

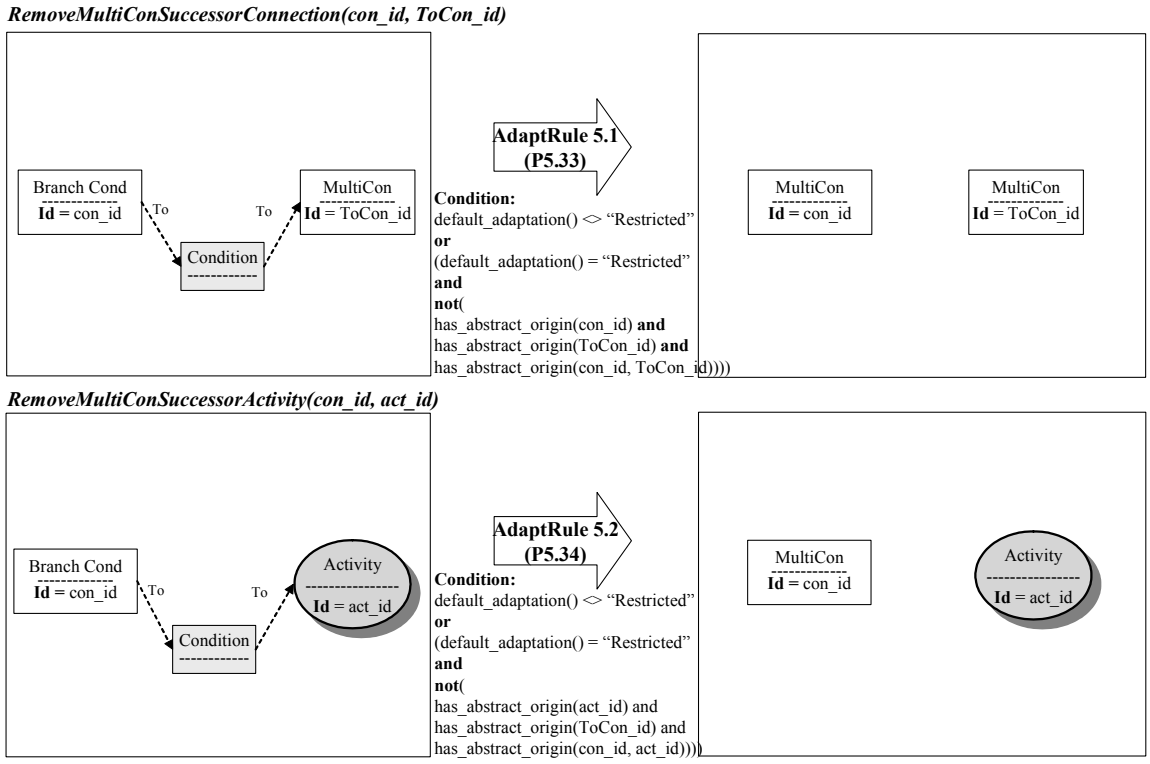

FIGURA A 74 - Regras AdaptRule 5.1 e 5.2 - funções *RemoveMultiConSuccessorConnection* e *RemoveMultiConSuccessorActivity*

A7.3.2 Regras para restringir a especialização de tipos

A substituição de componentes originais é restrita aos componentes que especializem os componentes descritos no *template* original. Por simplificação, esta seção apresenta somente a regra AdaptRule 1.6 (figura A 75) como exemplo para a especialização de atividades, embora o mesmo comportamento seja válido para a especialização de artefatos.

Segundo a figura A 75, seja a atividade concreta *Activity* (identificada por *act_id*), a atividade abstrata *AbsActivity* (identificada por *absact_id* e possuindo o tipo *abstype*), e o novo tipo solicitado pelo usuário (*new_type*), a mudança de tipo da *Activity* ocorre somente se *new_type* for um subtipo de *abstype*.

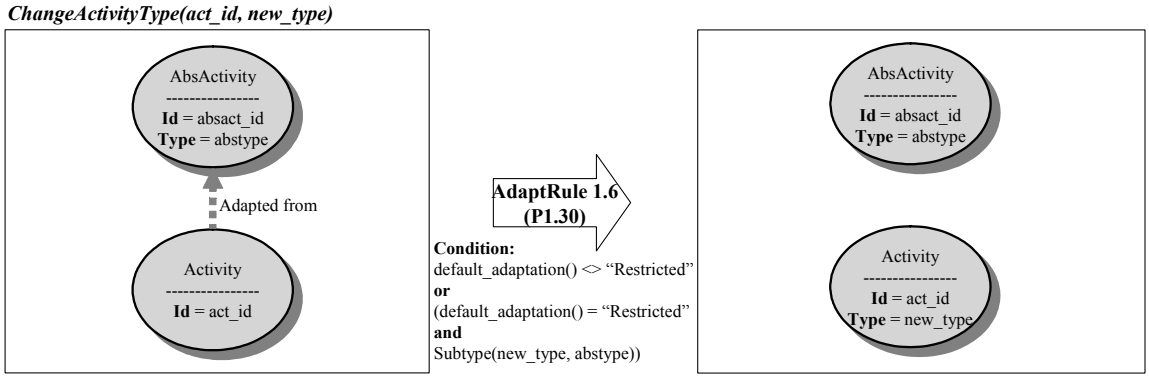

FIGURA A 75 - Regra AdaptRule 1.6 - função *ChangeArtifactType*

O teste *Adapted_From* usado na regra anterior é descrito pela regra *Test AF1* especificada abaixo.

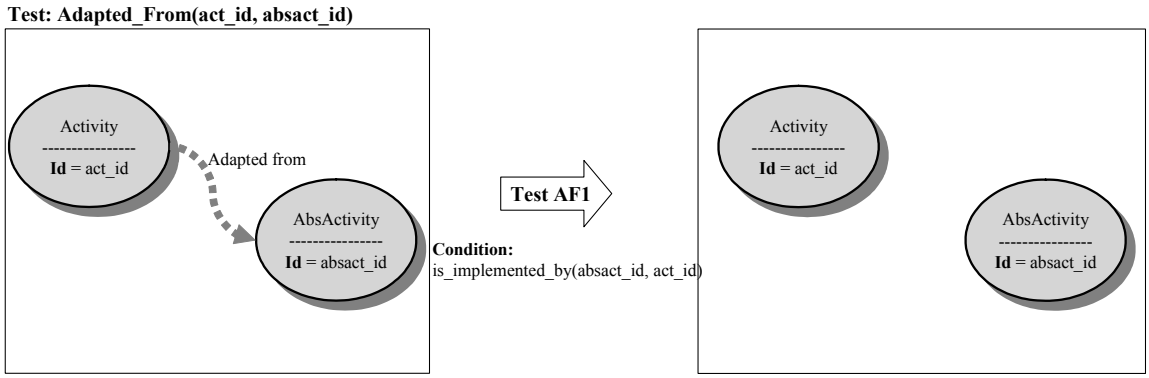

FIGURA A 76 - Regra de teste: *Adapted_From*

A7.3.3 Regras para restringir a especialização de conexões temporais

Na modelagem de um *template*, o projetista pode definir conexões temporais (tipos de dados *SimpleCon* e *MultCon*) em que as dependências associadas assumem o valor *end-start*, *start-start* ou *end-end*. Cada tipo de dependência possui uma semântica associada, o que influencia na ordem em que da execução das atividades [LIM2002d]. Alternativamente, o projetista pode deixar a cargo do usuário do *template* a definição exata da ordenação das atividades, deixando a dependência com valor indefinido. Assim, as regras de adaptação apresentadas na figura A 77 descrevem a adaptação para conexões múltiplas e simples que estão indefinidas no *template* original.

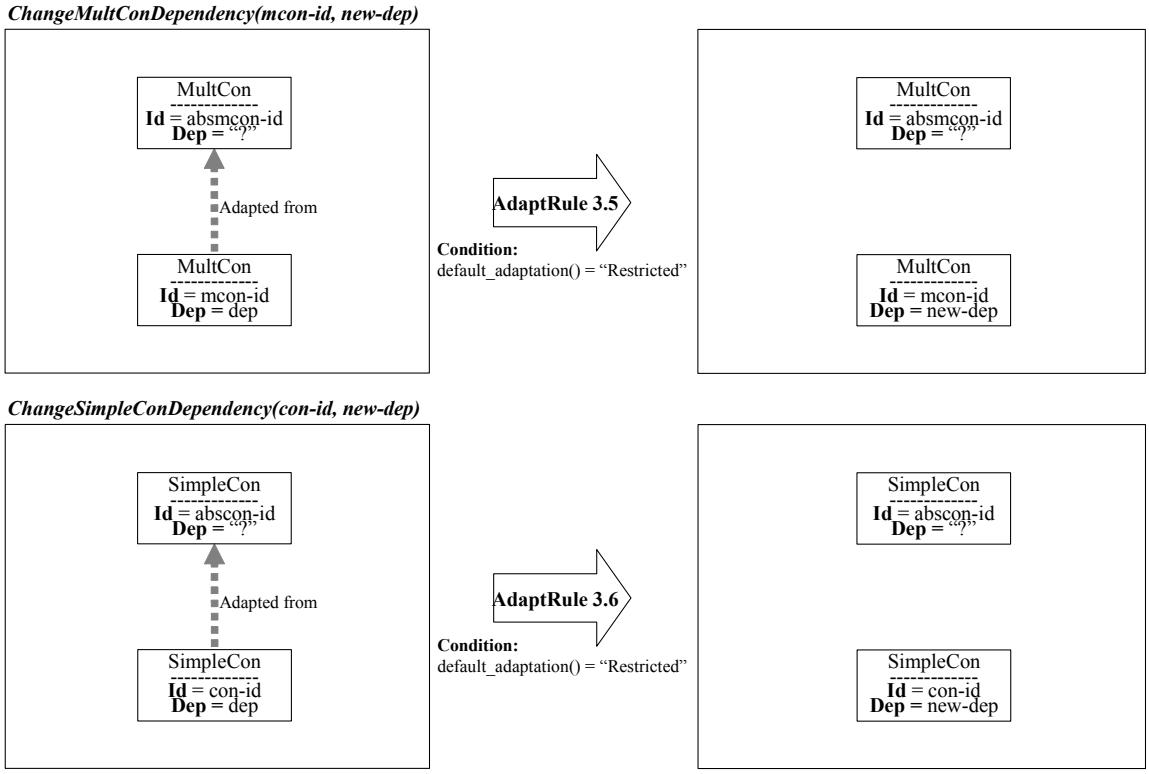

FIGURA A 77 - Regras AdaptRule 3.5 e 3.6 - funções *ChangeMultConDependency* e *ChangeSimpleConDependency*

Anexo 8 Semântica Algébrica para Políticas Estáticas

Esse anexo apresenta um resumo das principais funções que compõem a semântica algébrica para o interpretador de Políticas Estáticas descrito em [REI2001c].

