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

CARLA ALESSANDRA LIMA REIS

Uma Abordagem Flexível para Execução de Processos de Software Evolutivos

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Dr. Daltro José Nunes
Orientador

Porto Alegre, maio de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Reis, Carla Alessandra Lima

Uma Abordagem Flexível para Execução de Processos de Software Evolutivos / Carla Alessandra Lima Reis – Porto Alegre: Programa de Pós-Graduação em Computação, 2003.

277f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR-RS, 2003. Orientador: Nunes, Daltro José.

1. Modelos de processos de software 2. Execução de processos de software. 3. Ambientes de desenvolvimento de software. 4. Especificação algébrica. 5. Gramáticas de Grafos. I. Nunes, Daltro José. II. Título.

Para Rodrigo,
Adriana e José Antônio.

AGRADECIMENTOS

Agradeço a Deus, por ter terminado esse trabalho e por todas as bênçãos que recebo.

Ao meu marido Rodrigo, que me acompanhou de perto durante todo esse trabalho, e me ajudou, passando força, alegria e amor. Muito Obrigada por tudo! Um agradecimento especial à nossa filhinha Amanda, que se comportou direitinho na minha barriga para que eu terminasse o trabalho.

Aos meus pais José Antônio e Adriana que me incentivaram sempre e estiveram presentes em todos os momentos da minha vida. E também aos meus irmãos André e Vanessa. Obrigada pelas palavras de incentivo quando eu mais precisei.

Ao meu orientador, Prof. Daltro José Nunes, pela orientação, pela confiança no meu trabalho e por todos os ensinamentos passados nesse período de convívio.

Aos membros do grupo PROSOFT, em especial ao Heribert Schlebbe, do Instituto de Informática da Universidade de Sttutgart e ao Marcelo Abreu, bolsista do projeto, por toda ajuda na implementação do PROSOFT e do APSEE e por inúmeras sugestões ao trabalho.

A todos os amigos que tornaram o período em Porto Alegre mais divertido e nos farão voltar sempre (mesmo no inverno). Em especial a Duda e Lincoln, que se tornaram meus irmãos e sempre incentivaram e apoiaram em todos os momentos. Agradeço também a Idy, que foi grande companheira de conversas sobre assuntos que não tinham a ver com a tese.

Agradeço a CAPES e a Universidade Federal do Pará e ao seu Departametno de Informática pelo apoio financeiro e incentivo fundamental à realização desse trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADS	Ambiente de Desenvolvimento de Software
CASE	*Computer Aided Software Engineering* - Engenharia de Software Auxiliada por Computador.
CSCW	*Computer Supported Cooperative Work* – Trabalho cooperativo apoiado por computador
ECA	Evento-Condição-Ação
NAC	*Negative Application Condition* - Condição negativa de aplicação
PML	*Process Modeling Language* – Linguagem de modelagem de processos
PSEE	*Process-centered Software Engineering Environment* – Ambiente de desenvolvimento de software centrado em processo.
SGBD	Sistema de Gerência de Banco de Dados
SEE	*Software Engineering Environment* – Ambiente de Desenvolvimento de Software
UML	*Unified Modeling Language* – Linguagem de Modelagem Unificada
WfMS	*Workflow management systems* – Sistemas de Gerência de Workflow

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Buscando aumentar a qualidade de software, a área de Engenharia de Software tem produzido ferramentas para auxílio ao desenvolvimento de software assim como tem estudado e produzido formas de controlar o processo de desenvolvimento. A tecnologia existente para coordenação de atividades humanas - incluindo sistemas de *Workflow* e PSEE (*Process-Centered Software Engineering Environments*) - possui algumas limitações. Uma das principais é a falta de flexibilidade. Algumas características importantes de processos de software não têm sido levadas em consideração pela tecnologia disponível, como por exemplo: o aspecto humano envolvido, a dificuldade em definir todo o processo antes de executá-lo, o tratamento de mudanças durante a execução, dentre outras. Além disso, quando se considera a construção de PSEEs, nota-se a necessidade de uma abordagem formal adequada, que permita um melhor entendimento, análise e comparação com outros modelos.

Este trabalho apresenta e discute um modelo conceitual e mecanismos para contribuir para o aumento da flexibilidade e do nível de automação fornecidos para execução de processos. Para atingir os objetivos do trabalho, foi proposta uma arquitetura, denominada APSEE, construída a partir de um meta-modelo unificado que integra informações organizacionais, gerenciais, sobre artefatos do processo, e sobre ferramentas do ambiente. O modelo proposto define de forma rigorosa seus componentes. Assim, as propriedades de recursos e pessoas são descritas visando melhorar o controle de sua alocação, permitindo a construção de um mecanismo de apoio à instanciação de recursos e pessoas em processos. Além disso, uma linguagem de modelagem visual de processos de software é proposta. Por fim, é fornecido o mecanismo de execução de processos que permite modificações dinâmicas, verifica a consistência dessas modificações, e permite acompanhamento da execução de processos pelo gerente. Os diferentes componentes envolvidos na definição do modelo APSEE proposto foram especificados formalmente através de método algébrico e também através da abordagem de gramáticas de grafos, constituindo uma base semântica de alto nível de abstração que deu origem a um conjunto de protótipos implementados como ferramentas do ambiente PROSOFT-Java. A implementação também serviu para constatar a viabilidade do uso do APSEE como plataforma de integração para vários serviços de gerência de processos desenvolvidos como atividades de pesquisa no contexto do grupo PROSOFT.

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: PSEE, Processo de Software, Tecnologia de Processo de Software, Instanciação de Processos de Software, Execução de Processos de Software.

A Flexible Approach to Evolvable Software Process Enactment

ABSTRACT

Software Engineering evolved to increase software quality through the definition of tools to support both development and management processes. However, the existing technology to provide automated support for human activity coordination - mainly represented by Workflow Management Systems and Process-Centered Software Engineering Environments (PSEEs) - has some limitations. One of the most important limitations is related to the low level of flexibility provided by current tools. Existing technologies provide limited support for some important software process characteristics, which includes, for example, the humanistic aspect of software processes, the difficult to completely prescribe the process model in advance, and the lack of adequate support for dynamic changes on enacting processes. Besides, the construction of a PSEE demand an adequate formal approach for its specification, which can improve its understandability and analysis, while constitutes a basis for comparison with similar proposals.

This work presents and discusses a conceptual model and mechanisms which jointly aim to increase the level of flexibility and automation provided for software processes enactment. In order to reach this goal, a software-based architecture is proposed, named APSEE, which provides a unified meta-model that integrates organizational and management information, along with software artifact and tool support. The proposed meta-model rigorously defines its components. Therefore, resource and people properties are formally described in order to allow better allocation through an automated process instantiation mechanism. In addition, a visual software process modeling language is provided which is, in turn, related to the underlying meta-model. Finally, a flexible process enactment mechanism was specified to support dynamic changes on process models that work together with consistency check and monitoring mechanisms. The required software components for the proposed meta-model were specified using algebraic specification and graph grammar-based techniques which, in turn, were used to build prototypes for the Java-PROSOFT environment. This implementation was also useful to evaluate the feasibility of using APSEE as an integration platform for a number of process management services developed by PROSOFT research group.

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

Keywords: PSEE, Software Process, Software Process Technology, Software Process Instantiation, Software Process Enactment

1 INTRODUÇÃO

Buscando aumentar a qualidade de software, a área de Engenharia de Software tem produzido ferramentas para auxílio ao desenvolvimento de software, assim como tem estudado e produzido formas de controlar o processo de desenvolvimento. Recentemente o foco da área de Engenharia de Software foi deslocado de uma visão orientada a laboratório para uma visão mais orientada à indústria. Essa tendência reflete as necessidades da indústria de software de integrar técnicas de desenvolvimento de software com metodologias organizacionais e gerenciais para formar ambientes que atuem na coordenação de processos de software (WANG; KING, 2000).

Construir software é uma atividade cooperativa, que depende da qualidade de comunicação entre os desenvolvedores e os gerentes de desenvolvimento. Uma das áreas de maior destaque neste contexto é a Tecnologia de Processos de Software, que envolve a construção de ambientes e ferramentas que atuam na modelagem, execução[1], simulação e evolução de processos de desenvolvimento de software. Portanto, o processo de software corresponde ao conjunto de atividades realizadas desde a concepção até a liberação do produto de software. Uma forma de analisar e amadurecer tal processo ocorre através da sua descrição em um modelo de processo de software, a qual permite que o processo seja analisado, compreendido e automatizado (executado).

Um ADS (Ambiente de Desenvolvimento de Software) que permite a modelagem e execução de processos de software é denominado um ADS orientado ao processo (GIMENES, 1994) ou PSEE (*Process-Centered Software Engineering Environment*). A tecnologia existente para coordenação de atividades (incluindo sistemas de *Workflow* e PSEEs) possui algumas limitações. Uma das principais - destacadas por Fuggetta (2000) e Arbaoui et al. (2002) - é a falta de flexibilidade. Nesse sentido, observa-se que algumas características importantes de processos de software não têm sido levadas em consideração pela tecnologia disponível, como por exemplo: o aspecto humano envolvido, a dificuldade em definir todo o processo antes de executá-lo, o tratamento de mudanças durante a execução, dentre outros. Além disso, quando se considera a construção de ambientes para gerência do processo de software, nota-se a necessidade

[1] O termo utilizado mais freqüentemente na literatura é *process enactment*, que significa que o processo não será totalmente automatizado como sugere o termo execução, mas sim executado por pessoas e máquinas. Neste trabalho usaremos o termo “execução” por ser mais utilizado na literatura nacional. O termo “encenação” de processo também pode ser encontrado na literatura com o mesmo significado.

de uma abordagem formal adequada ao problema em questão, que permita um melhor entendimento, análise e comparação com outros modelos.

Essa tese de doutorado está inserida nesse contexto e tem como objetivo principal contribuir com o amadurecimento da tecnologia de processos de software através da integração de vários mecanismos de gerência de processos em um meta-modelo comum para auxiliar os usuários de PSEEs durante a gerência de processos. As seções a seguir mostram o contexto, introduzindo os problemas abordados, motivações para a sua solução e aponta a abordagem escolhida para atingir os objetivos estabelecidos.

1.1 Contexto e Terminologia Adotada

O trabalho intensivo na área de processo de software vem gerando muitas definições e terminologias. Alguns esforços no sentido de padronizar esta terminologia foram apresentados em (DOWSON, 1991; FEILER; HUMPHREY, 1993; LONCHAMP, 1993). Além disso, existem diversos paradigmas de modelagem de processo de software, bem como de execução de tais modelos. Essa seção descreve o contexto e a terminologia básica utilizada no texto.

1.1.1 Influências sobre a Tecnologia de Processos de Software

A tecnologia de processos de software tem suas origens relacionadas com as primeiras propostas de um ciclo de vida para o software, isto é, a definição de um conjunto bem definido de estágios que forneçam uma referência para o acompanhamento do desenvolvimento.

A tecnologia de ferramentas CASE (*Computer Aided Software Engineering*) foi precursora da cultura de construir ferramentas para apoiar o desenvolvimento de software. Esta tecnologia culminou no surgimento de ambientes de desenvolvimento de software, os quais fornecem uma base para integrar ferramentas CASE usadas durante todo o processo de software (NUNES, 1992). Mais recentemente, surgiu a necessidade de incorporar mecanismos para o controle do processo de desenvolvimento através dos ADSs. Os PSEEs surgiram neste contexto, permitindo a modelagem e a execução de processos de software, assim como o contínuo aperfeiçoamento do modelo executado.

Tem sido de grande importância para a área a definição de modelos de avaliação do processo de engenharia de software em organizações. As organizações de desenvolvimento de software têm procurado atender aos requisitos de tais modelos a fim de terem seu processo de desenvolvimento avaliado de forma positiva frente ao mercado, o que tem aumentado a adoção da tecnologia de processos de software. O CMM (*Capability Maturity Model*) (PAULK et al., 1994) e o SPICE (*Software Process Improvement and Capability Determination*) (EMAN et al., 1998) são exemplos de tais modelos.

Outro elemento importante para o progresso deste setor é a rápida proliferação de modelos de processos industriais ou acadêmicos, como por exemplo, o *Unified Process* (JACKOBSON et al., 1999), Programação Extrema (BECK, 2000), SCRUM (SCHWABER, 2001) e Catalysis (D'SOUZA, 2000), que são descrições complexas de propósito geral que precisam ser entendidas para serem adaptadas e usadas efetivamente em projetos de software. Portanto, as ferramentas de processo de software buscam

automatizar tarefas e auxiliar na tomada de decisão de gerentes de projetos (i.e., responsáveis pelo acompanhamento de projetos) e engenheiros de processo de software (i.e., responsáveis por determinar a estratégia organizacional).

1.1.2 Ambientes de Desenvolvimento de Software Orientados ao Processo

A Tecnologia de Processos de Software propõe o desenvolvimento e adoção de PSEEs para automatizar a gerência dos processos (GIMENES, 1994). Esta tecnologia traz benefícios, como por exemplo: melhor comunicação entre as pessoas envolvidas e a consistência do que está sendo feito; realização de algumas ações automáticas, "liberando" os seus usuários de tarefas repetitivas; fornecimento de informações sobre o andamento do processo quando necessário; possibilidade de reutilização de processo de software e a coleta automática de métricas (importantes para o controle e aperfeiçoamento de processos).

A gerência de processos de software está relacionada à tecnologia da coordenação (KLEIN, 1998; CONEN; NEUMANN, 1998), que depende de uma camada que forneça serviços de colaboração e outra de comunicação. Em um PSEE, a camada de comunicação deve permitir o compartilhamento de informações no ambiente. A camada de colaboração deve permitir aos participantes a atualização colaborativa de um conjunto de decisões compartilhadas. A camada de coordenação, por sua vez, deve garantir que as ações colaborativas dos indivíduos são coordenadas para obter o resultado desejado de forma eficiente.

Um modelo de processo descreve como a camada de coordenação deve trabalhar e é construído através de uma linguagem de modelagem do processo (PML – *Process Modeling Language*).

A literatura especializada apresenta diversos PSEEs. Dentre eles, pode-se citar: MARVEL (KAISER et al., 1988), EPOS (CONRADI et al., 1994), SPADE (BANDINELLI et al., 1994b), ENDEAVORS (TAYLOR; BOLCER, 1996) e DYNAMITE (HEIMAN et al., 1997). Em (GARG; JAZAYERI, 1996) são reunidos alguns dos artigos "clássicos" da área e nos trabalhos de (DERNIAME et al., 1999; GRUHN, 2002; ARBAOUI et al., 2002) são reunidos diversos aspectos mais recentes da tecnologia de processos de software. No país, algumas propostas foram desenvolvidas no âmbito de ambientes de modelagem e gerência de processos de software. Dentre elas, pode-se citar o ambiente ExPSEE (GIMENES, 2000) que permite uma "definição precisa e instanciação de atividades do processo de software em relação a artefatos, atores (agentes) e ferramentas utilizadas por atividades", e a Estação TABA (TRAVASSOS, 1994), que possui um "modelo recente para definição de processos que permite definir recursos humanos e ferramentas, dentre outros conceitos" (MACHADO, 2000).

1.1.3 CSCW, Workflow e Processos de Software

Diversas tecnologias têm surgido para auxiliar o envolvimento cooperativo de profissionais. Neste contexto, CSCW (*Computer Supported Cooperative Work*), *Workflow*, e Processos de Software se destacaram como tecnologias interrelacionadas que estabeleceram comunidades próprias na Ciência da Computação.

Há divergência na literatura acerca das fronteiras exatas entre estas três tecnologias (DUITSHOF, 1995; KRUKE, 1996; GEORGAKOPOULOS et al., 1994; OCAMPO; BOTELLA, 1998). Entretanto, pode-se afirmar que elas cooperam entre si, apesar de apresentarem muitas vezes terminologias conflitantes, fruto do desenvolvimento em paralelo (OCAMPO; BOTELLA, 1998). Além disso, seu relacionamento próximo é ainda evidenciado por terem como áreas principais três elementos chave comuns: comunicação, colaboração e coordenação (KRUKE, 1996).

Uma tendência atual é a concentração de esforços na uniformização dos conceitos e experiências nas três áreas (IPTW, 1999). Entretanto, a autora entende que processos de software possuem requisitos próprios geralmente mais complexos que processos organizacionais (tratados por sistemas de *workflow*). Por exemplo, uma importante característica de processos de software está nas mudanças no produto e processo durante o seu desenvolvimento. Sistemas de *workflow* são mais facilmente encontrados por terem conquistado uma parcela maior da indústria, e mais recentemente estão sendo propostos sistemas de *workflow* para suporte ao processo de software (TAYLOR; BOLCER, 1996; KAPPEL et al., 1998), que neste trabalho são tratados como PSEEs.

Finalmente, é inegável que os avanços na área de *workflow* podem beneficiar processos de software e vice-versa. Portanto, a proposta apresentada aqui leva em consideração tecnologias e ambientes desenvolvidos pelas duas áreas, com o objetivo de apoiar a definição e automação de processos em diferentes contextos.

1.1.4 Terminologia

No que concerne à terminologia a ser adotada no trabalho, cabe definir neste momento o conceito de recurso no contexto do modelo proposto. Apesar da questão da gerência de recursos de software ser bastante estudada em vários níveis (desde os sistemas operacionais até o nível de aplicação), não existe um consenso na literatura de Engenharia de Software acerca desse conceito. Em (LONCHAMP, 1993), o termo é definido como sendo qualquer pessoa ou coisa necessária para a realização de um passo do processo. Já em (WESTFECHTEL, 1999) um recurso é definido como qualquer entidade que realiza ou apóia a realização de uma atividade, o que exclui os documentos utilizados. Levando em consideração estas e outras descrições incluídas em (FUGGETTA; WOLF, 1996), pode-se definir três tipos principais de recursos diferentes listados na literatura:

- **Agentes ou Recursos humanos**: são as pessoas envolvidas no processo, no desenvolvimento ou na gerência das atividades;
- **Artefatos de Software**: documentos, componentes de software (inclusive código) e qualquer informação necessária para a realização da atividade ou produzida por esta;
- **Recursos de Apoio**: são todos os recursos que apóiam a realização de uma atividade da organização e podem ser compartilhados, como impressoras; consumidos, como recursos financeiros; ou de uso exclusivo como salas e recursos computacionais.

Apesar das definições da literatura levarem a esta classificação genérica de recursos, não é possível tratar todos os tipos da mesma forma. É necessário distinguir as características de cada tipo a fim de atender os requisitos de um sistema de gerência de processos, seja este de software ou de negócios. Não se pode considerar genericamente que quaisquer recursos são alocados e liberados para atividades, pois existem recursos que são totalmente consumidos e deixam de existir após seu uso (como recursos financeiros, por exemplo). Os artefatos de software, também considerados recursos por alguns autores (tais como (LERNER et al., 2000)), são, segundo a visão deste trabalho, utilizados, transformados ou produzidos por atividades, e não alocados para as mesmas.

Portanto, o termo recurso será utilizado aqui para denotar os recursos de apoio mencionados anteriormente. Os agentes e os artefatos de software, por terem características e necessidades específicas, merecem um tratamento conceitual e mecanismos de gerência diferenciados, que serão tratados neste trabalho. Com relação a outros termos a serem adotados que não têm problemas de consenso na literatura, estes serão apresentados no próximo capítulo.

1.2 Motivações

As sub-seções a seguir apresentam as principais motivações para o trabalho.

1.2.1 Importância da Tecnologia de Processos de Software

No Brasil, estudos publicados por (WEBER et al., 1995) indicam que, em virtude da baixa quantidade de profissionais capacitados e da enorme demanda por seus serviços, a indústria nacional de software é ainda incipiente e, muitas vezes, adota práticas artesanais durante o processo. Entretanto, é importante ressaltar que o desenvolvimento de software de qualidade, por sua vez, possui um enorme impacto para a economia de países em desenvolvimento, através da geração de empregos e receita para uma atividade de grande demanda em todo o mundo. Países como a Índia, cuja indústria de software, segundo (MOITRA, 2001), recebeu o reconhecimento global e movimentou 5,7 milhões de dólares americanos no ano fiscal de 1999-2000, consistem atualmente em pólos de exportação de produtos de alta tecnologia para os Estados Unidos, Europa e Japão.

Ainda segundo (MOITRA, 2001), um dos grandes obstáculos para o surgimento de uma indústria de software de porte em países em desenvolvimento consiste na disponibilização de ferramental tecnológico que atue decisivamente na gerência dos processos conduzidos nas organizações, controlando prazos, gerenciando a alocação de recursos e mantendo controle sobre os produtos resultantes, trazendo para esta área muito da disciplina que influencia o desenvolvimento bem sucedido de indústrias de outras áreas. Tais ferramentas, quando disponíveis, envolvem custos que ultrapassam em muitas vezes a capacidade de investimentos das organizações. Mesmo assim, algumas publicações nacionais reconhecem a importância do tópico (tais como (MOLINA, 2001)) e apostam na maciça adoção dessa tecnologia no país, em um futuro próximo.

A experiência internacional tem demonstrado que qualquer iniciativa de automação do processo de desenvolvimento de software é seguida de grandes benefícios econômicos para as organizações participantes (FUGGETTA; WOLF, 1996). Os

benefícios variam desde a simples constatação e solução de dificuldades específicas, relacionadas com a produtividade em projetos em andamento, até a certificação, a partir de avaliações formais, de que as empresas participantes atingiram um reconhecimento internacional da qualidade dos seus produtos e processos. Apesar destes claros benefícios, a grande maioria dos projetos acadêmicos nesta área não é utilizada amplamente pela indústria. Infelizmente, muitos ambientes propostos neste contexto ficam restritos a laboratórios e centros de pesquisa.

1.2.2 Algumas Limitações da Tecnologia Atual

A tecnologia atual para coordenação de atividades possui algumas limitações. Processos de software são orientados a pessoas, e as interações decorrentes dessa característica (pessoa-pessoa e pessoa-ferramenta) são freqüentemente imprevisíveis e muito variáveis. Este fato aumenta a complexidade do próprio processo e da gerência da sua execução. As atividades a serem coordenadas são muitas vezes criativas e a sua execução não pode ser totalmente definida antes do processo iniciar. Mudanças são necessárias no decorrer do processo e os ambientes devem tratar essas mudanças como parte do processo, permitindo que escolhas sejam feitas no decorrer da execução. A representação do processo através de linguagens de modelagem deve ser orientada a pessoas, fornecendo conceitos em alto nível de abstração, mas permitindo automação e adaptação. A seguir são resumidos alguns problemas encontrados na tecnologia de PSEEs que serviram de motivação para o presente trabalho:

- Os ambientes existentes têm, na maioria dos casos, um comportamento muito prescritivo. O modelo de processo é rígido e mudanças não são facilmente toleradas. O desenvolvedor deve obedecer à prescrição de um processo que foi desenvolvido sem levar em consideração a característica criativa da tarefa. A baixa aceitação de PSEEs, segundo (FUGGETTA, 2000), tem levado a um questionamento acerca da falta de flexibilidade dos mesmos. Ainda segundo Fuggetta, as PMLs devem permitir especificação de processos incompletos e informais, de forma incremental, mas que possam ser enriquecidos com detalhes, tornando-o mais formal quando necessário;
- Os PSEEs, tais como os *Workflow management systems* (WfMS), são sistemas de coordenação de tarefas que, por sua vez, competem por recursos limitados. Assim, a execução de processos depende da disponibilidade desses recursos e a especificação precisa desses recursos permite a análise e otimização da sua utilização. A maioria das abordagens encontradas na literatura, mesmo em avaliações de ambientes mais recentes como em (DERNIAME et al., 1999), não trata em profundidade a definição rigorosa dos recursos necessários para execução das atividades (LERNER et al., 2000; PODOROZHNY, 1999), o que impede análise e otimização de sua alocação. A modelagem precisa de recursos e sua gerência é um dos aspectos menos desenvolvidos da área de processos de software, segundo (HUFF, 1996). O raciocínio sobre recursos requer um entendimento das similaridades e diferenças entre os mesmos, assim como as necessidades precisas das atividades a serem coordenadas. Com essa informação é possível identificar em quais situações apenas um recurso em particular

pode ser adequado para a atividade. Outra motivação consiste na possibilidade de alterar o processo a ser executado para, por exemplo, substituir um recurso indisponível por outro compatível. Além disso, a especificação precisa pode facilitar a decisão sobre quando é necessário acrescentar ou adquirir um determinado recurso para tornar uma atividade mais rápida; ou ainda para gerenciar a alocação de recursos muito usados.

- Outra limitação corresponde à falta de mecanismos para auxiliar a escolha de desenvolvedores para tarefas específicas. Habilidades têm sido sugeridas para servirem de critério na alocação de pessoas (PLEKANOVA, 1998). No entanto, diversos fatores podem afetar a produtividade das pessoas, e a tomada de decisão deve basear-se em informações precisas sobre as características das pessoas, suas afinidades para atividades cooperativas e sobre seu histórico na organização. A análise e otimização da alocação de pessoas é um fator importante para aumentar a qualidade do software resultante, pois, segundo Pressman (2001), o fator determinante para essa qualidade é a adequação das pessoas envolvidas. Além disso, estratégias da organização devem ser adotadas pelo ambiente na alocação de pessoas. Por exemplo, se a organização tem como política alocar o desenvolvedor com mais habilidade para uma tarefa que está atrasada, então essa estratégia deve ser conhecida e obedecida pelo PSEE;
- O uso de conhecimento de informações sobre processos executados anteriormente na organização também é limitado na tecnologia atual. A coleta de métricas durante a execução de processos é um requisito muito citado para PSEEs (BANDINELLI et al., 1994b] e para desenvolvimento de software em geral (ROCHA et al., 2001), porém a utilização eficiente deste recurso nem sempre é tratada, o que tem diminuído o interesse em manter o esforço de coletar métricas. Esse conhecimento pode ser útil para auxiliar a tomada de decisão na alocação de pessoas e recursos no processo e, além disso, pode servir de base para simular a execução do processo de forma mais precisa. Desse modo, são necessários não somente mecanismos para coletar métricas, mas a integração de um modelo de métricas ao modelo de gerência de processos.

1.3 Objetivos Gerais

Levando em consideração as motivações apresentadas na seção anterior, o objetivo desse trabalho é incrementar o nível de automação da gerência de processos de software propondo uma abordagem flexível para execução de processos de software, através da proposta de um meta-modelo diferenciado que integre informações para atender diversas ferramentas de gerência de processos de software. Assim, os objetivos gerais do trabalho são:

- Especificar um meta-modelo conceitual que auxilie na construção e manipulação de modelos de processos de software, e que permita a integração de mecanismos para modelagem, instanciação e execução de processos de software. Além disso, pretende-se propor um mecanismo de execução que atenda aos requisitos do envolvimento humano em processos

de software, apoiando a execução de forma flexível e integrada à modelagem e instanciação;

- Demonstrar a viabilidade do modelo conceitual e mecanismos propostos através de um protótipo, isto é, uma implementação limitada que forneça a funcionalidade básica descrita pelo modelo conceitual;
- Mostrar que alguns exemplos são suportados pelo modelo proposto e que podem ser executados segundo os objetivos específicos do trabalho.

1.4 Contexto do Trabalho no Grupo de Pesquisa PROSOFT

O trabalho aqui apresentado está sendo desenvolvido no contexto do projeto PROSOFT (NUNES, 1992; 1994). Uma recente linha de pesquisa do projeto surgiu a partir do trabalho desenvolvido em (LIMA, 1998) e envolve a definição de ferramentas para apoiar a modelagem, execução, simulação (SILVA et al., 1999; SILVA, 2001; REIS et al., 2000a), e visualização (SOUZA, 2001a) de processos cooperativos (REIS 1998a; REIS et al., 1998b) de desenvolvimento de software, constituindo um gerenciador integrado de processos de software denominado APSEE (LIMA REIS et al., 2000b; REIS, 2002f; LIMA REIS et al., 2001c). Portanto, o trabalho aqui proposto está inserido no projeto PROSOFT, e busca definir um conjunto de ferramentas que possam ser integradas às restantes do ambiente, apoiando especificamente a execução de processos de software.

1.5 Organização do texto

O presente trabalho está organizado como segue: O capítulo 2 fornece uma visão geral sobre Execução de processos de Software através da apresentação de conceitos, formalismos, aspectos de interação com usuários, aspectos de flexibilidade e requisitos de execução. O capítulo 3 apresenta os objetivos específicos do trabalho e a visão geral do meta-modelo APSEE, juntamente com discusssões sobre as decisões tomadas nos componentes do meta-modelo. O capítulo 4 apresenta os mecanismos de gerência de processos propostos para apoiar modelagem, instanciação e execução de processos. O capítulo 5 apresenta a especificação formal do modelo e dos mecanismos propostos. O capítulo 6 apresenta o protótipo construído a partir da especificação do capítulo anterior. O capítulo 7 apresenta alguns estudos de caso sobre execução e instanciação de processos. O capítulo 8 apresenta as conclusões, análise da pesquisa, trabalhos relacionados e trabalhos futuros.

Foram adicionados apêndices ao texto a fim de apoiar a leitura do mesmo. Os apêndices estão organizados da seguinte forma: O apêndice A apresenta os tipos de dados do APSEE no paradigma PROSOFT; O apêndice B apresenta as regras de sintaxe da linguagem APSEE-PML; O apêndice C apresenta as regras de execução do APSEE; O apêndice D apresenta a especificação do interpretador de Políticas de Instanciação; O apêndice E apresenta as primitivas para definição de condições lógicas no APSEE.

2 EXECUÇÃO DE PROCESSOS DE SOFTWARE

A **execução de processos de software** é uma fase do ciclo de vida de processos de software na qual as atividades[2] modeladas são realizadas tanto pelos desenvolvedores (quando demandam agentes humanos) quanto automaticamente (quando demandam a invocação de ferramentas autônomas). Portanto, esta fase envolve questões importantes acerca de planejamento, controle, monitoração, garantia de conformidade com o processo modelado, treinamento, segurança e recuperação do processo (FEILER; HUMPHREY, 1993).

Neste capítulo será apresentada uma visão geral da execução de processos, interação com os usuários e questões acerca de flexibilidade no processo de software. O capítulo inclui, ainda, uma descrição das outras etapas do ciclo de vida de processos de software que estão intimamente relacionadas com a etapa de execução de processos. Além disso, são apresentados os requisitos da execução de processos considerados no trabalho.

2.1 Conceitos

A **execução de processos de software** ocorre quando um modelo de processo está pronto para execução (isto é, o processo está instanciado) e que leva em consideração a coordenação dos desenvolvedores, a interação entre as ferramentas e desenvolvedores, a garantia de que o processo está sendo executado conforme modelado, dentre outras questões dispersas em (LONCHAMP, 1993; DERNIAME et al., 1999; FEILER; HUMPHREY, 1993). Com o apoio de um ambiente de desenvolvimento de software é possível obter resultados relevantes acerca do andamento da execução além de melhorar a gerência do desenvolvimento. Portanto, apesar de não ser totalmente automatizada, a fase de execução depende de um mecanismo automatizado que compreenda o processo modelado e oriente os desenvolvedores (agentes) no decorrer do seu trabalho, assim como seja responsável por executar automaticamente algumas tarefas repetitivas.

Um **modelo de processo de software** é uma descrição abstrata do processo de software. Vários tipos de informação devem ser integrados em um modelo de processo de software para indicar quem, quando, onde, como e por que os passos são realizados (LONCHAMP, 1993). Para representar um modelo de processo de software é utilizada uma linguagem de modelagem do processo de software (PML). As **linguagens de processo de software** possuem algumas características específicas, que as distanciam

[2] Este capítulo, a menos quando explicitamente informado, pressupõe a modelagem de processos orientada a atividades.

das linguagens de propósito geral, como por exemplo, a necessidade de permitir a descrição de atividades que necessitam da interação humana para serem concluídas. Diversas classificações são encontradas na literatura com o objetivo de facilitar o estudo das linguagens de processo de software (REIS, 1999).

Um **modelo de processo instanciado** ou **processo executável** é um modelo de processo que apresenta características específicas relacionadas ao contexto da organização de desenvolvimento de software envolvida e aos prazos reais do projeto. São exemplos de informações necessárias para instanciação de processos: a alocação de recursos, a definição dos desenvolvedores e a descrição de prazos das atividades. O processo instanciado é o resultado da fase de instanciação no ciclo de vida de processos. A partir de um processo abstrato podem ser criados vários processos instanciados.

O **Mecanismo de Execução de processos ou Máquina de Execução** (*Process Engine*) interpreta o modelo de processo instanciado de acordo com a semântica da linguagem de modelagem (HUFF, 1996), gerenciando as informações do ambiente e orientando os desenvolvedores de acordo com esse modelo. Um dos objetivos principais desse mecanismo é manter o estado da execução do processo consistente com o estado real da realização das tarefas. Além disso, monitoração e coleta de métricas sobre o processo são atividades relacionadas com esse mecanismo.

De forma geral, durante a execução é necessário obter ***feedback*** da realização do processo necessário para manter a consistência entre o estado da execução e o estado da realização. Este *feedback* pode ser obtido de várias formas: o usuário pode agir explicitamente de modo a "avisar" que concluiu uma atividade; questionamentos podem ser feitos para o usuário sobre o andamento do processo; uma ferramenta pode gerar um evento automaticamente; uma consulta ao banco de dados do ambiente pode obter informações sobre artefatos criados; ou podem ser gerados eventos que sinalizem o andamento das atividades. Entretanto, na maioria dos casos, algum tipo de decisão do usuário é necessária para prover informação sobre eventos significativos.

A figura 2.1 ilustra os relacionamentos do mecanismo de execução com outros elementos de um PSEE. Na camada superior, rotulada como **Interação com o Usuário**, estão agrupadas as diferentes visões fornecidas para os usuários do sistema. A interação deve ser útil para que os usuários forneçam *feedback* acerca do andamento do processo: por exemplo, os desenvolvedores devem ser capazes de informar início e término de uma atividade, enquanto que os gerentes devem poder realizar modificações em processos em execução. Algum controle de consistência deve ser fornecido a fim de restringir as alterações dos usuários e manter a consistência geral dos processos em execução.

A interpretação de um processo executável pode implicar na ativação de ferramentas específicas, as quais estão agrupadas no item **Ferramentas** da figura. Os artefatos produzidos, manipulados e modificados a partir da execução do processo devem ser armazenados de maneira estruturada em um **Repositório**. A alocação de **Recursos** deve levar em consideração as necessidades expressas nos modelos de processos. Finalmente, o item **Métricas/Histórico** é útil para registrar métricas acerca da organização e seus processos, os quais podem ser consultados para avaliação do seu desempenho global.

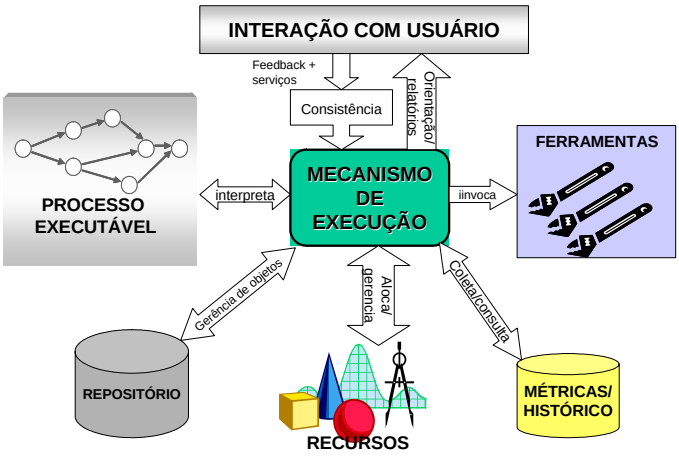


Figura 2.1: Interações entre o mecanismo de execução e outros componentes do PSEE (LIMA REIS, 2000).

A execução de processos de software está fortemente relacionada com a teoria da coordenação (GREENWOOD, 1995). De fato, segundo Malone e Crowston (1994), coordenação é a gerência das dependências entre atividades, e essas dependências podem ser: compartilhamento de recursos, relacionamentos de produção/consumo, restrições de simultaneidade e decomposição de tarefas. Assim, conforme Greenwood (1995), as dependências propostas por Malone e Crowston devem ser tratadas por mecanismos de execução de processos.

Para que um processo tenha suas dependências gerenciadas, sua modelagem deve prever as possíveis dependências entre atividades. Assim, o mecanismo de execução é fortemente influenciado pelo paradigma de modelagem utilizado e constituem, em última instância, os interpretadores das linguagens de modelagem. Por isso, a classificação de paradigmas de modelagem de processos de software também poderia ser adotada em execução. As diferenças entre um mecanismo de execução e outro utilizando o mesmo paradigma de modelagem geralmente recaem sobre a abordagem de interação com o usuário, o tratamento de segurança e autorização, a ênfase em monitoramento do processo através de métricas e a forma de interação com as outras ferramentas do ambiente, ou seja, questões de arquitetura do ambiente.

2.1.1 Domínios de Processos de Software

Existe uma distinção entre as definições de processo (estáticas) e a sua execução dinâmica. O envolvimento humano adiciona a imprevisibilidade na execução do processo, podendo gerar discordâncias entre as definições e a execução real (como por exemplo, atrasos e alterações nos processos para adicionar uma nova característica não prevista). Desta forma, existem três diferentes domínios de processos de software, apontados em (DOWSON; FERNSTRÖM, 1994): o domínio das definições de processo, o domínio da realização do processo e o domínio da execução da definição do processo.

O domínio das definições do processo contém modelos de processos ou fragmentos desses modelos, expressos em alguma notação. O domínio da realização do processo engloba as atividades ou ações conduzidas por pessoas e agentes não-humanos

(programas). O domínio da execução do processo definido preocupa-se com o que é necessário em um ambiente de desenvolvimento de software para suportar a execução de uma definição de processo. Na figura 2.2 são ilustrados os domínios de processo e seus relacionamentos.