A8.1 Especificação do algoritmo principal

```
Definição: verify_static_policies: APSEE, String → StaticPolEval

verify_static_policies(apsee, proc-id)
=       verify_static_policies_activities(apsee,
                ICS(StaticPolEval, create, proc-id, <current_date(), current_time()>),
                ICS(Processes, get_enabled_static_policies, select-Processes(apsee), <proc-id>) ∪
                ICS(Organization,get_enabled_static_policies,select-Organizaton(apsee)),
                ICS(Processes, get_set_activities, select-Processes(apsee), <proc-id>),
        proc-id))

Definição:
verify_static_policies_activities: APSEE, StaticPolEval, SetOfString, SetOfString, String → StaticPolEval

verify_static_policies_activities(apsee, staticPolEval, setOfPolicyIds, add(setOfActivityIds, act-id), proc-id)
=   verify_static_policies_activities(apsee,
        ICS(StaticPolEval, include_activity_log, staticPolEval,
                <act-id,
                verify_static_policies_activity(apsee, proc-id, act-id,
                        setOfPolicyIds ∪ ICS(Processes, get_enabled_static_policies,
                                select-Processes(apsee), <proc-id, act-id>))>),
                setOfPolicyIds, setOfActivityIds, proc-id))

verify_static_policies_activities(_, staticPolEval, _, empty-set, _) = staticPolEval

Definição: verify_static_policies_activity: APSEE, String, String, SetOfString → ActEvalLog

verify_static_policies_activity(apsee, proc-id, act-id, add(setOfPolicyIds, pol-id))
=       if      enabling_properties_satisfied(apsee, proc-id, act-id, pol-id)
        then    ICS(ActEvalLog, composition,
                        evaluate_policy_activity(apsee, proc-id, act-id, pol-id),
                        <verify_static_policies_activity(apsee, proc-id, act-id, setOfPolicyIds)>)
        else    verify_static_policies_activity(apsee, proc-id, act-id, setOfPolicyIds)

verify_static_policies_activity(_, _, _, empty-set) = empty-mapping
```

FIGURA A 78 - Funções que descrevem o algoritmo principal para a interpretação de Políticas Estáticas

A8.2 Avaliação de objetos expressos na seção *Enabling Properties* de uma política

Esta seção descreve as funções que avaliam as condições expressas como Propriedades de Habilitação (*Enabling Properties*) de um objeto da classe *StaticPolicies*. Todas as funções apresentadas nesta seção são definidas para o ATO APSEE, pois necessitam de informação do estado atual dos processos para realizarem as verificações desejadas. Por simplificação, serão apresentadas aqui somente funções que tratem de condições relacionadas aos objetos do tipo *Activity*. Uma descrição completa envolvendo os outros tipos de objetos associados a uma política está em [REI2001c].

A figura A 79 apresenta a especificação da função que recebe como argumentos um objeto APSEE, e os identificadores do processo, da atividade e da política sob avaliação. O item {1} verifica se o tipo APSEE associado com a interface da Política é *Activity*. O item {2} descreve a chamada à função *verify_conditions*, fornecendo como parâmetros o objeto APSEE, o atributo *EnablingProperties* da Política (item {4}, corresponde ao objeto de *PolCondition* a ser avaliado), o identificador do processo (*proc-id*) e da atividade em avaliação (*act-id*).

```
Definição: enabling_properties_satisfied: APSEE, String, String, String → Boolean

enabling_properties_satisfied(apsee, proc-id, act-id, pol-id)
=   if   ICS(Static Policies, get_interface_type, select-Policies(apsee), <pol-id>) = "Activity" {1}
    then verify_conditions(                                                                      {2}
                    apsee,
                    expand_resword_cond(                                                         {3}
                            apsee,
                            ICS(Static Policies, get_condition_enbl_prop,                        {4}
                                    ICS(Policies, get_static, select-Policies(apsee)), <pol-id>),
                            proc-id, act-id), proc-id, act-id)
         else    // comportamento análogo para os outros tipos de objetos do meta-modelo APSEE
```

FIGURA A 79 - Função que verifica se as pré-condições de uma Política Estáticas são satisfeitas (*enabling_properties_satisfied*)

A8.2.1 Expansão de palavras reservadas

A chamada à função *expand_resword_cond* localizada no item {3} da figura A 79 é responsável por uma importante etapa da interpretação de políticas estáticas. Toda expressão (contida em uma condição, isto é, instância de *PolCondition*) que possua referências para palavras reservadas (*any, all* e *no*) deve ser obrigatoriamente expandida antes de ser avaliada pela função *verify_conditions*. Na figura A 80 é apresentado um esquema gráfico para ilustrar a expansão de uma condição que faz uso de *any* para uma política-exemplo (*External Review*).

Toda ocorrência de uma palavra reservada possui o formato:

conjunto.**palavra-reservada**.chamada-de-método(parâmetros)

Onde:

- *conjunto* retorna um conjunto de objetos ou tipos APSEE;
- *palavra-reservada* assume o valor *any*, *all* ou *no*;
- *chamada-de-método* é uma chamada de função.

Assim, tal como mostrado no exemplo da figura A 80, o tratamento para ocorrências de *any* podem ser informalmente descritas da seguinte forma:

- o objeto *conjunto* é produzido (através da avaliação das expressões correspondentes);
- a condição original fornecida é substituída (expandida) por uma disjunção (*OR*) da lista de condições que avaliam cada um dos componentes.

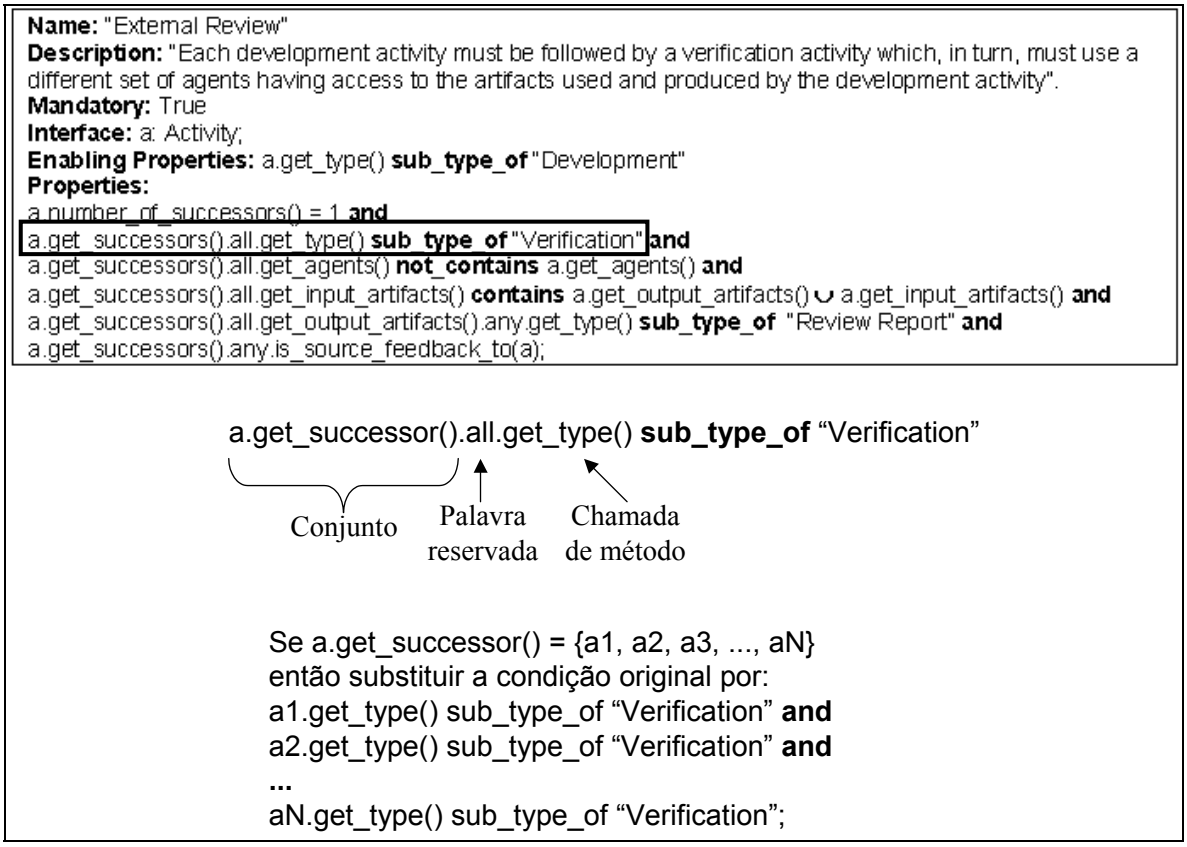

FIGURA A 80 - Ilustração do mecanismo de expansão de expressões com palavras reservadas

A função *expand_resword_cond*, descrita na figura A 81, avalia a condição recebida como argumento (*cond*). O resultado obtido é a composição {1} entre a expansão da primeira condição expressa no objeto *cond* (desmembrada pelas funções em {2}, {3} e {4}), com o resultado da chamada recursiva para a mesma função com as condições subseqüentes {5}.

```
Definição: expand_resword_cond: APSEE, PolCondition ∪ Null, String, String → PolCondition ∪ Null
expand_resword_cond(_, Null, _, _) = Null

expand_resword_cond(apsee, cond, proc-id, act-id)
=       ICS(PolCondition, composition,                                              {1}
                evaluate_resword(apsee,
                        ICS(PolConditon, get_expression, cond),                     {2}
                        ICS(PolCondition, get_comparison_type, cond),               {3}
                        ICS(PolCondition, get_related_expression, cond),            {4}
                        proc-id, act-id),
                <ICS(PolCondition, get_conn_type, cond),
        {5}     expand_resword_cond(apsee, ICS(PolCondition, get_conn, cond), proc-id, act-id)>)
```

FIGURA A 81 - Descrição da função *expand_resword_cond*

A figura A 82 apresenta as funções *evaluate_resword*, *expand_left* e *expand_set_cond* usadas na expansão de uma expressão que contém palavras reservadas.

```
Definição:
evaluate_resword: APSEE, PolExpression, ComparisonType, PolExpression, String, String → PolCondition

evaluate_resword(apsee, expression, null, _, _, proc-id, act-id)
=       evaluate_resword(apsee, expression, null, null, proc-id, act-id)

evaluate_resword(apsee, expression, _, null, proc-id, act-id)
=       if      ICS(PolExpression, get_type, expression) = "SingleOperand" and
                ICS(PolExpression, has_resword, expression)
        then    expand_left(apsee, expression, null, null, proc-id, act-id)
        else    ICS(PolCondition, create, expression)

evaluate_resword(apsee, exp1, comp, exp2, proc-id, act-id)
=       if      not(ICS(PolExpression, has_resword, exp1) or
                ICS(PolExpression, has_resword, exp2))
        then    ICS(PolCondition, create, exp1, <comp, exp2>)
        else    if      ICS(PolExpression, has_resword, exp1)
                then    expand_left(apsee, exp1, comp, exp2, proc-id, act-id)
                else    expand_right(apsee, exp1, comp, exp2, proc-id, act-id)

Definição:
expand_left: APSEE, PolExpression, ComparisonType ∪ Null, PolExpression ∪ Null, String, String → PolCondition

expand_left(apsee, exp1, comp, exp2, proc-id, act-id)
=       expand_left2(
                expand_set_cond(apsee, exp1, proc-id, act-id),
                ICS(PolExpression, get_resword_type, exp1),
                ICS(PolOperand, get_last_operator, ICS(PolExpression, get_operand, exp1)),
                comp, exp2)

expand_left2: PolObjValue, String, PolOperator, ComparisonType ∪ Null, PolExpression ∪ Null → PolCondition
expand_left2(add(empty-set, obj), _, operator, comp, exp2)
=       ICS(PolCondition, create,
                ICS(PolExpression, create, ICS(PolOperand, create, obj, <operator>)),
        <comp, exp2>)

expand_left2(add(set, obj), "Any", operator, comp, exp2)
=       ICS(PolCondition, create,
                ICS(PolExpression, create,
                        ICS(PolOperand, create, obj, <operator>)),
                <comp, exp2, "Or", expand_left2(set, "Any", operator, comp, exp2)>)

expand_left2(add(set, obj), "All", operator, comp, exp2)
=       ICS(PolCondition, create,
                ICS(PolExpression, create, ICS(PolOperand, create, obj, <operator>)),
                <comp, exp2, "And", expand_left2(set, "All", operator, comp, exp2)>)

Definição: expand_set_cond: APSEE, PolExpression, String, String → PolObjValue
/* Avalia a expressão até a ocorrência da palavra reservada
/* Exemplo: a.get_artifacts().any → avalia(a.get_artifacts())
expand_set_cond(apsee, expression, proc-id, act-id)
=       eval_expression(apsee, ICS(PolExpression, remove_resword, expression), proc-id, act-id)
```

FIGURA A 82 - Funções que tratam da expansão de palavras reservadas durante a avaliação de uma Política Estática.

A8.2.2 A função *verify_conditions*

A função *verify_conditions* descrita na avalia um objeto *cond* da classe *PolCondition* para uma atividade identificada pelo seu *proc-id* e *act-id*. A função verifica se a primeira condição expressa em *cond* está conectada logicamente por um *and* com a condição seguinte (em {1}), por um *or* (em {2}), ou se *cond* é a última condição a ser avaliada pela função ({3}).

```
Definição: verify_conditions: APSEE, PolCondition, String, String → Boolean

verify_conditions(apsee, cond, proc-id, act-id)
=       if      ICS(PolCondition, get_conn_type, cond) = "And"                          {1}
        then    and_eval_condition(apsee, cond, ICS(PolCondition, get_conn, cond), proc-id, act-id)

        else    if      ICS(PolCondition, get_conn_type, cond) = "Or"                   {2}
                then    or_eval_condition(apsee, cond, ICS(PolCondition, get_conn, cond),
                                proc-id, act-id)
                else    eval_condition(apsee, cond, proc-id, act-id)                    {3}
```

FIGURA A 83 - A função *verify_conditions*

As funções *and_eval_condition* e *or_eval_condition* são apresentadas na figura A 84.

```
Definição: and_eval_condition: APSEE, PolCondition, PolCondition, String, String → Boolean

and_eval_condition(apsee, cond, connected, proc-id, act-id)
=       eval_condition(apsee, cond, proc-id, act-id) and
        verify_conditions(apsee, connected, proc-id, act-id)

Definição: or_eval_condition: APSEE, PolCondition, PolCondition, String, String → Boolean

or_eval_condition(apsee, cond, connected, proc-id, act-id)
=       eval_condition(apsee, cond, proc-id, act-id) or
        verify_conditions(apsee, connected, proc-id, act-id)
```

FIGURA A 84 - Funções *and_eval_condition* e *or_eval_condition*

A função *eval_condition* (figura A 85) faz uso de funções auxiliares (*eval_relation* {1} e *eval_expression* {2, 3 e 5}) para avaliar um objeto de *PolCondition* e retornar um valor booleano. O passo {4} é necessário para converter um objeto booleano gerado pela função *eval_expression* {5} na forma de um objeto de *PolObjValue* em um *Boolean.*

```
Definição: eval_condition: APSEE, PolCondition, String, String → Boolean

eval_condition(apsee, cond, proc-id, act-id)
=       if      ICS(PolCondition, has_relation, cond)
        then    eval_relation                                                          {1}
                (apsee,
{2}             eval_expression(apsee, ICS(PolCondition, get_expression, cond), proc-id, act-id),
                eval_expression(apsee,
{3}                     ICS(PolCondition, get_related_expression, cond), proc-id, act-id),
                ICS(ComparisonType, to_string, ICS(PolCondition, get_comparison_type, cond)))
        else    ICS(PolObjValue, to_boolean,                                           {4}
{5}             eval_expression(apsee, ICS(PolCondition, get_expression, cond), proc-id, act-id))
```

FIGURA A 85 - Função que avalia se uma condição lógica é satisfeita para o estado da atividade do processo associada (*eval_condition*)

A função *eval_relation* (figura A 86) inicialmente verifica em {1} se os tipos sendo comparados são compatíveis entre si. Em {2}, é feita chamada à função auxiliar *eval_relation2* que apresenta diferentes especificações que unificam com o valor do objeto *comp* (de {3} a {7}).

```
Definição: eval_relation: APSEE, PolObjValue, PolObjValue, String → Boolean
eval_relation(apsee, obj1, obj2, comp)
=       if      ICS(PolObjValue, is_compatible, obj1, <obj2>)                    {1}
        then    eval_relation2(apsee, obj1, obj2, comp)                          {2}
        else    false

eval_relation2(_, obj1, obj2, "=") = ICS(PolObjValue, equals, obj1, <obj2>)      {3}

eval_relation2(apsee, obj1, obj2, "<>") = not(eval_relation2(apsee, obj1, obj2, "=")) {4}

eval_relation2(apsee, obj1, obj2, "SubTypeOf")                                   {5}
=       if      ICS(PolObjValue, get_type, obj1) = "Activity"
        then    ICS(Processes, is_subtype_of, apsee, <obj1, obj2)>)
        else    if      ICS(PolObjValue, get_type, obj1) = "Artifact"
                then    ICS(SwArtifacts, is_sub_type_of, apsee, <obj1, obj2>)

eval_relation2(apsee, obj1, obj2, "Contains") = ICS(PolObjValue, contains, obj1, <obj2>)  {6}

eval_relation2(apsee, obj1, obj2, "NotContains") = not (ICS(PolObjValue, contains, obj1, <obj2>) {7}
```