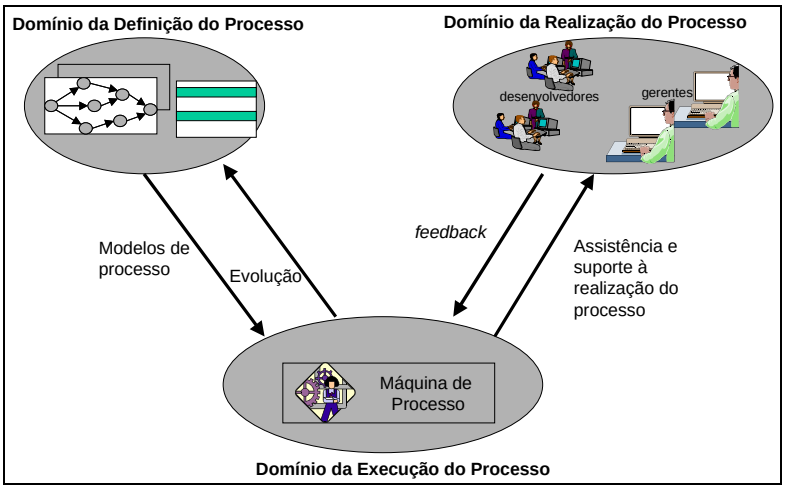


Figura 2.2: Domínios de Processos de Software (DOWSON; FERNSTRÖM, 1994).

2.1.2 A Execução no Ciclo de Vida de Processos de Software

Processos de software têm uma natureza evolucionária, devido à necessidade de melhoria e correção contínua e devido à instabilidade do ambiente operacional. Existe um ciclo de vida para processos de software análogo ao ciclo de vida de produtos de software. As atividades deste ciclo são chamadas de meta-atividades, e o ciclo de vida é chamado de meta-processo de software (DERNIAME et al., 1999).

Existem diversas propostas de ciclo de vida para processos de software. Por exemplo, Nguyen e Conradi (1994) mostram uma classificação de meta-processos, propõem fases para o meta-processo e comparam alguns ambientes europeus quanto à adoção das fases propostas. Por outro lado, Derniame et al. (1999) propõe um ciclo de vida de referência para processos de software chamado *PROMOTER Reference Model*.

A figura 2.3 mostra uma representação de um ciclo de vida de processos de software que inclui a fase de execução de processos.

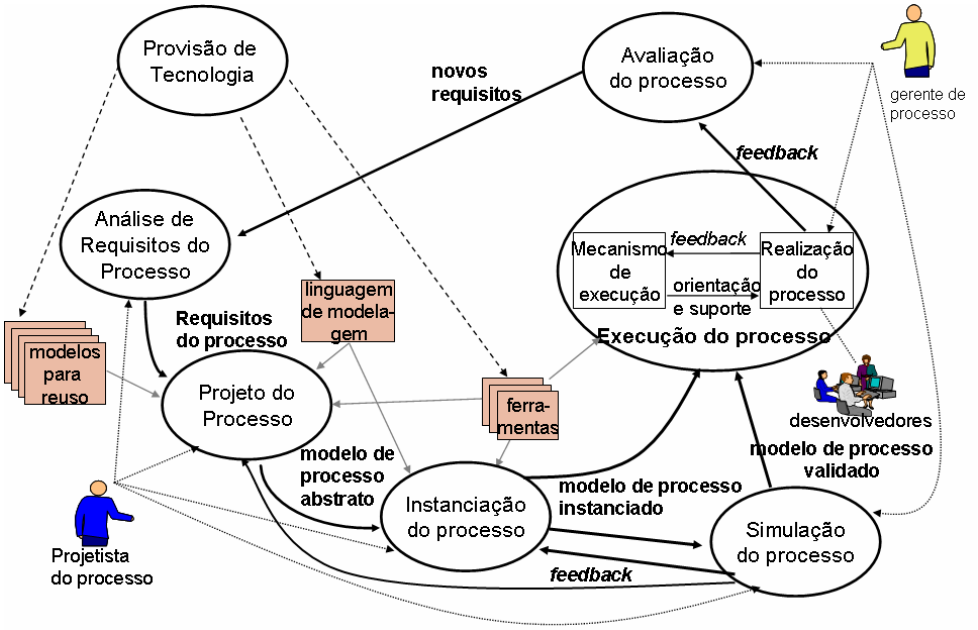


Figura 2.3: Ciclo de Vida de Processos de software.

As fases do meta-processo apresentado na figura 2.3 estão definidas em alto nível de abstração, de forma que cada fase pode ser decomposta em sub-fases de granularidade fina.

No ciclo de vida apresentado, a **Provisão de Tecnologia** inclui o fornecimento de tecnologia de suporte a produção de software e de modelos de processo (como as linguagens de modelagem de processo, modelos de processo prontos para reutilização e ferramentas para aquisição, modelagem, análise, projeto, simulação, evolução, execução e monitoração de modelos de processo); a **Análise de Requisitos do Processo** identifica requisitos para o projeto de um novo processo, ou novos requisitos para um processo existente. Os requisitos resultantes especificam os recursos e propriedades que o processo deve oferecer; O **Projeto do Processo** provê a arquitetura geral e detalhada do processo. Nesta etapa as linguagens de modelagem do processo são utilizadas; A **Instanciação do Processo** modifica a especificação do processo produzida pela atividade anterior acrescentando informações detalhadas sobre os prazos, agentes e recursos utilizados por cada atividade definida no processo; A **Simulação do Processo** permite verificação e validação dos processos definidos antes da execução; A **Execução do modelo de processo** utiliza o processo instanciado e o executa através da invocação de ferramentas para guiar e assistir a realização do processo no mundo real. Informações sobre o andamento do processo (*feedback*) são coletadas e analisadas durante a execução; A **Avaliação do Processo** provê informação quantitativa e qualitativa que descreve o desempenho de todo o processo em execução. A avaliação pode ocorrer em paralelo com a execução do modelo de processo e as informações adquiridas podem ser utilizadas nas futuras ocorrências da atividade de análise de requisitos.

No meta-processo é fundamental a participação de agentes humanos que operam na fase de execução do processo. Além disso, são necessários um projetista de processo, que é o responsável por descrever o processo a ser executado, e um gerente do processo

que deverá acompanhar a execução e a avaliação do processo, analisando seu desempenho. Em geral, os PSEEs estabelecem seu meta-processo através da provisão de tecnologia e paradigmas adotados para modelagem e execução de processos.

2.1.3 Questões tratadas pela execução

As questões tratadas pela execução de processos foram discutidas por vários autores, dentre eles Bandinelli et al. (1992), Finkelstein et al. (1994), Feiler e Humphrey (1993) e Dowson (1993). Todos concordam que o mecanismo de execução de um ADS deve tratar:

- **Automação de Processo**: Apoiar a coordenação de atividades e ativar automaticamente as atividades que podem ser executadas sem intervenção humana, através de uma integração com as ferramentas do ambiente;
- **Trabalho Cooperativo**: Apoiar a cooperação de pessoas trabalhando em um projeto de software através da orientação e assistência durante todo o ciclo de desenvolvimento;
- **Monitoração**: Prover diferentes visões do estado da execução do processo, permitindo que o gerente de projetos obtenha informações sobre o andamento real das atividades;
- **Registro da história do processo**: Coletar dados da evolução do processo para permitir que o processo melhore onde houver necessidade e seja corrigido para atender novos requisitos.

Existem também alguns aspectos que diferenciam mecanismos de execução e que devem ser levados em consideração na sua construção. Esses aspectos relacionam-se com a forma de tratar a influência humana na execução de processos, a preservação da consistência e recuperação de inconsistências, a capacidade de permitir modificação dinâmica (reflexão da linguagem de modelagem), a integração com ferramentas do ambiente, o controle de acesso a artefatos do processo, as versões dos artefatos e sua consistência, o tratamento das transações longas e das versões de modelos de processos. Portanto, as características de cada mecanismo de execução podem ser obtidas pelas decisões tomadas com relação aos aspectos citados.

A seção a seguir apresentará os principais formalismos de execução de processos encontrados na literatura.

2.2 Formalismos de Execução

O mecanismo de execução deve ser construído com base na semântica da linguagem de modelagem de processos adotada. Algumas vezes este componente trabalha com um formalismo considerado de baixo nível, que é obtido a partir de uma tradução do formalismo de alto nível da modelagem. Entretanto, a maioria dos PSEEs usa o mesmo formalismo considerado de baixo nível para a modelagem e execução de processos. Em (ARMENISE et al., 1992) são identificadas as principais abordagens de modelagem de processos e apresentada uma comparação entre vários ambientes e seus formalismos. Outras classificações também podem ser encontradas em (REIS, 1999) e (AMBRIOLA

et al., 1997) e, mais recentemente em (ARBAOUI et al., 2002). É importante notar que muitas abordagens não podem ser classificadas em um único paradigma, e são conhecidas como multi-paradigma.

Huff (1996) considera que uma nova geração de linguagens de modelagem de processos tem sido produzida, onde os seus construtores sintáticos são projetados especificamente para as particularidades de processos de software. Seguindo esta tendência, as linguagens de modelagem e formalismos de execução estão se tornando mais direcionados ao domínio de processos de software. Ainda segundo Huff, apesar de existirem formalismos básicos de execução (alguns são apresentados nesta seção), ainda é necessário trabalhar na integração de vários paradigmas para atender a grande diversidade de processos existentes. Outras necessidades apontadas são a importância do elemento humano no processo e a necessidade de notações gráficas de alto nível com mapeamento para uma semântica formal (HUFF, 1996; FUGGETTA, 2000).

Os formalismos de execução (e modelagem) mais conhecidos e relacionados a este trabalho são apresentados sucintamente nas seções a seguir.

2.2.1 Execução Procedimental

A execução procedimental considera que processos de software devem ser modelados em paradigmas similares à programação convencional, considerando a idéia introduzida por Osterweil de que um processo de software também é software (OSTERWEIL, 1987).

Neste paradigma, um processo é modelado através de instruções passo a passo que descrevem como executar o processo. Estruturas de controle - derivadas de linguagens de programação - especificam a ordem dos passos do processo. O programa de processo pode ser traduzido diretamente para código executável para conduzir o processo de desenvolvimento.

Em comparação com as abordagens baseadas em redes (incluindo Redes de Petri[3]), este paradigma é considerado de baixo nível. Todas as situações do processo têm que ser especificadas antes da execução iniciar. A execução de processos nesse caso é equivalente à execução de um programa. Entretanto, para lidar com a natureza dinâmica e não-determinística dos processos de software, as linguagens deste paradigma têm acrescentado o suporte a regras ECA (Evento-Condição-Ação, descritas na seção 2.2.3).

APPL/A (SUTTON et al., 1990) é a linguagem de modelagem do ambiente Arcadia e provê modelagem e execução procedimental. Trata-se de uma linguagem de programação baseada em Ada com extensões para *triggers*, relações persistentes, predicados no estado das relações e comandos de transações. LITTLE JIL (CASS et al., 2000) é outro exemplo de linguagem procedimental que foi proposta como a linguagem de segunda geração do grupo de pesquisa de Osterweil.

---

[3] Existe uma vasta bibliografia sobre Redes de Petri. Por exemplo, o site http://www.daimi.au.dk/PetriNets/ (acesso em jan./2003) contém informações indexadas sobre o assunto e a lista de bibliografias importantes.

2.2.2 Execução Baseada em Regras

Neste paradigma, as atividades do processo são modeladas como regras com pré e pós-condições e o mecanismo de execução se assemelha a uma máquina de inferência de um sistema especialista. Os ambientes Marvel (KAISER et al., 1988; 1990), Merlin (JUNKERMANN et al., 1994) e Amber (TONG et al., 1994) utilizam esse paradigma. A execução ou simulação de processos nestes ambientes é realizada pelas estratégias *forward* e *backward chaining* com algumas particularidades em alguns ambientes. Marvel, por exemplo, permite desligar os encadeamentos (*backward* ou *forward chaining*) quando for conveniente.

EPOS (CONRADI et al., 1994) é um ambiente multiparadigma onde o modelo de processo é representado através de redes de tarefas, *triggers* e regras. O paradigma de regras é usado no EPOS para permitir o encadeamento na rede de tarefas, pois cada tarefa possui vários tipos de pré-condições. Uma desvantagem dessa abordagem é que apesar do gerente poder monitorar a execução das regras, o processo de desenvolvimento geral não fica visível e é de difícil monitoração.

2.2.3 Execução Baseada em Regras ECA (Evento-Condição-Ação)

Este formalismo difere da execução baseada em regras devido à possibilidade de detectar eventos antes da execução da regra. Nesta abordagem o mecanismo de execução verifica a ocorrência de eventos e a pré-condição das regras, podendo assim disparar as regras que satisfazem a condição e o evento gerando, por sua vez, novos eventos a serem tratados.

O ambiente Adele (ESTUBLIER; CASALLAS, 1994; BELKHATIR et al., 1994) é um exemplo desse tipo de execução. A máquina de execução do Adele foi desenvolvida para suportar também máquinas de estado, Redes de Petri e abordagens orientadas a metas como no Marvel (KAISER et al., 1988) Além disso, é possível definir diferentes prioridades para os eventos e regras baseadas nos eventos.

Algumas deficiências dessa abordagem apontadas em (CONRADI et al., 1994) são: o formalismo é difícil para entendimento humano; há dificuldade de controlar a execução; faltam conceitos de alto nível como processo, espaço de trabalho, planejamento, entre outros; há dificuldade em responder questões simples como "*Onde estamos?*" e "*Qual a próxima tarefa?*". Entretanto, deve-se ressaltar que regras ECA têm sido utilizadas como formalismo de execução (baixo nível) para sistemas de *workflow* (SCHAL, 1998). Neste caso, a especificação do processo é realizada através de uma linguagem de alto nível, que é mapeada para regras ECA. *TrigsFlow* (KAPPEL et al., 1998) e APEL (ESTUBLIER et al., 1997) são exemplos dessa abordagem.

2.2.4 Execução Baseada em Redes de Petri

Nesta abordagem é utilizado o formalismo matemático de Redes de Petri. Uma Rede de Petri é um grafo direcionado com dois tipos de nodos: lugares e transições. Os lugares representam os predicados, objetos ou tipos de recursos enquanto as transições representam os passos do processo. Na modelagem de processos de software, artefatos e recursos são geralmente representados por *tokens* inseridos nos lugares. Os arcos expressam dependência entre lugares e transições e podem ser rotulados com pesos.

O comportamento de uma Rede de Petri é definido por regras para suas transições. A regra descreve quais *tokens* devem estar presentes nos lugares de entrada de uma transição e quais condições devem ser satisfeitas para os lugares de saída da transição a ser habilitada. Durante o disparo de uma transição, os *tokens* de entrada são removidos dos lugares de entrada e novos *tokens* são colocados nos lugares de saída. Através desse formalismo, é possível representar não-determinismo, visto que um lugar pode ser entrada para várias transições, e paralelismo. Deste modo, muitos ambientes adotam Redes de Petri para fornecer a semântica de execução de processos: SPADE (BANDINELLI et al., 1992) e *ProcessWeaver* (CHRISTIE, 1995; FERNSTRÖM, 1993) constituem exemplos deste uso.

O ambiente SPADE possui a linguagem SLANG baseada em *ER nets* que, segundo (BANDINELLI et al., 1992), são Redes de Petri temporizadas de alto nível. As ferramentas podem ser associadas às transições, o que acarreta sua chamada quando a transição for disparada. Existem também lugares de entrada do usuário que permitem a comunicação com os desenvolvedores. Cada instância de uma atividade é executada por um interpretador SLANG independente. Assim, durante a execução podem existir múltiplas cópias (de instâncias) executando concorrentemente, e os interpretadores podem trocar informações através de lugares na Rede de Petri compartilhados por várias atividades. Em SLANG, a consistência é garantida porque uma transição é sempre executada como uma transação atômica. Segundo Bandinelli et al. (1995), SLANG foi aplicada a vários processos de desenvolvimento industriais para detectar inconsistências, incompletude e oportunidades de melhoria dos processos adotados. Tais investigações demonstraram que os formalismos baseados em Redes de Petri ainda são muito complexos para serem compreendidos por usuários não técnicos (BANDINELLI et al., 1995).

2.2.5 Execução Baseada em Redes de Tarefas

Um modelo de processos baseado em redes de tarefas consiste em um grafo direcionado que representa a estrutura do processo. Em geral, os nodos representam atividades enquanto os arcos representam o fluxo de controle e de dados entre atividades. EPOS (CONRADI et al., 1994) e Dynamite (HEIMAN et al., 1997) são exemplos de ambientes que adotam esse paradigma. É importante notar que a semântica de execução para este paradigma não é necessariamente baseada em nenhum formalismo já existente, como Redes de Petri ou regras (considerados de baixo nível (ESTUBLIER et al., 1997)). Em geral, a semântica é definida em função dos tipos de fluxos de controle e dados providos pela linguagem, assim como em função dos construtores da linguagem propostos no seu meta-modelo. O ambiente Dynamite, por exemplo, provê uma semântica baseada em transformações de grafos.

O ambiente EPOS trabalha com redes de tarefas, regras e *triggers* e possui uma arquitetura em camadas composta pelo EPOSDB (um banco de dados), a linguagem de modelagem SPELL, a Estrutura de Atividades e Modelos de Processo de Aplicações. A camada de Estrutura de Atividades (*Tasking Framework*) permite a definição e a execução concorrente de redes de tarefas. A execução propriamente dita é realizada por um Gerenciador de Execução (*Execution Manager*) que gerencia os estados das atividades.

Dynamite (HEIMAN, 1997) utiliza uma representação gráfica baseada em redes de tarefas hierárquicas para apoiar o gerenciamento de processo de software. Neste ambiente, o gerente pode planejar o projeto, realizar algumas alterações nos modelos em execução, assinalar dinamicamente tarefas aos desenvolvedores, e organizar os artefatos dos processos. Os arcos da rede podem representar tanto o fluxo de controle entre atividades quanto o fluxo de dados.

2.3 Interação com Usuários Durante a Execução de Processos

Durante a execução de processos de software, as pessoas envolvidas devem receber orientação e assistência na realização de suas atividades, sem interferência no processo criativo (NGUYEN; WANG, 1997). O funcionamento do mecanismo de execução e algumas características do meta-modelo adotado por um PSEE são influenciados pelo tipo de interação escolhida/requerida para o ambiente. A fraca interação com os usuários de PSEEs tem sido apontada como causa para a falta de sucesso desses ambientes (FUGGETTA, 2000). Nesta seção são apresentados resumidamente os aspectos de interação com os usuários de PSEEs, os quais foram apresentados com mais detalhes em (SOUZA et al., 2001a; 2001b).

De modo geral, os usuários dos PSEEs podem estar atuando em dois papéis principais: gerente e desenvolvedor. **Gerente** é a denominação geral usada para designar aqueles que têm como responsabilidade a monitoração, gerência e manutenção do modelo de processo sendo executado, enquanto o **Desenvolvedor** atua no desenvolvimento de software realizando atividades que contribuem diretamente com a produção do mesmo. Ambos interagem com o PSEE durante a execução de um processo, porém possuem objetivos diferentes. A tabela 2.1 sintetiza os tipos de interação durante a execução, os objetivos e as ferramentas utilizadas por cada papel de usuário.

Tabela 2.1: Características dos tipos de Interação de usuários em PSEEs (adaptado de (SOUZA et al., 2001))

	Interação com o Gerente	**Interação com o Desenvolvedor**
Objetivos e papéis dos usuários	-Visão macro dos eventos que ocorrem durante a execução de processos. -Acompanhamento e monitoração dos eventos (desempenhados por desenvolvedores), prazos e recursos. -Manipulação do processo (ajustes durante a execução)	-Recebem orientação acerca das tarefas a serem desempenhadas. -Fornecem *feedback* sobre a performance
Ferramentas utilizadas	-Ferramentas para monitoração do processo -Ferramentas para modelagem de processos e planejamento de atividades	-Ferramentas CASE -Ferramentas de interação do PSEE (por exemplo, agendas, espaço de trabalho, etc.)

Durante a execução de processos, as questões básicas do gerente são relacionadas ao presente (por exemplo, "*Neste instante, qual a situação do(s) processo(s) em execução?*"), ao passado ("*Que eventos levaram a esta situação?*") e ao futuro ("*Quais as conseqüências de uma modificação na modelagem ou instanciação de um processo em execução?*"). Questões adicionais incluem "*Como o desempenho dos desenvolvedores envolvidos afeta o andamento dos processos?*" e "*Quais os recursos disponíveis e/ou mais adequados?*".

Os principais requisitos relacionados à interação com gerentes coincidem com os requisitos gerais de PSEEs encontrados dispersos em (CURTIS et al., 1992; YOUNG, 1994; GIMENES, 1994; AVRILIONIS et al., 1996). São eles:

- **Permitir diferentes visões de processos**: refere-se ao fornecimento de perspectivas com informações diferenciadas sobre o processo em andamento e operadores de visualização que permitem monitoração do processo. As perspectivas mais comuns são a funcional, comportamental, organizacional, informacional e tecnológica (CURTIS et al., 1992). Exemplos de PSEEs que fornecem esse tipo de informação são ProcessWeaver (CHRISTIE, 1995; FERNSTRÖM, 1993; CERN, 1996), Dynamite (HEIMAN, 1997) e Endeavors (YOUNG, 1994; ENDEAVORS, 2000);
- **Mostrar o estado atual e o histórico do processo**: esta característica permite que o ambiente forneça opções de reversibilidade, registro das decisões e navegação sobre o estado presente e passado do processo. Os ambientes *Dynamite* (HEIMAN, 1997) e EPOS (CONRADI et al., 1994) permitem reversibilidade e estruturam as informações através de hiperdocumentos (Dynamite) e versões (EPOS e Dynamite).
- **Permitir modificação dinâmica do processo durante a execução:** essa característica é declarada como sendo fornecida pela maioria dos ambientes. Exemplos são *Dynamite*, EPOS, *ProcessWeaver*, Endeavors e SPADE. Entretanto a abordagem para tratamento de mudanças no processo nem sempre trata as conseqüências das mudanças, deixando a cargo do gerente as decisões sobre como tratar essa propagação e manter o processo consistente;
- **Fornecer independência do formalismo de modelagem de processos:** essa característica está bastante relacionada com as diferentes visões do mesmo processo. Porém aqui a ênfase está em permitir diferentes representações para o processo independente do formalismo original utilizado para modelagem. Por exemplo, uma representação textual para um formalismo gráfico ou ainda uma notação diferenciada para representar detalhes que não são visíveis no formalismo original. Os ambientes *ProcessWeaver*, *Dynamite*, SPADE e EPOS fornecem esse tipo de independência;
- **Fornecer monitoração de eventos e tratamento de exceções na execução de processos**: Gerentes necessitam perceber exceções e estabelecer mecanismos para prever um tratamento automático (uso de *triggers*) de acordo com políticas da organização, ou restrito a um projeto

ou tipo de processo. A maioria dos ambientes utiliza *triggers* ou regras ECA para monitoração de eventos e tratamento de exceção.

Do ponto de vista do desenvolvedor também há necessidade de interação adequada por parte do PSEE. Manter o processo em execução sincronizado com a realização é o objetivo da interação do PSEE com desenvolvedores. Apesar da dificuldade em obter essa sincronização, as informações fornecidas pelo desenvolvedor (*feedback* do processo) devem ser as mais precisas possíveis, sem aumentar excessivamente a carga de trabalho e o excesso de informação para o desenvolvedor.

Segundo Fuggetta (2000), a pouca aceitação da tecnologia de PSEEs tem sido creditada à dificuldade de convencer desenvolvedores a usá-los e, segundo Kobialka e Lewerentz (1998), isto está fortemente relacionado ao paradigma de interação adotado.

A interação com o desenvolvedor revela uma questão importante de controle: "*Quem controla o processo: o PSEE ou o desenvolvedor?*". Este relacionamento pode ser altamente prescritivo (o PSEE dita as ações dos desenvolvedores) ou variar até uma forma mais branda de controle e aconselhamento (mais liberdade para realizar tarefas).

Durante a execução de processos, as questões básicas do desenvolvedor são "*Onde estou?*" e "*Qual o próximo passo?*". Questões adicionais são "*Que eventos levaram a essa situação?*" ou ainda "*Como o trabalho dos outros afeta o meu?*". A interface do PSEE deve ajudar a responder essas questões e permitir que o *feedback* necessário seja informado. Deste modo, alguns requisitos dos PSEEs durante interação com os desenvolvedores (definidos a partir dos propostos em (BANDINELLI et al., 1996; KOBIALKA; LEWERENTZ, 1998; POHL, 1999)) são listados a seguir:

- Orientar desenvolvedores na realização de suas tarefas;
- Fornecer visualização adequada das tarefas do processo;
- Obter *feedback* do andamento do processo;
- Fornecer visualização dos estados do processo (atual e anteriores) e mecanismo de *undo*;
- Facilitar comunicação e cooperação com outros desenvolvedores;
- Flexibilizar a interação (adequação do paradigma de interação ao usuário).

O desenvolvedor necessita visualizar, selecionar e realizar ações sobre itens de trabalho. Existem diversas abordagens e classificações quanto à orientação do desenvolvedor (DOWSON; FERNSTRÖM, 1994; AMBRIOLA, 1997; BANDINELLI et al., 1994a; BANDINELLI et al., 1996). Porém, observa-se uma falta de padronização nos conceitos e classificações fornecidos. Por isso, cabe aqui distinguir entre o **tipo de orientação ao desenvolvedor** e o **paradigma de interação com o desenvolvedor**. O primeiro define **COMO** o PSEE se relaciona com o desenvolvedor, ou seja, qual o grau de controle do desenvolvimento das tarefas. O segundo especifica as principais metáforas e conceitos que são visíveis aos desenvolvedores, ou seja, **O QUE** o desenvolvedor visualiza quando interage com o PSEE. As seções a seguir apresentam essa distinção.

2.3.1 Tipos de Orientação de Processo

Os tipos de orientação de processo (ou tipos de suporte à execução) definem o grau de controle da máquina de processos sobre os desenvolvedores. Dowson e Fernström (1994) identificaram os tipos de orientação aos desenvolvedores durante a execução de processos da seguinte forma:

- **Orientação passiva**: provê aos desenvolvedores, quando solicitado, informações para ajudar a decidir as ações a serem tomadas de acordo com a definição do processo. Também permite que o usuário tenha uma visão do estado atual do processo ou uma capacidade de explorar os efeitos de ações hipotéticas;
- **Orientação ativa**: A informação é oferecida sem interferência do usuário nas circunstâncias especificadas na modelagem. Um exemplo de orientação ativa é a adição de tarefas em uma lista de tarefas do usuário ou a notificação de eventos importantes;
- **Obrigação do processo**: Força usuários a executar o processo de acordo com a modelagem. A única maneira de obter isso é controlando o acesso dos usuários aos dados e ferramentas de forma indireta, restringindo o acesso a um subconjunto de ferramentas ou dados, ou diretamente, invocando ferramentas para objetos apropriados. Esta coação pode prevenir um usuário de executar qualquer ação não permitida na modelagem, porém consiste em uma característica considerada negativa.

Dowson e Fernström (1994) também definem a quarta categoria chamada **Automação do processo**. Neste tipo de suporte uma parte do processo modelado é executada automaticamente. Esta categoria tem como característica a execução de pedaços do processo executados com pouca ou nenhuma intervenção humana. Por não se tratar de um tipo de orientação ao desenvolvedor, decidiu-se não considerá-la para a classificação.

Os limites entre os tipos de orientação não são precisamente definidos. Por exemplo, a invocação por parte do PSEE de uma ferramenta interativa para um desenvolvedor pode ser considerada como orientação ativa (pois o usuário pode preferir não utilizá-la), obrigação (porque não partiu do usuário a sua chamada), ou automação de processo (se pouca ou nenhuma interação é requerida).

2.3.2 Paradigmas de Interação

Os paradigmas de interação com o desenvolvedor especificam as principais metáforas e conceitos que são visíveis aos desenvolvedores do ambiente. Um PSEE geralmente adota um paradigma de interação como sendo o único mecanismo de comunicação entre o PSEE e o desenvolvedor. A partir do estudo dos ambientes existentes e da literatura especializada, foram encontrados quatro paradigmas de interação principais, cuja classificação é proposta a seguir:

- **Interação orientada a tarefas**: Este paradigma adota o uso de listas de tarefas a fazer. O uso de agendas que gerenciam a lista de tarefas de um determinado desenvolvedor é a abordagem mais comum. O desenvolvedor,

por sua vez, visualiza o processo como um conjunto de tarefas a serem realizadas. O ambiente SPADE (BANDINELLI et al., 1996) é um exemplo de utilização deste paradigma;

- **Interação orientada a documentos**: Este paradigma utiliza o conceito de espaços de trabalho ou *workcontexts* onde estão armazenados os documentos a serem manipulados pelos desenvolvedores. Cada documento é associado com um *menu* que provê a lista de ações que podem ser selecionadas para o documento selecionado. A execução de uma ação em um documento pode afetar documentos que estejam no mesmo espaço de trabalho ou em outros. O desenvolvedor visualiza o processo como um conjunto de documentos a serem criados ou manipulados. Por exemplo, o ambiente Merlin (JUNKERMANN et al., 1994) utiliza este paradigma;
- **Interação orientada a metas**: Este paradigma utiliza o conceito de metas a serem atingidas, sem distinguir quais tarefas devem ser realizadas ou quais documentos devem ser manipulados. O desenvolvedor visualiza o processo como um conjunto de metas a serem atingidas. No ambiente Marvel (KAISER et al., 1988), por exemplo, a interação é baseada na visualização do conjunto de regras que um usuário pode ativar. Estas regras representam as metas a serem atingidas. O PSEE PEACE (ARBAOUI; OQUENDO, 1994) também adota o mesmo paradigma;
- **Interação orientada a ferramentas**: Neste paradigma a orientação ao processo está totalmente integrada nas ferramentas de desenvolvimento. Não existem ferramentas adicionais para gerenciar o trabalho do desenvolvedor, nem mesmo para obtenção de *feedback*, o desenvolvedor utiliza apenas as ferramentas usuais de desenvolvimento de software. Cada ferramenta de desenvolvimento (por exemplo, um editor de diagramas Entidade-Relacionamento) "sabe" o que o desenvolvedor pode ou não fazer em um determinado documento. O ambiente PRIME (POHL, 1999) é um dos poucos ambientes que provê este tipo de interação.

Além desses paradigmas, é possível identificar abordagens intermediárias. Por exemplo, o ambiente *ProcessWeaver* fornece uma agenda que mostra *workcontexts*. Ela é composta de descrições de tarefas e dos documentos a serem usados ou produzidos. Também faz parte do paradigma de interação a definição das metáforas utilizadas para interação e cooperação entre desenvolvedores. Por exemplo, em abordagens orientadas a tarefas, a principal forma de apoiar cooperação é através da criação e delegação/distribuição de tarefas.

De forma geral, a interação dos desenvolvedores com o PSEE é influenciada pelo formalismo de modelagem utilizado e pelo tipo de orientação adotada (passiva, ativa, obrigação). A influência do tipo de orientação adotada sobre o paradigma de interação pode ser exemplificada pelo fato de que uma orientação por obrigação do processo dificilmente funcionaria em uma interação orientada a metas. O raciocínio utilizado também vale para orientação passiva com interação orientada a tarefas.

Alguns autores (KOBIALKA; LEWERENTZ, 1998; BANDINELLI et al., 1994a) propõem que a interação seja independente do formalismo de representação do

processo. Segundo Bandinelli et al. (1994a), a interação com o desenvolvedor pode ser "**acoplada**" ou "**desacoplada**" em relação ao formalismo de modelagem de processos. Em resumo, na abordagem "acoplada" a interação está ligada ao paradigma de modelagem do ambiente, o que força os desenvolvedores seguirem um paradigma de interação pré-definido, mas também libera os projetistas de processo de gerenciar a consistência entre o processo executado e a interação com o desenvolvedor. Utilizando uma abordagem "desacoplada", é possível definir diferentes estilos de interação (orientado a metas, orientado a tarefas, a documentos ou híbrido) para diferentes tipos de usuários e etapas do processo. Entretanto, a desvantagem desta abordagem está na necessidade de incluir a garantia da consistência no modelo de processo.

2.4 Flexibilidade na Execução de Processos

Modelos de processos de software possuem características peculiares porque envolvem pessoas realizando tarefas criativas. Não é possível prever antecipadamente todo o desenvolvimento de software. Assim, o processo deve poder ser construído aos poucos e o mecanismo de processos deve lidar com processos incompletos. Além disso, processos de software envolvem incerteza e não-determinismo. Escolhas entre caminhos alternativos devem ser permitidas durante a execução, e essas escolhas podem depender de resultados de atividades anteriores. Esses são requisitos gerais que, quando atendidos, podem aumentar a flexibilidade da execução de processos.

Flexibilidade é uma característica bastante citada na literatura sobre ambientes de gerência de processos, sejam processos de software ou de negócios (*workflow*). Maior atenção foi dedicada ao tema em decorrência da constatação de que processos são executados por pessoas, e que falhas no tratamento dado a esses usuários podem inviabilizar a utilização da tecnologia e ainda ocultar seus benefícios. Além disso, flexibilidade tem sido requerida por todas as fases da evolução de processos de software, não se restringindo somente à fase de execução. Cugola (1998) discute o problema da baixa adoção de PSEEs pela indústria e aponta a falta de flexibilidade dos PSEEs como a mais provável causa para o problema. Por isso, várias abordagens para tratamento de flexibilidade em PSEEs têm sido propostas, tais como: (HEINL et al., 1999; BIDER; KHOMYAKOV, 2000; SLISKI, 2001; JOERIS; HERZOG, 1999; AALST, 1999; HAGEN; ALONSO, 2000).

Apesar das propostas existentes, não há um consenso sobre a definição de flexibilidade na área de processos. Em ambientes de gerência de *workflow*, segundo Heinl et al (1999), flexibilidade consiste em permitir modificações dinâmicas no processo e a escolha de diferentes caminhos alternativos de execução. Já em propostas para PSEEs, segundo Arbaoui et al. (2002), uma linguagem de modelagem de processos multiparadigma é considerada flexível, assim como é considerado flexível o PSEE que fornece diferentes paradigmas de interação com o usuário dependendo da atividade do processo. A flexibilidade de um PSEE também pode ser interpretada como a possibilidade de integrar diferentes ferramentas em sua arquitetura (GRUNDY, 2002) ou ainda como a possibilidade de criar novos fluxos de controle para a linguagem de modelagem (JOERIS; HERZOG, 1999). Porém, é comum encontrar a mudança

dinâmica de processos[4] (ou suporte a evolução de processos) como uma característica de flexibilidade (por exemplo, em (HEINL et al., 1999; BOGIA; KAPLAN, 1995; AALST, 1999)).

A partir do que foi encontrado na literatura sobre o assunto, constata-se que o conceito de flexibilidade de execução de processos de software é usado para definir a *propriedade de mecanismos de execução que toleram a informalidade, o envolvimento humano e processos incompletos sem deixar que os processos tornem-se inconsistentes*. E, sendo o mecanismo de execução o componente que interpreta um modelo descrito em uma PML, pode-se dizer que a PML contribui com a flexibilidade de execução através de construtores adequados que facilitem o uso da mesma.

2.4.1 Aspectos Gerais da Flexibilidade na Execução de Processos

Este trabalho considera que a flexibilidade da execução é uma propriedade que atende ao usuário de um PSEE e não aos desenvolvedores do PSEE, e que requer:

- **Modificação dinâmica durante a execução**: Deve ser possível alterar o processo, seja o conteúdo de atividades ou o fluxo de controle durante a execução, sendo que essa modificação deve ser tratada para que o estado do processo se mantenha consistente. A partir dessa possibilidade são derivadas as seguintes:
  - **Execução de processos incompletos**: Como é possível modelar o processo enquanto está sendo executado, então é possível que o processo inicie incompleto. Segundo Fuggetta (2000) as PMLs devem permitir que sejam modelados processos incompletos e informais que possam ser posteriormente detalhados se necessário;
  - **Instanciação das atividades do processo durante a execução**: As informações sobre pessoas e recursos para atividades poderão estar disponíveis somente após o início da execução de um processo. Esta característica permite que seja iniciada a execução de um processo com partes abstratas que serão instanciadas somente quando estiverem aptas a executar. Dessa forma, a alocação de pessoas e recursos pode ser adiada.
- **Escolhas entre caminhos alternativos**: Essa escolha geralmente é prevista na modelagem, se a linguagem usada fornecer recursos para tal. O projetista de processo pode prescrever alguns caminhos alternativos no processo e condições lógicas a serem satisfeitas para que o caminho correto seja escolhido. Essas condições somente serão verificadas no momento em que o caminho tiver que ser escolhido. Portanto, dependendo da situação corrente, o processo pode mudar de rumo sem que o projetista tenha que monitorar essa situação;
- **Possibilidade de adaptação ao usuário desenvolvedor**: Essa adaptação possui foco na interação com o usuário e corresponde a reconhecer e agir de acordo com o perfil do mesmo durante a execução. Utilizando-se de

[4] A mudança dinâmica de processos é descrita como evolução *on-line* em (ARBAOUI et al., 2002).

métricas, informações gerenciais e informações fornecidas pelo próprio usuário, o ambiente pode representar as atividades do processo de maneira diversa e adequar a orientação ao processo em função desse perfil;

- **Gerência e tratamento de eventos** com possibilidade de modificação automática do processo: Ao invés de monitorar exaustivamente a execução do processo, o gerente pode programar o mecanismo de execução para agir em determinadas situações de forma automática. Para isso, o mecanismo de execução deve manter o registro de todos os eventos ocorridos e ainda deve possibilitar a definição de estratégias para tratamento desses eventos. Com isso, é possível que o próprio mecanismo de execução seja capaz de modificar o processo durante a execução em resposta a algum evento. Essa característica também foi citada por Fuggetta (2000) quando afirma que os PSEEs devem ser tolerantes e gerenciar inconsistências e desvios no modelo de processo.

Já que a modelagem pode ocorrer durante a execução, considera-se que qualquer recurso que aumente a flexibilidade da modelagem contribui com a flexibilidade da execução, como por exemplo, o uso de técnicas de reutilização de processos. Por ser de grande importância para o conceito de flexibilidade na execução de processos, o tópico sobre mudanças dinâmicas será tratado na seção seguinte.

2.4.2 Mudanças Dinâmicas

Mudanças ocorrem com freqüência em um modelo de processo de software. Algumas mudanças são pré-planejadas ou não alteram tanto a execução do processo enquanto outras podem alterar toda a estrutura do processo. Existem mudanças estáticas, ou seja, realizadas antes da execução. Por exemplo, normalmente as definições de um processo necessitam ser especializadas, generalizadas, adaptadas a diferentes classes de projetos (reutilização de processo) ou aperfeiçoadas de acordo com mudanças organizacionais ou experiências anteriores. Outras mudanças são dinâmicas, alterando definições enquanto os processos são executados em função de falhas detectadas, necessidade de aperfeiçoamento ou simplesmente porque alguns aspectos da execução de processos não puderam ser determinados antecipadamente.

As mudanças dinâmicas durante a execução de processos são bastante comuns e difíceis de tratar, necessitando de suporte adequado por parte da linguagem de modelagem e do mecanismo de execução. Além de afetar a execução e a continuação do processo, as mudanças dinâmicas podem requerer modificações compensatórias na realização do processo (isto é, para tratar a ocorrência de possíveis efeitos colaterais). Por exemplo, pode ser necessário apagar informações, desfazer operações, retirar itens da lista de tarefas dos usuários, ou cancelar tarefas que causarão efeitos externos (como por exemplo, cancelar reuniões com o cliente).

A necessidade de permitir mudanças dinâmicas implica na inclusão de recursos específicos no formalismo de modelagem e no mecanismo de execução. Ferramentas de inspeção e análise podem ser necessárias para que o gerente de processo seja capaz de inspecionar o estado da execução do processo e de analisar o impacto de mudanças.

Em ambientes de *workflow* são geralmente tratados processos de negócio ou de manufatura, onde cada atividade ocorre para vários casos, sendo que um caso é, por exemplo, uma compra ou a fabricação de um item. Portanto, quando ocorre mudança no processo, é necessário pensar sobre como isso afetará os casos que ainda não chegaram no ponto da mudança. Além disso, deve ser decidido se a mudança afetará somente os casos novos ou também os que estão em andamento. A mudança que ocorre em um processo que afeta vários casos é chamada de evolucionária (AALST, 1999), pois tem origem na necessidade de aperfeiçoamento ou otimização do processo em questão para vários casos. Várias abordagens para tratar mudanças dinâmicas em *workflow* restringem-se a mudanças evolucionárias, como por exemplo, os trabalhos de Sadiq et al. (2000a; 1999; 2000c), Aalst (1999), e Bider e Khomyakov (2000).

Ao contrário da maioria das abordagens de *workflow*, onde mudanças dinâmicas devem tratar vários casos executando em um processo, a tecnologia de processos de software lida com mudanças para casos específicos, o que é chamado de mudança *ad-hoc* (AALST, 1999). Processos de software são, em geral, específicos para um software sendo desenvolvido (ARBAOUI et al., 2002), e cada processo sendo executado corresponde a um caso. As mudanças buscam prover soluções específicas para atender ao usuário de um caso ou para tratar exceções. Um dos principais problemas com mudanças *ad-hoc* é tratar a consistência de uma instância específica de processo, evitando mudanças que possam introduzir *deadlocks* ou afetem atividades anteriores.

O que se observa dos PSEEs existentes na literatura é que a maioria adota um paradigma particular de gerência de mudanças (como por exemplo (JØRGENSEN, 2001; SLISKI et al., 2001; KRAPP, 1998; SADIQ et al., 2000a)) ou não possui facilidades para isso.

Arbaoui et al. (2002) propõe uma comparação entre PSEEs onde um dos requisitos a serem comparados é o suporte à evolução de processos. A capacidade de lidar com mudanças dinâmicas faz parte desse suporte à evolução. As estratégias para tratar mudanças são duas: evolução *off-line* e *on-line*. Na *off-line* a mudança ocorre antes da execução, enquanto na *on-line* ocorre durante a execução. Quando um processo necessita de mudança *on-line*, as seguintes situações são possíveis (ARBAOUI et al., 2002):

- **Caso 1**: A modificação está sendo realizada em um fragmento que ainda não está em execução. Nesse caso, o processo aceita a mudança realizada;
- **Caso 2**: A modificação está sendo realizada em um fragmento anterior ao ponto em que o processo está sendo executado, mas o estado do processo é coerente com as modificações. Portanto não há necessidade de modififcar o estado atual;
- **Caso 3**: A modificação está sendo realizada em um fragmento anterior ao ponto em que o processo está sendo executado, porém o estado do processo não é consistente com a mudança realizada, o que ocasiona um desvio do processo. Duas abordagens são possíveis: tolerar a mudança e gerenciar o desvio ou reexecutar o processo para que o processo observado volte a um estado consistente;

- **Caso 4**: A modificação está sendo realizada no fragmento de processo que está em execução. Nesse caso os estados do processo observado e do processo modelado podem estar inconsistentes e recai na mesma situação do caso 3.

Em (ARBAOUI et al., 2002) são mostrados três tipos de solução para evolução *on-line* considerando os casos apresentados:

- **Solução A** (*Interfragment on-line evolution support*): Trata os dois primeiros casos de propagação de mudança. O fragmento de processo sendo modificado ainda não está sendo executado e pode ser modificado ou substituído (casos 1 e 2);
- **Solução B** (*Interfragment on-line evolution support with interfragment reenactment*): O fragmento de processo pode ser modificado (após sua execução) e reexecutado (caso 3);
- **Solução C** (as duas soluções anteriores com *Internal-fragment reenactment*): Permite a modificação do fragmento de processo mesmo que esteja sendo executado. A execução é suspensa, o fragmento é modificado e reexecutado (caso 4).

2.5 Requisitos de Execução de Processos

Esta seção apresenta os requisitos de execução de processos de software divididos em três categorias principais: requisitos de prescrição, requisitos de interação e requisitos de flexibilidade. Esses requisitos estão refletindo a revisão da literatura apresentada neste capítulo.

2.5.1 Requisitos de Prescrição

Em geral um mecanismo de execução precisa garantir que o modelo de processo seja executado da forma como foi prescrito. Os requisitos da tabela 2.2 lidam com a interpretação do modelo de processo e são geralmente satisfeitos (nem sempre simultaneamente) pela tecnologia já desenvolvida na área. Trata-se das primeiras questões a serem satisfeitas levantadas acerca de execução de processos (GIMENES, 1994; BANDINELLI et al., 1992; FROELICH, 1994; DOWSON; FERNSTRÖM, 1994).

2.5.2 Requisitos de Interação

Além de garantir que o processo seja executado conforme prescrito, uma importante atribuição do mecanismo de execução, freqüentemente negligenciada, é interagir com seus usuários: gerentes e desenvolvedores em geral. Conforme já apresentado na seção 2.3, os gerentes são os usuários que modelam o processo e monitoram sua execução, podendo decidir sobre as questões do ambiente e do processo sendo executado. Os desenvolvedores, por sua vez, estão envolvidos com atividades do processo, e não têm necessidade de compreender questões de modelagem de processos ou da arquitetura do ambiente.

Os requisitos de interação foram levantados por Sousa et al. (2001) segundo as perspectivas dos gerentes e desenvolvedores que atuam no processo. Souza et al. também compara PSEEs existentes quanto ao atendimento dos requisitos. Na seção 2.3 foram apresentadas as características de interação com usuários de PSEEs e também foram introduzidos os requisitos para cada tipo de usuário. Os requisitos de interação foram classificados e são organizados na tabela 2.3.

Tabela 2.2: Requisitos de prescrição de processos.

Id	**Requisitos de Prescrição**	
R1	**Fluxo de Controle**	A seqüência de atividades no processo modelado deve ser obedecida de acordo com o paradigma de execução adotado (execução ativa, passiva ou obrigação).
R2	**Automação de Processo**	Ativar automaticamente as atividades que podem ser executadas sem intervenção humana, através de uma integração com as ferramentas do ambiente.
R3	**Gerência de Objetos**	O armazenamento persistente de todos os dados envolvidos no processo, com controle dos direitos de acesso e gerência de versões.
R4	**Registro da história do processo**	Coletar dados da evolução do processo para permitir que o processo melhore onde houver necessidade e seja corrigido para atender novos requisitos.
R5	**Coleta de Métricas**	Prover mecanismos para coleta automática de dados sobre o processo, o produto e os participantes, assim como geração de estimativas quando possível.
R6	**Iteração**	Algumas seqüências de atividades podem necessitar de repetição. Deste modo, a ativação automática de uma atividade a ser repetida é tarefa do mecanismo de execução.
R7	**Restrições e alocação de recursos**	Respeito às restrições da execução e gerência da alocação de recursos a fim de saber quem está trabalhando com o que e quais recursos são necessários para quais atividades.

Tabela 2.3: Requisitos decorrentes da interação humana durante execução.

Requisitos de interação com gerentes, desenvolvedores e entre desenvolvedores			
R8 - Interação com Gerentes	**R8.1**	**Permitir diferentes visões de processos**	Os gerentes devem poder monitorar o processo utilizando diferentes perspectivas
	R8.2	**Mostrar estado atual e histórico do processo**	Permitir reversibilidade do estado atual e registro das decisões (*design rationale*).
	R8.3	**Modificação dinâmica do processo durante a execução**	Permitir que o gerente altere o modelo sendo executado de acordo com algumas restrições e que possa definir o modelo enquanto o processo executa (processo incompleto)
	R8.4	**Independência do formalismo de modelagem de processos**	Permitir que o gerente visualize o processo em formatos diferentes.
	R8.5	**Monitoração de eventos e tratamento de exceções.**	Informar ao gerente os eventos ocorridos e fornecer mecanismo para tratá-los automaticamente.

R9- Interação com desenvolvedores	**R9.1**	**Orientar desenvolvedores nas suas tarefas**	Informar desenvolvedores sobre tarefas atribuídas a eles, quais documentos devem manipular e quais os resultados esperados.
	R9.2	**Fornecer visualização adequada das tarefas do processo**	Definir como o desenvolvedor irá visualizar o processo. A visualização pode ser orientada a tarefas, orientada a documentos, tarefas e documentos, metas ou orientada a ferramentas (SOUZA et al., 2001).
	R9.3	**Obter *feedback* do andamento do processo**	O *feedback* pode ser fornecido explicitamente pelo desenvolvedor (formulários, agendas) ou diretamente pelo PSEE (análise de eventos ocorridos).
	R9.4	**Fornecer visualização dos estados do processo (atual e anteriores) e mecanismo de *undo***	Mesma visualização definida para o gerente, mas restrita aos interesses de desenvolvedores.
	R9.5	**Flexibilizar a interação**	Permitir que a orientação do processo possa ser adaptada durante execução. Esta interação é geralmente determinada pela PML.
R10 - Interação entre desenvolvedores	**R10.1**	**Permitir comunicação informal**	Através de mecanismos para comunicação síncrona e assíncrona.
	R10.2	**Permitir gerência de reuniões e horários**	Fornecer mecanismos para discutir detalhes sobre a definição de prazos e horários de grupo
	R10.3	**Permitir monitoração de produtos e processos**	Permitir que os desenvolvedores acompanhem o andamento do processo, estando conscientes, por exemplo de quem manipulou produtos
	R10.4	**Controlar o acesso aos objetos**	Identificar usuários e seus papéis e definir direitos de acesso aos objetos
	R10.5	**Múltiplos níveis de compartilhamento de objetos**	Permitir compartilhamento de objetos de qualquer tamanho e granulosidade variável.
	R10.6	**Registro do histórico dos objetos e mecanismos de undo e redo.**	Gerenciar versões dos objetos e permitir desfazer e refazer as últimas alterações.

Além dos requisitos apresentados, algumas propostas encontradas na literatura apontam para a necessidade de descentralização da execução do processo, como ocorre no ambiente SerendipityII (GRUNDY, 1998), que permite monitoração de processos onde os desenvolvedores podem atuar desconectados.

2.5.3 Requisitos de Flexibilidade

A partir dos aspectos discutidos na seção 2.4 sobre flexibilidade na execução de processos, a tabela 2.4 apresenta os requisitos de um mecanismo de execução com relação à flexibilidade. Deve-se ressaltar que alguns requisitos de flexibilidade são redundantes em relação aos requisitos de interação já apresentados na seção anterior. Isso ocorre com os requisitos R11 (mesmo do R8.3), R15 (mesmo do R9.2) e R16 (mesmo do R8.5).

Tabela 2.4: Requisitos de flexibilidade na execução de processos.

Id	Requisitos de Flexibilidade	
R11	**Modificação dinâmica durante a execução**	O ambiente deve permitir alteração do processo durante a execução mantendo a sua consistência. A alteração pode atingir o fluxo de controle (dependências entre atividades), o conteúdo das atividades (por exemplo, o *script* e datas que prescrevem o comportamento em uma atividade), a alocação de desenvolvedores e recursos, dentre outras.
R12	**Execução de Processos Incompletos**	O processo deve iniciar sua execução mesmo que alguns componentes estejam faltando, como por exemplo, atividades, fluxos de controle e detalhes de alocação.
R13	**Instanciação do processo durante execução**	As informações sobre pessoas e recursos para atividades poderão ser definidas durante a execução. Esta característica permite que seja iniciada a execução de um processo com partes abstratas, que serão instanciadas somente quando estiverem aptas a executar. Dessa forma, a alocação de pessoas e recursos pode ser adiada.
R14	**Escolhas entre caminhos alternativos**	No decorrer da execução podem surgir decisões previstas ou não em tempo de modelagem. O mecanismo de execução deve poder tomar decisões automaticamente com base em condições que consultam o estado corrente do processo ou informações do histórico do ambiente. Essa tomada de decisão consiste em uma escolha de caminho alternativo para continuação do processo.
R15	**Adaptação ao usuário**	O mecanismo de execução deve adaptar a sua interação de acordo com o perfil do usuário. Isso evita que o ambiente torne-se muito intrusivo para desenvolvedores experientes e permite que ele auxilie mais os desenvolvedores novatos.
R16	**Gerência e tratamento de eventos**	O mecanismo de execução deve manter o registro de todos os eventos ocorridos e ainda deve possibilitar a definição de estratégias para tratamento desses eventos. Assim, é possível que o próprio mecanismo de execução seja capaz de modificar o processo durante a execução em resposta a algum evento. Além disso, esse requisito livra o gerente da tarefa de monitorar todas as ocorrências do processo.

3 VISÃO GERAL DO META-MODELO APSEE

A proposta de uma solução flexível para execução de processos de software envolve a definição de um meta-modelo e a integração de ferramentas para fornecer os serviços requeridos. Como o contexto do trabalho está inserido na área de Engenharia de Software, as vantagens da solução proposta terão maior efeito se esta for integrada a um ambiente de desenvolvimento de software (ADS) que permita às pessoas envolvidas o acesso integrado às ferramentas de desenvolvimento, ferramentas de gerência de processos e serviços adicionais para aumentar a qualidade do produto.

A partir de estudos sobre a área de ADS (LIMA REIS, 1999; 2000) foi constatada a necessidade de vários serviços para integrar ferramentas e gerenciar o repositório de um ambiente. Além disso, a construção de um ADS envolve várias áreas da computação, como por exemplo, tecnologia de banco de dados, interação homem-computador, segurança, sistemas distribuídos, dentre outras. Portanto, optou-se por determinar um foco específico para o trabalho no sentido de prover componentes de gerência de processos de software para um ADS já existente, sem tratar as questões inerentes à construção de ADS genéricos.

A abordagem para gerência de processos proposta neste trabalho foi denominada de APSEE. A especificação da abordagem foi desenvolvida com a finalidade de ser genérica o suficiente para ser adotada em outro ambiente, porém, para permitir avaliações, foi integrada ao ADS PROSOFT (NUNES, 1992; 1994). Esta integração não será detalhada neste capítulo, porém é importante caracterizar a evolução do modelo de gerência de processos adotado neste ambiente. Um mecanismo de gerência de processos de software foi proposto para o ambiente PROSOFT em (LIMA REIS, 1998) como dissertação de mestrado. 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. Posteriormente o mecanismo GP foi estendido para permitir o desenvolvimento de um simulador de processos de software baseado em conhecimento ((SILVA et al., 1999; REIS et al., 2000a; SILVA, 2001)). A partir dessa versão ficou clara a necessidade de construir uma abordagem que integre serviços para tratar as várias fases da evolução de processos de software, assim como prover flexibilidade ao ambiente. Neste sentido, um meta-modelo comum para atender essas fases tornou-se requisito essencial.

Grande parte do meta-modelo do APSEE foi desenvolvido neste trabalho com a finalidade de contribuir para a construção de melhores serviços de gerência de processos. Portanto, o trabalho propõe, além do meta-modelo, alguns mecanismos que contribuem com o aumento da flexibilidade na execução de processos, sendo que outros

serviços necessários para PSEEs, como visualização de processos, reutilização e simulação são objeto de outros trabalhos e estão fora do escopo deste.

Neste capítulo serão apresentados os objetivos específicos do trabalho e será apresentada uma arquitetura geral do ambiente em três camadas na seção 3.3. A arquitetura do ambiente APSEE será utilizada para orientar a sua apresentação e fornecer suporte para o atendimento dos requisitos citados acima. Em seguida, será descrita a camada do meta-modelo APSEE e serão discutidas as suas contribuições para os serviços de gerência de processos. Estes, por sua vez, serão apresentados no capítulo 4 com base no meta-modelo proposto, pois o seu entendimento requer uma apresentação prévia do mesmo. O capítulo 5 detalhará como os diferentes aspectos da proposta do meta-modelo foram especificados utilizando métodos formais. Portanto, o objetivo deste capítulo é mostrar o meta-modelo como sendo a estrutura na qual a proposta do trabalho foi construída. Assim, detalhes do funcionamento do ambiente serão discutidos posteriormente com ênfase na proposta de aumento da flexibilidade na execução de processos.

3.1 Objetivos Específicos

Com relação aos requisitos de execução de processos apontados no capítulo 2, são objetivos específicos deste trabalho:

- Atender os requisitos de prescrição R1 a R7, da seção 2.5.1, sendo que o requisito R5, que trata de coleta automática de métricas será atendido parcialmente, ou seja, o meta-modelo será construído para que a coleta de métricas seja feita, mas o mecanismo de coleta automática não será objetivo do trabalho;
- Atender os requisitos de interação R8 (interação com gerentes) e R9 (interação com desenvolvedores). O requisito R10 (interação entre desenvolvedores) não é objetivo deste trabalho pois o ambiente onde o APSEE será integrado, PROSOFT, já possui uma solução para esse requisito, chamada *PROSOFT Cooperativo* (REIS, 1998a; REIS et al., 1998b). Quanto aos requisitos do grupo R8, vale ressaltar que o R8.1, R8.4 e R8.5 não serão diretamente tratados neste trabalho. E quanto aos do grupo R9, apenas o R9.5 não será tratado;
- Atender os requisitos de flexibilidade R11 a R14. O requisito R15 não será tratado enquanto que o requisito R16 será tratado parcialmente.

3.2 Considerações sobre a abordagem escolhida

A abordagem escolhida para atender os requisitos mencionados na seção anterior resulta de decisões acerca do formalismo de modelagem, dos componentes do meta-modelo, dos mecanismos adicionais necessários para atender os objetivos e da forma de especificar tais soluções. A seguir são apresentadas algumas considerações acerca dessas escolhas, que influenciaram a construção do meta-modelo apresentado nesse capítulo assim como dos mecanismos a serem apresentados nos próximos capítulos.

3.2.1 Decisões sobre a escolha do formalismo de modelagem

Existem várias discussões acerca da escolha do melhor formalismo para modelagem de processos. No entanto, publicações mais recentes (FUGGETTA, 2000; JOERIS; HERZOG, 1999; ESTUBLIER et al., 1997) têm dado maior importância para linguagens de modelagem de alto nível, com representação gráfica e que facilitem a tarefa de modelagem provendo conceitos mais próximos aos seus usuários, sem forçar o conhecimento sobre detalhes de execução. Ao mesmo tempo, estas linguagens necessitam gerar processos que possam ser executados. Portanto, também necessitam incluir detalhes de execução em sua semântica.

Analisando os diversos formalismos de modelagem e execução encontrados na literatura (alguns já citados no capítulo 2), observa-se que a maioria mapeia conceitos da área de processos de software para construções em linguagens e paradigmas já existentes. Huff (1996) considera que esses formalismos fazem parte da primeira geração de linguagens de modelagem e destaca a necessidade de linguagens mais direcionadas para processos e que integrem vários paradigmas de modelagem para expressar os diferentes aspectos envolvidos com esse domínio. Outras necessidades importantes, como expressar processos incompletos e em diferentes níveis de complexidade e detalhamento também tem influenciado a construção de formalismos de processos de mais alto nível (FUGGETTA, 2000). Portanto, uma nova geração de formalismos para modelagem e execução de processos ainda está em desenvolvimento. Sendo que a principal sugestão de Huff neste sentido é que sejam propostos formalismos de alto nível, fáceis de entender, cuja semântica seja mapeada para um modelo formal, como por exemplo, redes de Petri.

O mecanismo de execução de processos é fortemente influenciado pelo formalismo de modelagem adotado. Assim, foram investigados os paradigmas existentes de modelagem e as possibilidades de inserir informações que auxiliassem na obtenção de flexibilidade, que é um dos objetivos do trabalho. Foi escolhida a proposta de um meta-modelo de processos orientado a redes de atividades. Nesse meta-modelo, as conexões entre as atividades devem ser detalhadas com relação ao fluxo de controle do processo e elementos adicionais devem ser integrados ao processo para incrementar o paradigma de modelagem. A idéia de criação de um novo formalismo em oposição à adoção de algum já existente surgiu da observação de Huff (1996), já citado, que diz que são necessários formalismos de alto nível, cuja semântica possa ser mapeada para um modelo formal. Considerando a dificuldade em encontrar um formalismo que atendesse essa necessidade, optou-se pela proposta de um novo formalismo com vários tipos de controle entre atividades visando aumentar a flexibilidade. A semântica do formalismo terá o detalhamento das estruturas de dependência propostas. Além disso, mesmo que fosse adotado um formalismo já existente, seria necessário especificar uma extensão a esse formalismo para que o mesmo atendesse os requisitos.

3.2.2 Decisões sobre a abordagem para aumento de flexibilidade

Para aumentar a flexibilidade, o caminho da execução poderá ser alterado automaticamente dependendo do fluxo de controle e da situação corrente. Além disso, poderão existir processos onde os artefatos produzidos por uma atividade podem estar sendo utilizados por outra, em paralelo (compartilhamento de artefatos).

Um meta-modelo organizacional com definições precisas sobre os recursos e pessoas será provido para permitir automação de sua instanciação através de estratégias definidas pelo usuário. Políticas de Instanciação serão propostas para representar essas estratégias que podem basear-se em informações históricas sobre os componentes do processo. Para auxiliar na definição e aplicação de tais políticas, um meta-modelo de conhecimento de processos é proposto permitindo definição de métricas para quaisquer componentes do processo.

Adicionalmente, a gerência dos objetos cooperativos do ambiente é baseada em (REIS, 1998a; REIS et al., 1998b) e será acrescida de informações importantes para seu uso do ponto de vista de gerência de processos.

3.2.3 Decisões sobre a especificação do modelo

A especificação formal das operações mais importantes do sistema será realizada. Especificação formal é a aplicação de matemática na especificação dos requisitos de software. Seu objetivo é superar a falta de precisão e ambigüidade inerentes às especificações informais. Métodos formais têm sido uma área ativa de pesquisa que tem evoluído para ser utilizada na prática da engenharia de software (NISSANKE, 1999) e em outras áreas para descrever o comportamento dos sistemas (semântica), bem como possibilitar a verificação de propriedades destes sistemas e da correção de suas implementações (DÈHARBE et al., 2000). Para dar semântica à execução de processos, a especificação formal ajudará na definição do mapeamento entre formalismos de alto nível, tais como baseados em grafos e mecanismos de execução de baixo nível, tais como os baseados em regras (JOERIS; HERZOG, 1999). A semântica definirá como um modelo de processos deverá ser interpretado em alto nível de abstração;

O contexto onde o trabalho está inserido é o ambiente PROSOFT (NUNES, 1994), que permite desenvolvimento formal de programas e a integração do sistema proposto. O grupo de pesquisa vem trabalhando com tecnologia de processos de software há alguns anos. O PROSOFT vem sendo re-implementado em Java desde 1997 e as propostas deste trabalho deverão ser integradas ao mesmo a fim de testar suas idéias.

3.3 Componentes Principais do Modelo APSEE

Os componentes envolvidos com a gerência de processos são fortemente relacionados entre si. Entretanto, é possível identificar algumas camadas relevantes que agrupam componentes e ferramentas que compõem modelo proposto e que são utilizadas para definir sua estrutura visando facilitar o seu entendimento. Uma visão geral do funcionamento do APSEE é apresentada na figura 3.1 através da separação em três camadas principais: meta-modelo, mecanismos de gerência de processos e nível de interação com o usuário.

O meta-modelo unificado (camada inferior) foi proposto para facilitar a integração de diferentes serviços de gerência de processos de software (que estão na camada do meio). As diferentes fases de evolução de um processo no ambiente APSEE são capturadas pelo meta-modelo unificado. É importante observar que os componentes centrais do modelo são a linguagem de modelagem e o mecanismo de execução, pois

interagem com todos os outros componentes. Essa interação, representada através das setas, indica a chamada de serviços entre os componentes do ambiente.

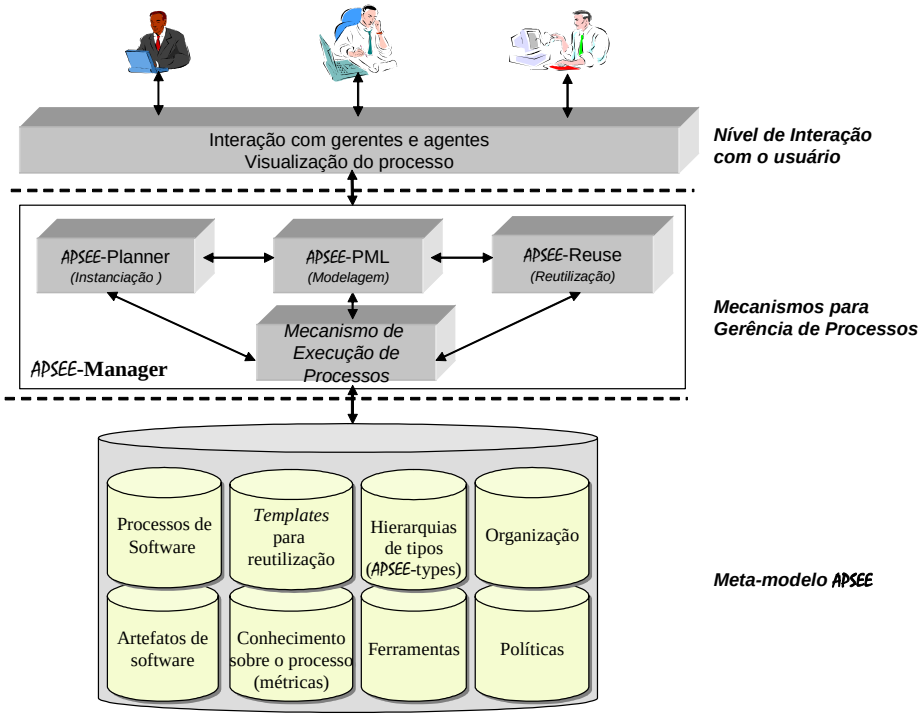


Figura 3.1: Visão Geral do modelo APSEE.

Uma breve descrição dos componentes da figura é apresentada a seguir, porém sem entrar nos detalhes sobre os relacionamentos existentes entre os mesmos (o que será feito na seção 3.4).

- Componentes do meta-modelo:
  - **Processos de Software**: Modelos de processo de software em seus diferentes estados (modelados – em nível abstrato ou instanciado, e em diferentes estados de execução);
  - ***Templates* para reutilização**: descrevem processos abstratos que podem ser reutilizados em diferentes contextos (REIS, 2002f);
  - **Hierarquias de Tipos (**APSEE**-Types)**: relacionadas aos componentes do APSEE e utilizadas na descrição de processos abstratos e de reutilização, além de permitir o raciocínio sobre elementos do processo de forma genérica;
  - **Organização**: Incorpora o modelo de recursos de apoio utilizados pelas atividades, o modelo de pessoas da organização (agentes), suas habilidades, afinidades, cargos e grupos de trabalho;
  - **Artefatos de software**: Correspondem aos itens de dados manipulados, criados e utilizados durante o desenvolvimento de software. Exemplos de artefatos são documentos e código;

  - **Ferramentas**: Informação sobre as ferramentas disponíveis no ambiente para realização das atividades;
  - **Políticas**: São regras definidas pelo usuário que permitem estabelecer quando um modelo de processos está correto do ponto de vista da organização (políticas estáticas), como atividades devem ser instanciadas (políticas de instanciação) e que ações realizar na ocorrência de eventos durante execução (políticas dinâmicas). Políticas podem ser habilitadas na organização, em processos ou atividades específicas. Políticas Estáticas foram propostas por Reis (REIS et al., 2002b; 2001b; REIS, 2002f) enquanto políticas de instanciação serão apresentadas posteriormente no capítulo 4;
  - **Conhecimento sobre o Processo**: É o componente que permite definir e armazenar métricas e estimativas para os componentes do processo, as quais podem ser consultadas dinamicamente durante a execução do mesmo;
- Mecanismos para Gerência de Processos:
  - **APSEE-PML**: Linguagem para modelagem de processos que permite definição de instâncias de processos de software e seus relacionamentos com os outros componentes do modelo;
  - **Mecanismo de Execução do Processo**: Coordena as atividades do processo em execução através da interpretação da linguagem de modelagem de processos APSEE-PML;
  - **APSEE-*Planner***: Auxilia na instanciação de processos de software através do uso de Políticas de Instanciação definidas pelo usuário;
  - **APSEE-*Reuse***: Componente de apoio à reutilização de processos responsável pela criação, recuperação e adaptação de *templates* reutilizáveis (REIS et al., 2001a; REIS, 2002f);
- Interação e Visualização do processo:
  - Este componente provê mecanismos de interação especializados para os diferentes usuários do ambiente. Por exemplo, agendas para os agentes, e facilidades de visualização e manipulação de processos para projetista/gerentes.

3.4 Meta-Modelo Geral do APSEE

Nesta seção serão apresentados e discutidos os componentes do meta-modelo e seus relacionamentos (camada inferior da figura 3.1) através da notação UML (BOOCH et al., 1998)[5]. Esta notação é utilizada neste capítulo por ser largamente conhecida e facilitar o entendimento do modelo em alto nível de abstração. Além disso, será apresentado apenas o Modelo Conceitual do APSEE (ou seja, sem detalhes de projeto).

[5] O meta-modelo proposto tem servido de base para construção de ferramentas em um projeto que envolve cooperação internacional (PROSOFT). Assim, optou-se por construir os diagramas correspondentes em inglês.

A figura 3.2 mostra o diagrama de pacotes do meta-modelo APSEE ilustrando os componentes a serem detalhados e seus relacionamentos de dependência. Alguns componentes do meta-modelo são refinados em novos pacotes. Isto ocorre com os pacotes *Organization*, *Policies* e *SoftwareProcesses*. Pode-se observar uma dependência cíclica entre os pacotes *Software Processes* e *Organization* que indica uma forte interação entre os mesmos. Apesar de não ser recomendado manter dependências cíclicas entre pacotes na UML (AMBLER, 2002), tal dependência foi mantida nesse caso para facilitar o entendimento do modelo e separar as classes da organização das classes dos processos.

O pacote *ProcessReuse* (que aparece na figura 3.2), que contém os *templates* com processos reutilizáveis, está fora do escopo deste trabalho, tendo sido apresentado em (REIS, 2002f). Os demais pacotes são discutidos nas subseções a seguir.

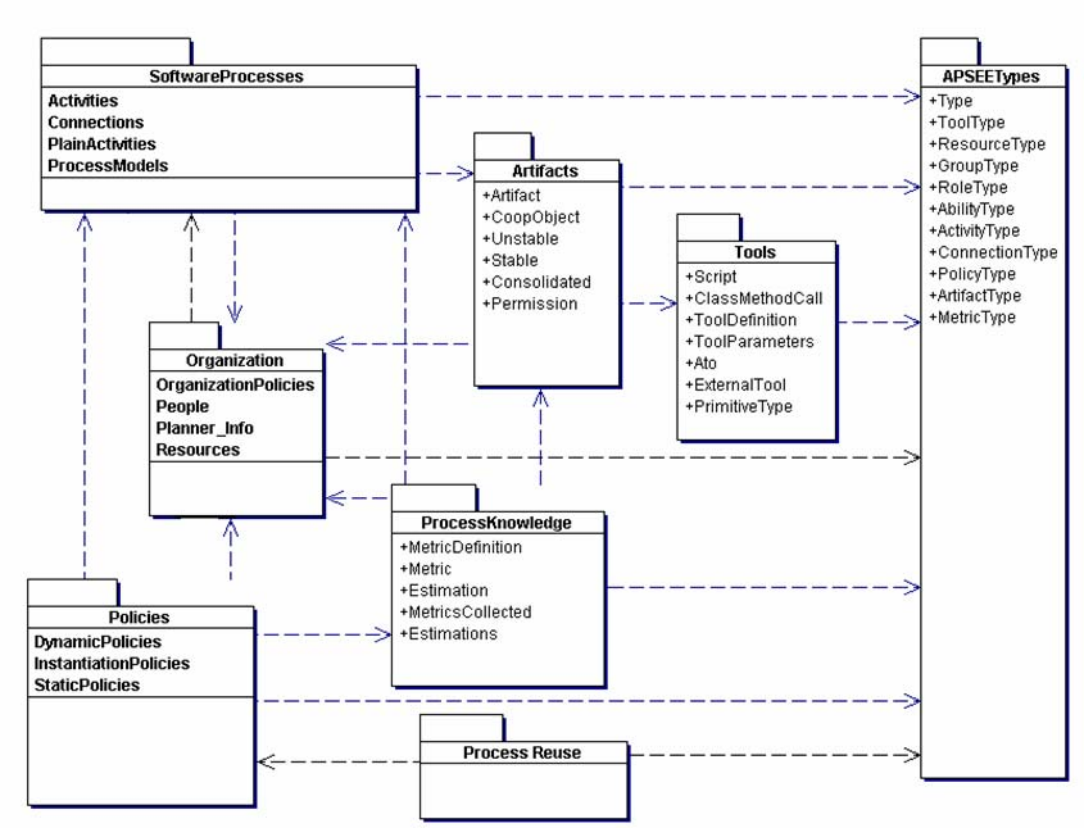


Figura 3.2: Meta-modelo APSEE representado através de um diagrama de Pacotes UML.

3.5 APSEE-Types - Hierarquia de Tipos

Os principais componentes da arquitetura APSEE são tipados, ou seja, são classificados através de tipos predefinidos. O pacote *APSEE-Types* contém as hierarquias de tipos para os componentes do APSEE. Neste caso, são propostas as hierarquias de tipos principais, fornecidas juntamente com o modelo, mas o usuário do ambiente poderá criar novas hierarquias de tipos e instâncias para hierarquias existentes de acordo com as necessidades da organização ou do processo.

As hierarquias de tipos propostas inicialmente apresentadas na figura 3.3 são: recursos, cargos, grupos, habilidades, métricas, atividades, conexões, políticas, artefatos e ferramentas.

O pacote *APSEE-Types* exerce um papel importante na descrição do modelo, visto que a maioria dos componentes do meta-modelo APSEE estão associados a um tipo (*Type*) que deve existir na hierarquia correspondente. As hierarquias de tipos relacionadas aos componentes do APSEE permitem que sejam descritos processos abstratos que podem ser refinados conforme necessário para execução ou usados para reutilização e ainda apóiam a descrição de mecanismos que lidam com elementos genéricos de processos. Essa estrutura de tipos aumenta a flexibilidade do modelo, o que permitiu a descrição de regras gerais (políticas) que se referem a tipos e não a instâncias, e de uma abordagem para reutilização de processos (REIS 2002f).

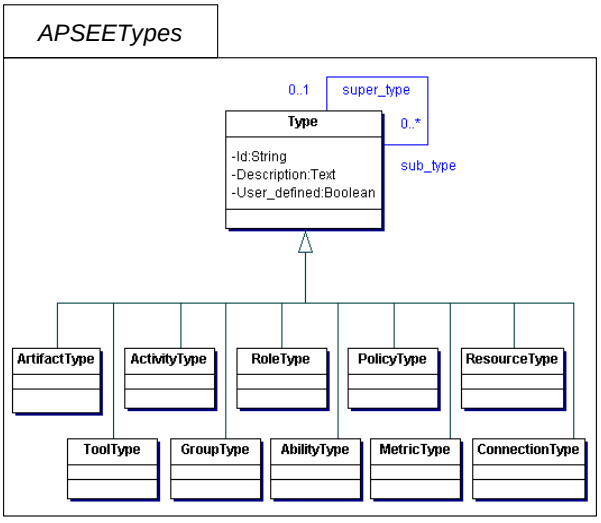


Figura 3.3: Pacote *APSEE-Types* - Hierarquia de tipos do APSEE.