FIGURA A 86 - Função *eval_relation*

A8.2.3 Avaliação de expressões com a função *eval_expression*

A figura A 87 apresenta a função *eval_expression* que retorna uma instância de *PolObjValue* para cada objeto de *PolExpression* avaliado. Se a instância de *PolExpression* a ser avaliada possuir um único operando (verificado em {1}) será chamada a função *eval_operand* ({2}). Como os parâmetros de um operando são constituídos por expressões, é necessário avaliar previamente os parâmetros de cada um dos operadores através da função *evaluate_parameters* (item {3}). Finalmente, caso a instância de *PolExpression* for constituída de um operando seguido de um tipo de expressão e uma nova expressão, a função *eval_operand* retorna o resultado obtido a partir da chamada de *apply_expression_operator* (item {4}). Vale ressaltar que as funções *eval_relation* e *get_method_result* não apresentam todos os casos permitidos na linguagem.

```
Definição: eval_expression: APSEE, PolExpression, String, String → PolObjValue
eval_expression(apsee, expression, proc-id, act-id)
=       if      ICS(PolExpression, get_type, expression) = "SingleOperand"          {1}
        then    eval_operand(apsee,                                                 {2}
                evaluate_parameters (                                               {3}
                        apsee, ICS(PolExpression, get_operand, expression), proc-id, act-id),
                proc-id, act-id)
        else    apply_expression_operator                                           {4}
                (apsee,
                eval_operand(apsee, ICS(PolExpression, get_operand1, expression), proc-id, act-id),
                eval_expression(apsee, ICS(PolExpression, get_operand2, expression),
                                proc-id, act-id),
                ICS(PolExpression, get_operator, expression))

Definição: evaluate_parameters: APSEE, PolOperand, String, String → PolOperand
evaluate_parameters(apsee, operand, proc-id, act-id)
=       ICS(PolOperand, create, ICS(PolOperand, get_object, operand),
                evaluate_parameters2
                        (apsee, ICS(PolOperand, get_Operators, operand), proc-id, act-id))

Definição: evaluate_parameters2: APSEE, ListOfOperators, String, String → ListOfOperators
evaluate_parameters2(_, empty-list, _, _) = empty-list
evaluate_parameters2(apsee, cons(operator, operator-list), proc-id, act-id)
=       if      ICS(PolOperator, get_Parameters, operator) = empty-list
        then    cons(operator, evaluate_parameters2(apsee, operator-list, proc-id, act-id)
        else    cons(ICS(PolOperator, create_methodCall,
                ICS(PolOperator, get_methodID, operator),
                <evaluate_parameters3(apsee, ICS(PolOperator, operator, get_Parameters),
                proc-id, act-id)>, evaluate_parameters2(apsee, operator-list, proc-id, act-id))

Definição: evaluate_parameters3: APSEE, ListOfParameters, String, String → ListOfParameters
evaluate_parameters3(_, empty-list, _, _) = empty-list
evaluate_parameters3(apsee, cons(parameter, parameter-list), proc-id, act-id)
=       cons(eval_expression(apsee, parameter, proc-id, act-id),
                evaluate_parameters3(apsee, parameter-list, proc-id, act-id))
```

FIGURA A 87 - Funções *eval_expression* e *evaluate_parameters*

A figura A 88 apresenta *eval_operand*, função que avalia um objeto da classe *PolOperand*, retornando o resultado como um *PolObjValue*. Se o operando possuir uma lista vazia de operadores (item {1}), então a função verifica se o objeto manipulado pelo *PolObject* associado é uma referência à interface da política (item {2}): caso positivo, a função retorna uma instância de *PolObjValue* contendo a referência para a atividade sob avaliação (item {3}). Caso contrário, a avaliação do operando retorna o valor armazenado no objeto associado (item {4}).

Se o operando sendo avaliado possuir uma lista de operadores associados (i.e., a verificação do item {1} da figura A 88 falhou) a função verifica se o primeiro elemento da lista é uma referência à *MethodCall* (item {3}). Desse modo, a função aplica o primeiro método no operando associado (item {5}) e aplica recursivamente *eval_operand* para o resultado obtido (item {4}). O item {6} refere-se a um objeto *PolOperand* que possua como primeiro elemento da lista de operadores um objeto que não referencie uma chamada de método (i.e., o objeto *PolOperand* não está bem formado).

```
Definição: eval_operand: APSEE, PolOperand, String, String → PolObjValue
eval_operand(apsee, operand, proc-id, act-id)
=       if      ICS(PolOperand, empty_list_operators, operand)                          {1}
        then    if      ICS(PolObject, is_interface, ICS(PolOperand, get_object, operand))  {2}
                then    ICS(PolObjValue, create_activity, proc-id, act-id)
                else    ICS(PolObject, get_objValue, ICS(PolOperand, get_object, operand))
        else    if      ICS(PolOperator, get_type,
                        ICS(PolOperand, get_first_operator, operand)) = "MethodCall"    {3}
                then    eval_operand(apsee,                                             {4}
                                apply_method_operator(apsee, operand, proc-id, act-id), {5}
                                proc-id, act-id)
                else    error                                                           {6}
```

FIGURA A 88 - Função *eval_operand*

A função *apply_expression_operator* é definida na figura A 89. Essa função obtém o resultado da avaliação de uma expressão composta por dois objetos de *PolObjValue*. Inicialmente (em {1}) é verificado se os dois objetos são compatíveis entre si (uma característica necessária para a boa formação da política sob avaliação). Em seguida (item {2}) é avaliado o tipo de *operator* associado (*Union* ou *Intersection*) para determinar a chamada às funções correspondentes (itens {3} ou {4}).

```
Definição:
apply_expression_operator: APSEE, PolObjValue, PolObjValue, ExpressionType → PolObjValue
apply_expression_operator(apsee, obj1, obj2, exp-type)
=       if      ICS(PolObjValue, is_compatible_type, obj1, <obj2>)              {1}
        then    if      ICS(ExpressionType, get_type, exp-type) = "Union"       {2}
                then    ICS(PolObjValue, union, obj1, <obj2>)                   {3}
                else    ICS(PolObjValue, intersection, ob1, <obj2>)             {4}
        else    error
```

FIGURA A 89 - Função *apply_expression_operator*

A função *apply_method_operator* está descrita na figura A 90. O objetivo desta função é avaliar um objeto *PolOperand* recebido, retornando um novo objeto devidamente avaliado. Se o objeto contido no operador avaliado for uma referência à interface da política (condição verificada no item {1} da especificação), então certamente o objeto será do tipo *Activity* e a especificação do item {2} construirá um novo objeto de *PolOperand* que armazenará o resultado da função *get_method_result_activity*. Caso contrário, a avaliação segue com a chamada à função genérica *apply_method_object* (no item {3}) que, como definido seguir, possui uma especificação para todos os tipos de objetos APSEE manipulados pela política. É incluída somente a definição para a função *get_type()* (item {4}) no tipo *Activity*[51] {5}.

[51] O leitor interessado na definição semântica das outras funções suportadas deve consultar [REI2001c].

```
Definição: apply_method_operator: APSEE, PolOperand, String, String → PolOperand
apply_method_operator(apsee, operand, proc-id, act-id)
=       if      ICS(PolObject, operand_is_interface, operand)                    {1}
        then    ICS(PolOperand, create,
                        get_method_result_activity                               {2}
                        (apsee, proc-id, act-id,
                        ICS(PolMethods, get_method_name,
                                ICS(PolOperator, get_methodId,
                                        ICS(PolOperand, get_first_operator, operand))),
                        <ICS(PolOperand, get_tail_operators, operand)>)
        else    ICS(PolOperand, create,                                          {3}
                        apply_method_object
                        (apsee,
                        ICS(PolObject, get_ObjValue, ICS(PolOperand, get_object, operand)),
                        ICS(PolObjValue, get_type,
                                ICS(PolObject, get_ObjValue,
                                        ICS(PolOperand, get_object, operand))),
                        ICS(PolOperand, get_first_operator, operand)),
                        <ICS(PolOperand, get_tail_operators, operand)>)

Definição: apply_method_object: APSEE, PolObjValue, String, Operator → PolObjValue
apply_method_object(apsee, objValue, "Activity", operator)                       {4}
=       get_method_result_activity
                (
                apsee,
                ICS(PolApseeObj, get_ProcessID, ICS(PolObjValue, get_apsee_obj, objValue)),
                ICS(PolApseeObj, get_ActivityID, ICS(PolObjValue, get_apsee_obj, objValue)),
                ICS(PolMethods, get_method_name, ICS(PolOperator, get_methodId, operator)),
                ICS(PolOperator, get_list_parameters, operator)
                )

Definição: get_method_result_activity: APSEE, String, String, String, ListOfString → PolObjValue
get_method_result_activity(apsee, proc-id, act-id, "get_type", empty-list)       {5}
=       ICS(PolObjValue, create_activity_typeID,
                ICS(Processes, get_type, select-Processes(apsee), <proc-id, act-id>))
```

FIGURA A 90 - Exemplos de funções que definem a semântica para métodos que manipulam atividades de processos

A8.3 Avaliação de objetos expressos na seção *Properties* de uma política

A função *evaluate_policy_activity*, apresentada na figura A 91 é análoga à *enabling_properties_satisfied* (apresentada anteriormente na figura A 79). A diferença é o resultado da avaliação: nesse caso, o resultado é um objeto da classe *ActEvalLog*, retornando um mapeamento vazio {1} ou mapeamentos que contenham *Errors* {2} ou *Warnings* {3} que porventura surjam da avaliação de uma política estática.

```
Definição: evaluate_policy_activity: APSEE, String, String, String → ActEvalLog

evaluate_policy_activity(apsee, proc-id, act-id, pol-id)
=       if      verify_conditions
                (
                apsee,
                expand_resword_cond
                        (
                        apsee,
                        ICS(StaticPolicies, get_condition_properties,
                                ICS(Policies, get_static, select-Policies(apsee)),
                        <pol-id>),
                        proc-id, act-id
                        ),
                proc-id, act-id
                )
        then    empty-mapping                                                   {1}
        else    if      ICS(StaticPolicies, is_mandatory,
                                select-Static(select-Policies(apsee)), <pol-id>)
                then    ICS(ActEvalLog, create, pol-id,  <"error">)             {2}
                else    ICS(ActEvalLog, create, pol-id,  <"warning">)           {3}
```

FIGURA A 91 - Função *evaluate_policy_activity*

Bibliografia

[AGG2002] THE ATTRIBUTED Graph Grammar System: An Interpreter for Attributed Graph Transformation with Integration of Java Programs. Berlin: Technischen Universität Berlin. Disponível em: <http://tfs.cs.tu-berlin.de/agg/>. Acesso em: abr. 2002.

[ALE99] ALENCAR, F.M.R. **Mapeamento da Modelagem Organizacional em Especificações Precisas.** 1999. Tese (Doutorado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife.

[AMB99] AMBLER, S.W. **Process Patterns:** Building Large-Scale Systems Using Object Technology. Cambridge: SIGS, 1998.

[AVR96a] AVRILIONIS, D.; CUNIN, P. Process Model Reuse Support - The OPSIS Approach. In: BOEHM, B.; KELLNER, M.; PERRY, D. (Ed.) INTERNATIONAL SOFTWARE PROCESS WORKSHOP, 10., 1996, Ventron, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p. 37-40. Disponível em: <http://horus.imag.fr/Les.Personnes/Pierre-Yves.Cunin/PUBLICATIONS/ispw10.html>. Acesso em: jan. 2000.

[AVR96b] AVRILIONIS, D.; CUNIN, P.; FERNSTRÖM, C. OPSIS: A View Mechanism for Software Processes which Supports Their Evolution and Reuse. INTERNACIONAL CONFERENCE ON SOFTWARE ENGINEERING, 18., 1996, Berlin, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p. 25-29. Disponível em: <http://horus.imag.fr/Les.Personnels/Pierre-Yves.Cunin/PUBLICATIONS/publications.html>. Acesso em: fev. 2000.

[BAN94] BANDINELLI, S.; FUGGETTA, A.; GHEZZI, C.; LAVAZZA, L. SPADE: An Environment for Software Process Analysis, Design and Enactment. In: FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology**. Tauton: Research Studies Press, 1994. p. 223-248.

[BAR2000] BARDOHL, R. **GenGED** - Visual Definition of Visual Languages based on Algebraic Graph Transformation. 2000. Thesis (Computer Science PhD) - Technische Universität Berlin, Berlin.

[BHH2000] BARROCA, L.; HALL, J.; HALL, P. (Ed.) **Software Architectures:** Advances and Applications. London: Springer, 2000.

[BER98] BERGMANN, R.; STAHL, A. Similarity Measures for Object-Oriented Case Representations. In: EUROPEAN WORKSHOP ON CASE-BASED REASONING, EWCBR, 1998. **Proceedings…** [S.l: s.n.], 1998.

[BER99] BERGMANN, R. et al. **Developing Industrial Case-Based Reasoning Applications**. Berlin: Springer-Verlag, 1999. (Lecture Notes in Artificial Intelligence, v. 1612).

[BOO94] BOOCH, G. **Object-Oriented Analysis and Design with Applications**. [S.l.]: Benjamim Cummings, 1994.

[BOO98] BOOCH, G. et al. **The Unified Modeling Language User Guide.** New York: Addison-Wesley, 1998.

[BOL99] BOLCHER, G.A.; KAISER, G. SWAP: Leveraging the Web to Manage Workflow. **IEEE Internet Computing**. Los Alamitos: IEEE Computer Society Press, v. 3, n. 1, Jan-Feb, 1999.

[BOT2001] BOTHA, R.A.; ELOFF, J.H.P. Designing Role Hierarchies for Access Control in Workflow Systems. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 25., 2001. **Proceedings…** Los Alamitos: IEEE Computer Society, 2001.

[CAR97] CARLSEN, S.; GJERSVIK, R. Organizational Metaphors as Lenses for Analyzind Workflow Technology. In: INTERNATIONAL CONFERENCE ON SUPPORTING GROUP WORK, 1997, Phoenix. **Proceedings…** New York: ACM Press, 1997.

[COH86] COHEN, B.; HARWOOD,W.T; JACKSON, M.I. **The Specification of Complex Systems.** Wokinghan: Addison-Wesley, 1986.

[COL2001] COLE, J.; DERRICK, J.; MILOSEVIC, Z.; RAYMOND, K. Author Obliged to Submit Paper before 4 July: Policies in an Enterprise Specification. In: SLOMAN, M.; LOBO, J.; LUPU, E. (Ed.) Policies for Distributed Systems and Networks. International Workshop POLICY 2001, Bristol, UK. **Proceedings…** Berlin: Springer-Verlag, 2001 p. 1-18.

[CON95] CONRADI, R.; CHUNNIAN, L. Process Modeling Languages: One or Many? In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 4. Noordwijkerhout, The Netherlands. **Proceedings…** Berlin: Springer-Verlag, 1995. Disponível em <http://www.idi.ntnu.no/grupper/su/publ/pdf/pml-ewspt95.pdf>. Acesso em: abr. 2002.

[CON98] CONRADI, R.; FUGGETTA, A.; JACCHERI, M. Six Theses on Software Process Research. In: EUROPEAN WORKSHOP SOFTWARE PROCESS TECHNOLOGY, 6., EWSPT, 1998, Weybridge, UK. **Proceedings…**, Berlin: Springer, 1998. (Lecture Notes in Computer Science, v. 1487)

[DAM98] DAMI, S. et al. APEL: a Graphical Yet Executable Formalism for Process Modeling. **Automated Software Engineering**, special issue on Software Process Technology, [S.l.], v. 5, n. 1, Jan. 1998.

[DAV95] DAVIS, A.M. **201 Principles of Software Development.** New York: McGraw-Hill, 1995.

[DAU92] DAUDT, R. **Uma Proposta de Extensão do PROSOFT Básico.** 1992. Trabalho de Diplomação. (Bacharelado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[DEH2000] DÉHARBE, D.; MOREIRA, A.; RIBEIRO, L.; RODRIGUES, V. Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes. **Revista de Informática Teórica e Aplicada**. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, v. 6, n.1, Set. 2000.

[DER99] DERNIAME, J.; KABA, B.; WASTELL, D. (Ed.) **Software Process:** Principles, Methodology and Technology. Berlin: Springer-Verlag, 1999. (Lecture Notes in Computer Science, v. 1500).

[DOW91] DOWSON, M.; NEJMEH, B.; RIDDLE, W. Fundamental Software Process Concepts. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS MODELLING, 1., 1991, Milan, Italy. **Proceedings...** [S.l.]: AICA Press, 1991.

[EMA98] EMAN, K.; DROUIN, J.N.; MELO, W. (Ed.) **SPICE**: The theory and practice of software process improvement and capability determination. Los Alamitos: IEEE Computer Society Press, 1998.

[ENG97] ENGELS, G.; HECKEL, R.; TAENTZER, G.; EHRIGH, H. A View-Oriented Approach to System Modelling Based on Graph Transformation. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE, ESEC, 1997. **Proceedings…** Berlin: Springer-Verlag, 1997. (Lecture Notes in Computer Science, v. 1301).

[ELL96] ELLMER, E.; MERKL, D.; QUIRCHMAYR, G.; MIN TJOA, A. Process Model Reuse to Promote Organizational Learning in Software Development. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 20., 1996. **Proceedings…** Los Alamitos: IEEE Computer Society Press, Aug. 1996. p. 21-26.

[EST96] ESTUBLIER, J.; DAMI, S. About Reuse in Multi-Paradigm Process Modelling Approach. In: BOEHM, B.; KELLNER, M.; PERRY, D. (Ed.) INTERNATIONAL SOFTWARE PROCESS WORKSHOP, 10., 1996, Ventron, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p. 77-79.

[EST99] ESTUBLIER, J. Is a Process Formalism an ADL? In: INTERNATIONAL PROCESS TECHNOLOGY WORKSHOP, IPTW, 1., 1999, Villars de Lans, France. **Proceedings…** [S.l.]: International Software Process Association (ISPA), 1999.

[FEI93] FEILER, P.; HUMPHREY, W. Software Process Development and Enactment: Concepts and Definitions. In: INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS, ICSP, 2., 1993. **Proceedings…** Berlin, Germany: IEEE Computer Society Press, 1993.

[FIN94] FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology.** Taunton: Research Studies Press, 1994.

[FRA94] FRAKES, W. Systematic Software Reuse: a Paradigm Shift. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 3., 1994, Rio de Janeiro. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1994.

[FRA99] FRANCH, X.; RIBÓ, J. Some Reflexions in Modeling of Software Process. In: INTERNATIONAL PROCESS TECHONOLOGY WORKSHOP, 1., IPTW, 1999, Villars de Lans, France. **Proceedings...** [S.l.]: International Software Process Association (ISPA), 1999.