É importante observar que a hierarquia de tipos *ActivityType* serve para designar tanto os tipos de atividades quanto os de processos de software pois atividades podem ser decompostas em processos (o modelo de atividades será apresentado na seção 3.7.2).

3.6 *Organization* - Informações sobre a Organização

A figura 3.4 mostra o detalhamento do pacote *Organization* que contém os aspectos relacionados à estrutura organizacional envolvida. As informações sobre a organização foram divididas em subpacotes, distinguindo informações sobre as pessoas, recursos, sobre as políticas adotadas pela organização e seus projetos, contendo ainda informações de instanciação (*Planner_Info*).

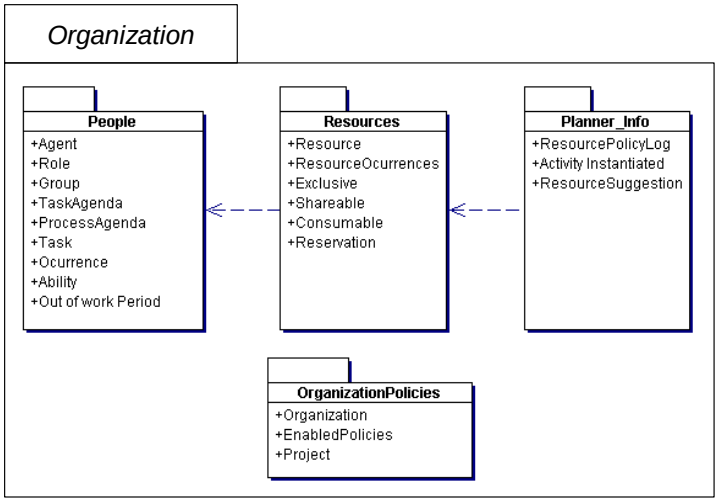


Figura 3.4: Pacote *Organization*.

3.6.1 *OrganizationPolicies* - Projetos e Políticas da Organização

No pacote *OrganizationPolicies* (ver figura 3.5) são destacados os projetos da organização e são indicadas as políticas adotadas pela mesma (políticas estão descritas na seção 3.11). Os projetos refletem, na verdade, informações derivadas de processos de software que são executados na organização. Um projeto refere-se a um processo que ocorre na organização (processos são definidos na seção 3.7), indicando a data de início e fim reais e quais são os artefatos finais (artefatos são definidos na seção 3.8), por exemplo, código-fonte e executáveis produzidos pelo projeto. A organização pode ainda habilitar algumas políticas para serem adotadas por todos os seus processos. Os atributos *automatic_instantiation* e *policy_evaluation_priority_local* da classe *Organization* descrevem configurações necessárias para o mecanismo de instanciação a ser apresentado na seção 4.2 do capítulo 4.

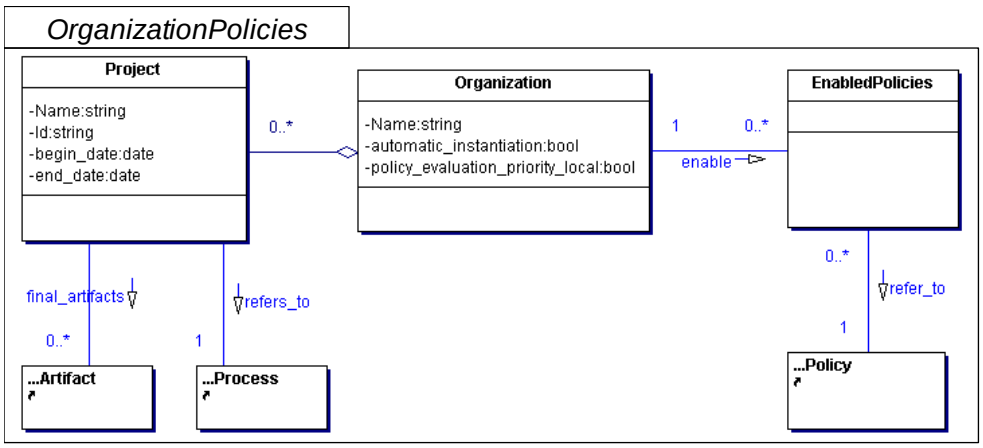


Figura 3.5: Pacote *Organization.OrganizationPolicies*.

3.6.2 *People* - Pessoas da Organização

Informações sobre as pessoas e recursos envolvidos com o processo de software são normalmente encontradas em PSEEs da literatura, com algum detalhamento de suas características. Entretanto a maioria das abordagens não distingue entre pessoas e recursos (PODOROZHNY et al., 1999). Assim, pessoas (ou agentes) são normalmente tratados como recursos de apoio, e são alocados para atividades, enquanto que os recursos de apoio nem sempre possuem informação detalhada sobre seu estado, impedindo verificação automática e adoção de estratégias para alocação. A já citada falta de consenso na literatura (LONCHAMP, 1993; WESTFECHTEL, 1999; FUGGETTA; WOLF, 1996) sobre o conceito de recurso pode ser a causa desse problema. Portanto, o modelo aqui proposto distingue o termo recurso de pessoa (ou agente), propondo uma definição mais detalhada dos componentes organizacionais, a fim de que se possa controlar e otimizar seu uso ou alocação em processos de software.

A partir desta distinção foi possível detalhar separadamente o modelo de pessoas através do pacote *People*, apresentado nesta seção, e do pacote *Resources*, a ser apresentado na seção 3.6.3.

As informações sobre as pessoas da organização visam apoiar os componentes de gerência de processos auxiliando na escolha de desenvolvedores para tarefas específicas. Essas informações são de extrema importância, pois a adequação das pessoas envolvidas no processo é, segundo Pressman (PRESSMAN, 2001), um fator determinante para o aumento da qualidade do software resultante. Neste contexto, as habilidades e cargos das pessoas têm sido sugeridos como critérios na sua escolha para atividades de um processo (PLEKANOVA, 1999). No entanto, diversos fatores adicionais podem afetar a produtividade das pessoas (PRESSMAN, 2001), e a tomada de decisão deve levá-los em consideração. Apesar da impossibilidade de mapear todos os fatores que influenciam na produtividade de um desenvolvedor, são propostas algumas características que podem auxiliar nas tomadas de decisão sobre pessoas, tais como: afinidades entre os agentes para atividades cooperativas, os diferentes graus de habilidades e o registro sobre seu histórico na organização (no pacote *ProcessKnowledge*).

O diagrama de classes do pacote *People* contém todas as informações relacionadas às pessoas da organização (chamadas de *Agent* no modelo), seus grupos, cargos, habilidades, afinidades e agenda de tarefas (ver figura 3.6). Seus componentes são descritos pelas subseções a seguir.

3.6.2.1 Agentes

A classe *Agent* provê informações sobre os agentes, que incluem quaisquer pessoas envolvidas com o processo de software (desenvolvedores, gerentes e usuários, por exemplo). Os agentes podem também ser compreendidos como os atores do processo. O atributo *cost_hour* indica o custo por hora do agente e permite calcular o custo de uma atividade em função do custo dos agentes, ou ainda permite selecionar agentes pelo seu custo.

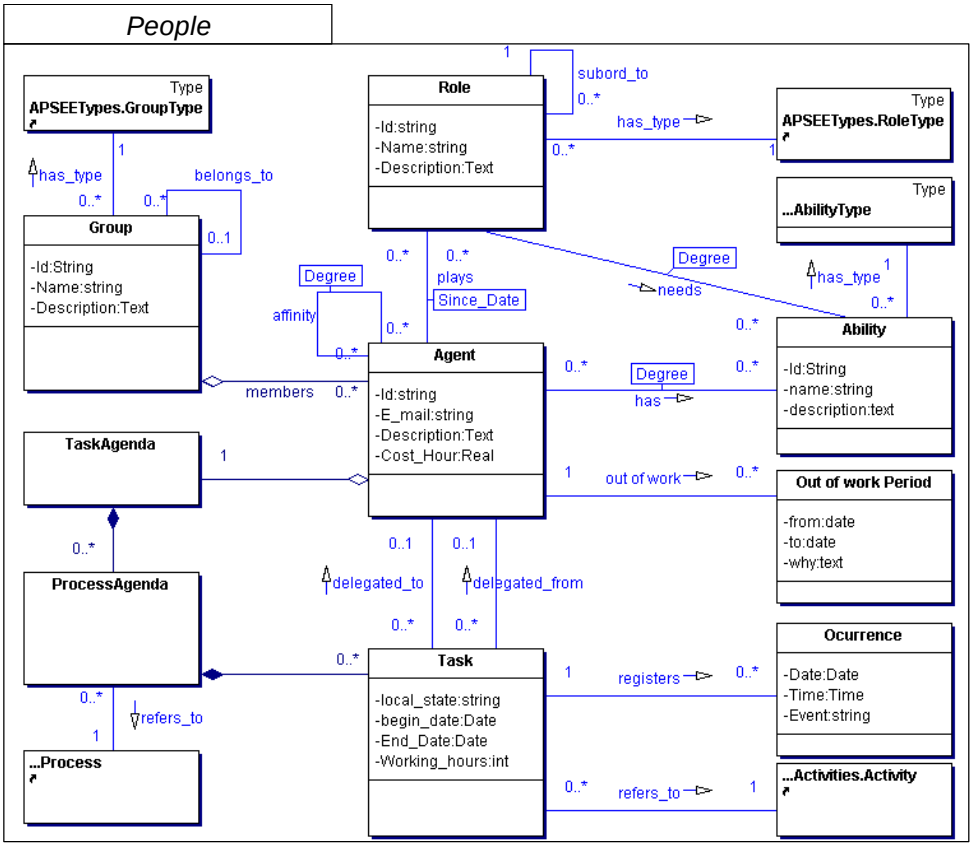


Figura 3.6: Pacote *Organization.People* – Informações sobre as pessoas envolvidas no processo de software.

A afinidade entre agentes é a medida que indica quão bem dois agentes trabalham juntos. A associação *affinity* representa essa informação e indica um grau de 0 a 1 para cada ocorrência da associação. O grau de afinidade entre agentes é uma métrica subjetiva sobre a realização de tarefas cooperativas. A idéia de armazenar afinidades entre os agentes veio do meta-modelo para simulação de processos de software proposto por Silva (2001), sendo que essa contribuição foi incorporada ao modelo deste trabalho. A simulação utiliza essa informação para predizer a realização das atividades de um processo. Acredita-se que alta afinidade pode levar a atividade a ser executada com sucesso. Além de auxiliar a simulação de processos, a determinação da afinidade entre agentes pode servir de critério para alocação de agentes em atividades cooperativas.

A classe *Agent* possui uma associação com a classe *Out_Of_Work_Period.* Para cada agente podem ser definidos períodos em que ele/ela não estará disponível para as atividades da organização (por exemplo, férias ou períodos de licença). Isto serve para prevenir sua alocação em atividades futuras previstas para os mesmos períodos e também facilita a tomada de decisão quando o agente deixa de trabalhar em atividades que já iniciaram. Por exemplo, o mecanismo de execução pode detectar que o agente deixou de trabalhar e automaticamente delegar a tarefa a outro ou avisar o gerente de processos.

3.6.2.2 Cargos

Os cargos da organização são representados pela classe *Role*. A associação *subord_to* indica a estrutura hierárquica dos cargos. Além disso, são indicadas para cada cargo quais habilidades (e seus graus mínimos) um agente deve possuir para atuar no mesmo. Os cargos possuem tipos na hierarquia de tipos de cargos no *APSEE-Types* (por exemplo, "programador", "gerente", "analista" podem ser tipos de cargos, enquanto que "programador_nível_I", "programador_nível_II" são cargos do tipo "programador") e para cada cargo que o agente ocupa, é armazenada a data de início da ocupação do cargo (atributo *since_date*).

3.6.2.3 Habilidades

A classe *Ability* representa habilidades reconhecidas pela organização para o desenvolvimento de software. Habilidades estão associadas a tipos de habilidades na hierarquia *APSEE-Types*. Um agente pode ter várias habilidades e para cada ocorrência dessa associação é determinado um grau da habilidade (de 0 a 1). Assim como a afinidade entre agentes, a habilidade é uma métrica subjetiva e foi incorporada a este modelo a partir do modelo apresentado em (SILVA, 2001). A atribuição do grau de habilidade de um agente não é automatizada, porque diferentes fatores podem influenciar na habilidade de uma pessoa, tais como: experiência, formação e comparação com outros agentes, além da natureza dinâmica desse atributo (o grau de habilidade dos agentes pode ser aumentado). Além disso, devem existir critérios para aumento ou diminuição da habilidade dos agentes e esses são fatores muito subjetivos e difíceis de serem automatizados. A organização pode definir os graus de habilidades de seus agentes utilizando métodos como entrevistas, coleta de informações sobre os agentes, testes, etc. O importante é que os graus atribuídos sejam consistentes com a formação e experiência dos agentes em comparação com os outros. As habilidades dos agentes fornecem um conhecimento importante para a sua alocação em atividades do processo.

3.6.2.4 Grupos

A classe *Group* armazena informações sobre os grupos da organização. A composição dos grupos é representada pela associação *belongs_to* com o objetivo de facilitar a representação da estrutura da organização em grupos e sub-grupos de trabalho. Os grupos possuem tipos na hierarquia de tipos de grupos no *APSEE-Types* (por exemplo, "revisão", "programação" e "gerência"). Um grupo possui vários agentes como membros.

3.6.2.5 Agendas

A maioria dos PSEEs existentes adotam agendas de tarefas como principal tipo de interação com desenvolvedores (BANDINELLI et al., 1996). As agendas dos agentes são o mecanismo principal de comunicação entre o mecanismo de execução de processos do APSEE e os agentes. Neste modelo, um agente pode participar de vários processos. Assim, cada agenda representa a visão do agente sobre o processo em execução, ou seja, atividades alocadas a ele e os documentos que pode manipular.

A classe *TaskAgenda* possui uma agenda para cada processo que o agente participa (*ProcessAgenda*), e esta, por sua vez, está associada a um processo do APSEE (classe *Process*) e é composta de várias tarefas (*Task*) alocadas para o agente naquele processo. Instâncias de *Task* estão associadas a atividades de processos (classe *Activity*) e armazenam o estado local da execução da atividade para o agente. Como vários agentes e grupos podem estar envolvidos na mesma atividade, o estado da execução da atividade para um agente pode ser diferente do estado global da mesma. Portanto, na agenda do agente o atributo *local_state* da classe *Task* pode assumir os seguintes valores (estados):

- ***Waiting**:* prevista para o agente, mas não pronta para começar;
- ***Ready**:* pronta para começar;
- ***Active**:* o agente está trabalhando na atividade;
- ***Paused**:* o agente solicitou pausa da atividade;
- ***Finished**:* o agente concluiu a atividade;
- ***Delegated**:* a atividade foi delegada para outro agente (antes de ser iniciada);
- ***Cancelled**:* a atividade foi cancelada antes de iniciar;
- ***Failed**:* a atividade falhou por decisão dos agentes ou do gerente.

A agenda de atividades também guarda as datas de início e fim da atividade (*begin_date, end_date*), pra qual agente a atividade foi delegada (*delegated_to*), qual agente delegou a atividade (*delegated_from*), tempo total de trabalho nesta atividade (*working_hours*), e um registro de todas as ocorrências da agenda (associação com classe *Ocurrence*).

O atributo *event* da classe *Ocurrence* na agenda do agente indica quando a atividade foi adicionada na agenda (*added*), quando foi delegada para o agente (*delegated*), e a transição dos estados do atributo *local_state* (*to_ready, to_active, to_paused, to_finished, to_failed, to_cancelled*).

No capítulo 4 a seção 4.3 trata da execução de processos no APSEE detalhará a manipulação da agenda dos agentes pelo mecanismo de execução.

3.6.3 *Resources* - Recursos da Organização

A construção de formalismos de modelagem de processos e melhores mecanismos de execução não têm priorizado o aprofundamento na gerência de recursos que necessitam de controle durante a execução de processos. A importância desse controle pode ser observada pela necessidade de otimização na alocação de recursos, a fim de evitar atrasos desnecessários e pela possibilidade de manter o estado da realização do processo no mundo real consistente com o modelo sendo executado. Esta consistência entre o estado da definição e da realização do processo pode ser aumentada se o modelo de gerência de recursos for especificado de forma precisa em relação à realidade da organização (DOWSON; FERNSTRÖM, 1994).

O modelo de recursos do APSEE consiste de tipos de recursos, instâncias de recursos e relacionamentos entre os mesmos. Além de prover um mecanismo para descrever e organizar os recursos disponíveis na organização, o modelo de recursos pode ser usado para controlar dinamicamente o acesso concorrente aos recursos, escalonar sua

utilização, analisar sua disponibilidade na organização e verificar consistência de seu uso. A figura 3.7 apresenta o diagrama de classes do pacote *Resources*.

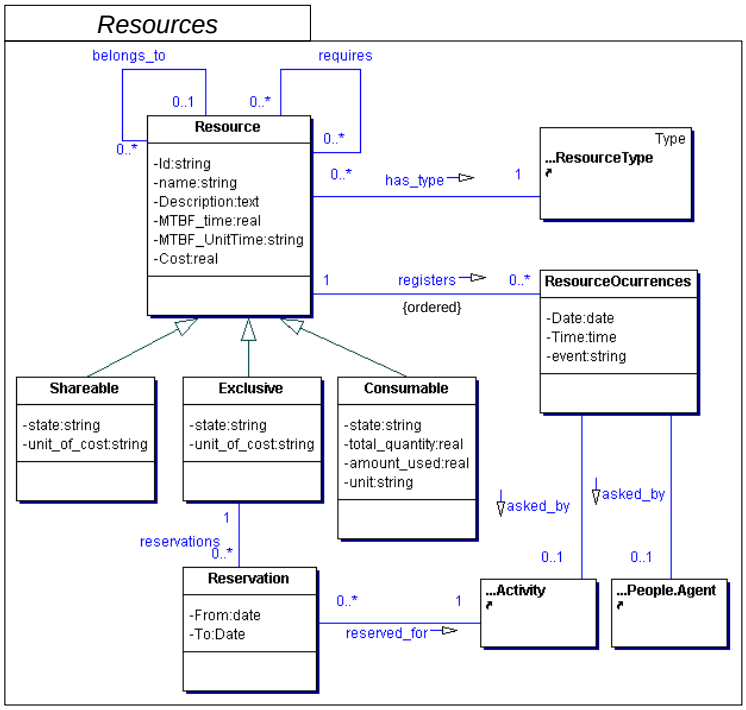


Figura 3.7: Pacote *Organization.Resources* – Informações sobre os recursos da organização.

Utilizando o componente *APSEE-Types* foi definida uma hierarquia de tipos de recursos cujo significado é apresentado a seguir:

- ***Exclusive*** (Exclusivos): São recursos que não podem ser alocados para várias atividades simultaneamente. Portanto, caso duas atividades necessitem do recurso ao mesmo tempo, uma delas deverá aguardar pelo término da outra. Exemplos de recursos exclusivos são computadores e salas;
- ***Shareable*** (Compartilháveis): Podem ser utilizados por várias atividades simultaneamente, sem necessidade de alocação exclusiva. Assim, não é necessário que a atividade faça alocação e liberação desse tipo de recurso. Uma impressora conectada em rede é um exemplo de recurso compartilhável;
- ***Consumable*** (Consumíveis): Este tipo de recurso não pode ser utilizado novamente como os anteriores. Uma instância de recurso consumível pode ser utilizada total ou parcialmente por uma atividade. Deste modo, uma vez consumido, o recurso passará para o estado "*Finished*" e não mais poderá ser alocado. Recursos financeiros e papel são exemplos de recursos consumíveis.

Cada recurso possui um identificador (*id*), um nome e uma descrição textual (*description*). As demais características comuns a todos os recursos são:

- **Tempo médio entre falhas** (*MTBF - Medium Time Between Failure*), através dos atributos *MTBF_time* e *MTBF_UnitTime:* armazenam o tempo e a unidade de tempo entre falhas do recurso, respectivamente (por exemplo: 2 meses). Esta informação pode ser obtida a partir da lista de ocorrências do recurso e permite que o projetista de processo decida pelo recurso mais adequado e confiável quando estiver lidando com atividades críticas do processo, por exemplo;
- Associação de **composição** entre recursos *belongs_to*: A disponibilidade e alocação de recursos podem ser influenciadas por essa associação. Por exemplo, se uma sala é composta de três computadores, e é alocada para uma atividade, então seus componentes também serão alocados. Se apenas um dos computadores da sala estiver alocado para uma atividade, isto não implica que a sala também deva estar;
- Associação ***requires***: indica um conjunto de recursos requeridos para alocação. Este relacionamento também modifica a disponibilidade e alocação de recursos. Porém, neste caso, o fato de um recurso requerer outros não significa que os têm como componentes. Uma impressora pode requerer papel enquanto uma licença de software pode requerer um computador configurado para executar o software. Quando um recurso é alocado, são também alocados todos os recursos necessários especificados neste relacionamento;
- Uma **lista de ocorrências** (*ResourceOcurrence*), com as modificações feitas no recurso, indicando data e hora, evento (por exemplo, mudança de estado do recurso, alocação), identificação da atividade que gerou a ocorrência e do agente que solicitou. Através da lista de ocorrências é possível saber a taxa de utilização de um recurso ou quem solicitou mudanças de estado.

As características adicionais das subclasses de recursos são detalhadas nas seções a seguir.

3.6.3.1 Recursos de Uso Exclusivo (Exclusive Resources)

Recursos de uso exclusivo podem ser alocados por atividades (em momentos diferentes), reservados para alocação futura e indisponibilizados por defeito. A seguir são detalhadas as suas características:

- **Estado do recurso** (*state*), que pode ser disponível (*available*), alocado (*locked*) ou com defeito (*defect*). A alocação e liberação de um recurso são realizadas automaticamente pelo ambiente. A mudança para o estado *Defect* somente pode ser realizada por um usuário que gerencie o ambiente. A

figura 3.8 apresenta o diagrama de transição de estados de um recurso exclusivo[6];

- **Custo** e **Unidade de tempo do custo** (*Cost* e *Unit_Cost*), que definem o custo de utilização do recurso em função do tempo. Por exemplo, o custo de um equipamento é definido como R$100,00 por hora de utilização (*Cost* = 100,00 e *Unit_Cost* = "hora");
- **Reservas** de um recurso (classe *Reservation*), que definem para qual atividade e período o recurso está reservado. As reservas servem para planejamento de uso futuro de recursos e têm o potencial de melhorar a utilização de recursos por garantir que um recurso específico estará disponível no momento da alocação.

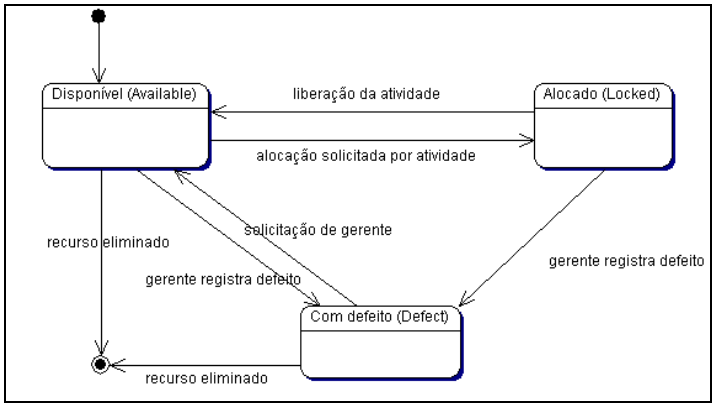


Figura 3.8: Transição de estados dos recursos de uso exclusivo.

3.6.3.2 Recursos Compartilháveis (Shareable Resources)

Recursos compartilháveis podem ser utilizados por várias atividades, mas sua alocação exclusiva não se faz necessária, porque sua utilização é momentânea e pode ser escalonada. Este é o caso de impressoras, aparelhos de fax e telefones para citar os recursos de um escritório, por exemplo. Entretanto, é útil registrar sua utilização por parte das atividades do processo. Portanto as seguintes características foram definidas para recursos compartilháveis:

- **Estado** do recurso (*state*), que pode ser disponível (*available*) ou não-disponível (*not_Available*). Como este tipo de recurso não é alocado e liberado para atividades específicas, o seu estado permanece sempre disponível até que um usuário gerente torne o recurso não disponível[7]. O diagrama de transição de estados de um recurso compartilhável é apresentado na figura 3.9;

[6] As figuras que contém diagramas de estados de recursos ilustram os estados em português e em inglês para facilitar a correspondência com a especificação do modelo, pois na especificação formal e implementação do modelo os estados são representados em inglês.

[7] Esta indisponibilidade pode tanto significar defeito como indisponibilidade por decisão da organização.

- **Custo** e **Unidade de tempo do custo** (*Cost* e *Unit_Cost*), que definem o custo de utilização do recurso em função do tempo da mesma forma que os recursos de uso exclusivo da seção 3.6.3.1;

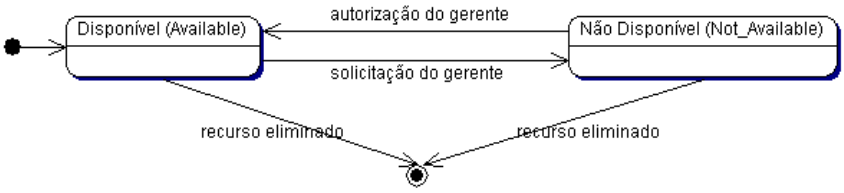


Figura 3.9: Transição de estados dos recursos consumíveis.

3.6.3.3 Recursos Consumíveis (Consumable Resources)

Recursos consumíveis são transformados a partir de sua utilização/consumo. O registro de que uma atividade consumiu um determinado recurso é importante para o sistema porque permite prever a necessidade de reposição caso outras atividades também o requeiram. Além disso, tal registro permite calcular com mais precisão os custos de um processo de software ou ainda controlar sua utilização através da autorização de uso. A seguir as características adicionais de um recurso compartilhável:

- **Estado** do recurso (*state*), que pode ser disponível (*available*), terminado (*finished*) ou não autorizado (*not_authorized*). Não existe alocação e liberação deste tipo de recurso, e sim uso ou consumo. Duas ou mais atividades podem estar usando o mesmo recurso consumível, desde que não ultrapassem o uso da quantidade total disponível do mesmo. Os estados *available* e *finished* são tratados automaticamente pelo ambiente. O gerente pode solicitar que o recurso passe para o estado *Not_Authorized*, conforme mostra o diagrama de transição de estados da figura 3.10;
- **Quantidade Total** (*TotalQuantity*) e **Usada** (*amount used*) do recurso. O uso deste tipo de recurso é registrado através de adição no atributo *amount_used*. É possível repor a quantidade de um recurso (*TotalQuantity*), fazendo com que o mesmo volte para o estado disponível;
- O atributo ***Unit*** guarda a unidade de medida do recurso. Por exemplo, para um papel, a unidade pode ser resmas ou folhas. E o atributo *cost*, neste caso, guarda o custo unitário do recurso (útil para recursos não financeiros).

Semelhante aos recursos *Shareable*, este tipo de recurso não requer alocação exclusiva, porém a lista de ocorrências armazena a utilização do mesmo. Além disso, atividades que necessitam deste tipo de recurso não são realizadas caso o mesmo não esteja autorizado ou tenha terminado.

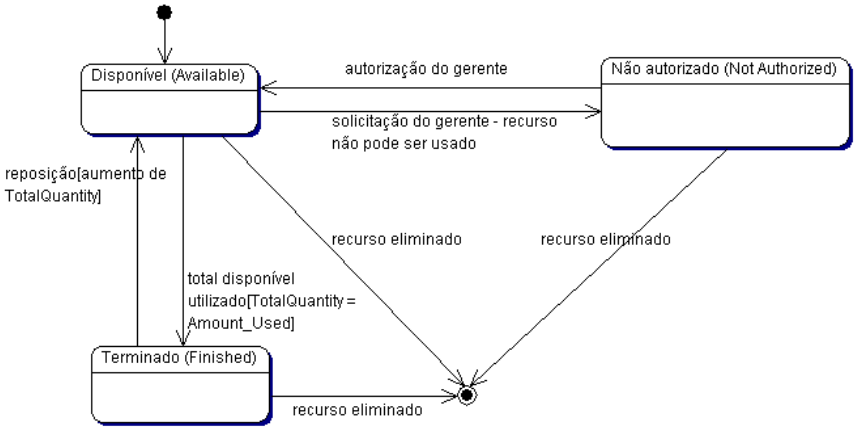


Figura 3.10: Transição de estados de recursos consumíveis.

3.7 Processos de Software

O pacote *SoftwareProcesses* (ver figura 3.11) foi dividido em sub-pacotes. Seus componentes são os modelos de processos de software (*ProcessModels*), atividades (*Activities*), Atividades simples (*PlainActivities*) e conexões (*Connections*). O pacote *ProcessModels* é o componente principal do pacote *SoftwareProcesses* e tem como objetivo descrever as características de um processo de software e seu relacionamento com os outros componentes do meta-modelo.

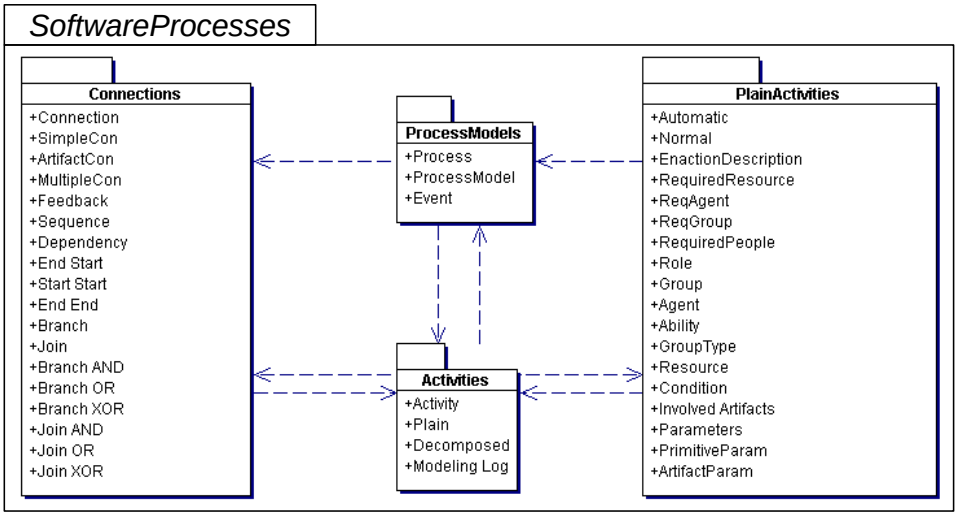


Figura 3.11: Pacote *SoftwareProcesses*.

Os aspectos de execução de processos de software são o foco principal deste trabalho. Entretanto, quando a modelagem de processos tem como objetivo a execução, e não somente a documentação do processo, deve-se considerar a influência do formalismo de modelagem no mecanismo de execução, pois este último deve obedecer a semântica da linguagem de modelagem.

O modelo proposto buscou integrar o máximo possível de aspectos sobre coordenação de processos de software. Esses aspectos derivam da necessidade de

representar vários tipos de dependências no modelo. Tipos de dependência foram propostos por Malone e Crowston (1994) na área de teoria da coordenação, a qual está fortemente relacionada com coordenação de processos de software (GREENWOOD, 1995). Por isso, as dependências foram levadas em consideração para a construção do meta-modelo e separação do mesmo em diferentes aspectos relevantes. Portanto, considerando as camadas de coordenação, colaboração e comunicação definidas por Klein (1998) e citadas na seção 1.1.2 deste texto, o presente trabalho contribui com a camada de coordenação, mas é integrado a um ambiente que provê as demais camadas.

Alguns aspectos já são conhecidos na literatura sobre o assunto, como as perspectivas de Curtis et al. (1992) que são: **funcional** (o que – atividades), **organizacional** (quem faz – pessoas), **comportamental** (quando – fluxo de controle) e **operacional** (como é feito – ferramentas). Assim como na proposta de Kappel et al. (1998), utilizamos adicionalmente o aspecto **informacional**, que representa o fluxo de dados entre atividades.

Um aspecto inovador sendo utilizado é a habilitação de **políticas**, que complementam a linguagem de modelagem e influenciam na utilização do modelo somente nos pontos onde estão habilitadas. Outros aspectos técnicos que demonstraram utilidade para os objetivos do trabalho e são integrados no meta-modelo de processo são: o registro do **histórico** de ocorrências, as **informações sobre a execução** associadas a cada elemento relevante e associação de **tipos** para construção de processos abstratos. Desta forma a figura 3.12 ilustra os aspectos que estão integrados no modelo de processos proposto. Os aspectos funcionais do modelo (processos de software, que são explicados nesta seção) formam a base para a integração dos outros aspectos, que estão sendo apresentados no decorrer deste capítulo. Os aspectos integrados aos processos de software no meta-modelo proposto são: políticas habilitadas em partes dos processos; as informações sobre o estado da execução que são encontradas nas atividades de processos; os aspectos operacionais, que são integrados através das atividades automáticas; os aspectos informacionais, que são explicitados pelas conexões de artefato; os aspectos comportamentais que são definidos pelas conexões de controle; os aspectos organizacionais, definidos pelo pacote *Organization*; as hierarquias de tipos, do pacote *APSEE-Types*; e o histórico do processo, representado pelos eventos armazenados a cada passo de execução.

As subseções seguintes apresentam o meta-modelo de processos de software proposto para o APSEE de acordo com a divisão dos componentes no pacote *SoftwareProcesses*.

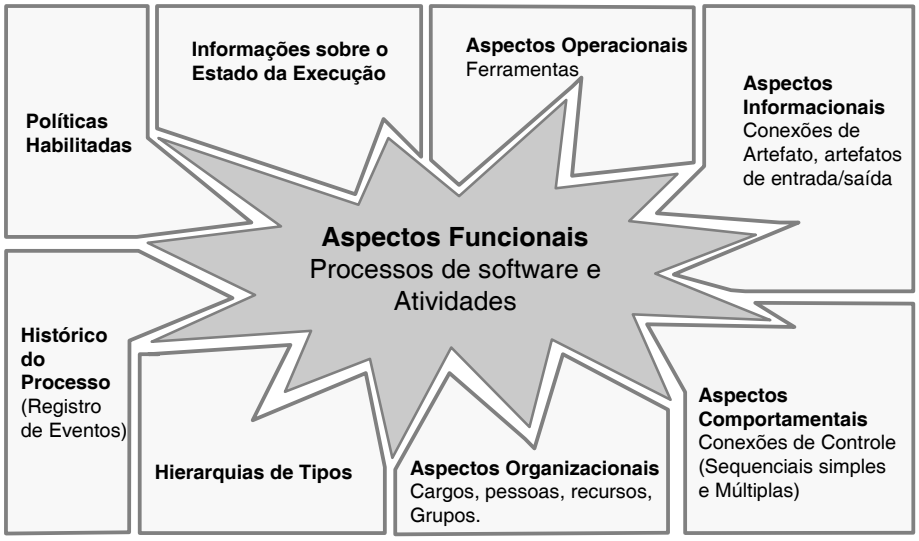


Figura 3.12: Aspectos envolvidos com o modelo de processos no APSEE.

3.7.1 Modelos de processos

O pacote *SoftwareProcesses*.*ProcessModels* (ver figura 3.13) contém o modelo para descrever processos de software em diferentes estados de modelagem e de execução. Um processo da classe *Process* possui um identificador, um tipo na hierarquia de tipos de atividades, um estado e um modelo de processo. O estado do processo (*p_state*) pode ser: *Not_Started*, (o processo não foi iniciado), *Enacting* (o processo está em execução) ou *Finished* (o processo foi concluído). A distinção entre processo (*Process*) e modelo de processo (*ProcessModel*) se faz necessária para determinar a situação geral do nodo raiz da rede de atividades, pois modelos de processo são definidos de forma recursiva (são compostos de atividades que podem ser decompostas em novos modelos de processo). Portanto, uma instância de *Process* faz o papel de raiz de um modelo de processo de software.

O modelo de processo (*ProcessModel*) é composto basicamente de atividades e conexões entre atividades. O estado do modelo de processo (*pm_state*) é determinado dinamicamente em função de seu conteúdo e pode receber os seguintes valores:

- ***Requirements****:* o modelo de processo não possui nenhuma atividade, apenas uma descrição textual de seus requisitos (atributo *requirements*), ou ainda, possui somente atividades decompostas que estão no estado *requirements*;
- ***Abstract***: o modelo de processo possui atividades que não estão relacionadas ao contexto da organização, ou seja, são genéricas. Neste estado as atividades indicam apenas os cargos necessários, os tipos de recurso e tipos de artefato;
- ***Instantiated***: neste estado o modelo de processo já possui atividades instanciadas e pode ser executado;

- ***Enacting***: pelo menos uma das atividades do modelo de processo está sendo executada;
- ***Finished***: todas as atividades do modelo de processo foram concluídas;
- ***Failed***: todas as atividades do modelo de processo falharam. Esta situação é possível através da solicitação de um gerente. As conseqüências da falha são tratadas pelo mecanismo de execução;
- ***Cancelled***: o modelo de processo foi cancelado e todas as atividades estão canceladas. Também é necessária a solicitação de um gerente para que o processo assuma este estado. A diferença com relação à falha é que somente processos não iniciados podem ser cancelados;

Mixed: este estado é caracterizado quando os componentes do processo estão em diferentes estados sem caracterizar nenhum dos estados anteriores. Por exemplo, um processo que possui uma atividade falhada, outra concluída e uma outra atividade decomposta no estado *requirements* está no estado *Mixed*.