[FRA2000] FRANCH, X.; RIBÓ, J. Searching for Expressiveness, Modularity, Flexibility and Standardisation in Software Process Modeling. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 14., 2000. **Anais...** João Pessoa, Brasil: Sociedade Brasileira de Computação, 2000.

[FRA2002] FRANCH, X.; RIBÓ, J. **Supporting Process Reuse in PROMENADE.** Barcelona: Departament de Llenguatges i Sistemes Informàtics, Universitat Politécnica de Catalunya, 2002. (Research Report, No. LSI-02-14-R). Disponível em: <http://www.lsi.upc.es/dept/techreps/html/R02-14.html>. Acesso em: fev. 2002.

[FUG2000] FUGGETTA, Alfonso. Software Process: A Roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2000, Limerick, Ireland. **Proceedings...** New York: ACM Press, 2000.

[FUN99] FUNK, P.J.; CRNKOVIC, I. Reuse, Validation and Verification of System Development Processes. In: INTERNATIONAL WORKSHOP ON THE REQUIREMENTS ENGINEERING PROCESSES, DEXA, 1999. Florence, Italy. **Proceedings...** Los Alamitos: IEEE Computer Society Press, Sept. 1999. Disponível em: <http://www.mrtc.mdh.se/publications/0022.pdf> Acesso em: dez. 2000.

[GAM94] GAMMA, E. et al. **Design Patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

[GIM93] GIMENES, I.M.; McDERMID, J.A. Investigating and Formalising the Development of HIS within PSEEs. In: SCHÄFER, W. (Ed.) INTERNATIONAL SOFTWARE PROCESS WORKSHOP, ISPW, 8., 1993, Wadern. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993, p.90-94.

[GIM94] GIMENES, I.M. **Uma Introdução ao Processo de Engenharia de Software:** Ambientes e Formalismos. Trabalho apresentado na Jornada de Atualização em Informática, 13., 1994, Caxambu.

[GOD99] GODART, C.; MOLLI, P.; PERRIN, O. Modeling and enacting processes: Some Difficulties. In: INTERNATIONAL PROCESS TECHNOLOGY WORKSHOP, IPTW, 1., 1999, Villars de Lans, France. **Proceedings…**[S.l.]: International Software Process Association (ISPA), 1999.

[GOS96] GOSLING, J.; JOY, Bill.; STEELE, Guy. **The Java Language Specification.** Disponível em: <http://java.sun.com/doc/books/jls/index.html>. Acesso em: mar. 2002.

[GRA96] GRANVILLE, L.Z.; SCHLEBBE, H. **Distributed PROSOFT**: management of tools and memory. Stuttgart: [s.n.], 1996. 26 p. (Technical Report)

[GNA2001] GNATZ, M.; MARSCHALL, F. **Towards a Living Software Development Process Based on Process Patterns**. Berlin: Springer-Verlag, 2001. (Lecture Notes in Computer Science, v. 2077).

[GNA2002] GNATZ, M. **1st Workshop on Software Development Process Patterns.** Call for Papers, 2002. Disponível em: <http://www.forsoft.de/zen/sdpp02/>. Acesso em abr. 2002.

[HAL90] HALL, A. Seven Myths of Formal Methods. **IEEE Software**, Los Alamitos, v. 7 n. 5 , p. 11-20, Sept. 1990.

[HEI91] HEIMBIGNER, D. **Software Process Modelling Example for ISPW7 Call for Participation.** 1991. Disponível em: <http://www.cs.colorado.edu/~dennis/ISPW7-Solutions/ispw7ex.pdf>. Acesso em: abr. 2002.

[HUM89] HUMPHREY, W.S. **Managing the Software Process.** New York: Addison-Wesley, 1989.

[ISO87] ISO-Information Processing Systems-Open Systems Interconection. **LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour**. [S.l.], 1987. (Technical Report, No. DIS 8807).

[ISO92] ISODA, S. Experience Report on Software Reuse Project: Its Structure, Activities and Statistical Results, In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 14., 1992, Melbourne. **Proceedings...** New York: ACM Press, 1992.

[JAC83] JACKSON, M. **System Development.** New York: Prenctice-Hall International, 1993.

[JAC95] JACCHERI, M.; AMERIO, E.; MALNATI, G. SENESI, P. **E3 p-draw: a tool to produce and reuse software process models.** Torino: Dipartamento de Automatica e Informatica, Politecnico de Torino. (Technical Report). Disponível em: <http://www2.umassd.edu/SWPI/e3/tool.pdf>. Acesso em: maio 2002.

[JAC98] JACCHERI, M.L.; PICCO, G.P.; LAGO, P. Eliciting Software Process Models with the E3 language. **ACM Transactions on Software Engineering and Methodology**, New York: ACM Press, v. 7, n. 4, p. 368 - 410, Oct. 1998.

[JØR2001] JØRGENSEN, H.; CARLSEN, S. **Writings in Process Knowledge Management: Management of Knowledge Captured by Process Models**. Oslo: SINTEF Telecom and Informatics, 2001. (Technical Report, No. STF40 A00011). Disponível em: <http://www.informatics.sintef.no/~hdj/>. Acesso em: jul. 2001.

[KAP98] KAPPEL, G.; RAUSCH-SCHOTT, S.; RETSCHITZEGGER, W. Coordination in Workflow Management Systems - A Rule-Based Approach. In: CONEN, W.; NEUMANN, G. (Ed.). Coordination Technology for Collaborative Applications: Organizations, Processes and Agents. Berlin: Springer-Verlag, 1998. (Lecture Notes in Computer Science, v. 1364).

[KEL98] KELLNER, M. Reuse of Process Elements. In: BOEHM, B.; KELLNER, M.; PERRY, D. (Ed.). In: INTERNATIONAL. SOFTWARE PROCESS WORKSHOP, 10., 1996, Ventron, France. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1998. p. 19-23.

[KLE98] KLEIN, M. Coordination Science: Challenges and Directions. In: CONEN, W.; NEUMANN, G. (Ed.). **Coordination Technology for Collaborative Applications**: Organizations, Processes and Agents. Berlin: Springer, 1998. (Lecture Notes in Computer Science, v. 1364).

[KÖR96] KÖRBES, F. **Implementação de um Mecanismo de Herança no PROSOFT.** 1996. Projeto de Diplomação (Bacharelado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[KRA98] KRAPP, C.A. **An Adaptable Environment for the Management of Development Processes**. 1998. Ph.D. Thesis (Computer Science Ph.D) - Aachen Universität, Germany.

[KRU96] KRUKE, V. **Reuse in Workflow Modeling.** 1996. Diploma thesis (Computer Science Ph.D) - Norwegian University of Science and Technology, Norway. Disponível em: <http://www.pvv.ntnu.no/~crukis>. Acesso em: jan. 2000.

[KRU2000] KRUCHTEN, P. **The Rational Unified Process**: An Introduction. 2nd ed. [S.l.]: Addison-Wesley, 2000.

[LIM98] LIMA REIS, C.A. **Um Gerenciador de Processos de Software para o Ambiente PROSOFT.** 1998. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[LIM2000] LIMA REIS, C.A. **Ambientes de Desenvolvimento de Software e seus Mecanismos de Execução de Processos de Software.** 2000. Exame de Qualificação (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[LIM2001a] LIMA REIS, C.A. A Abordagem APSEE para Modelagem e Gerência de Recursos em Ambientes de Processos de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 15., 2001. **Anais...** Rio de Janeiro: IME/SBC, 2001.

[LIM2001b] LIMA REIS, C.A. **Visit Report of a Research Mission in Germany: from January, 15th to March 23th, 2001.** Porto Alegre: CNPq, DLR, 2001. (Relatório de Pesquisa). Disponível em: <http://www.inf.ufrgs.br/~clima>. Acesso em abr. 2002.

[LIM2002a] LIMA REIS, C.A.; REIS, R.Q.; ABREU, A.; NUNES, D.J. APSEE: Um Modelo Formal e Flexível para Execução de Processos de Software. In: WORKSHOP IBEROAMERICANO DE INGENIERÍA DE REQUISITOS Y AMBIENTES SOFTWARE, IDEAS, 5., 2002. **Memórias...** Havana: CYTED, 2002.

[LIM2002b] LIMA REIS, C. A.; REIS, R.Q.; SCHLEBBE, H.; NUNES, D.J. A Policy-based Resource Instantiation Mechanism to Automate Software Process In: SOFTWARE ENGINEERING & KNOWLEDGE ENGINEERING, SEKE, 14., 2002, Ischia, Italy **Proceedings…** New York: ACM Press, Jul. 2002.

[LIM2002c] LIMA REIS, C.A.; REIS, R.Q.; SCHLEBBE, H.; NUNES, D.J. Resource Instantiation Policies in Software Process Environments. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 26., 2002. Oxford, England. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2002.

[LIM2002d] LIMA REIS, C.A. **APSEE**: **Um Modelo Flexível para Execução de Processos de Software**. 2002. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

[LON93] LONCHAMP, J. A Structured Conceptual and Terminological Framework for Software Process Engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE PROCESS, ICSP, 2., Berlin, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1993. p. 41-53.

[LOP97] LOPES, C. I. V. **D: a language framework for distributed programming.** 1997. Ph.D. Thesis (Computer Science Ph.D.) - Northeast University, Northeast, USA.

[LOT95] LOTT, C.M.; HOISL, B.; ROMBACH, D. The Use of Roles and Measurement to Enact Project Plans in MVP-S. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 4., 1995, Noordwijkerhout, Holland. **Proceedings…** Berlin: Springer-Verlag, 1995 (Lecture Notes in Computer Science, v. 913).

[LUD2001] LUDEWIG, J. **SESAM: A Software Project Simulator.** Disponível em: <http://www.informatik.uni-stuttgart.de/ifi/se/research/sesam/>. Acesso em: jan. 2001.

[MAL99] MALONE, T. et al. Tools for inventing organizations: Toward a handbook of organizational processes. **Management Science**, [S.l.], v. 45, n. 3, p. 425-443, March, 1999. Disponível em <http://citeseer.nj.nec.com/malone93tools.html>. Acesso em: jul. 2001.

[MAS2002] MASPEGHI Workshop on MAnaging SPEcialization/Generalization Hierarchies. Call for Papers. Disponível em: <http://www.lirmm.fr/~huchard/MASPEGHI/>. Acesso em: mai. 2002.

[MIC2002] MICROSOFT Visio. Disponível em: <http://www.microsoft.com/office/visio/>. Acesso em: 2002.

[MIL95] MILI, H. et al. Reusing Software: Issues and Research Directions. **IEEE Transactions on Software Engineering.** Los Alamitos, v. 21, n. 6, p. 528-562, Jun. 95.

[MOF93] MOFFETT, J.D.; SLOMAN, M. Policy Conflict Analysis in Distributed System Management. **Journal of Organizational Computing,** v. 4, n.1. p. 1-22, 1993. Disponível em: <http://www.cs.york.ac.uk/hise/bib.html>. Acesso em: nov. 2001.

[MOF94] MOFFETT, J.D.; McDERMID, J.A. Policies for Safety-Critical Systems: the Challenge of Formalisation. In: INTERNATIONAL WORKSHOP ON DISTRIBUTED SYSTEMS: OPERATIONS AND MANAGEMENT, DSOM, 5., 1994. **Proceedings…** Los Alamitos: IEEE Computer Society Press, Oct. 1994. Disponível em: <http://www.cs.york.ac.uk/hise/bib.html>. Acesso em: nov. 2001.

[MOR97] MORAES, S. W. **Um Ambiente Expert para Apoio ao Desenvolvimento de Software.** 1997. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[MÜN97] MÜNCH, J.; SCHMITZ, M.; VERLAGE, M. Tailoring groβer Prozeβmodelle auf der Basis von MVP-L. In: WORKSHOP DER FACHGRUPPE: VORGEHENSMODELLE - EINFÜHRUNG, BETRIEBLICHER EINSATZ, WERKZEUG-UNTERSTÜTZUNG UND MIGRATION, 4. **Proceedings…** [S.l: s.n.], 1997.

[MUR96] MURER, T.; WÜRTZ, A.; SCHERER, D.; SCHWEIZER, D. GIPSY: **Generating Integrated Process Support Systems:** Project Overview. 1996. Zürich: Institut für Technische Informatik und Kommunikationsnetze, Eidgenössische Technische Hochschule Zürich, Swiss Federal Institute of Technology. (Technical Report, No. 22). Disponível em: <http://www.tik.ee.ethz.ch/~murer/pubRaw/TIK-Report22.pdf>. Acesso em: abr. 2002.

[NGU97] NGUYEN, M.;WANG, A. Total Software Process Model Evolution in Epos (Experience Report). In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 1997, Boston, Massachusetts **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1997.

[NIS99] NISSANKE, N. **Formal Specfication:** Techniques and Applications. Berlin: Springer Verlag, 1999.

[NUN92] NUNES, D.J. Estratégia Data-driven no Desenvolvimento de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 6., 1992, Gramado. **Anais...** [S.l.]: Sociedade Brasileira de Computação, 1992, p. 81-95.

[NUN93] NUNES, D.J. **Verbesserung der software-qualitat durch verifikation der korrektheit der implementierung im projekt Prosoft** : ein vorschlag zur formalisierung des models. Stuttgart: Universität Stuttgart, 1993. (Technical Report, No. 1993/2.)

[NUN94] NUNES, D.J. **PROSOFT**: Um Ambiente de Desenvolvimento de Software baseado no Método Algébrico**.** Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1994. **(**Relatório de Pesquisa Interno). Disponível em: <http://www.inf.ufrgs.br/~prosoft>. Acesso em: jul. 2002.

[NUN2001] NUNES, I.D.; REIS, R.Q.; LIMA REIS, C.A.; NUNES, D.J. Componentes de percepção para o ambiente PROSOFT cooperativo. In: ENCUENTRO CHILENO DE COMPUTACIÓN - JORNADAS CHILENAS DE COMPUTACION, 9., 2001, Punta Arenas. **Anales...** Punta Arenas: Universidad de Magallanes, 2001.

[OST87] OSTERWEIL, L. Software Processes are Software Too. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 9. Monterey, CA. **Proceedings...** Los Alamitos: IEEE Computer Society Press. 1987.

[OST92] OSTERWAG, E.; HENDLER, J.; PRIETO DÍAZ, R.; BRAUN, C. Computing similarity in a reuse library system: an AI-based approach. **ACM Transactions on Software Engineering and Methodology**. New York: ACM Press, v. 1 , n. 3, 1992. p 205-228.

[PAU94] PAULK, M. **The Capability Maturity Model:** Guidelines for Improving the Software Process. [S.l.]: Addison-Wesley, 1994.

[PER91] PERRY, D.E. Policy-Directed Coordination and Cooperation. In: INT. SOFTWARE PROCESS WORKSHOP, ISPW, 7., Yountville, California **Proceedings…** [S.l.: s.n.], 1991. Disponível em: <http://www.ece.utexas.edu/~perry/work/papers/ispw7.ps.gz> Acesso em: abr. 2000.

[PER96] PERRY, D.E. Practical Issues in Process Reuse. In: INT. SOFTWARE PROCESS WORKSHOP, ISPW, 10., France. **Proceedings...** Los Alamitos: IEEE Computer Society Press, June 1996.

[PER97] PERRY, D.E. Using Process Modeling for Process Understanding. In: SOFTWARE PROCESS IMPROVEMENT, 1997, Barcelona, Spain. **Proceedings…** [S.l.: s.n.], 1997. Disponível em: <http://www.ece.utexas.edu/~perry/work/papers/spi97.ps.gz> Acesso em: abr. 2000.

[PRE2001] PRESSMAN, R.S. **Software Engineering:** A Practitioner's Approach. European Adaptation. London: McGraw-Hill, 2001.

[PRI94] PRIETO-DÍAZ, R. The Disappearance of Software Reuse. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 3., 1994, Rio de Janeiro. **Proceedings…** Los Alamitos, California: IEEE Computer Society Press, 1994.

[RAN2002] RANGEL, G. **Uma Ferramenta de Prototipação de Software em Ambiente de Especificação Formal.** 2002. Plano de Estudos e Pesquisa (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul. Porto Alegre.

[REI98] REIS, R.Q. **Uma Proposta de Suporte ao Desenvolvimento Cooperativo de Software no Ambiente PROSOFT.** 1998. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[REI99] REIS, R.Q.; NUNES, Daltro José. **Uma Avaliação dos Paradigmas de Linguagens de Processo de Software.** 1999. Trabalho Individual II (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[REI2000] REIS, R.Q.; NUNES, D.J. **Reutilização de Processos de Software**, 2000. Exame de Qualificação número 46 (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[REI2001a] REIS, R. Q.; LIMA REIS, C.A. NUNES, D.J. Automated Support for Software Process Reuse: Requirements and Early Experiences with the APSEE model. In: INTERNATIONAL WORKSHOP ON GROUPWARE, 7. Darmstadt, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2001.

[REI2001b] REIS, R.Q. **Report for Brazil-Germany Research Mission.** Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, CNPq-DLR, 2001. Disponível em: <http://www.inf.ufrgs.br/~quites>. Acesso em: mai. 2002.

[REI2001c] REIS, R.Q. **APSEE-StaticPolicy**: Sintaxe, semântica algébrica e exemplos de uma linguagem para verificação automática de políticas estáticas em modelos de processos de software. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2001. (Relatório de Pesquisa, No. 311).

[REI2001d] REIS, R.Q.; LIMA REIS, C.A.; NUNES, D.J. APSEE-StaticPolicy: Verificação de Políticas Estáticas em Modelos de Processos de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 15., 2000. **Anais...** [S.l.]: Sociedade Brasileira de Computação, 2000.

[REI2001e] REIS, R.Q. **Descrição da Metodologia INRECA para desenvolvimento de aplicações CBR através de Modelos Reutilizáveis de Processos de Software do ambiente APSEE**. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2001. (Relatório de Pesquisa, No. RP-322).

[REI2001f] REIS, R.Q. **A Similarity Mechanism to Assist the Retrieval of Reusable Software Process Models**, Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2001. (Relatório de Pesquisa Interno). Disponível em: <http://www.inf.ufrgs.br/~prosoft>. Acesso em: abr. 2002.

[REI2002a] REIS, R.Q.; LIMA REIS, C.A.; YAMIN, A.; AUGUSTIN, I.; NUNES, D.J.; GEYER, C. Towards a Software Process Model to Support the Design of Mobile Computing Applications. In: WORLD CONFERENCE ON INTEGRATED DESIGN & PROCESS TECHNOLOGY, IDPT, 6. Pasadena, CA. **Proceedings...** [S.l.]: Society for Design and Process Science, 2002.

[REI2002b] REIS, R.Q.; LIMA REIS, C.A.; SCHLEBBE, H.; NUNES, D.J. Automatic Verification of Static Policies on Software Process Models. In: WANG, Y.; BRYANT, A. (Ed.) **Annals of Software Engineering.** Special volume on Process-based Software Engineering. Boston, MA: Kluwer Academic Publishers, v. 14, Oct. 2002.