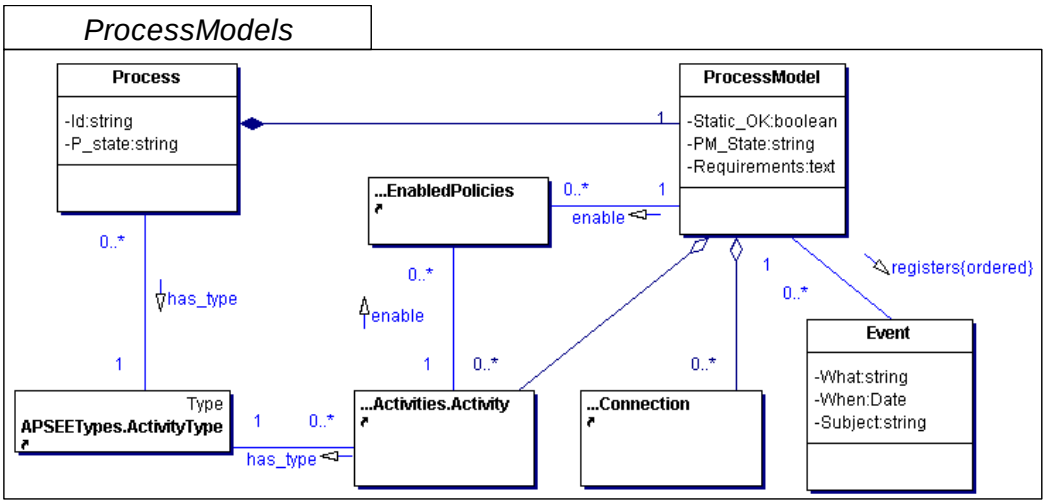


Figura 3.13: Pacote *SoftwareProcesses.ProcessModels*.

Além dessas informações o modelo de processo registra todas as mudanças de estado como eventos ocorridos na classe *Event* e habilita políticas (associação com a classe *EnabledPolicies*) que passam a ser válidas para seus componentes (atividades e sub-processos). Finalmente, na classe *ProcessModel*, o atributo *static_ok* indica se o modelo de processo satisfaz as políticas estáticas habilitadas.

Outra observação que se faz necessária consiste na capacidade de evolução de um modelo de processo. Um modelo de processo no estado *Requirements* transforma-se em um modelo no estado *Abstract* ou *Instantiated*, dentre outras mudanças de estado, conforme será visto na seção 4.3. Portanto, se o usuário pretende criar um processo genérico, que esteja no estado *Abstract*, deverá armazená-lo como *template* para reutilização conforme proposto no trabalho de (REIS, 2002f).

3.7.2 Atividades de um processo

As atividades de um processo de software estão descritas nos pacotes *Activities* e *PlainActivities* (apresentado na próxima seção). O diagrama de classes do pacote *Activities* é apresentado na figura 3.14.

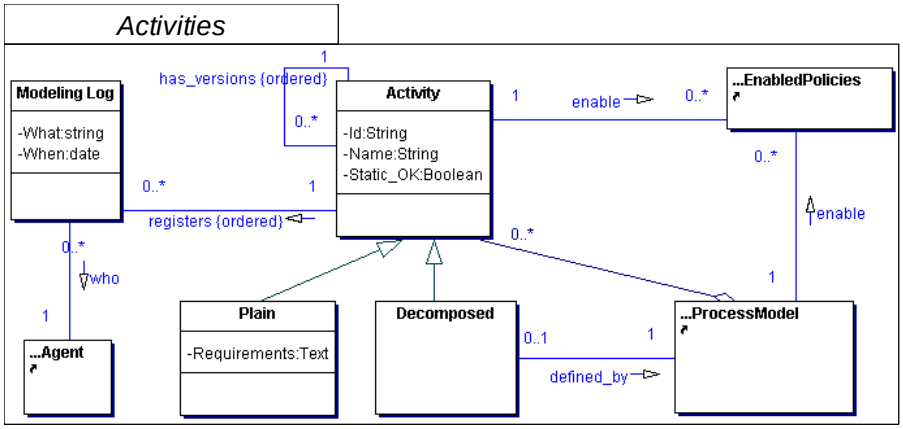


Figura 3.14: Pacote *SoftwareProcesses.Activities*

Uma atividade pode ser simples (*Plain*) ou decomposta (*Decomposed*). Se a atividade for decomposta, então é definida por um novo modelo de processo e seu conteúdo é representado pela classe *ProcessModel* mostrada na seção anterior. Caso contrário, trata-se de uma atividade simples (ou folha) na decomposição do modelo de processo.

As características comuns a todas as atividades são: a habilitação de políticas (associação com a classe *EnabledPolicies*); a indicação de que as políticas estáticas estão satisfeitas (atributo *static_ok*); o registro das modificações (de modelagem) através da classe *Modeling_Log;* e as versões da atividade (associação *has_versions*).

3.7.3 Atividades simples – Normais e Automáticas

O diagrama de classes do pacote *PlainActivities* é mostrado na figura 3.15 e detalha a definição de atividades simples (*Plain*) como normais ou automáticas. Atividades automáticas (classe *Automatic*) não consomem recursos nem tempo e são realizadas através de chamadas a operações de ferramentas integradas no ambiente (por exemplo, compilar código). Já as atividades normais (classe *Normal*) necessitam de recursos, agentes e envolvem artefatos de entrada e saída, os quais são descritos de forma abstrata (tipos) e instanciada (identificadores dos componentes). Além disso, indicam condições que devem ser avaliadas antes e depois da execução da atividade (pré e pós-condição, respectivamente). Ambos os tipos de atividade possuem uma descrição de sua execução (classe *EnactionDescription*) que indica o estado da atividade, suas datas de início e fim e o registro de eventos ocorridos (classe *Event*).

A idéia de armazenar os tipos necessários na descrição da atividade serve para aumentar a flexibilidade da definição do processo. O projetista de processo pode, durante a modelagem, definir apenas o tipo de recurso necessário e decidir qual recurso vai ser efetivamente utilizado no início da execução da atividade. Além disso, a

descrição do processo através dos tipos necessários permite que um modelo de processo possa ser reutilizado em outro contexto, ou mesmo em outra organização.

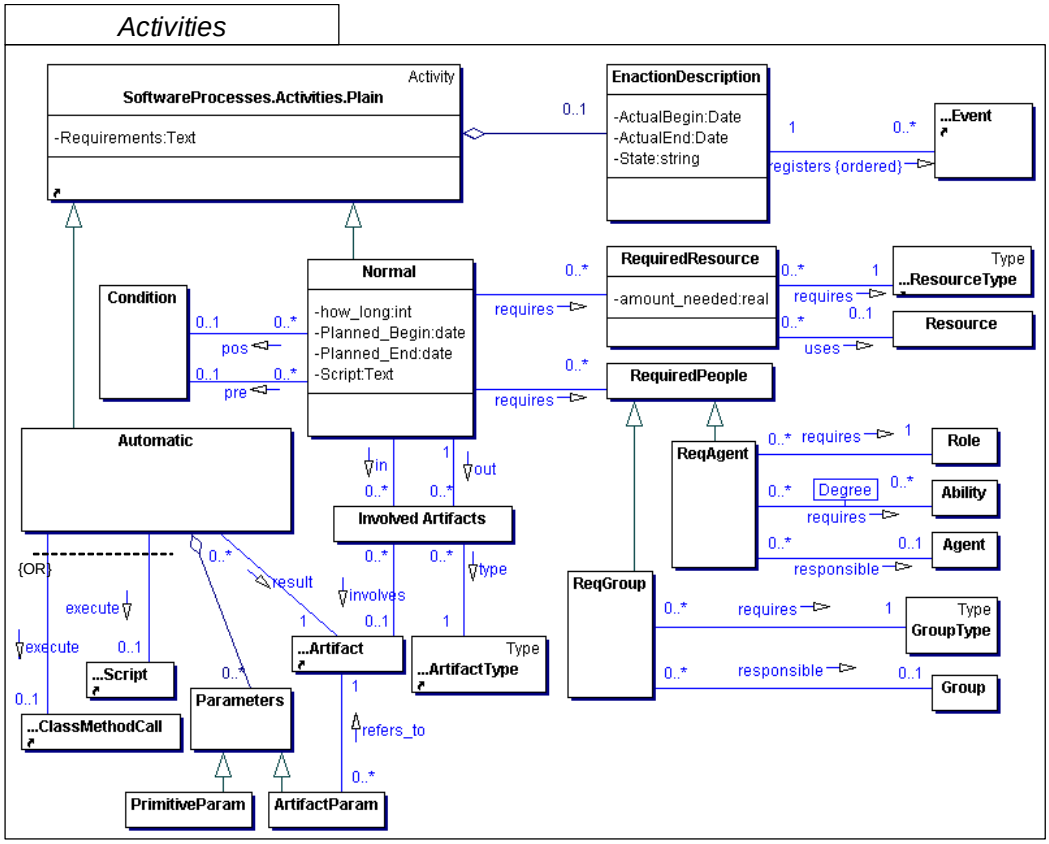


Figura 3.15: Pacote *SoftwareProcesses.PlainActivities*

Atividades normais possuem cronograma abstrato com duração em dias (*how long*), datas de início e fim planejadas (*planned_begin* e *planned_end*), o *script* com os objetivos da atividade; e pré e pós-condições para a sua execução. Enquanto atividades automáticas podem ser executadas a partir de um script de sistema operacional definido pelo usuário (classe *Script*) ou uma chamada a método de uma ferramenta integrada no ambiente (classe *ClassMethodCall*). O projetista de processos pode indicar quais os parâmetros (classe *Parameters*) a serem utilizados na chamada automática e qual o artefato de saída a ser gerado ou modificado (classe *Artifact*). O pacote *Tools*, a ser apresentado na seção 3.9, apresenta com mais detalhes a definição de chamadas automáticas.

O estado da atividade (*state*) na classe *EnactionDescription* é o atributo que armazena a situação de cada atividade e serve de referência para a transição de estados de todo o processo de software. Os possíveis estados de uma atividade simples são descritos a seguir:

- ***Waiting**:* as dependências da atividade ainda não estão satisfeitas;
- ***Ready**:* pronta para começar;

- ***Active**:* a atividade está sendo realizada pelos agentes responsáveis (pelo menos um agente está trabalhando na atividade);
- ***Paused**:* todos os agentes solicitaram pausa da atividade;
- ***Finished**:* a atividade foi concluída. Se a atividade for cooperativa, significa que todos os agentes concluíram;
- ***Cancelled**:* a atividade foi cancelada antes de iniciar;
- ***Failed**:* a atividade falhou após o início por decisão dos agentes ou do gerente.

3.7.4 Conexões entre atividades

As conexões servem para interligar atividades representando o fluxo de controle e de dados do processo. Para a definição do fluxo de controle foram levados em consideração os tipos de dependência necessários e desejáveis para processos de software, alguns dos quais encontrados em ambientes da literatura, como Trigsflow (KAPPEL et al., 1998) Dynamite (KRAPP, 1997; 1998) e a linguagem Promenade (RIBÓ; FRANCH, 2000; 2001). O modelo resultante leva em consideração a necessidade de modelagem de processos utilizando fluxos de controle de alto nível que possam ser refinados durante a execução e que expressem vários tipos de precedência entre atividades. Portanto, são propostos os fluxos de controle de seqüência, de *feedback* (retorno a uma atividade anterior) e envolvendo várias atividades (*branch* e *join*). Esses fluxos de controle e de dados são representados no modelo através de conexões que são chamadas de: conexões simples, conexões múltiplas e conexões de artefato.

As conexões simples podem ser de seqüência ou de *feedback*, enquanto que as conexões múltiplas são *branch* e *join* combinados com operadores lógicos. As conexões simples e múltiplas também possuem um tipo de dependência (*end-start*, *start-start*, *end-end*) que influencia diretamente na execução das atividades e pode representar diferentes dependências encontradas em processos reais. As conexões de artefato não influenciam no fluxo de controle do processo, servindo para descrever o fluxo de informações entre atividades, mas podendo ser ligadas a conexões múltiplas.

O diagrama de classes do pacote *Connections* é mostrado na figura 3.16 e seus componentes representam as conexões tendo como elemento central atividades (classe *Activity*). Os tipos de conexão e dependências são detalhados a seguir.

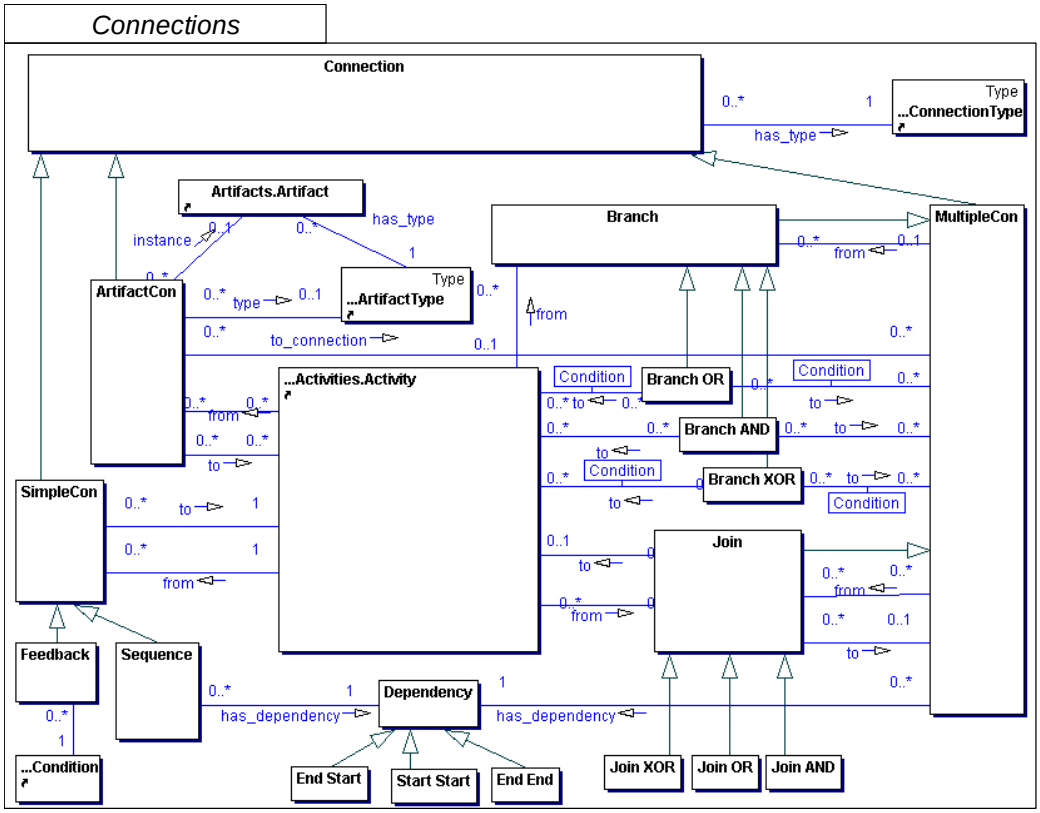


Figura 3.16: Pacote *SoftwareProcesses.Connections*.

- **Conexão Simples (*SimpleCon*)**: Define o fluxo de controle entre duas atividades (origem e destino). Este fluxo pode ser de seqüência ou de *feedback*.
  - **Conexão simples de seqüência**: o fluxo de seqüência indica que a atividade destino depende da atividade origem. Os tipos de dependência entre atividades podem ser:
    a) ***End-Start***: a atividade destino somente pode iniciar quando a origem terminar;
    b) ***Start-Start***: a atividade destino pode iniciar somente após o início da origem;
    c) ***End-End***: a atividade destino somente pode terminar quando a origem terminar;
  - **Conexão simples de *Feedback***: O fluxo de *feedback* é utilizado para indicar o retorno a uma atividade já realizada desde que uma condição seja satisfeita. Nesta conexão a atividade destino (alvo) é uma atividade anterior à origem e a sintaxe (a ser explicada na seção 4.1) define que deve haver um fluxo de execução entre a atividade destino (alvo) e a origem do *feedback*;
- **Conexão Múltipla (*MultipleCon*)**: Também define o fluxo de controle, porém pode envolver várias atividades e conexões múltiplas. Existem dois

tipos de conexão múltipla, ambos subdivididos pela aplicação de operadores lógicos (AND, EX-OR, In-OR) e tipos de dependência (*end-start, start-start, end-end*):

- ***Branch***: possui uma atividade ou conexão múltipla origem e várias atividades ou conexões múltiplas destino. Um exemplo de aplicação é que a partir do término da atividade origem, todas as atividades destino podem começar (*Branch AND end-start*). A semântica da conexão depende do operador lógico associado (And - todas, XOR – somente uma ou OR – um subconjunto de), da condição lógica associada a atividade destino, quando o operador for OR ou XOR e do tipo de dependência utilizado (*end-start, start-start* ou *end-end*);
- ***Join***: possui várias atividades ou conexões múltiplas origem e uma atividade ou conexão múltipla destino. Uma aplicação dessa conexão seria que assim que todas as atividades origem terminarem, então uma atividade destino pode começar (*Join AND end-start*). Os operadores lógicos e tipos de dependências também se aplicam ao *Join*;

- **Conexão de Artefato (*ArtifactCon*)**: Define o fluxo de dados do processo através da passagem de artefatos entre atividades. Um artefato produzido por uma atividade pode servir de entrada para outra e essa informação é explicitada através da conexão de artefato. Assim, uma conexão de artefato pode ter como origem várias atividades e como destino várias atividades ou conexões múltiplas (*Branch* ou *Join*). Este último caso de conexão múltipla como destino serve para simplificar a definição do modelo de processo uma vez que a semântica é a mesma de ligar a conexão de artefato com cada atividade/conexão destino da conexão múltipla. O efeito da simplificação é mais claro com conexões *Branch* quaisquer e com *Join* que possuem como destino outra conexão múltipla.

A reunião e refinamento no mesmo meta-modelo desses vários tipos de conexões é uma contribuição importante. Alguns tipos de conexão foram propostos a partir de fluxos de controle já existentes, porém com mais características não encontradas em outros ambientes da área, como por exemplo, ligação entre conexões múltiplas, uso de condições para definir automaticamente o retorno (*feedback*), a escolha do destino através de condições em conexões *Branch Or/XOR* e possibilidade de refinamento da conexão (definição do tipo de dependência, por exemplo) durante execução. Além disso, as conexões propostas possuem semântica formal (definida no próximo capítulo), e sua definição foi fortemente encorajada pelo requisito de formalismos de alto nível com semântica formal (HUFF, 1996). A maioria dos PSEEs encontrados na literatura provê fluxos de controle simples de dependência *end-start* entre atividades, o que é muito restritivo em se tratando de processos de desenvolvimento de software.

3.8 *Artifacts* – Objetos do Ambiente

Uma das principais tarefas dos ambientes de desenvolvimento de software é armazenar, estruturar e controlar o desenvolvimento de informação, produtos de software e os objetos gerados durante o desenvolvimento de software. Esses objetos são, por natureza, mais complexos que os objetos tratados por sistemas de banco de dados tradicionais, pois são mais estruturados e requerem operações complexas. Além disso,

são fortemente inter-relacionados e sensíveis ao tempo (alguns objetos possuem diferentes versões) e espaço (as versões podem estar distribuídas) (GODART; CHAROY, 1994).

O ambiente deve prover um módulo que seja responsável por controlar o acesso e a evolução de objetos compartilhados. Geralmente este módulo é conhecido como Repositório do ambiente e é implementado através de um sistema de gerência de banco de dados (SGBD). As versões dos documentos e produtos de software devem ser gerenciadas para permitir cooperação e consistência.

O ambiente ao qual o APSEE está sendo integrado é o ambiente PROSOFT, o qual já possui um componente para tratar de gerência de objetos, chamado PROSOFT Cooperativo (REIS, 1998a; REIS et al., 1998b). Este componente trata objetos de qualquer tipo no ambiente, guardando seu estado, gerenciando o seu acesso e suas versões. O componente PROSOFT Cooperativo foi integrado na arquitetura do APSEE dentro do componente *Artifacts*, apresentado na figura 3.17.

Apesar da existência de um modelo para gerência de objetos cooperativos (REIS, 1998a; REIS et al., 1998b), a classe *Artifact* foi criada para acrescentar mais atributos que se mostraram importantes do ponto de vista de gerência de processos de software. Assim, um artefato possui as seguintes características:

- Um **identificador** (*id*), um **tipo** na hierarquia de tipos de artefatos do APSEE-Types e uma **descrição**;
- **Políticas** habilitadas (*EnabledPolicies*) associadas ao artefato. Políticas estáticas (REIS et al., 2001b) podem garantir que propriedades do artefato são atendidas;
- O **objeto cooperativo** em si (*CoopObject*), que representa o uso do componente Objeto Cooperativo definido em (REIS, 1998a; REIS et al., 1998b), com as mesmas características;
- Uma relação de **derivação** entre artefatos (associação *derived_from*), indicando qual artefato serviu de base para a construção deste. Neste caso, pode-se representar a informação de que um diagrama de classes em UML derivou de uma especificação de requisitos do sistema;
- Uma relação de **composição** entre artefatos (associação *belongs_to*). Por exemplo, um artefato que representa uma especificação de requisitos pode conter vários diagramas e textos.

A classe que representa o objeto cooperativo associado ao artefato é *CoopObject*. Conforme definido em (REIS, 1998a; REIS et al., 1998b), um objeto cooperativo possui versões (neste caso, associadas a um artefato), identifica qual o objeto ancestral (associação *ancestral*), possui atributos para indicar sua localização no sistema (*pathname* e *filename*), atributos para indicar quando foi modificado (*last_modify_date* e *last_modify_time*) e por quem (associação *last_modify* com *Agent*) e finalmente indica como foi gerado através da associação com *ToolDefinition*, *Script* ou *ClassMethodCall*.

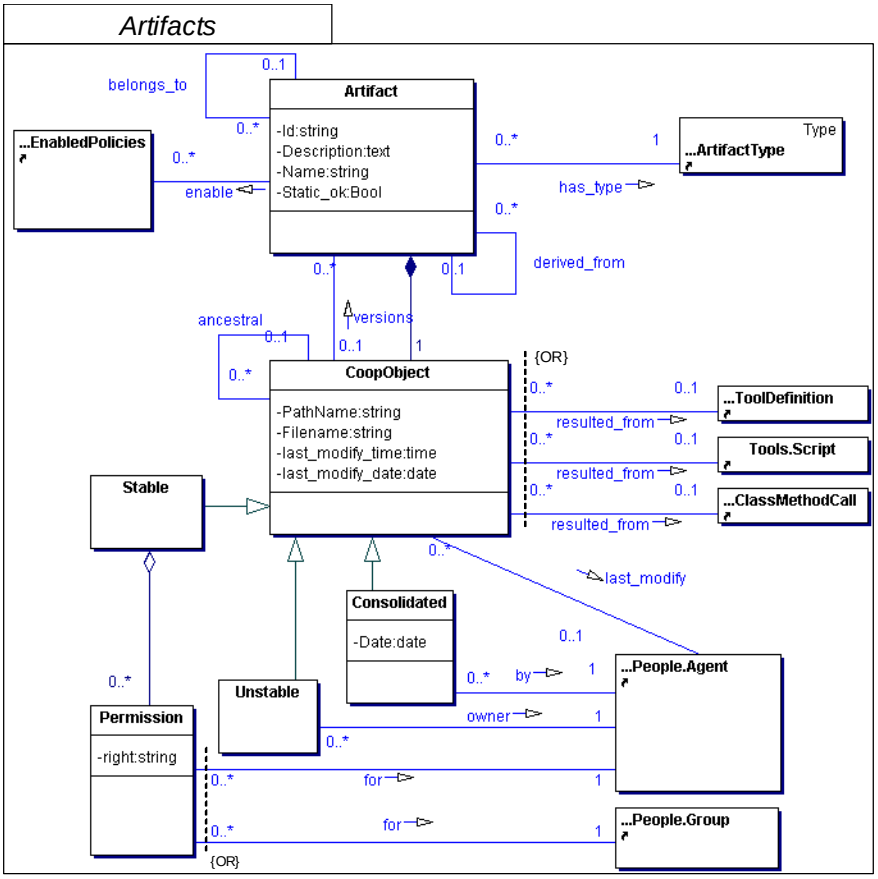


Figura 3.17: Pacote *Artifacts*.

Existem três tipos de objetos cooperativos: Instável (*Unstable*), Estável (*Stable*) e Consolidado (*Consolidated*). Quando criado, o objeto é instável e, nesse caso, é indicado o agente que o criou (associação *owner*) como sendo o seu único usuário. Quando o criador decide compartilhar o acesso ao objeto, ele solicita que o objeto torne-se estável e pode, nesse caso, definir as permissões de acesso ao objeto através da classe *Permission*. Cada permissão pode ser dada a um agente ou a um grupo. O atributo *right* indica se o direito concedido é de gerência (nesse caso o usuário pode manipular o artefato e suas permissões), leitura (o usuário não pode modificar o objeto), escrita (o usuário pode modificar o objeto) ou nenhum (o usuário não possui direitos de acesso ao objeto). Neste estado o objeto pode ser editado cooperativamente pelos seus usuários.

Um usuário com direito de gerência pode solicitar consolidação do objeto, que passa a ser instância da classe *Consolidated*. Neste estado, o objeto não pode ser modificado, porém podem ser geradas versões do mesmo.

3.9 *Tools* – Ferramentas do Ambiente

As informações sobre as ferramentas integradas no ambiente são apresentadas no diagrama de classes da figura 3.18. A classe *ToolDefinition* define ferramentas do

ambiente como sendo *ATOs*[8] (ferramentas do ambiente PROSOFT) ou ferramentas externas ao PROSOFT. Essas ferramentas estão associadas a tipos de ferramentas no *APSEE-Types* e também identificam quais tipos de artefatos podem manipular (associação com *ArtifactType*).

Devido à necessidade de executar atividades automáticas, são definidos os *scripts* (classe *Script*) e as chamadas de métodos de classes (classe *ClassMethodCall*). Um *script* define a execução de operações baseadas em primitivas do sistema operacional ou do ambiente operacional de apoio. Assim, o usuário pode definir um conjunto de operações do sistema operacional em um arquivo e cadastrar esse arquivo como um *script* que pode ser chamado por uma atividade automática. *Scripts* podem estar associados a parâmetros e definem o tipo de artefato resultante (associação *result*).

Uma instância de *ClassMethodCall* define uma chamada a um método Java e, assim como os *scripts*, define o tipo de artefato resultante e pode necessitar de parâmetros. O método sendo chamado pode ser de qualquer classe instalada no sistema, porém precisa ser previamente definido como instância nesta classe.

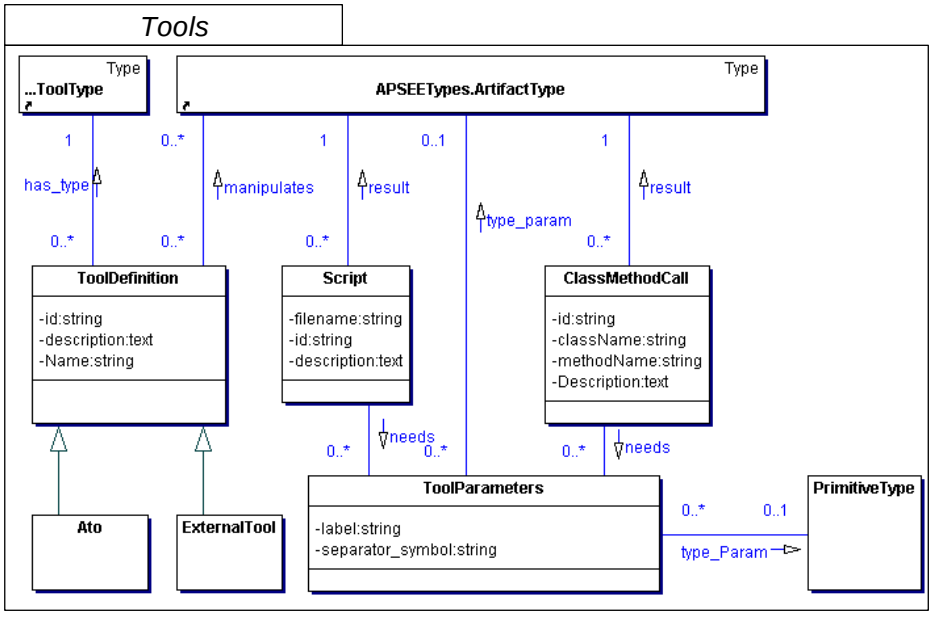


Figura 3.18: Pacote *Tools*.

3.10 *ProcessKnowledge* – Conhecimento sobre os Componentes do Modelo

O componente *ProcessKnowledge* do APSEE (ver figura 3.19) funciona como uma base de conhecimento sobre todos os componentes do meta-modelo. A construção desse componente foi baseada nas pesquisas sobre gerência de processos e projetos, métricas de processo e produto, qualidade de software e assuntos relacionados. Dentre os trabalhos pesquisados podemos citar: Rocha et al. (2001), Pressman (2001), Jones

[8] ATO – Ambiente de Tratamento de Objetos.

(1996), Fernandes (1995), Florac (1999), Franca (1998) e Humphrey (1989; 1995; 1997). A pesquisa nesta área indica que o conhecimento útil a ser armazenado sobre a realização de processos de desenvolvimento de software, além da própria descrição do processo executado, pode ser modelado como métricas. Métricas são fundamentais para qualquer disciplina de engenharia, pois permitem conhecer e comparar o assunto sendo estudado. Sem métricas, somente julgamentos subjetivos podem ser feitos sobre o desempenho de um processo de software. Além disso, a partir de métricas, melhores estimativas podem ser obtidas (PRESSMAN, 2001). A importância das métricas é consenso na literatura sobre Engenharia de Software, porém a coleta de métricas úteis ainda é sugerida como uma tarefa manual, como em (HUMPHREY, 1997). Isto se deve à falta de ambientes integrados de gerência de processos que permitam consultar conhecimento sobre processos executados e transformá-los em métricas úteis, ou que auxiliem a entrada de métricas em um repositório de conhecimento do ambiente.

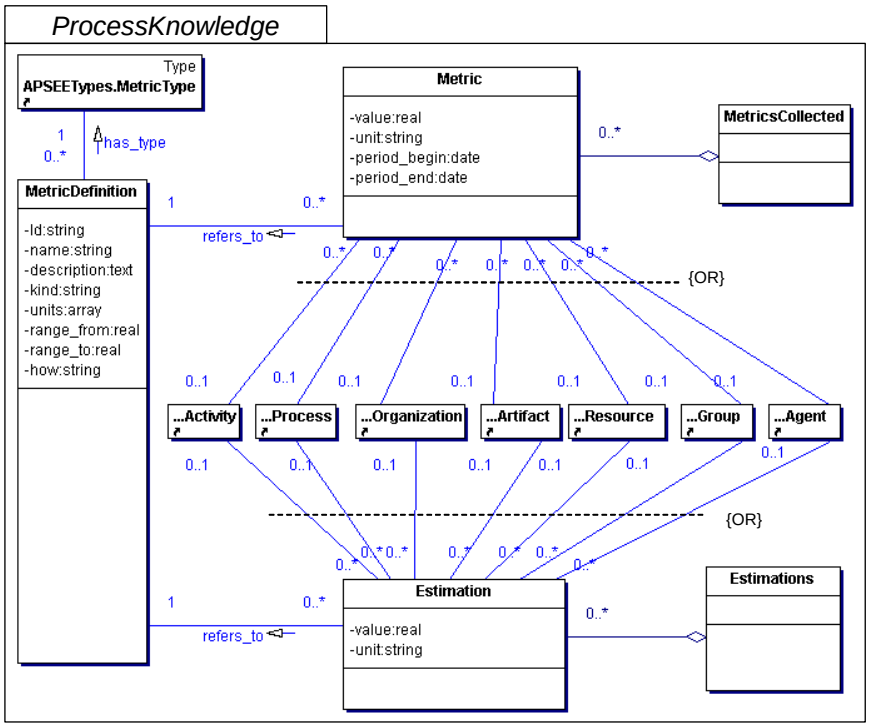


Figura 3.19: Pacote *ProcessKnowledge*.

Apesar do avanço na tecnologia de processos de software, as soluções propostas em PSEEs são, na maioria das vezes, isoladas, dificultando a reutilização de conhecimento adquirido anteriormente. O uso de PSEEs pela indústria tem sido limitado devido à dificuldade inerente de sua adoção, problemas com a interação com os usuários, dificuldade de utilização da PML adotada e baixa capacidade de reutilização de conhecimento adquirido anteriormente (FUGGETTA, 2000; SOUZA et al., 2001). A abordagem APSEE contribui propondo integração das ferramentas necessárias ao ciclo de vida proposto em um meta-modelo comum. O componente *ProcessKnowledge* está sendo integrado para ser consultado nas fases de modelagem, instanciação, simulação e execução, além de permitir estudos empíricos acerca do impacto de algumas métricas em aspectos importantes, como por exemplo, na qualidade do software produzido.

É importante observar que o conhecimento armazenado nas classes apresentadas nesta seção é o conhecimento factual (ABEL, 1994) sobre os componentes do modelo, enquanto que o conhecimento heurístico que pode ser definido pelo usuário é representado por políticas.

A classe *MetricDefinition* permite definir métricas importantes para o processo e para a organização enquanto que as classes *MetricsCollected* e *Estimations* armazenam instâncias de métricas associadas aos componentes do modelo.

Florac (1999), Rocha et al. (2001) e Jones (1996) definem várias métricas que servirão como conjunto inicial de métricas definidas no APSEE. As características principais sobre as instâncias da classe *MetricsDefinition* são:

- Possuem um identificador único, estão associadas a um tipo de métrica em APSEE-Types, possuem um nome e uma descrição textual;
- O tipo de métrica sendo definida (*Kind*): de processo, de produto ou individual;
  - **Métrica de Processo**: indica uma característica de um processo ou atividade. Por exemplo, esforço de um processo, tempo total do processo, percentual de atividades atrasadas, percentual de atividades concluídas, dentre outros;
  - **Métrica de Produto**: indica uma característica de um artefato produzido ou consumido por atividades. Podem ser métricas de tamanho, por exemplo, pontos por função, ou de qualidade, por exemplo, defeitos por ponto por função;
  - **Métricas Individuais**: indicam características de pessoas (agentes e grupos) e recursos envolvidos no processo. Estas métricas também podem se aplicar a cargos ou à organização como um todo. Podem ser definidas métricas que representam a produtividade de agentes, quantidade de defeitos produzidos, dentre outros. O uso de métricas individuais têm sido motivo de controvérsia entre os autores da área (PRESSMAN, 2001). Entretanto, podem-se definir alternativas para o uso controlado das mesmas.
- Unidades de medida da métrica (*unit*). Cada métrica pode ter várias unidades de medida. Por exemplo, a métrica “Tamanho” pode ser medida por “linhas de código” ou “pontos por função”. Entretanto a definição de métricas é livre e pode ser criada uma métrica diferente para tamanho em linhas de código e outra para tamanho em pontos por função;
- Intervalo de validade da métrica (*range_from* e *range_to*). Pode ser útil armazenar o valor mínimo e máximo permitido para uma métrica sendo definida. Por exemplo, a afinidade total em um grupo de desenvolvimento (obtida a partir das afinidades individuais) somente pode ser válida entre os valores 0 e 1.
- Como a métrica pode ser obtida (*how*). Neste campo pode ser registrado o método utilizado para medir, como por exemplo, “cálculo automático”, “contagem manual” ou “chamada de operação de uma ferramenta”, dentre outros.

É importante observar que os itens associados a métricas e estimativas nesse modelo são os propostos nesse trabalho, e por isso têm a limitação de associar o conhecimento (métrica ou estimativa) a uma instância de processo. A partir da integração deste modelo com o proposto no Apsee-*Reuse* (REIS, 2002f), será possível associar métricas para um processo genérico (*template*), agrupando nesse caso o conhecimento sobre todas as ocorrências de execução daquele *template*.

3.10.1 Métricas

As métricas coletadas na classe *MetricsCollected* contêm as instâncias de métricas (classe *Metric*) que estão associadas a definições de métricas (*MetricDefinition*). Cada métrica possui as seguintes características:

- Uma associação indicando a definição da métrica em *MetricDefinition*;
- Uma associação indicando o elemento medido. Neste caso, o elemento pode ser um artefato, um processo, uma atividade, um recurso, um agente, um grupo ou toda a organização. Deve-se observar que o elemento medido deve ser compatível com o tipo de métrica definido pela associação com *MetricDefinition*;
- O valor medido (*value*);
- A unidade de medida (*unit*), que deve ser uma unidade válida para a métrica;
- O período da medição (*period_begin, period_end*). Em alguns casos, pode-se medir várias vezes o mesmo elemento do processo variando o período da medição.

3.10.2 Estimativas

Estimativas permitem que a evolução de um processo ou de um produto sejam avaliadas continuamente. As estimativas podem ser comparadas com as métricas coletadas, e por isso, possuem estrutura semelhante, mostrada na figura 3.19.

A principal diferença entre os atributos das estimativas e das métricas coletadas está no período da medição (atributos *period_begin* e *period_end*) da classe *MetricsCollected*, que não se mostrou necessário em *Estimations*. O principal motivo é que estimativas para os elementos de processo e produto não são medidas por períodos. Dificilmente estimativas são utilizadas para prever a qualidade de um artefato em um determinado período, normalmente se estima a qualidade final do mesmo. Da mesma forma, não se estima o esforço de uma atividade num determinado período, e sim para o período em que ela é executada. Por outro lado, o período é um atributo importante em métricas pois pretende-se medir constantemente o desempenho de pessoas, recursos, grupos e organização. E essas medições são periódicas devido à natureza permanente desses componentes. Por exemplo, não há muita vantagem em estimar a produtividade do desenvolvedor, e sim utilizar sua produtividade anterior para estimar a qualidade do produto que o desenvolvedor irá construir. Devem-se utilizar métricas com um certo cuidado quando se trata de produtividade pessoal, pois esta pode ser influenciada por inúmeros fatores, os quais nem sempre podem ser previstos com clareza, segundo (PRESMAN, 2001).

3.11 *Policies* - Políticas de Processo

Segundo Feiler e Humphrey (1993), políticas de processos de software consistem em "princípios que conduzem o desenvolvimento e/ou a execução de processos de software". No modelo APSEE, 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 contém critérios da organização definidos pelo usuário e pode estar habilitada em uma atividade, em um processo (estando habilitada em todas as atividades componentes), ou na organização (isto é, em todos os processos existentes na organização), permitindo a sua reutilização em diferentes contextos.

Na abordagem APSEE podem ser definidos três tipos principais de políticas com diferentes finalidades. São elas:

- **Políticas Estáticas:** Apóiam os construtores de modelos de processo, pois definem regras sintáticas para a formação dos modelos. Podem ser úteis para definir formalmente boas práticas de gerenciamento de projetos que podem ser reutilizadas em diferentes processos. As políticas estáticas foram propostas por Reis e apresentadas em (REIS, 2002f; REIS et al., 2002b);
- **Políticas de Instanciação:** Definem critérios para a alocação de recursos e pessoas utilizadas por uma organização em contextos ou processos específicos, podendo ser baseadas no histórico da execução de processos anteriores;
- **Políticas Dinâmicas:** Baseiam-se nos eventos que podem ocorrer durante a execução de processos e realizam ações em resposta aos eventos de acordo com estratégias definidas pelo usuário. Este tipo de política pode ser representado por regras ECA – Evento-Condição-Ação e pode ser encontrado mais facilmente em PSEEs que utilizam o paradigma de execução baseada em regras (citado na seção 2.2.2).

No meta-modelo APSEE existe um componente que armazena as políticas disponíveis na organização definido no pacote Policies, representado na figura 3.20. Na figura estão representados os pacotes que definem os três tipos de políticas citados.

A definição de Políticas de Instanciação é proposta neste trabalho como complemento à definição de processos de software. Portanto, o meta-modelo de políticas está associado ao meta-modelo de processos de software. Com essa associação, é possível auxiliar a instanciação de recursos e pessoas objetivando aumentar o nível de automação e a flexibilidade da execução de processos.

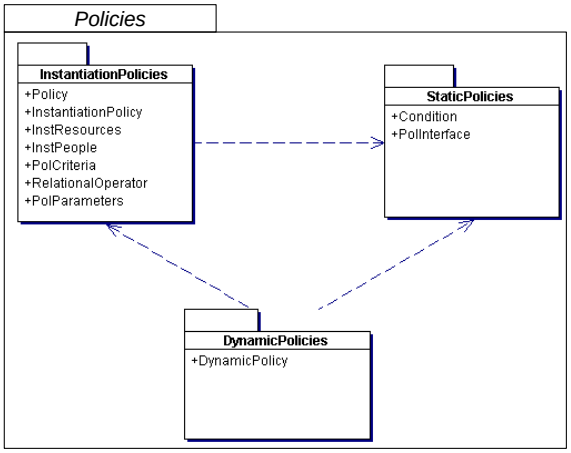


Figura 3.20: Pacote *Policies*.

3.11.1 *InstantiationPolicies* - Políticas de Instanciação

O apoio existente atualmente para a gerência de processos claramente falha em prover soluções adequadas para a instanciação de processos (CHANG et al., 2001). Por exemplo, não é comum encontrar PSEEs que especificam precisamente a estrutura do modelo organizacional, o que os faz diminuir o potencial de aproveitamento dessa estrutura para auxiliar na definição de estratégias para a fase de instanciação. De acordo com Chang et al (2001), a maioria das técnicas e ferramentas existentes apenas provê rastreamento passivo de projetos e suporte a relatórios. Por isso, os gerentes são obrigados a tomar todas as decisões baseando-se em sua experiência pessoal.

O suporte automatizado à instanciação depende de meta-modelos adequados (LERNER et al., 2000) e envolve a definição de formalismos para expressar estratégias de alocação a serem selecionadas dinamicamente. Como resultado, mecanismos de instanciação devem prover aos gerentes e projetistas informações que apontem para recursos e agentes disponíveis e mais adequados a atividades específicas. Esses mecanismos possuem alguns requisitos que influenciam na definição do meta-modelo, são eles:

- A necessidade de definir precisamente as propriedades dos recursos e agentes no meta-modelo do PSEE. Somente com informações detalhadas sobre esses componentes é possível fazer sugestões com critérios mais adequados, e não somente a partir da disponibilidade. Este requisito está sendo atendido pelo meta-modelo através do pacote *Organization*;
- A literatura especializada fornece descrições informais de estratégias genéricas de instanciação (por exemplo, (DAVIS, 1995)). Entretanto, para prover suporte automatizado à instanciação, é importante fornecer paradigmas adequados para expressar formalmente essas estratégias;
- É importante que as estratégias de alocação possam ser definidas em componentes de processo de baixa granularidade, permitindo que regras gerais sejam refinadas ou ainda que casos específicos sejam tratados de forma adequada. Neste caso, a habilitação de políticas em atividades que podem ser decompostas permite o atendimento a este requisito;

- O mecanismo de instanciação deve tratar adequadamente a escalabilidade das estratégias de alocação, já que o aumento da quantidade dessas estratégias pode aumentar a quantidade de conflitos. Os conflitos ocorrem quando várias estratégias de alocação podem ser adotadas simultaneamente para o mesmo caso;
- A seleção de estratégias de instanciação pode depender do estado corrente do processo ou da organização. Por exemplo, a estratégia a ser utilizada para sugestão de um recurso para uma atividade atrasada pode ser diferente da estratégia utilizada para o mesmo caso se o atraso não ocorre;
- Instanciação automática de processos deve estar disponível. Em alguns casos pode ser útil prevenir atrasos na execução de processos (por exemplo, quando uma atividade não puder executar por falta de detalhes de instanciação). Entretanto, a decisão sobre a habilitação ou não de instanciação automática deve depender do gerente de processos.

Considerando esses requisitos, as estratégias de alocação de recursos e agentes em atividades de processos de software são propostas através do conceito de Políticas de Instanciação. Esse conceito busca definir aspectos que podem ser habilitados em diferentes pontos de processos e visam orientar sobre a melhor estratégia para instanciação em cada situação.

O meta-modelo de políticas de instanciação é apresentado na figura 3.21. Neste modelo, políticas de instanciação dividem-se em políticas de instanciação de recursos e de pessoas.

As características comuns às políticas de instanciação de recursos e pessoas são:

- Identificador da política (*Id*); nome da política (*name*); e descrição textual (*description*) da política; tipo da política (associação com classe *PoliceType*);
- Interface da Política (*PolInterface*). Especifica o tipo de objeto APSEE que será tratado na condição da política, ou seja, o objeto cujas propriedades são avaliadas. A interface pode ser *Resource* ou *Activity*. O valor *default* deste atributo é *Activity*;
- Condição (*Condition*). Condição lógica que deve ser satisfeita para que os critérios da política sejam seguidos no momento da instanciação. As condições são formadas através de funções pré-definidas de acesso e verificação dos componentes do ambiente. As condições permitem que a situação corrente do processo seja avaliada para que a política seja aplicada;
- Critérios de Restrição (associação *RestrictBy* com classe *PolCriteria*). Se as condições forem satisfeitas, os critérios de restrição definidos pelo usuário são levados em consideração para instanciação. Estes critérios ajudam a diminuir o espaço de busca por recursos compatíveis a serem alocados;
- Critério de Ordenação (associação *OrderBy* com classe *PolCriteria*). Após aplicação dos critérios de restrição, o critério de ordenação é utilizado para ordenar a sugestão de recursos para o usuário. Caso não haja critério de ordenação, a sugestão será ordenada pelo menor custo.

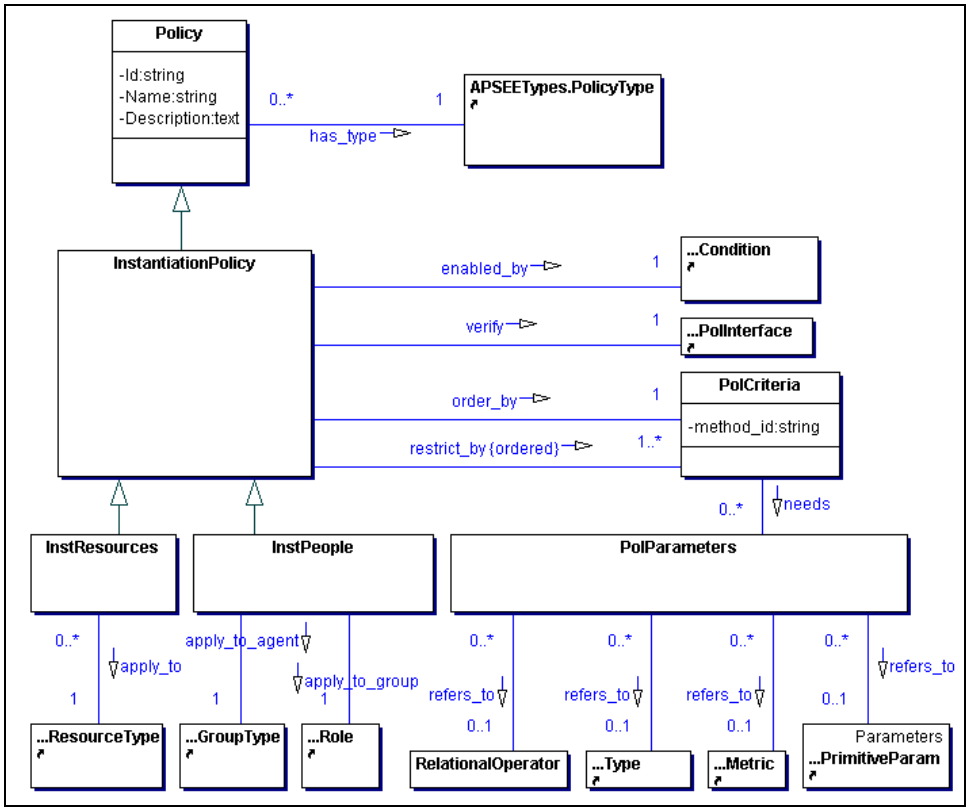


Figura 3.21: Pacote *InstantiationPolicies*.

A diferença entre as políticas para recursos e para pessoas está no tipo ao qual a política se aplica. Cada política de instanciação de recursos pode ser aplicada a um tipo de recurso (associação *apply_to* com classe *ResourceType*), enquanto cada política de instanciação de agentes pode ser aplicada a um cargo (*Role*) ou a um tipo de grupo (*GroupType*).

Uma atividade pode ter várias políticas de instanciação de recursos (e de agentes) habilitadas, porém cada uma instancia um tipo de recurso. Por exemplo, uma política que se aplica ao tipo “Handheld” somente será levada em consideração em atividades que necessitam de recursos desse tipo ou seus subtipos (“PalmOS” e “PocketPC”, por exemplo);

O conceito de políticas é ortogonal ao meta-modelo de processos, de recursos e de agentes. A figura 3.22 ilustra como as políticas podem ser habilitadas em atividades e processos e como cada política se aplica a um tipo de recurso.

A próxima seção apresentará o meta-modelo onde o controle da aplicação das políticas é armazenado. Posteriormente, a seção 4.2 do próximo capítulo apresentará o funcionamento do mecanismo de instanciação que usa o modelo de políticas apresentado.

3.12 *Planner_Info* - Meta-modelo de Apoio à Instanciação de Processos

As políticas de instanciação são levadas em consideração na sugestão dos recursos a partir da definição dos tipos necessários. Esta sugestão considera o tipo do recurso, a disponibilidade do recurso para o período da atividade e a disponibilidade dos recursos requeridos.

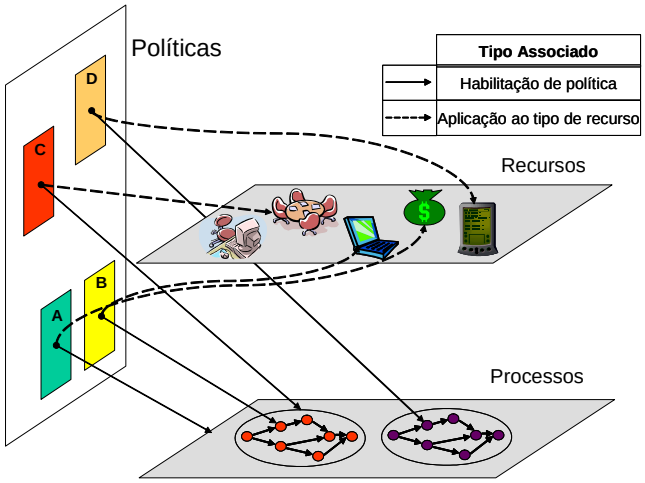


Figura 3.22: Visão em camadas da associação de políticas a processos e recursos.

Para registrar as sugestões fornecidas pelo mecanismo de instanciação (chamado *APSEE-Planner*), foi criado o pacote *Planner_Info*, apresentado na figura 3.23. Apesar do pacote estar inserido no contexto da organização, está sendo apresentado por último porque sua apresentação requer conceitos do pacote de políticas apresentado na seção anterior. As classes apresentadas neste pacote dizem respeito à instanciação de recursos, mas uma estrutura análoga é utilizada para instanciação de agentes.

O pacote apresentado possui como classe principal a classe *ResourcePolicyLog*, composta de vários registros de instanciação de atividades (classe *ActivityInstantiated*). Para cada atividade instanciada são registradas várias sugestões de instanciação de recursos (classe *ResourceSuggestion*). Uma sugestão referencia a política adotada, o tipo de recurso requerido, uma lista de recursos sugeridos e o recurso escolhido (pelo gerente ou automaticamente).

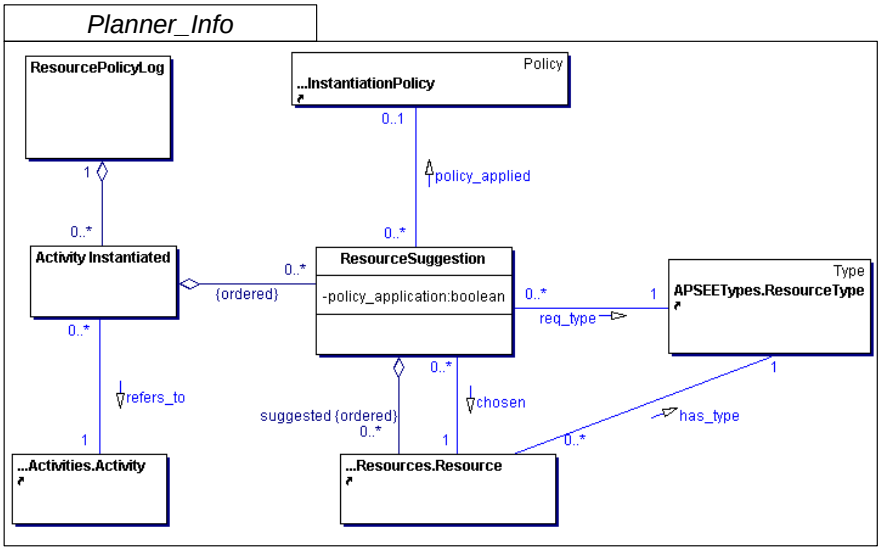


Figura 3.23: Pacote Organization.*Planner_Info* usado para registrar a instanciação de processos.

4 MECANISMOS DE GERÊNCIA DE PROCESSOS DE SOFTWARE DO APSEE

Este capítulo apresenta os componentes da abordagem APSEE propostos com o objetivo de aumentar o nível de automação e flexibilidade na gerência de processos de software. Apesar desses componentes estarem fortemente integrados ao meta-modelo apresentado no capítulo anterior, este capítulo concentra-se no nível intermediário da arquitetura do APSEE[9], mais especificamente nos componentes APSEE-PML, APSEE-*Planner* e Mecanismo de Execução de Processos (destacados na figura 4.1, que mostra os elementos da figura 3.1 que interessam neste capítulo).

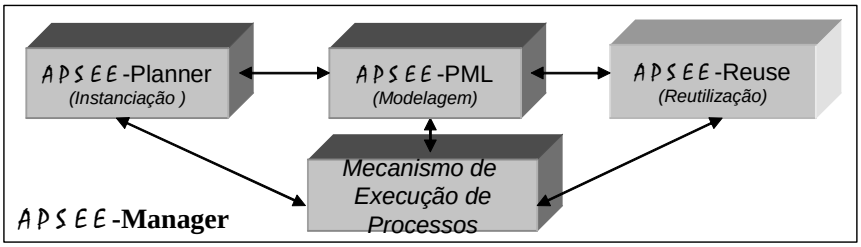


Figura 4.1: Mecanismos de Gerência de Processos - Nível intermediário da arquitetura do APSEE.

Para atender os requisitos de execução flexível de processos - descritos no capítulo 2 - foi necessário propor uma linguagem de modelagem e um mecanismo para instanciação de processos, além do mecanismo de execução propriamente dito, os quais são apresentados neste capítulo. Como esses componentes atendem diferentes fases do ciclo de vida de um processo de software, a integração dessas fases no meta-modelo proposto também será apresentada neste capítulo.

4.1 Modelagem de Processos de Software no APSEE

A modelagem de processos de desenvolvimento de software requer formalismos de alto nível para descrever aspectos de coordenação das atividades e relacionamento com a organização. Estes formalismos devem ser convenientes não somente para representação, mas também para execução. Com base nisso é proposto um formalismo

[9] A separação do conteúdo dos capítulos por nível de arquitetura visa melhorar a organização do texto.

de modelagem para o APSEE que permite representação gráfica de processos criados a partir do meta-modelo mostrado no capítulo anterior.

A linguagem de modelagem de processos de software do APSEE (denominada APSEE-PML) é baseada em redes de atividades que podem ser decompostas. Neste formalismo, um modelo de processo pode ser construído a partir de símbolos gráficos conectados e o detalhamento do relacionamento com os outros componentes do modelo (como por exemplo, as informações organizacionais e descrição dos artefatos, entre outros) é feito através de formulários específicos que apóiam essa tarefa. A gramática da linguagem é definida formalmente no próximo capítulo, onde as dependências e restrições de consistência para a construção de processos serão apresentadas. Cabe ressaltar que os símbolos adotados para a linguagem podem ser modificados futuramente visando melhorar a visualização do modelo. Na verdade é prevista a proposta futura de um mecanismo que transforme a notação gráfica do processo de acordo com a necessidade do usuário.

O meta-modelo APSEE considera a necessidade de apoiar execução de forma flexível, permitindo alternativas de caminhos de execução, retorno a atividades anteriores dependendo de uma condição, mudanças dinâmicas, possibilidade de instanciação automática de partes do processo durante a execução, dentre outros requisitos desta proposta já discutidos no capítulo 2.

Na linguagem de modelagem do APSEE, os nodos são atividades e conexões entre atividades, sendo que as conexões detalham o fluxo de controle e de dados do processo. Com o uso do paradigma de redes de tarefas não há um mapeamento direto para um formalismo de execução de processos, como por exemplo, regras ou redes de Petri. Por outro lado, há uma certa liberdade em propor construtores diferenciados para a linguagem, que simplifiquem o entendimento da modelagem e que tenham uma semântica bem definida.

Além da definição do formalismo como uma rede de tarefas, optou-se por distinguir no detalhamento dos componentes as informações abstratas (através dos tipos), informações concretas ou instanciadas e informações para a execução. Esta distinção permite que o mesmo meta-modelo seja utilizado em diferentes fases da evolução de um processo. Por exemplo, na fase de modelagem abstrata é possível definir um processo sem associá-lo a elementos da organização (recursos e agentes), somente associando tipos, que posteriormente serão instanciados. Enquanto que para a fase de execução, o meta-modelo fornece informações específicas, como o estado de execução e registro de eventos, permitindo também associação com elementos adicionais que influenciam na execução (políticas habilitadas).

Nesta seção será apresentada uma descrição informal da notação da linguagem e exemplos de seu uso considerando a parte visual da mesma. Já formulários usados para detalhar os componentes não visuais da linguagem são apresentados somente no capítulo 6.

4.1.1 Atividades

A figura 4.2 mostra a representação adotada para atividades de processos de software distinguindo as atividades decompostas, normais e automáticas. Atividades

decompostas serão definidas através de um modelo de processo, enquanto as atividades do tipo automática e normal são chamadas de atividades-folha.

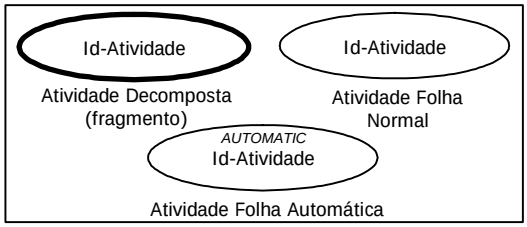


Figura 4.2: Notação para Atividades na APSEE-PML.

É importante observar que uma atividade decomposta, apesar de possuir alguns atributos próprios, sempre estará associada a um modelo de processo de software, e isto determina a estrutura hierárquica do processo. Essa decomposição é útil como mecanismo de abstração de processos. Um exemplo que ilustra a estrutura de processo com atividades decompostas é apresentado na figura 4.3. Neste exemplo, o processo é definido em três níveis e possui atividades decompostas (A1, e A1.1) e atividades-folha (A2, A1.2, A1.3, A1.1.1 e A1.1.2), sendo que, por simplificação, não estão definidos os fluxos de controle e de dados entre as atividades.

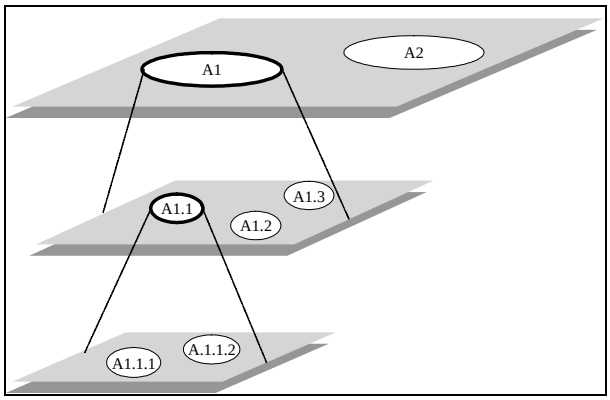


Figura 4.3: Exemplo de decomposição de atividades de um processo.

A definição de uma atividade-folha é feita através de formulários que auxiliam o preenchimento das associações necessárias descritas no meta-modelo de processos de software mostrado na seção 3.7. Essas associações definem, por exemplo, recursos requeridos, agentes envolvidos, pré e pós condições, ferramentas que executam as atividades automáticas, dentre outros detalhes. No caso da definição de recursos e agentes, opcionalmente pode ser fornecido um auxílio através do uso de políticas de instanciação habilitadas para a atividade.

4.1.2 Conexões Simples: Seqüência e *Feedback*

A figura 4.4 mostra a representação adotada para conexões simples entre atividades. A conexão de seqüência tem início e fim em atividades diferentes (que podem ser de qualquer tipo) e o nodo que a representa é um retângulo contendo o tipo de dependência escolhido (“*end-start*”, “*start-start*”, “*end-end*” ou “*?*”, quando ainda não foi decidido o tipo de dependência). Uma restrição importante é que a definição de uma conexão de

seqüência não pode inserir ciclos no processo, ou seja, não pode haver um fluxo de controle no sentido contrário ao sentido da conexão de seqüência entre as duas atividades envolvidas.

A conexão de *feedback*, por sua vez, é representada por uma seta pontilhada com destino à atividade que deve ser repetida após o término ou falha da atividade origem da seta. Uma condição lógica é associada a essa conexão. Essa condição pode verificar a situação de um artefato, métricas, atrasos ou ainda consultar o gerente responsável para autorizar o *feedback*. O projetista do processo deve levar em consideração que essa conexão será avaliada automaticamente e reativará a atividade destino assim como suas sucessoras, e esse fragmento de processo será repetido até que a condição seja falsa. Outra característica importante é que pode ser definida uma conexão de *feedback* com origem e destino na mesma atividade, e isso pode ser usado para que a execução da atividade seja repetida até que uma determinada condição seja satisfeita. Sempre que uma atividade é repetida, é criada uma nova versão da sua descrição e as versões anteriores são mantidas.

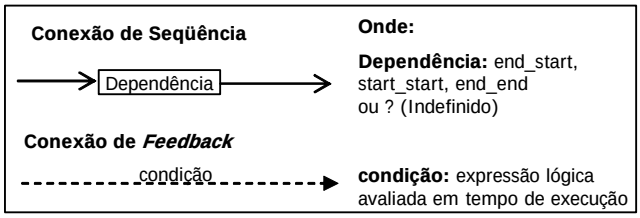


Figura 4.4: Notação para Conexões Simples na APSEE-PML.

Um exemplo de modelo de processo contendo três atividades normais e duas conexões simples é apresentado na figura 4.5. Neste exemplo, a *atividade1* não possui dependências, enquanto que a *atividade2* somente pode ser iniciada após o término da *atividade1*, e a *atividade3* pode ser iniciada após o início de *atividade1* (conexão *start-start*). Nesse caso, devido às dependências existentes, *atividade1* e *atividade3* poderão, eventualmente, ser executadas em paralelo.

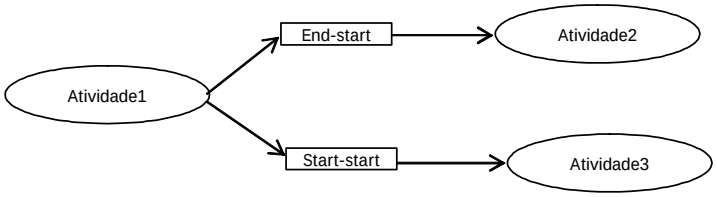


Figura 4.5: Exemplo de processo na APSEE-PML com conexões simples de sequência.

Na figura 4.6 é ilustrada a tentativa de inclusão de uma conexão entre a *atividade3* e a *atividade1*. Como a *atividade3* já depende do início da *atividade1*, não é possível acrescentar uma conexão que definiria que a *atividade1* depende do término da *atividade3* para ser iniciada. Também não é possível acrescentar uma conexão *start-start* no mesmo sentido; porém, uma conexão *end-end* de *atividade3* para *atividade1* seria permitida pois não insere conflito no modelo de processo. Portanto, regras são necessárias para garantir a consistência dos modelos de processos, as quais são descritas formalmente no capítulo 5 a seguir.

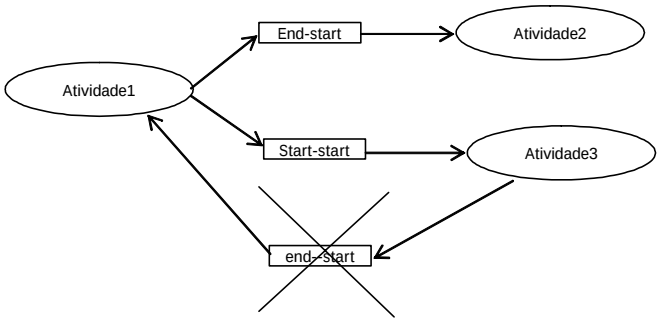


Figura 4.6: Exemplo de processo com ciclo de dependências. Restrição da linguagem impede criação da terceira conexão simples.

Um exemplo de uso da conexão de *feedback* é mostrado na figura 4.7. A conexão de *feedback* tem origem na *atividade2* e destino na *atividade1*. Foi possível definir essa conexão porque existe um fluxo de controle entre a *atividade1* e *atividade2*, ou seja, a *atividade2* depende direta ou indiretamente da *atividade1*. Nesse exemplo, após o término ou falha da *atividade2*, se a condição especificada na conexão for verdadeira, a execução retorna para a *atividade1*, que é repetida, assim como sua sucessora, a *atividade2*.

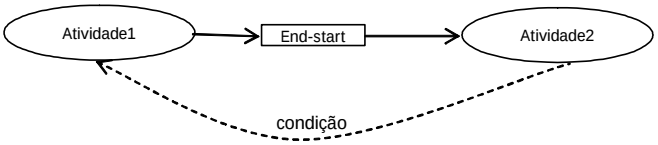


Figura 4.7: Exemplo de processo com conexão de *feedback*.

A condição para que a conexão de *feedback* seja executada é um termo de uma linguagem restrita para condições do APSEE. Exemplos de termos de condição válidos são[10]:

- **"True"**: após o término da *atividade2* a *atividade1* sempre será repetida. Porém é possível alterar a condição de retorno durante a execução;
- **"False"**: o *feedback* não será ativado;
- **"ask_manager()"**: é solicitada decisão do usuário gerente antes de ativar o *feedback*;
- **"a.get_output_artifacts.any.getMetric("defects_percentage", >, 20)"**: Esse exemplo de condição é útil para quando se deseja repetir um fragmento de processo por causa da falta de qualidade dos artefatos gerados. Nesta sentença, a atividade origem do *feedback* é avaliada (representada pelo símbolo "a" no início da expressão), todos os seus artefatos de saída são obtidos (*get_output_artifacts*) e para cada um deles é verificada a métrica "*defects_percentage*" (porcentagem de defeitos). Se a

[10] Uma linguagem específica para definição de condições para o modelo APSEE foi proposta em [REI02f].

métrica armazenada tiver valor maior que 20 para qualquer artefato de saída (*any*), então o *feedback* é ativado.

Outras condições podem ser definidas para avaliar o *feedback*. Primitivas para criação de condições lógicas são mostradas no Apêndice E.

4.1.3 Conexões Múltiplas: *Join* e *Branch*

A representação de conexões múltiplas na APSEE-PML é feita através de um retângulo dividido onde a parte superior indica o tipo de conexão (*Branch* ou *Join*) e o operador lógico associado (*AND, OR, XOR*), enquanto que a parte inferior indica o tipo de dependência (*end-start*, *start-start*, *end-end*). A figura 4.8 mostra a representação gráfica para conexões múltiplas. Por definição, uma conexão *Join* (J) pode ter vários antecessores e somente um sucessor, enquanto que a conexão *Branch* (B) pode ter apenas um antecessor e vários sucessores. Na APSEE-PML, os antecessores e sucessores de conexões múltiplas podem ser atividades ou conexões múltiplas (conexões simples não podem participar de conexões múltiplas[11]). Portanto, o fluxo de controle de um processo pode ser definido mais detalhadamente através da combinação de várias conexões múltiplas encadeadas.

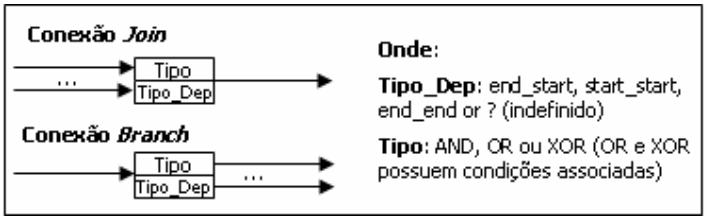


Figura 4.8: Notação para Conexões Múltiplas na APSEE-PML.

O significado das conexões múltiplas já foi apresentado informalmente na seção 3.7.4 deste capítulo. Entretanto, cabe lembrar que conexões *Branch OR* e *XOR* refletem uma escolha entre possíveis caminhos de execução (*XOR* – somente um caminho, e *OR* – vários caminhos). Cada possibilidade de caminho deve estar associada a uma condição, sendo que essas condições são definidas através da mesma linguagem de condições usada para conexões de *feedback*, mencionada anteriormente.

Exemplos de usos de conexões múltiplas na linguagem são mostrados nas figuras a seguir.

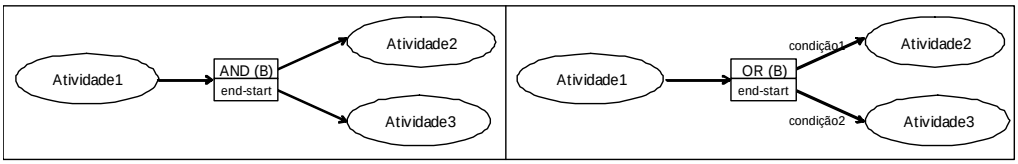


Figura 4.9: Exemplos de uso de conexões *Branch*.

[11] O motivo para que conexões simples não possam ser ligadas diretamente a conexões múltiplas é que uma conexão múltipla já expressa o tipo de dependência requerido das atividades antecesssoras (*end-start*, *start-start*, *end-end*) e uma combinação com conexão simples torna ambíguo qual o tipo de dependência desejado.

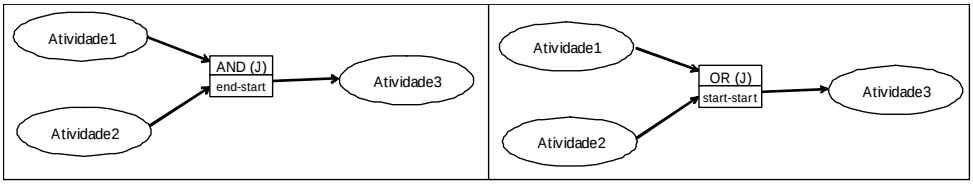


Figura 4.10: Exemplos de uso de conexões *Join*.

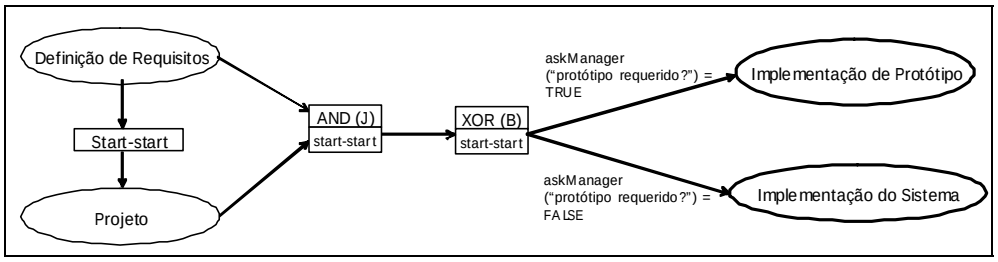


Figura 4.11: Exemplo de uso de conexões múltiplas encadeadas.

Existem restrições de consistência que devem ser observadas na definição de conexões múltiplas. Se a combinação de conexões múltiplas fosse livre, alguns problemas poderiam surgir devido a conflitos estruturais no processo, como por exemplo, ciclos entre os fluxos de controle das atividades. Um exemplo de conflito por ciclo é ilustrado na figura 4.12. Se não detectados corretamente, estes conflitos impedirão o andamento do processo. A detecção pode ser feita durante a edição de processos ou pode ser necessário o uso de métodos de verificação de processos que analisam o modelo de processo e apontam onde estão os conflitos.

Um método para detecção de conflito de *deadlock* e de falta de sincronização foi proposto por Sadiq e Orlowska (2000b). No caso, conflito de *deadlock* ocorre quando há uma tentativa de sincronizar com *Join*-AND vários caminhos alternativos gerados por um *Branch* XOR (ver figura 4.13), enquanto que falta de sincronização ocorre quando após um *Branch*-AND, os caminhos paralelos são sincronizados com um *Join*-OR ou XOR, ao invés de um *Join*-AND. Isso faz com que o primeiro caminho executado habilite o *Join*, sem esperar que os outros caminhos terminem (ver figura 4.14).

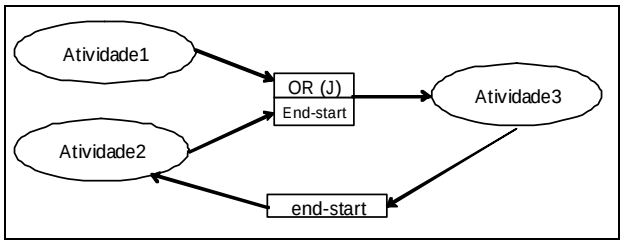


Figura 4.12: Exemplo de dependência cíclica (conflito).

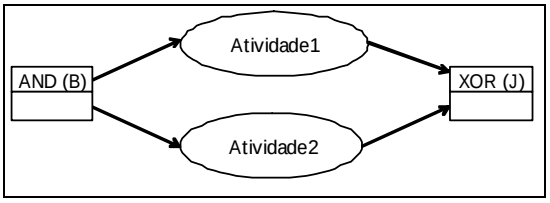


Figura 4.13: Conflito de *Deadlock*

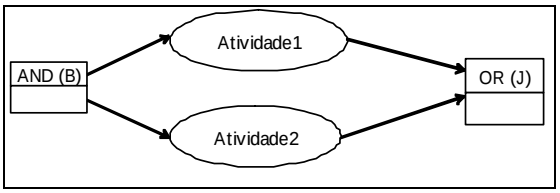


Figura 4.14: Conflito de falta de sincronização.

Essas restrições de consistência são fornecidas pela APSEE-PML da seguinte forma[12]:

- **Dependência cíclica**: A inclusão de ciclos no processo é verificada no momento da definição dos antecessores e sucessores das conexões múltiplas, ou seja, durante a edição do processo. O mesmo procedimento é adotado para conexões simples;
- **Conflito de *Deadlock***: Quando o conflito sendo inserido está na forma como mostrado na Figura 4.13 (sendo que *atividade1* e *atividade2* podem ser substituídas por conexões múltiplas), então é detectado em tempo de edição. Porém, quando os componentes de processo existentes entre o *Branch* inicial e o *Join* final são compostos de várias atividades e conexões, então é necessário utilizar um método como o proposto em (SADIQ; ORLOWSKA, 2000b] para detectar esse conflito.

  A dificuldade de tratar esse conflito em tempo de edição é que essa decisão depende dos outros ramos do *Branch*, que podem conter vários componentes. Há uma grande possibilidade de combinações sem conflito mesmo utilizando *Branch*-AND e *Join*-XOR, como ilustrado no exemplo da Figura 4.15, onde três caminhos paralelos são executados, sendo que os dois primeiros caminhos são sincronizados em um *Join*-And, porém o *Branch*-XOR do terceiro caminho faz com que apenas um dos antecessores do *Join*-XOR final seja habilitado, o que caracteriza um processo sem conflito de *deadlock*:

[12] Uma especificação formal dessas restrições é fornecida através da gramática da linguagem no capítulo 5.