[REI2002c] REIS, R.Q.; LIMA REIS, C.A.; SCHLEBBE, H.; NUNES, D.J. APSEE-Reuse: Automated Support for Software Process Reuse. In: WORKSHOP IBEROAMERICANO DE INGENIERÍA DE REQUISITOS Y AMBIENTES SOFTWARE, IDEAS, 5., 2002. **Memórias...** Havana: CYTED, 2002.

[REI2002d] REIS, R.Q.; LIMA REIS, C.A.; SCHLEBBE, H.; NUNES, D.J. Towards an Aspect-Oriented Approach to Improve the Reusability of Software Process Models. In: INTERNATIONAL WORKSHOP ON EARLY ASPECTS: ASPECT-ORIENTED REQUIREMENTS ENGINEERING AND ARCHITECTURE DESIGN. Enschede, The Netherlands, Apr. 2002. **Proceedings…** New York: ACM Press. 2002.

[REI2002e] REIS, R.Q.; LIMA REIS, C.A.; SCHLEBBE, H.; NUNES, D.J. Early Experiences on Promoting Explicit Separation of Details to Improve Software Processes Reusability. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 26., 2002. Oxford, England. **Proceedings…** Los Alamitos: IEEE Computer Society Press.

[RIB2000] RIBEIRO, L. **Métodos Formais para Especificação**: Gramáticas de Grafos. In: ESCOLA DE INFORMÁTICA DA SBC SUL, 8. Maio, 2000. Santa Maria: Universidade Federal de Santa Maria, 2000.

[RIB2001] RIBÓ, J.M.; FRANCH, X. Building Expressive and Flexible Process Models using na UML-based approach. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 8., 2001. **Proceedings…** Berlin: Springer-Verlag, 2001. (Lecture Notes in Computer Science, v. 2077)

[RIB2002] RIBÓ, J.M.; FRANCH, X. **Supporting Process Reuse in PROMENADE.** Barcelona: LSI - Universitat Politècnica de Catalunya, 2002. (Technical Report, No. LSI-02-14-R) Disponível em: <http://griho.udl.es/josepma/tesi/report.pdf>. Acesso em: abr. 2002.

[ROM2000] ROMBACH, D. Die Experience Factor: Lernen in der Softwareentwicklung. In: WORKSHOP DER FACHGRUPPE, 6. Kaiserslautern, Germany. **Proceedings...** Kaiserslautern: Vorgehensmodelle, Fraunhofer IRB Verlag, 1999.

[ROM2001] ROMBACH, D. Comunicação pessoal realizada na Uni-Kaiserslautern. Kaiserslautern: IESE, 13 de fevereiro de 2001.

[ROM95] ROMBACH, D.; VERLAGE, M. Directions in Software Process Research. **Advances in Computers**, [S.l.: s.n.] v. 41, 1995.

[ROS85] ROSS, D. Applications and Extensions of SADT. **IEEE Computer**, New York, v. 18, n.4, p.25-35, Apr. 1985.

[SAT2001] SATYANARAYANAN, M. Pervasive Computing: Vision and Challenges. **IEEE Personal Communications**, Los Alamitos, v.8, n.4, p. 10-17, 2001.

[SCH94] SCHLEBBE, H. **Distributed PROSOFT** : report on a working stay at the institute of computer science of the state university of Rio Grande do Sul (UFRGS) at Porto Alegre, Brazil from May 1 to June 15, 1994. Stuttgart: Universität Stuttgart, Fakultät Informatik, 1994. (Technical Report).

[SCH97] SCHLEBBE, H.; SCHIMPF, S. **Reenginering of PROSOFT in Java**, Stuttgart: Universität Stuttgart, Fakultät Informatik, 1997. (Technical Report).

[SCH2002] SCHLEBBE, H. **Java-PROSOFT Manual.** Stuttgart: Universität Stuttgart 2002. Disponível em: <http://www.informatik.uni-stuttgart.de/ifi/bs/schlebbe/prosoft_doc/guide>. Acesso em: jul. 2002.

[SIL99] SILVA, F.; LIMA REIS, C.A.; REIS, R.Q.; NUNES, D.J. Um modelo de simulação de processos de software baseado em agentes cooperativos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 13. **Anais...** Florianópolis: UFSC, 1999. p. 163-178.

[SIL2000] SILVA, R.P. **Suporte ao desenvolvimento e uso de frameworks e componentes.** 2000. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SIL2001] SILVA, F. **Um Modelo de Simulação de Processos de Software Baseado em Conhecimento para o Ambiente PROSOFT.** 2001. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SOT2001] SOTO, M. **Evaluation of Existing Commercial Process Engines for Software Development**, Kaiserslautern: Fachbereich Informatik, Universität Kaiserslautern, Aug. 2001. (White paper)

[SOU2001] SOUSA, A.L.R.; LIMA REIS, C.A.; REIS, R.Q.; NUNES, D.J. **Interação Humana Durante Execução de Processos de Software:** Classificação e Exemplos. 2001. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul. (Relatório de Pesquisa, No. RP 310).

[SOU2002] SOUSA, A.L.R. **Uma Ferramenta de Apoio a Visualização e Representação de Modelos de Processos de Software.** 2002. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SUN02a] SUN Microsystems. **Java Runtime Environment.** Disponível em: <http://www.java.sun.com>. Acesso em: mar. 2002.

[SUN02b] SUN Microsystems. **JavaTM Remote Method Invocation (RMI).** Disponível em: <http://java.sun.com/products/jdk/rmi/>. Acesso em: mar. 2002.

[SUT97] SUTTON, S.; LERNER, B.; OSTERWEIL, L. **Experience Using the JIL Process Programming Language to Specify Design Process.** Amhest: Computer Science Department, University of Massachusetts, 1997. (Technical Report, No. 97-68). Disponível em: <http://laser.cs.umass.edu/~blemer/icse98-stan.ps>. Acesso em: mar. 2002.

[TOT95] TOTLAND, T.; CONRADI, R. A Survey and Classification of Some Research Areas Relevant to Software Process Modeling. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 4. Noordwijkerhout, The Netherlands. **Proceedings…** Berlin: Springer-Verlag, 1995.

[TUL89] TULLY, C. Representing and Enacting the Software Process. **ACM SigSoft Software Engineering Notes**. New York: ACM Press, v. 14, n. 4, June 1989.

[VAS97] VASCONCELOS JUNIOR, F.; WERNER, C.M. Suporte dos Padrões à Evolução de Processos de Desenvolvimento de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 11. **Anais...** Fortaleza: Sociedade Brasileira de Computação, 1997. p.131-146.

[VAN2000] VAN DER AALST, W.M.P.; JABLONSKI, S. Dealing with Workflow Change: Identification of Issues and Solutions. **International Journal of Computer Systems, Science, and Engineering**. London: CRL Publishers, v. 15, n. 5, 2000. p. 267-276. Disponível em: <http://tmitwww.tm.tue.nl/staff/wvdaalst/Publications/publications.html>. Acesso em: dez. 2001.

[WAN2000] WANG, Y.; KING, G. **Software Engineering Processes**: Principles and Applications. Boca Raton: CRC Press, 2000.

[WAT91] WATT, D. **Programming Language Syntax and Semantics.** New York: Prentice-Hall, 1991.

[WEL99a] WELLEN, U.; GRUHN, V. Software Process Landscaping: An Approach to Structure Complex Software Processes. In: INTERNATIONAL PROCESS TECHNOLOGY WORKSHOP, IPTW, 1., 1999, Villars de Lans, France. **Proceedings…** [S.l.]: International Software Process Association (ISPA), 1999.

[WEL99b] WELLEN, U.; GRUHN, V. Process Landscaping: Modelling Complex Business Processes. **European Journal of Engineering for Information Society Applications.** v. 1, n. 3, May 1999. Disponível em: <http://www.ejeisa.com/nectar/journal/03/012.htm>. Acesso em: abr. de 2002.

[YAM2001] YAMIN, A. C., AUGUSTIN, I., BARBOSA, J. L. V., SILVA, L. C., GEYER, C. F. R. Explorando o Escalonamento no Desempenho de Aplicações Móveis Distribuídas In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 2001, Pirenópolis, Goiás. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2001. p. 1-8.

[YAS94] YASUMOTO, K.; HIGASHINO, T.; TANIGUCHI, K. Software Process Description using LOTOS and Its Enaction. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 16. **Proceedings...** New York: ACM Press. Disponível em: <http://www.acm.org/pubs/citations/proceedings/soft/257734/p169-yasumoto>. Acesso em: feb. 1999.

[YE2000] YE, Y.; FISCHER, G.; REEVES, B. Integrating active information delivery and reuse repository systems. In: INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE, 8. San Diego, California. **Proceedings…** New York: ACM Press, 2000. p. 60-68.

[YOU95] YOURDON, E. **Declínio e Queda dos Analistas e Programadores.** São Paulo: Campus, 1995.

[VAS98] VASCONCELOS JUNIOR, F.M.; WERNER, C.M.L. Organizing the Software Development Process Knowledge: An approach based on Patterns. In: **International Journal of Software Engineering and Knowledge Engineering**. [S.l.]: World Scientific Publishing, v. 8, n. 4, 1998. pp. 461-482.

[ZIR95] ZIRBES, S.F. **A Reutilização de Modelos de Requisitos de Sistemas por Analogia: Experimentação e Conclusões**. 1995. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.