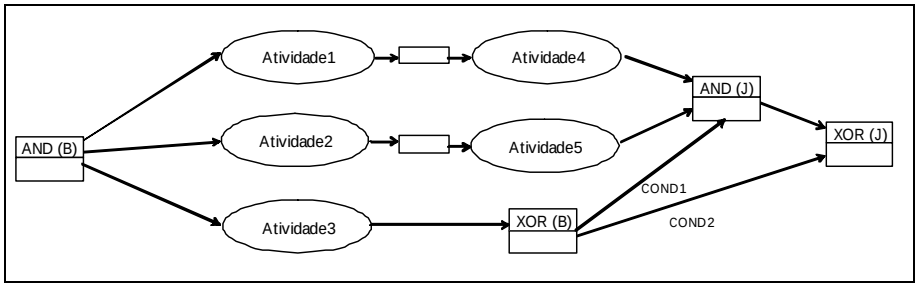


Figura 4.15: Exemplo de processo sem conflito de *deadlock*.

- **Falta de Sincronização**: Conflitos de falta de sincronização (mesmo simples como o da Figura 4.14) não são tratados durante a edição, sendo proposto o uso de um método como o de Sadiq e Orlowska (2000b) para detectá-los.

4.1.4 Conexões de Artefato

Artefatos são quaisquer documentos utilizados ou produzidos por atividades do processo, portanto podem representar diagramas, código fonte, manuais e objetos do próprio APSEE, como por exemplo, um modelo de processo. Uma conexão de artefato define o fluxo de dados do processo através da passagem de artefatos entre atividades. A representação de uma conexão de artefato é uma caixa tridimensional (sombreada) contendo o identificador do artefato (a conexão refere-se aos artefatos existentes no pacote *Artifacts*) e o tipo do artefato. Tanto atividades quanto conexões múltiplas *Branch* e *Join* podem estar ligadas a uma conexão de artefato e não há limite para a quantidade de ligações. A figura 4.16 mostra a notação para as conexões de artefato na APSEE-PML.

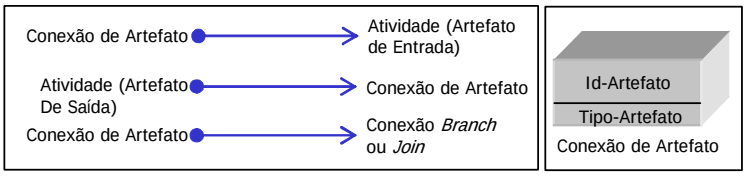


Figura 4.16: Notação para Conexões de Artefato na APSEE-PML.

Apesar de atividades normais poderem declararar quais são seus artefatos de entrada e saída através de formulários textuais, as conexões de artefato são úteis porque tornam visível a passagem de dados entre atividades de um processo. Assim, o projetista pode escolher quais artefatos deseja visualizar como conexão no diagrama que representa o processo. A linguagem adota algumas regras de consistência para lidar com conexões de artefato, com o objetivo de facilitar a modelagem de processos:

- Quando uma conexão de artefato tem como destino uma atividade normal, o artefato da conexão torna-se um artefato de entrada desta atividade. Porém, o fato de existir um artefato de entrada em uma atividade não implica que exista uma conexão de artefato ligada à mesma. Raciocínio análogo se aplica a artefatos de saída. Por exemplo, a Figura 4.17 apresenta uma conexão de artefato ligada a duas atividades. A partir dessa ligação, através das regras de consistência implementadas no editor da linguagem,

as informações da *atividade1* são atualizadas automaticamente para acrescentar o *artefato1* como artefato de saída, enquanto o mesmo artefato é inserido como artefato de entrada na *atividade2*;

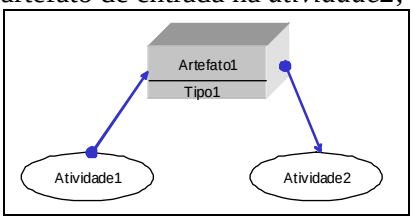


Figura 4.17: Exemplo de uso de conexão de artefato.

- Quando uma conexão de artefato é ligada a uma atividade decomposta, é criada uma conexão de artefato idêntica (referenciando o mesmo artefato) no sub-processo da atividade decomposta. Porém uma conexão de artefato existente em um processo não necessariamente deve existir no nível superior. Essa consistência é ilustrada pelo processo do exemplo da Figura 4.18, onde o *Artefato 1* conectado à atividade *AtividadeDec* no plano superior foi automaticamente inserido na descrição do plano inferior;

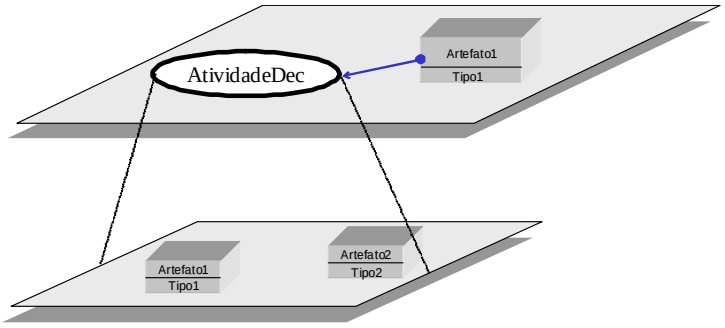


Figura 4.18: Exemplo de consistência de conexões de artefato entre níveis de processo.

- Uma conexão de artefato que tem como destino uma conexão múltipla é equivalente à ligação da conexão de artefato a cada sucessor direto dessa conexão múltipla. Portanto, as consistências citadas acima podem ser aplicadas mesmo que a conexão de artefato não esteja diretamente ligada à atividade. Esta característica serve para simplificar o modelo de processo. Entretanto, uma conexão de artefato não pode ter como origem uma conexão múltipla. Apesar de possível, essa simplificação foi evitada na linguagem para não causar a impressão de que conexões múltiplas produzem artefatos. A Figura 4.19 mostra a equivalência entre dois processos usando conexões de artefato;

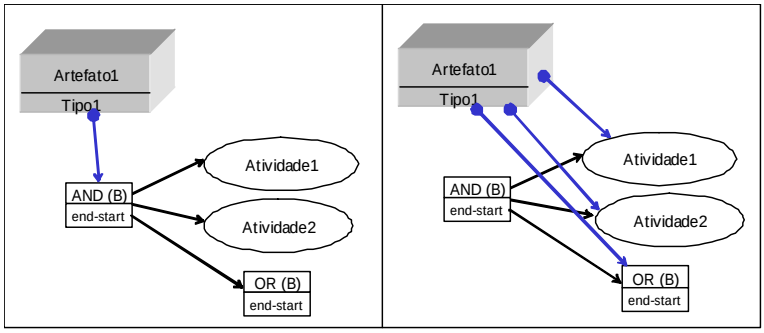


Figura 4.19: Conexão de artefato ligada à conexão múltipla é equivalente à ligação com sucessores da conexão múltipla.

- Considerando que artefatos possuem relacionamento de composição, conforme explicado na seção 3.8, a ocorrência de conexões que referenciam artefatos compostos em um processo pode ser representada através de um símbolo especial de composição conforme mostrado na Figura 4.20 (O *artefato1* é composto por *artefato1.1* e *artefato1.2*). Entretanto, deve-se notar que não existe composição entre conexões de artefato e sim entre os artefatos referenciados pelas conexões. O símbolo de composição entre conexões é representado automaticamente quando artefatos compostos são referenciados em conexões do mesmo processo.

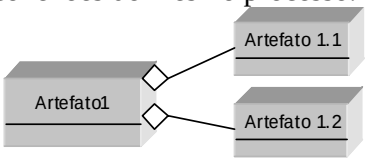


Figura 4.20: Exemplo de representação da composição de conexões de artefatos.

4.1.5 Exemplo de Modelo de Processo na APSEE-PML

Um exemplo de modelo de processo construído através da linguagem APSEE-PML é apresentado na figura 4.21. Trata-se de um processo para desenvolvimento de sistemas móveis apresentado em (REIs, 2002a). Neste exemplo pode-se observar o fluxo de controle e de dados do processo entre atividades decompostas. Além disso, o exemplo traz o encadeamento entre conexões múltiplas, conexões de *feedback* e várias conexões de artefato. Estudos de caso envolvendo modelagem de processos nesse formalismo serão apresentados no capítulo 7.

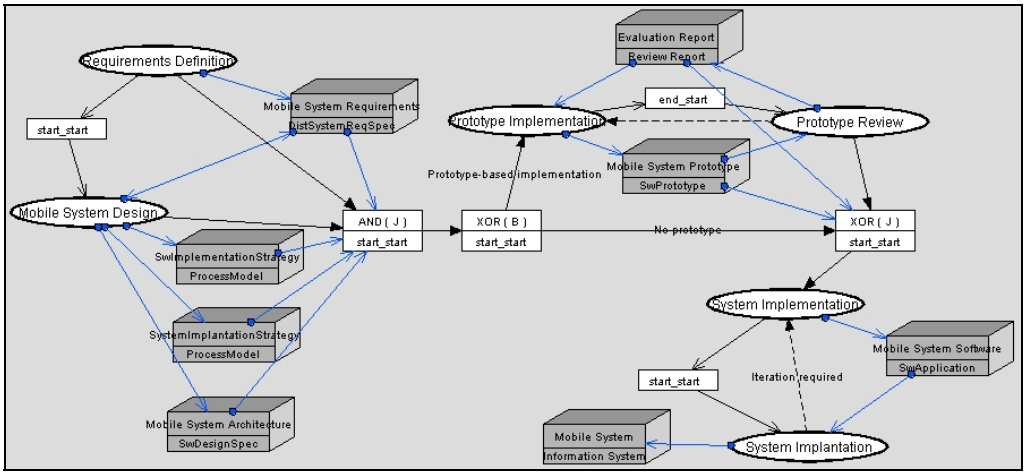


Figura 4.21: Processo para desenvolvimento de aplicações móveis (REIS et al., 2002a).

4.2 Instanciação de Processos de Software no APSEE

A instanciação de processos de software é a transformação de um modelo de processo em processo executável (DERNIAME et al., 1999) através do planejamento do cronograma e alocação de desenvolvedores e recursos para as tarefas, dentre outros ajustes. Nessa fase do meta-processo de software é necessário o conhecimento acerca do estado da organização, o contexto da execução do processo e informações de projetos anteriores. Considerando que essas informações podem ser complexas e são inter-relacionadas, a falta de atenção e suporte a essa fase pode ter como conseqüência uma alocação inadequada de recursos. Apesar de a instanciação ser tratada como uma fase do ciclo de vida de processos de software, na abordagem proposta, a instanciação pode ocorrer juntamente com a execução e a modelagem.

A instanciação possui uma natureza desafiadora quando se considera que nas organizações de desenvolvimento de software o conjunto de recursos é limitado, as pessoas possuem especialidades diversas, e que estes aspectos devem ser adaptados a diferentes atividades de forma adequada (DERNIAME et al., 1999). Vários recursos são utilizados, consumidos e/ou alocados durante o processo de desenvolvimento de software. Os recursos que apóiam a execução das atividades devem ser tratados de forma distinta das pessoas que as executam. Dada a importância de ambos, essa distinção objetiva tratar mais adequadamente seus aspectos específicos.

As organizações freqüentemente adotam estratégias específicas para instanciar processos, através de modelos de custos e ferramentas de gerência de projetos para obter melhores resultados econômicos. Entretanto, as ferramentas de gerência de processos diferem dos mecanismos convencionais de escalonamento de projetos com respeito a questões específicas que reforçam a necessidade de suporte adequado à fase de instanciação, tais como:

- Os processos de software não são completamente definidos antes da execução, já que execução e modelagem são tarefas que podem ocorrer simultaneamente. Portanto, a instanciação pode ocorrer durante a execução

e depende de decisões rápidas para não prejudicar o andamento do processo;

- As características das pessoas, incluindo habilidades e experiências passadas, assim como informações relativas aos recursos, como métricas e disponibilidade, influenciam de forma decisiva na realização do processo. Portanto, o conhecimento sobre essas informações deve estar disponível para que a instanciação seja realizada adequadamente;
- A natureza dinâmica dos processos pode influenciar a alocação de recursos e pessoas. Os gerentes freqüentemente devem re-alocar processos em decorrência de modificações dinâmicas. Por isso, a escolha de estratégias de alocação deve levar em consideração o estado corrente da execução do processo assim como a disponibilidade atual e futura (esperada) dos recursos e agentes.

Além disso, é importante ressaltar que apesar das técnicas de estimativas de custos serem usadas como estratégia para definir a instanciação em alguns casos, elas cobrem apenas um aspecto da instanciação. A natureza da execução de processos sugere que múltiplos critérios são necessários. Esses critérios podem ser definidos através de políticas.

Um mecanismo de instanciação automática de processos baseado em estratégias definidas pelo usuário é proposto neste trabalho. O mecanismo utiliza o conceito de Políticas de Instanciação, cujo modelo foi apresentado na seção 3.11. Essa proposta considera que uma solução para o problema da flexibilidade em PSEEs inclui uma solução para o problema da instanciação nesses ambientes. De fato, se o objetivo principal é a execução flexível de processos de software e a instanciação pode ocorrer durante a execução, um mecanismo de instanciação automatizado deve contribuir para o aumento da flexibilidade. Para isso, um formalismo adequado para representar as políticas do usuário deve complementar a APSEE-PML, assim como o funcionamento do mecanismo deve prever a flexibilidade necessária. Assim, propõe-se a capacidade de instanciar atividades automaticamente durante a execução quando essa necessidade é detectada, ou seja, quando a atividade estiver prestes a iniciar e não estiver instanciada. A vantagem desta abordagem está na adoção de políticas definidas pelo usuário que podem ser habilitadas em pontos específicos do processo e que, por estar integrada ao modelo APSEE, pode utilizar tanto características dos recursos e agentes quanto informações históricas armazenadas como métricas.

As seções seguintes detalham a proposta do mecanismo de instanciação definindo seu funcionamento, a linguagem para definição de políticas de instanciação e explicações sobre sua semântica, além de exemplos de sua utilização.

4.2.1 Definição de Políticas de Instanciação

Considerando que a política representa um aspecto adicional ligado a componentes de processo, é suficiente uma representação textual para descrevê-la, porém é possível representar a sua ligação com um componente de processo graficamente. Assim, para definir uma política de instanciação o usuário utiliza uma linguagem textual para a qual foi proposta a gramática apresentada na figura 4.22.

Os componentes essenciais de uma política já foram apresentados na seção 3.11.1 do capítulo anterior. São eles: identificador da política, interface, tipo ao qual a política se aplica, condições para que a política seja adotada, critérios de restrição e critério de ordenação.

<ResInstant_Policy> → **Id** <string> **(**(**Name** <string>) (**; Description** <string>) **;**
PolicyTypeID <type_id> **;**
Interface <pol_interface> **;**
ApplyToType <type_id>;
Conditions <condition> **;**
RestrictBy <RestrictionCriteria> **;**
OrderBy <OrderCriteria>)
<pol_interface> → <label> **:** <type>
<condition> → <policy_operand> (<opt_relation>) (<opt_connection>)
<opt_relation> → <comparison> <policy_operand>
<comparison> → **> | < | >= | <= | = | <> | contains | not_contains** | **sub_type_of**
<opt_connection> → <conn_type> <condition>
<conn_type> → **and** | **or**
<policy_operand> → <policy_object> **.** <operators>
<operators> → <policy_operator>*
<policy_operator> → <method_id> <parameters> | <reserved_word> | ∪ | ∩
<parameters> → <policy_operand>*
<reserved_word> → **any** | **all** | **no**
<RestrictionCriteria> → <inst_method>* /* vários critérios de restrição podem ser definidos */
<OrderCriteria> → <inst_method>/* apenas um critério de ordenação pode ser definido */
<label> → <string>
<type> → **Activity | Resource**
<policy_object> → <string>
<inst_method> → <inst_method_id> <inst_method_parameter>*
<inst_method_id> → <string>
<inst_method_parameter> → <integer> | <real> | <string> | <boolean> | <relational_op>
<relational_op> → **equal | different | greaterThan | lessThan | ...**

Figura 4.22: Gramática da linguagem *ResourceInstantiationPolicies*

O trecho que define a condição da política utiliza a mesma sintaxe e semântica das condições usadas no trabalho de Reis (2002f), propostas especificamente para o APSEE. Essa linguagem define um conjunto de funções aplicáveis aos tipos *Activity*, *Artifact*, *Process*, *Resources*, *Agent*, *Groups*, *Roles* e *Connections*, consultando informações sintáticas acerca do modelo de processo em questão. Estas funções são usadas na definição das condições da política de instanciação de recursos (no item *method_id*). O conjunto de funções disponíveis para o programador é fixo, determinado por primitivas disponíveis no mecanismo APSEE. Algumas funções disponíveis são mostradas na figura 4.23 (uma lista mais completa de funções está disponível no apêndice E).

Get_roles → set of roles	Is_active → bool
Is_performed_by_agents → bool	Get_sucessors → set of activities
Get_duration → real	Get_sucessors_oftype(type) → set of activities
Get_agents →set of agents	Get_artifact_connections_to(activity) → set of activities
Is_enacting →bool	Get_types_reqResources → set of string
Is_late_to_begin → bool	Get_input_artifacts → set of artifacts
Is_waiting → bool	Get_output_artifacts → set of artifacts

Figura 4.23: Operações sobre atividades para políticas de instanciação.

Por sua vez, os critérios das políticas são definidos no presente trabalho, assim como a semântica da interpretação das mesmas. Como apresentado anteriormente, o objetivo

dos critérios de restrição é estabelecer um conjunto inicial de recursos/agentes adequados a uma atividade, enquanto que o objetivo do critério de ordenação é posicionar os elementos do conjunto resultante de recursos/agentes de forma a facilitar a escolha do mais adequado.

Alguns dos critérios de restrição possíveis são mostrados na Tabela 4.1, enquanto que critérios de ordenação são mostrados na Tabela 4.2.

Tabela 4.1: Critérios de Restrição na instanciação de recursos.

Critério (parâmetros)	Significado
Belongs_to (tipo_de_recurso)	Restringir recursos a sugerir para aqueles que pertencem a algum recurso do tipo dado como parâmetro. Por exemplo, solicitar que sejam sugeridos apenas os recursos que pertençam à salas.
NotBelongs_to (tipo_de_recurso)	Restringir recursos a sugerir para aqueles que não pertencem a recurso do tipo dado como parâmetro. Por exemplo, solicitar que sejam sugeridos apenas os recursos que não pertençam a salas.
No_Requirements	Restringir os recursos a serem sugeridos para aqueles que não requerem nenhum outro recurso para serem utilizados.
Requires (tipo_de_recurso)	Restringir recursos a sugerir para aqueles que requerem um recurso do tipo dado como parâmentro.
NotRequires (tipo_de_recurso)	Restringir recursos a sugerir para aqueles que não requerem o recurso dado como parâmetro.
GetMetric(id_métrica, < | > | = | <= | >= | <>, valor)	Restringir os recursos a sugerir para aqueles que atendem à comparação com o valor de uma métrica existente. Por exemplo: getMetric("taxa_utilização_ano", < , 30), ou seja, os recursos que têm menos de 30% de taxa de utilização ao ano.
Cost(< | > | = | <= | >= | <>, valor)	Restringir os recursos a sugerir para aqueles que atendem à comparação com o custo. Por exemplo: Cost(< , 30), ou seja, os recursos cujo custo de utilização seja menor que 30.
MTBF(< | > | = | <= | >= | <>, valor, unidade de tempo)	Restringir os recursos a sugerir para aqueles que atendem à comparação com o atributo MTBF. Por exemplo: MTBF(>= , 60, "dias"), ou seja, os recursos cujo tempo médio entre falhas seja maior ou igual a 60 dias.
Restrições específicas para recursos do tipo Consumables	
Consumable_New	Restringir recursos consumíveis a sugerir para aqueles que nunca foram usados
Max_PercentUsed(valor)	Restringir os recursos consumíveis a serem sugeridos para aqueles com pertentual de uso menor que o valor dado como parâmetro.
AmountAvailable(< | > | = | <= | >= | <>, valor)	Restringir os recursos consumíveis para aqueles que satisfazem a condição de disponibilidade expressa nos parâmetros dados. Por exemplo, AmountAvailable(>,10) significa que se deve restringir os recursos para aqueles em que restam mais de 10 unidades não consumidas.

Tabela 4.2: Critérios de Ordenação na instanciação de recursos.

Critério	Significado
Low_Cost	Ordenar recursos pelo menor custo
High_Cost	Ordenar recursos pelo maior custo.
High_Mtbf	Ordenar pelos recursos que falham menos.
Low_Mtbf	Ordenar pelos recursos que falham mais.
Metric_higher(id_métrica)	Sugerir recursos em ordem decrescente do valor da métrica cujo identificador é passado como parâmetro.
Metric_lower(id_métrica)	Sugerir recursos em ordem crescente do valor da métrica cujo identificador é passado como parâmetro.

4.2.2 Exemplos de políticas de instanciação

Exemplos de políticas de instanciação de recursos de acordo com a sintaxe fornecida são mostrados nas figuras a seguir (figura 4.24, figura 4.25, figura 4.26 e figura 4.27). Uma atividade pode ter várias políticas aplicadas ao mesmo tempo desde que não tratem de recursos do mesmo tipo (isto é verificado através do atributo *ApplyToType*). Os exemplos demonstram que uma política pode verificar o estado do processo a fim de sugerir melhores recursos para uma atividade, como por exemplo, se a atividade estiver atrasada os recursos a serem sugeridos são diferentes dos recursos sugeridos normalmente. As quatro políticas apresentadas poderiam estar habilitadas em uma mesma atividade decomposta.

Os critérios utilizados nos campos *restrictBy* e *OrderBy* foram apresentados na Tabela 4.1 e Tabela 4.2 anteriormente. Os métodos chamados no campo *Conditions* fazem parte das funções embutidas na linguagem (apresentadas na seção anterior).

ID:	**“**Pol_Inst1**”**
Name:	“Instanciação para salas”
Description:	“Se tamanho do software a ser produzido pela atividade for maior que 100 pontos por função e o número de agentes da atividade for maior que 5, então restringir para salas que tenham tido taxa de utilização no ano inferior a 30% e custem menos que 10 (unidades de custo) e ordenar pelas salas de maior tamanho”
Interface:	a: Activity;
ApplyToType:	“Room”
Conditions:	a.get_output_artifacts.any.getEstimation(“num_pontos_por_função”) **>** 100 **and** a.numberofagents() **>** 5
RestrictBy:	getMetric(“taxa_utilização_ano”, <, 30) Cost(<, 10)
OrderBy:	Metric_higher(“tamanho”)

Figura 4.24: Política “Instanciação para salas”.

ID:	**“**Pol_Inst2**”**
Name:	“Atividade atrasada envolvendo cliente e uso de computadores handheld”
Description:	“Se atividade está atrasada e é do tipo encontro com cliente, então obter computadores handheld que tenham taxa de utilização anual menor que 10% e tempo médio entre falhas maior que 30 dias e ordená-los pelo maior MTBF”
Interface:	a: Activity;
ApplyToType:	“handheld”
Conditions:	a.is_late_to_begin() **and** a.get_type() **sub_type_of** “Reunião_Cliente”
RestrictBy:	getMetric(“taxa_utilização_ano”, <, 10) MTBF(>, 30, “dias”)
OrderBy:	High_MTBF

Figura 4.25: Política “Instanciação para handhelds”.

ID:	**“**Pol_Inst3**”**
Name:	“Atividade de codificação atrasada com recursos consumíveis”
Description:	“Se atividade do tipo codificação está atrasada e necessita de recursos consumíveis, então obter consumíveis que tenham pelo menos 70% de disponibilidade e ordenar por menor custo”
Interface:	a: Activity;
ApplyToType:	“Consumable”
Conditions:	a.is_late_to_begin() **and** a.get_type() **sub_type_of** “Codificação”
RestrictBy:	MaxPercentUsed(30)
OrderBy:	Low_Cost

Figura 4.26: Política “Instanciação para quaisquer consumíveis”.

ID:	**“**Pol_Inst4**”**
Name:	“Atividade de codificação requerendo impressora”
Description:	“Se atividade do tipo codificação produz código fonte então obter impressoras que não requeiram papel colorido e que requeiram papel A4 e ordenar pelas de maior velocidade”
Interface:	a: Activity;
ApplyToType:	“Printer”
Conditions:	a.get_output_artifacts.any.get_type() **sub_type_of** “CódigoFonte” **and** a.get_type() **sub_type_of** “Codificação”
RestrictBy:	NotRequires(“PapelColorido”) Requires(”PapelA4”)
OrderBy:	Metric_Higher(“Velocidade”)

Figura 4.27: Política “Instanciação para impressora”.

Exemplos de políticas de instanciação de agentes são mostrados nas figuras a seguir (figura 4.28 e figura 4.29). Deve-se observar que, conforme já explicado no capítulo 3, as políticas para instanciação de agentes diferem no campo ***ApplyTo***, onde o usuário deve distinguir entre ***Agent*** ou ***Group*** e definir o tipo de agente ou grupo ao qual a política se aplica.

ID:	**“**People_Inst1**”**
Name:	“Programador para atividade atrasada”
Description:	“Se atividade está atrasada para começar e envolve programação, selecionar programadores que possuam experiência em programação e que tenham produtividade maior que 2 pontos por função por dia. Ordenar por agentes que atrasam menos”
Interface:	a: Activity;
ApplyTo:	**Agent** “Programmer”
Conditions:	a.is_late_to_begin() **and** a.get_type() **sub_type_of** “coding”
RestrictBy:	experience_in(“coding”) getMetric(“productivity_FP_day”, >, 2)
OrderBy:	metric_lower(“late”)

Figura 4.28: Programador para atividade atrasada.

ID:	**“**People_Inst2**”**
Name:	“Treinamento de analistas OO”
Description:	“Se a atividade é do tipo treinamento de analistas OO, selecionar os agentes sem experiência neste tópico e ordenar de acordo com a mais alta afinidade do grupo.”
Interface:	a: Activity;
ApplyTo:	**Agent** “analyst”
Conditions:	a.get_type() **sub_type_of** “training_OO_analysts”
RestrictBy:	noExperience_in(“OOanalysis”)
OrderBy:	higher_Affinity

Figura 4.29: Treinamento de analistas de orientação a objetos.

Através dos exemplos pode-se observar que é possível reutilizar políticas em diferentes processos, domínios de aplicação e até diferentes organizações, desde que as hierarquias de tipos sejam compatíveis. As métricas armazenadas sobre os componentes do processo podem ser consultadas nas políticas, assim como podem servir de critério de ordenação para recursos. Além disso, a linguagem focaliza a consulta a tipos e não a instâncias de elementos de processo, o que aumenta a capacidade de reutilização da política em outros cenários.

4.2.3 Funcionamento do APSEE-Planner

As políticas de instanciação podem estar habilitadas em atividades específicas, modelos de processos e na organização como um todo. As atividades herdam as políticas estabelecidas para o processo em que estão contidas e os processos, por sua vez, herdam as políticas que estão habilitadas na organização em geral. O nível de habilitação das políticas influencia na prioridade da sua aplicação, ou seja, políticas habilitadas em uma atividade, por serem mais específicas, possuem maior prioridade no momento da instanciação. Se as políticas da atividade não puderem ser aplicadas, são consideradas em seguida as políticas do processo e, por último, as habilitadas na organização.

A partir de um processo de software (ou fragmento) modelado com informações abstratas (com a PML adotada pelo ambiente), a fase de instanciação passa por duas etapas. A figura 4.30 mostra um diagrama usando a própria APSEE-PML (apresentada na seção 4.1) para descrever essas etapas. Na primeira etapa (“Geração de Sugestões” na figura citada), são geradas as sugestões de instanciação, que levam em consideração o processo (ainda total ou parcialmente abstrato), as políticas definidas e o estado dos recursos e agentes. Essas sugestões são armazenadas como sugestões de instanciação que estão representadas através das classes do pacote *Organization.Planner_Info*, apresentado na seção 3.12. A segunda etapa (atividade rotulada como “Instanciação do Processo” na figura) gera como resultado um modelo de processo de software instanciado[13] através do uso das sugestões obtidas da etapa anterior.

[13] Trata-se de uma modificação no modelo origem, e não a criação de um novo modelo de processo.

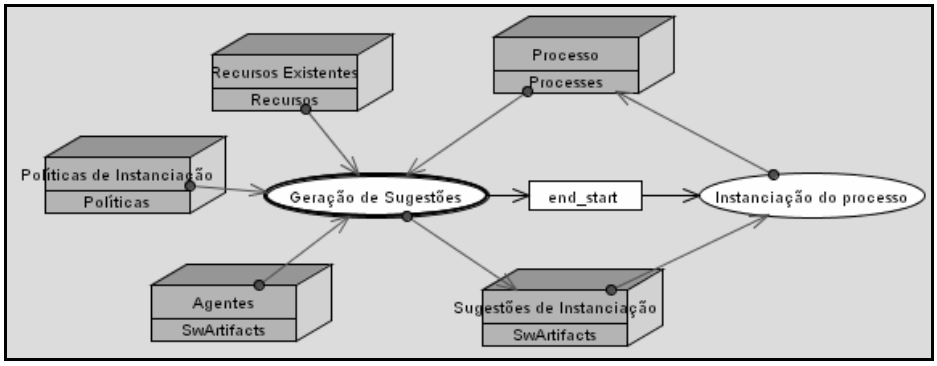


Figura 4.30: Fases da Instanciação de um processo através de políticas.

A figura 4.31 ilustra o relacionamento entre os níveis abstrato e instanciado na definição de processos. No nível abstrato do exemplo, o processo consiste de uma atividade *a1*, e possui a política *PolInst1* habilitada. Além disso, a atividade *a1* especifica que os tipos de recursos requeridos são um computador portátil e um recurso financeiro. No nível instanciado pode-se observar a alocação de instâncias de recursos para a atividade *a1*.

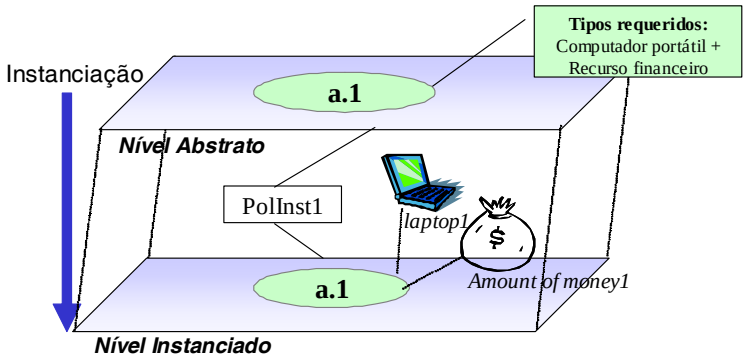


Figura 4.31: Instanciação de um processo de software

Enquanto no nível abstrato só estão definidas referências aos tipos de recursos requeridos por uma atividade, no nível instanciado o projetista pode definir instâncias reais da organização. Em uma mesma atividade, alguns elementos podem estar definidos nos dois níveis ou em um nível somente, não havendo restrição quanto aos componentes já instanciados pelo usuário.

O usuário pode receber auxílio à instanciação de recursos nos seguintes momentos:

- **Durante a modelagem de uma atividade** (antes de submeter o processo à execução): Neste caso, por solicitação do usuário, o mecanismo de instanciação verifica para cada tipo de recurso requerido se existem instâncias disponíveis. Se o recurso for exclusivo, então deve estar sem reservas para o período da atividade[14]. Se for consumível, deve haver uma quantidade suficiente para uso pela atividade. Por fim, se for do tipo

[14] A reserva não muda o estado do recurso. Apenas impede que outras atividades o utilizem no mesmo período

compartilhável, não deve estar no estado *Defect*. Além disso, caso o recurso requeira ou seja composto de outros recursos, estes devem ser verificados quanto à possível disponibilidade (i.e., devem estar sem defeito e sem reservas para o mesmo período);

- **Durante a execução do processo** (imediatamente antes da atividade começar): Neste caso a atividade pode estar no estado *Waiting* ou *Ready*. Prevendo a necessidade de definição dos recursos, o mecanismo de execução ativa a instanciação que age da mesma forma que no item anterior, para que o início da atividade não seja atrasado.

A seção a seguir apresenta como é realizada a primeira etapa do processo de instanciação: Geração de Sugestões.

4.2.4 Geração de Sugestões de Instanciação

Esta seção explica informalmente como são geradas sugestões de instanciação. Antes de implementado, este mecanismo de instanciação foi especificado algebricamente. Essa especificação será apresentada posteriormente. No entanto, as principais decisões tomadas acerca do funcionamento do mecanismo de instanciação encontram-se nesta seção.

A instanciação de recursos pode ser solicitada para um processo de software, um fragmento (porção) de processo ou para uma atividade folha do processo. Para cada atividade folha de um processo existe o componente *RequiredResources* e o componente *RequiredPeople*, mostrados na figura 3.15, os quais contêm a lista de recursos e pessoas necessários para execução da atividade. Antes da instanciação, apenas os tipos dos recursos necessários estão definidos. Portanto a instanciação baseia-se no conteúdo das referidas classes. Para cada recurso requerido de uma atividade é necessário sugerir uma lista de recursos que possam ser utilizados. Neste nível é necessário levar em consideração quais recursos do tipo requerido estão disponíveis, quais políticas estão habilitadas para esta atividade, e ainda escolher uma política para ser interpretada a fim de gerar a lista de sugestões.

Uma política de instanciação atrelada a uma atividade será interpretada através da avaliação das condições especificadas na mesma e posterior seleção dos recursos mais adequados (de acordo com os critérios escolhidos). Critérios restritivos sempre são avaliados antes dos critérios de ordenação. Quando nenhuma política puder ser aplicada, então um **algoritmo padrão** obtém as sugestões, restringindo os recursos a sugerir àqueles que: a) satisfazem o tipo solicitado na atividade; b) estão disponíveis no período necessário (juntamente com seus recursos requeridos e componentes). Depois os recursos resultantes da etapa anterior são ordenados pelo critério de menor custo.

A seguir são apresentadas as etapas da instanciação de recursos por políticas considerando uma atividade e um recurso requerido (a instanciação para agentes é análoga). Essas etapas são combinadas para obter uma lista de recursos sugeridos para um item de recurso solicitado. Assim, a instanciação de uma atividade requer que várias listas de sugestões sejam geradas. A figura 4.32 mostra o encadeamento destas etapas como um detalhamento da geração de sugestões de instanciação, modelada como atividade decomposta na APSEE-PML. Cabe observar que este processo é uma decomposição da atividade “Geração de Sugestões” mostrada na figura 4.30, sendo que

todas as suas atividades são executadas automaticamente pelo sistema e o artefato final desse processo é *Sugestões de Instanciação* que, no modelo especificado, é uma instância da classe *ResourcePolicyLog*, já citada.

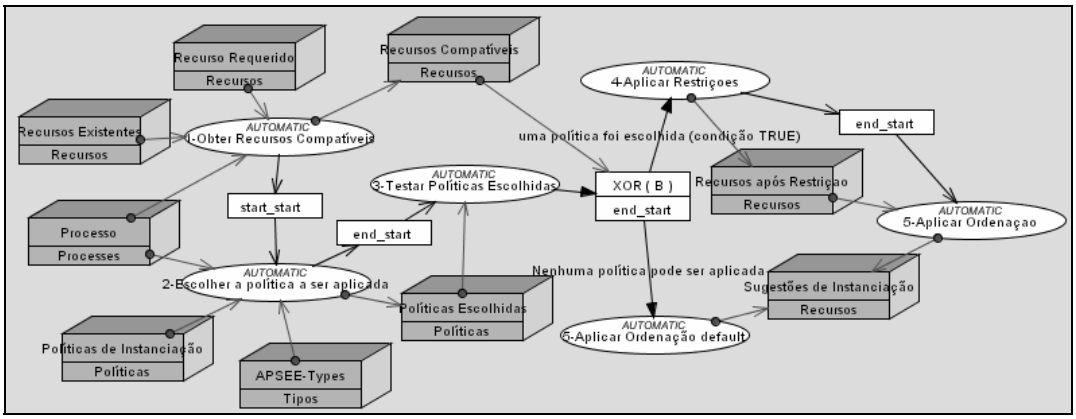


Figura 4.32: Detalhamento da Geração de Sugestões para Instanciação.

Os itens a seguir definem como cada passo do processo da figura 4.32 é executado.

- **Obter recursos compatíveis**: O conjunto inicial de recursos compatíveis para cada recurso requerido é obtido pelos critérios de tipo e disponibilidade para o período da atividade:
  - **Por tipo**: Com o fornecimento do tipo para cada recurso requerido de uma atividade, o primeiro passo da instanciação é obter o conjunto de recursos do tipo requerido. Por exemplo, se o recurso requerido é "computador", então todos os recursos do tipo computador e de seus sub-tipos são selecionados (incluindo *notebooks*, *handhelds*, entre outros).
  - **Por Disponibilidade**: é verificada a disponibilidade dos recursos exclusivos e consumíveis. Para recursos exclusivos são verificadas as reservas (classe *Reservation* no pacote *Organization.Resources*) e para recursos consumíveis é verificado se a quantidade necessária (atributo *amount_needed*) é menor ou igual à disponível (*total_quantity* – *amount_used* do *consumable_resources*). Alguns dos recursos disponíveis são compostos e/ou requerem outros recursos, portanto sua disponibilidade também deve ser verificada:
    a) **Componentes**: Os componentes de cada recurso selecionado anteriormente devem estar disponíveis no período da atividade ou com defeito. Se o recurso necessário é uma sala, seus componentes não devem estar alocados ou reservados para outras atividades;
    b) **Requeridos**: Recursos que são requeridos pelos recursos selecionados anteriormente devem estar disponíveis no período da atividade e não podem estar com defeito.

  O conjunto resultante desta etapa contém uma lista de recursos que são totalmente compatíveis com a atividade em questão. E, se forem escolhidos

ao final do processo, possuem recursos componentes e requeridos compatíveis;

- **Escolher a política a ser aplicada:** Dado um tipo de recurso requerido para uma atividade específica, a política a ser usada para instanciá-lo deve ser procurada primeiramente entre as políticas habilitadas na atividade. Em seguida entre as políticas habilitadas no processo e eventuais processos acima deste e por último são verificadas as políticas habilitadas na organização como um todo.

No caso de existirem várias políticas habilitadas a serem aplicadas, o algoritmo deve ordená-las, pois apenas uma política deve ser utilizada para instanciar o recurso. Para ordenar, utiliza-se a proximidade de habilitação (políticas habilitadas localmente têm mais prioridade) e, se várias políticas estiverem no mesmo nível de habilitação utiliza-se o campo *ApplyToType* fornecido na política. A figura 4.33 mostra um exemplo de habilitação de políticas em vários níveis e os tipos aos quais as políticas se aplicam. A atividade X habilita as políticas P1 e P2, o processo que a contém habilita a política P3. Os processos de níveis superiores e a organização também habilitam suas políticas. Neste exemplo pretende-se instanciar a atividade X que requer um recurso do tipo "Sala de Aula Prática". A figura 4.34 mostra uma parte da hierarquia de tipos de recursos a partir do tipo "Sala". Pode-se observar que neste caso existem duas políticas (P1 e P2) habilitadas na atividade que são aplicáveis ao tipo requerido. Neste instante, deve-se decidir qual das duas verificar primeiro (se a condição da política não for satisfeita, a próxima aplicável será verificada).

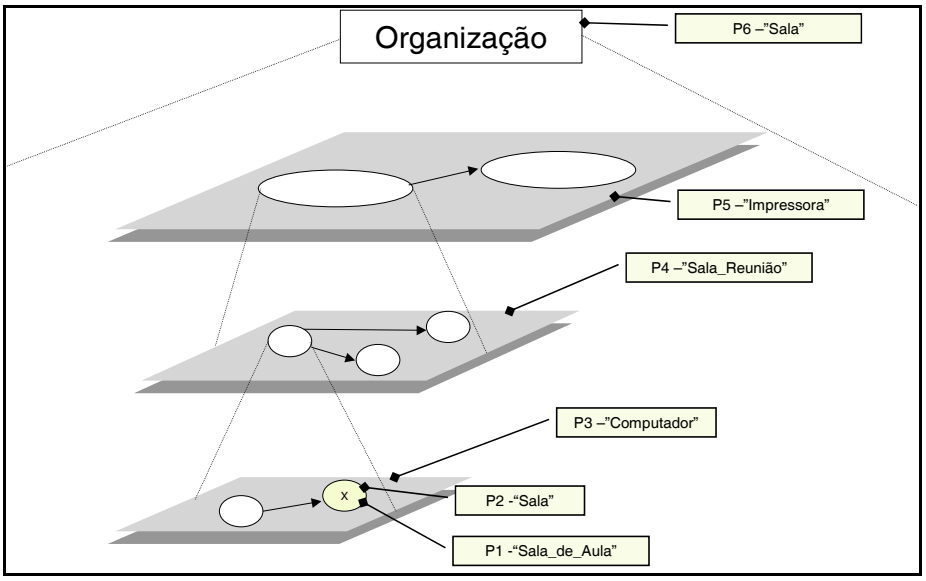


Figura 4.33. Habilitação de políticas em vários níveis.

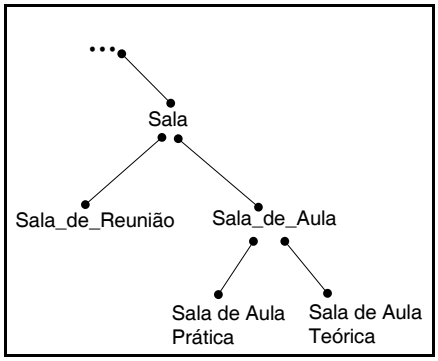


Figura 4.34. Exemplo de hierarquia de tipos de salas.

O critério de ordenação de políticas utilizado corresponde à escolha da política que se aplica ao tipo mais próximo do tipo requerido. São calculadas as distâncias entre o tipo requerido e o tipo da política na árvore de tipos e é obtida uma lista em ordem crescente de distâncias de tipos. No exemplo dado, o tipo requerido "Sala de Aula Prática" é comparado aos tipos das políticas P1 e P2, "Sala de Aula" e "Sala", respectivamente. A política P1 se aproxima mais do tipo requerido e será aplicada. Deve-se ressaltar que não há possibilidade de haver no mesmo nível duas políticas que se apliquem ao mesmo tipo, o que facilita a ordenação e a escolha da política a ser aplicada. Entretanto, é possível que nenhuma das políticas aplicáveis tenha sua condição satisfeita. Neste caso, os níveis superiores do processo e organização devem ser consultados e o processo de ordenação de políticas pelo tipo se repete. No exemplo, poderia ocorrer a instanciação através da política P6.

- **Testar Políticas Escolhidas**: Nesta etapa o mecanismo de instanciação dispõe de uma lista ordenada de políticas de instanciação (artefato "Políticas Escolhidas" na Figura 4.32). A condição da primeira política é testada. O campo *Conditions* da política permite verificar o estado atual do processo de desenvolvimento, o tamanho do software a ser desenvolvido, dentre outras métricas. A semântica da verificação de condições é definida em (REIS, 2002f) Para nossos propósitos, convém considerar que um método de avaliação retornará um valor booleano para o resultado da condição.

  Se o valor resultante for verdadeiro, então a política escolhida será aplicada, caso contrário, o mecanismo tentará aplicar a próxima política compatível identificada na segunda etapa. Se nenhuma das políticas da lista tiver condição verdadeira, a instanciação prossegue com o algoritmo *default*.

- **Aplicar critérios de restrição**: Os critérios de restrição do campo *RestrictBy* (apresentados na Tabela 4.1) são aplicados na ordem em que foram definidos na política. Nesta etapa é obtido um novo conjunto com recursos que atendem aos critérios definidos pelo usuário na política (artefato "Recursos após Restrição" na Figura 4.32). Deve-se observar que é possível a obtenção de um conjunto vazio de recursos caso a política seja muito restritiva.

- **Ordenar resultado**: A quinta e última etapa utiliza o critério fornecido no campo *OrderBy* (critérios da Tabela 4.2) para ordenar o conjunto resultante da quarta etapa. O valor utilizado para ordenar é apresentado para o usuário juntamente com a lista de sugestões com o objetivo de facilitar a decisão (caso a instanciação não seja automática). Por exemplo, caso o conjunto de salas tenha sido ordenado pelo menor custo, o usuário recebe uma lista de sugestões como mostrado a seguir, e pode avaliar o impacto da sua escolha:

Sala 1 – R$30,00 por hora

Sala 2 – R$31,00 por hora

...

Caso o algoritmo *default* tenha sido utilizado, então os recursos estarão ordenados pelo menor custo.

4.3 Execução de Processos de Software no APSEE

Durante a **execução de processos de software** as atividades modeladas e instanciadas são realizadas tanto pelos desenvolvedores quanto automaticamente. Esta fase envolve questões importantes acerca de planejamento, controle, monitoração, garantia de conformidade com o processo modelado, treinamento, segurança e recuperação do processo (FEILER; HUMPHREY, 1993).

O Mecanismo de Execução de Processos interpreta o modelo de processo instanciado e gerencia as informações do ambiente e desenvolvedores de acordo com esse modelo. Durante a execução é necessário obter *feedback* da realização do processo, o que permite manter a consistência entre o estado da execução e o estado da realização. O mecanismo de execução de processos proposto para o APSEE leva em consideração os requisitos levantados no capítulo 2 que enfatiza a importância dada ao envolvimento humano no processo, com o objetivo de aumentar a flexibilidade da execução e suportar processos criativos e incerteza. Assim, a execução de processos no APSEE tem como principais características:

- Permitir execução de processos incompletos e modificação dinâmica: O processo pode continuar sendo modelado enquanto algumas atividades já estão em execução. A modelagem do processo também pode ocorrer em partes do processo que já iniciaram. Nesse caso, mudanças dinâmicas são permitidas desde que não afetem a consistência do modelo de processo. A criação de um meta-modelo unificado e a construção de um mecanismo de execução que interage com as fases de modelagem e execução possibilita essa característica;
- Possibilitar a escolha dentre várias alternativas para o fluxo de execução dependendo da situação corrente (conexão *branch* com condições da APSEE-PML). Isto inclui consulta a conhecimento sobre processos executados anteriormente. Essa característica é possível devido à estrutura das conexões de controle propostas no meta-modelo;

- Permitir a repetição de atividades a partir da verificação da condição da conexão de *feedback*, sendo que as condições podem consultar situação corrente ou métricas armazenadas sobre o histórico do processo;
- Fornecer a instanciação automática para o processo ou seus fragmentos, utilizando para isso as políticas de instanciação definidas pelo usuário. Nesse caso as informações providas pelo meta-modelo acerca da organização são extrema valia para o funcionamento da instanciação. Essa característica integra o mecanismo de instanciação e de execução pois permite sincronizar seu funcionamento levando em consideração as políticas definidas pelo usuário;
- Fornecer um registro histórico dos processos, o que traz a possibilidade de disparo de políticas dinâmicas, que correspondem a regras ECA (Evento-Condição-Ação) que podem ser interpretar eventos ocorridos no processo e determinar ações como conseqüência. Além disso permitem auxiliar na modelagem e na obtenção de novos requisitos para processos;
- O mecanismo de execução obedece a semântica da linguagem APSEE-PML. Essa semântica é definida formalmente no paradigma do PROSOFT (algébrico) combinado com gramática de grafos. A especificação formal da semântica da APSEE-PML permite tratar o problema em alto nível de abstração, provê uma base para rastreamento e análise de impacto de alternativas, e define as conseqüências das transições de estados de forma resumida e unificada. Essa semântica será apresentada no próximo capítulo;
- Estar integrado com o mecanismo de gerência de objetos fornecido pelo componente *SoftwareArtifacts*, que estende o PROSOFT COOPERATIVO (REIS, 1998a; REIS et al., 1998b);
- Permitir execução de processos colaborativos, através de atividades que podem ser realizadas por várias pessoas e também através de atividades diferentes que podem ser realizadas em paralelo compartilhando artefatos de saída (por exemplo, para atividades conectadas através da dependência *start-start*).
- Permitir visualização da execução através do formalismo gráfico e executável para modelagem de processos. O mesmo formalismo é usado para modelar processos e acompanhar sua execução;
- Para facilitar a interação com os usuários e obter *feedback* sobre a realização do processo são fornecidas agendas de tarefas para cada agente envolvido no processo.

A execução trabalha com transformações em vários níveis, atendendo às solicitações de vários usuários ao mesmo tempo e continuamente verificando o estado corrente do processo. As seções a seguir mostram o comportamento da execução durante essas transformações para atingir os objetivos citados anteriormente. Assim, será apresentada a execução do ponto de vista de uma única atividade normal ou automática, do ponto de vista do desenvolvedor e do processo como um todo. Deve-se observar que, para uma compreensão geral da execução, é necessário levar em consideração também a semântica das conexões entre atividades.

4.3.1 Execução de uma Atividade Normal

Nessa seção será analisada a execução para cada atividade normal, ou seja, que precisa de pessoas e recursos. A figura 4.35 apresenta um diagrama de transição para os possíveis estados de uma atividade normal, sendo que os estados e seu significado foram apresentados na seção 3.7.3 do capítulo anterior. Deve-se ressaltar que ainda são levados em consideração a pré e pós-condições (se existirem) para início da atividade.

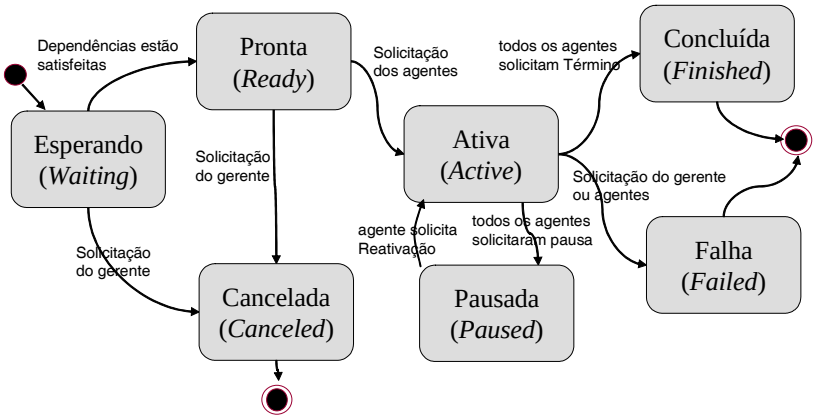


Figura 4.35: Transição de estados de uma atividade normal[15].

As transições de estado neste caso dependem da intervenção do gerente ou dos agentes. Deve-se observar que essas transições ocorrem para cada atividade do processo individualmente. O mecanismo de execução recebe as solicitações e dispara as transições quando necessário. Como atividades normais podem envolver diversos agentes, algumas transições dependem da concordância de todos os envolvidos. Como citado na seção anterior, para que os agentes saibam o estado em que a atividade se encontra e possam solicitar mudanças de estado eles dispõem de uma Agenda de tarefas. Em uma agenda, os estados da atividade são registrados do ponto de vista do agente. Por exemplo, se o agente termina a atividade, não significa que a atividade realmente foi concluída, pois ainda é necessário aguardar manifestação de todos os agentes envolvidos.

Uma atividade na agenda do agente pode assumir os estados mostrados na figura 4.36. O estado "*Delegada (Delegated)*" significa que a atividade foi repassada para outro agente. A delegação de tarefas modifica a descrição do processo (pois altera a alocação de agentes), e somente pode ser feita antes da atividade ser ativada (iniciada pelo agente). Em todo caso, o gerente do processo pode modificar essa alocação em outros momentos, dependendo da necessidade, através do editor gráfico da APSEE-PML.

[15] Os diagramas de transição de estados apresentados nesse capítulo seguem o estilo adotado no restante do trabalho. Isto é, para cada estado é fornecido o rótulo em português e em inglês (sendo que este último corresponde ao nome do atributo usado na especificação e implementação do modelo).

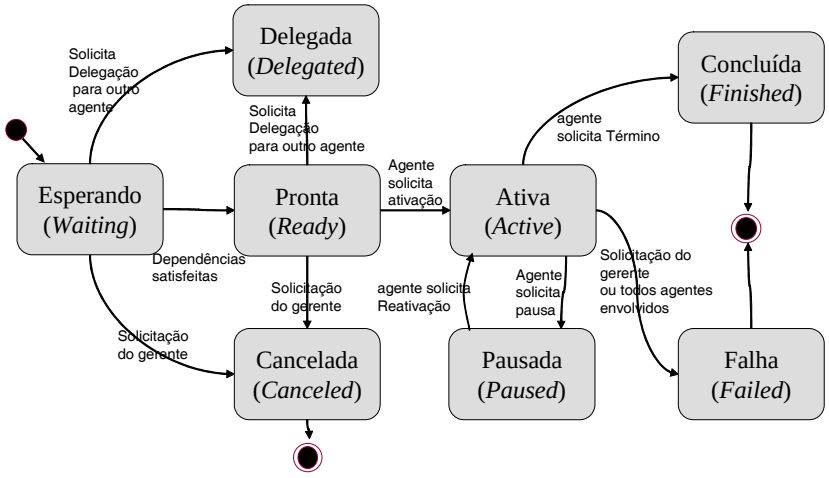


Figura 4.36: Transição de estados de atividade na agenda do agente.

Outra característica importante da execução de uma atividade normal é a alocação de recursos.Para que a atividade torne-se ativa, os recursos necessários devem ser alocados ou consumidos. No caso de indisponibilidade de um recurso, a atividade ficará bloqueada até que os recursos estejam disponíveis[16]. A gerência de recursos durante a execução é apresentada através da dinâmica da execução de uma atividade. O diagrama de estados mostrado na figura 4.37 descreve os estados de uma atividade em execução e o seu relacionamento com a gerência de recursos. A tabela 4.3 apresenta as operações de gerência de recursos, ilustradas na figura anterior, que o mecanismo de execução do ambiente necessita.

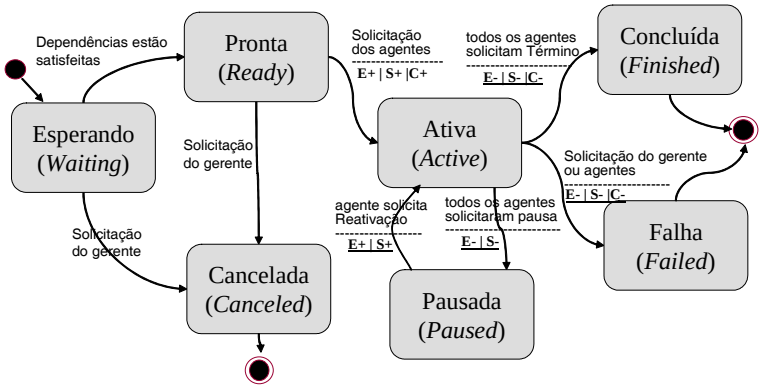


Figura 4.37: Transição de estados de atividade normal com gerência dos recursos.

[16] Este comportamento aumenta a importância de modelar precisamente os recursos *a priori*, pois permite escolher um substituto que esteja disponível. Ou, caso este não exista, aponta a necessidade de aquisição do mesmo.

Tabela 4.3: Operações de Gerência de Recursos

Operações / RECURSOS	Para uso do recurso	Para liberação do recurso
Recursos Exclusivos	**E+ Alocar_Recurso**: Se o recurso está disponível, muda estado para *locked* e registra início do uso no *log*.	**E– Liberar_Recurso**: Muda estado para *available* e registra fim do uso no *log*.
Recursos Compartilháveis	**S+ Inicia_uso**: Se o recurso está disponível registra uso do recurso no *log*.	**S– Termina_Uso**: Registra o fim do uso do recurso no *log*.
Recursos Consumíveis	**C+ Consome_Recurso**: Se recurso está disponível e é suficiente, aumenta *amount_used* com *amount_needed*. Registra consumo no *log*.	**C– Registra_fim_consumo**: Registra fim de consumo do recurso pela atividade no *log*.

A execução de atividades normais também pode levar à chamada do mecanismo de instanciação automática. Se a atividade não estiver instanciada, ou estiver com agentes ou recursos indefinidos, isso é detectado e, dependendo da escolha do usuário, o próprio sistema pode preencher as alocações em aberto, de acordo com as políticas habilitadas. Caso o usuário tenha definido que a instanciação não será automática[17], então o gerente receberá uma mensagem avisando que a atividade está pronta para executar e necessita de instanciação.

4.3.2 Execução de uma Atividade Automática

As atividades automáticas executam sem intervenção humana. Portanto, não são repassadas para agendas de agentes nem necessitam de alocação de recursos. Seu diagrama de transição de estados, apresentado na figura 4.38, é uma simplificação do diagrama para atividades normais. Quando a atividade automática possui suas dependências satisfeitas, incluindo sua pré-condição, ela é imediatamente executada, de acordo com a sua definição (isto é, uma chamada de método ou *script* de sistema operacional).

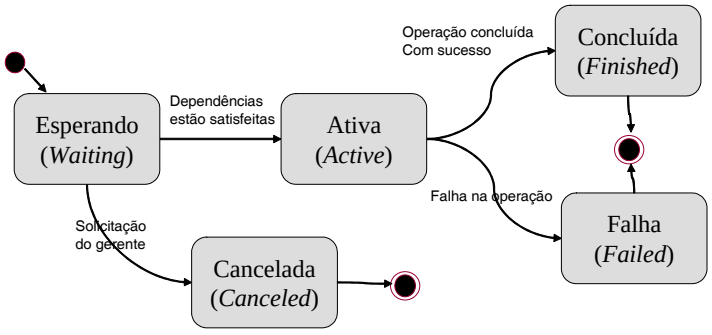


Figura 4.38: Transição de estados de uma atividade automática.

[17] Isso é definido através do valor para um atributo da classe *Organization*, pertencente ao pacote *OrganizationPolicies* da Figura 3.5.

4.3.3 Execução de um Modelo de Processo de Software

Um processo de software no APSEE é definido através de uma instância da classe modelo de processo (conforme apresentado na figura 3.13). O processo (mais geral) possui somente três estados: *não iniciado*, *executando* e *concluído*. Assim que o gerente decide solicitar execução do processo, este passa para o estado "executando" e o modelo de processo passa também a registrar seu estado de execução. Além disso, o gerente de processos pode solicitar que o processo inicie a sua execução mesmo antes de defini-lo completamente. O gerente também percebe a transição de estados de um modelo de processos e a propagação das transições das atividades-folha para os níveis superiores.

No momento em que o gerente decide iniciar a execução, ocorre a inclusão das atividades correspondentes nas agendas dos agentes envolvidos no processo. Como visto nas seções anteriores, um modelo de processo de software é composto de atividades que, por sua vez, possuem estados. O estado de um modelo de processo depende do estado das atividades componentes e deve refletir também a situação na qual as atividades ainda estão sendo modeladas. Portanto, para que o ambiente identifique em que fase do ciclo de vida encontra-se um processo, é proposto o diagrama de transição da figura 4.39.

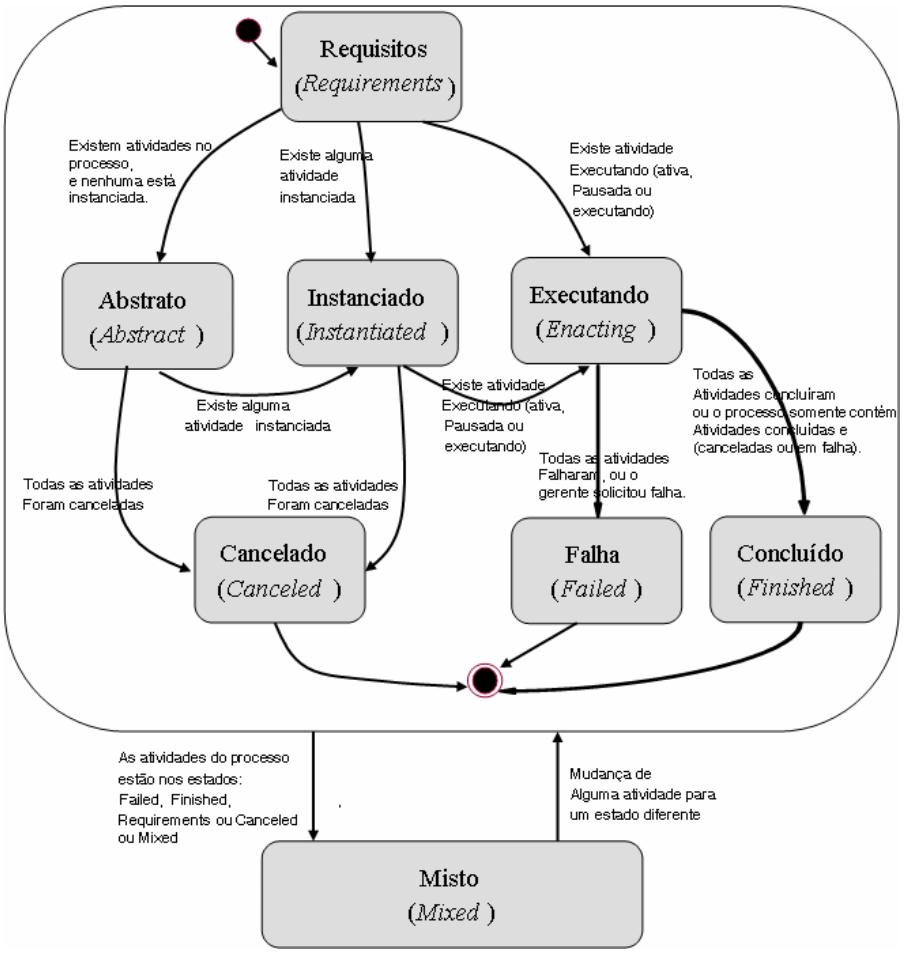


Figura 4.39: Transição de estados de um processo de software (e atividade decomposta).

Como é mostrado na figura 4.39, o modelo de processo é iniciado no estado "Requisitos" quando a sua definição resume-se a uma descrição dos seus requisitos e ainda não há atividades componentes. A partir do momento em que é criada alguma atividade no processo, este pode passar para o estado "abstrato" ou "instanciado". O processo assume o estado "Abstrato" se nenhuma atividade estiver pronta para executar, ou seja, estão definidos somente os cargos necessários e não foram definidos agentes para a sua realização (e, de forma análoga, para recursos). Caso alguma atividade já possua essa definição, então o modelo de processo é considerado "instanciado" (podendo ser executado, mesmo que esteja ainda incompleto). O modelo de processo continua instanciado à medida que novas atividades forem sendo incluídas no mesmo, sejam elas abstratas ou instanciadas.

Quando ocorre a instanciação de uma atividade componente de um modelo de processo que já está em execução, também é atualizada a agenda dos agentes responsáveis. Quando as atividades desse processo iniciarem, seus diagramas de estados serão executados em paralelo com as transições de estados de modelo de processos (figura 4.39), sendo que há uma monitoração interna desses estados para sincronizar com os de processo

O gerente pode solicitar o cancelamento do processo antes da execução, sendo que todas as atividades componentes são canceladas, incluindo as decompostas. Se o processo não for cancelado, então, a partir do início da execução (solicitada pelos agentes) de qualquer atividade do processo, este passa para o estado "executando". Nesse caso, outras atividades podem continuar sendo instanciadas no modelo, sem alterar o estado geral do processo. Essa característica contribui para a obtenção de flexibilidade durante a execução.

Se, estando o processo no estado "executando", o gerente decide que o mesmo não deve continuar sendo executado, então é possível falhar o processo todo: isso faz com que todas as atividades também passem para o estado de falha.

Para que o processo seja "concluído", todas as suas atividades devem estar concluídas, ou então algumas concluídas e outras em falha ou canceladas.

Em qualquer momento da evolução de um modelo de processo, este pode passar para o estado "misto" (*mixed*). Este estado foi criado para distinguir os processos que não satisfazem os requisitos para estarem nos demais estados. Por exemplo, quando o processo contém alguma atividade decomposta no estado *mixed* ou tem atividades nos quatro estados a seguir: *failed*, *finished*, *requirements* e *cancelled*. A condição para deixar este estado *mixed* é a mudança de qualquer atividade componente do modelo para um estado diferente dos quatro citados.

4.3.4 Fluxo de controle da Execução através de Conexões

O uso de conexões de controle (simples e múltiplas) durante a modelagem de processos permite definir as dependências entre atividades do processo. Para determinar o fluxo de controle do processo a partir das conexões, o mecanismo de execução do APSEE leva em consideração:

- Conexões simples entre duas atividades com o tipo de dependência (*end-start*, *start-start*, *end-end*), indicando quando a segunda atividade pode começar ou terminar;
- Conexões simples de *feedback* contendo condições que habilitam o retorno a uma atividade anterior. A repetição de um fragmento de processo pode ser definida em tempo de modelagem ou essa conexão pode ser acrescentada em tempo de execução quando, a critério do usuário, se percebe que algumas atividades devem ser refeitas. No término de uma atividade, se houver conexão de *feedback* e a condição associada for verdadeira, então são criadas versões para a atividade alvo do *feedback* e as suas sucessoras (propagação de *feedback*) e a execução anterior das atividades não é excluída do processo, podendo ser repetidas quantas vezes forem necessárias;
- Conexões múltiplas *Branch*. Várias atividades sucessoras podem ser escolhidas dependendo do operador lógico utilizado (*AND*, *XOR* ou *OR*). Conexão *Branch-AND* possui o mesmo significado que várias conexões simples entre a atividade origem e as atividades destino. *Branch-XOR* escolhe apenas uma das atividades destino a partir da avaliação das condições de ativação para cada uma. *Branch-OR* permite que um subconjunto das atividades destino possa ser habilitado, também a partir das avaliações de condições. Além dos operadores lógicos, o tipo de dependência influencia na execução. Por exemplo, *Branch-XOR end-start* escolhe uma das atividades destino a partir do término da origem. Caso a dependência for *start-start*, a atividade destino pode ser habilitada após o início da origem;
- Conexões múltiplas *Join*. Várias atividades origem podem habilitar a execução de uma atividade destino dependendo do operador lógico e da dependência utilizados. Por exemplo, *Join-XOR-end-start* significa que a atividade destino pode ser habilitada caso uma das origens termine. *Join-OR* implica que a atividade destino somente pode ser habilitada se um subconjunto das atividades origem terminarem (esse subconjunto deve possuir mais de um elemento). *Join-AND* somente habilita a atividade destino se todas as origens terminarem, sendo este caso equivalente ao uso de várias conexões simples para diferentes origens e um mesmo destino;
- As conexões múltiplas (tanto *join* quanto *branch*) possuem um atributo booleano "*Fired*" que guarda o estado da conexão. Assim que a conexão está habilitada, as atividades de destino podem ser habilitadas;
- Conexões de artefato influenciam a execução ainda por indicarem quais artefatos devem ser inseridos no espaço de trabalho dos desenvolvedores antes da atividade iniciar, e quais artefatos podem ser retirados e podem ter seus estados alterados quando a atividade termina. Nesse caso, trata-se de liberar acesso a arquivos no momento adequado.

4.3.5 Propagação Automática de Falha e Cancelamento

As atividades de um processo podem ser interrompidas (falhar) após terem iniciado a sua execução ou podem ser canceladas (somente antes de começarem a execução).

Devido a mudanças no processo, pode ser que o gerente perceba problemas na execução de uma atividade e decida que a mesma não deve mais continuar. Para tratar falhas e cancelamentos, o mecanismo de execução adota o seguinte procedimento:

- **Propagação de Cancelamento**: Quando uma atividade é cancelada (ainda não iniciou sua execução), suas sucessoras também o são, e no caso de atividades normais, os respectivos agentes são notificados através da agenda. As sucessoras podem ter diferentes dependências em relação a atividade cancelada. Por isso, no caso da sucessora estar participando de uma conexão múltipla do tipo *Join*, ela somente será cancelada se todas as antecessoras foram canceladas. Isto previne o cancelamento total de um *Join OR* ou *XOR* quando apenas uma atividade antecessora foi cancelada. Além disso, quando uma atividade decomposta é cancelada, as sub-atividades que a compõem também o são. A propagação de cancelamento é feita automaticamente e atinge somente as atividades que ainda não iniciaram, que dependem da atividade cancelada e que não estão falhadas. O gerente pode então efetuar mudanças nas atividades e retomar sua execução, se for necessário.

  A Figura 4.40 mostra um exemplo de processo antes e depois da atividade A ser cancelada. Pode-se observar que a atividade B é cancelada, mas as outras (C e D) continuam no mesmo estado. A atividade D somente será cancelada se a atividade C também for.

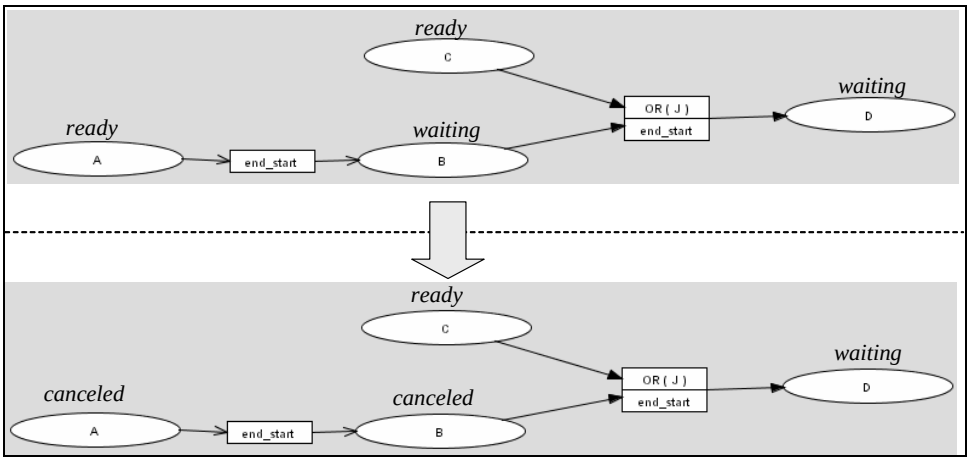


Figura 4.40: Exemplo de propagação de cancelamento de atividade.

- **Propagação de Falha**: Quando uma atividade é interrompida por falha, isso significa que já havia sido iniciada. Portanto, o tratamento da propagação de falhas é mais cuidadoso: as sucessoras falham somente se dependerem do fim da atividade falhada. Daí, conexões *start-start* não propagam falhas. Quando ocorre a falha em atividades normais, os recursos alocados devem ser liberados e os agentes envolvidos devem ser notificados imediatamente. A falha é propagada para todas as sucessoras que dependem da atividade falhada, inclusive as sucessoras que ainda não iniciaram. Da mesma forma que no cancelamento, as conexões *Join OR* e

XOR somente propagam a falha caso todas as antecessoras tenham falhado (porém, uma combinação de falhas e cancelamentos propaga a falha).

A Figura 4.41 apresenta um exemplo de propagação de falha em um processo. No exemplo, a atividade A é falhada pelo gerente e essa falha é propagada para a atividade C. A atividade B não altera seu estado porque dependia somente do início de A (o que de fato ocorreu), e a atividade D não falha por causa do *Join OR* (assim como no cancelamento).

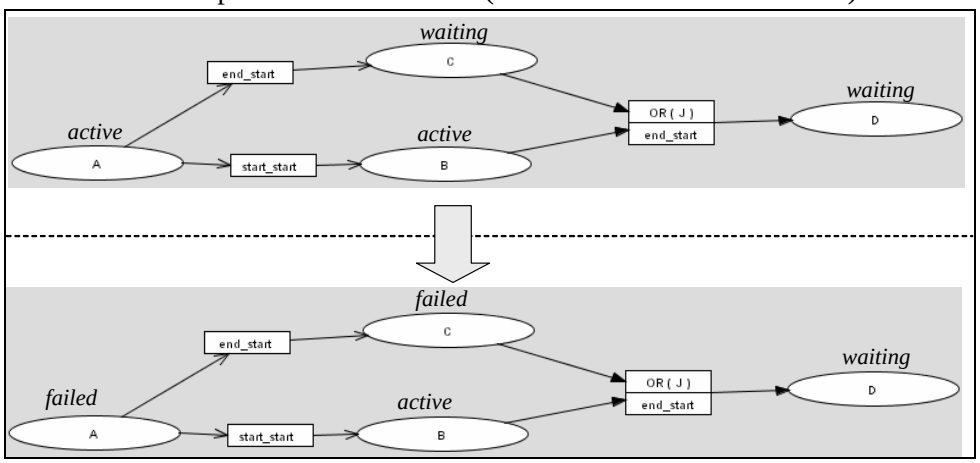


Figura 4.41: Exemplo de propagação de falha de atividade.

Existem regras a serem apresentadas no próximo capítulo que detalham a propagação de cancelamento de atividades no processo em todos os possíveis casos.

4.3.6 Registro de Eventos

Todas as ocorrências do processo geram eventos que são armazenados em registros. O registro de eventos é feito em cada componente do processo. Por exemplo, para saber os eventos das atividades de um processo é necessário observar os eventos de cada atividade. É registrado o identificador do evento (por exemplo: *failed*) e a data e hora de sua ocorrência. O registro de eventos está presente em todos os níveis do meta-modelo apresentado no capítulo 3 e foi proposto por ser um requisito básico da execução de processos e por facilitar a criação futura de diversos mecanismos, tais como: mecanismo de descoberta de conhecimento e Políticas Dinâmicas. Ambos são discutidos na seção de trabalhos futuros das conclusões no capítulo 8:

4.4 Integração entre as fases do Ciclo de Vida de Processos no APSEE

Como já mencionado no capítulo 2, o objetivo principal do ciclo de vida é apoiar a produção, refinamento e evolução de processos de software: assim, é importante que o meta-processo seja bem definido e seguido. Segundo (NGUYEN; CONRADI, 1994), meta-processos têm sido pouco estudados e seu potencial não tem sido bem explorado. Alguns PSEEs adotam uma parte do meta-processo, apenas como conseqüência de sua arquitetura, e se restringem às fases de modelagem e execução de processos de software.

A evolução dos processos de software depende do meta-processo adotado, o qual descreve como e quando o processo pode ser modificado, como um processo executado afeta a criação de novos processos e quais fases contribuem para o seu aperfeiçoamento.

O ciclo de vida de processos de software adotado no APSEE é baseado nos requisitos de automação da gerência do processo e flexibilidade na execução. Portanto, foram propostas fases adicionais que, em conjunto, não são encontradas em meta-processos da literatura, e que auxiliam a evolução do processo a atingir os requisitos do trabalho aqui apresentado. A figura 4.42 apresenta o meta-processo do APSEE. Utilizou-se a própria APSEE-PML para representar as fases do meta-processo APSEE, assim como seus artefatos e fluxo de controle (com a sintaxe já apresentada no decorrer deste capítulo).

O ciclo de vida inicia-se com a fase de **análise de requisitos do processo** que produz requisitos para o processo a ser desenvolvido. A fase de **modelagem** resulta em um modelo de processo abstrato através do uso da PML do APSEE. Também é possível modelar *templates* (processos abstratos) para reutilização. Esta fase pode ser auxiliada pela fase de **recuperação e adaptação de processo** (reutilização), que pode fornecer *templates* de processo adequados à situação corrente. A fase de **instanciação de processos** modifica o processo de software modelado para produzir um modelo de processo instanciado, ou seja, com recursos e agentes alocados e cronogramas definidos. Antes da execução, o processo instanciado pode passar por uma fase de **validação** que é auxiliada pela simulação de processos de software, a qual permite antever problemas de alocação e prazo no processo instanciado. Como resultado da fase de validação, pode ser necessário retornar a fases anteriores para refinamento do modelo, até que este esteja pronto para execução.

Durante a **execução** a máquina de processos coordena a interação com gerentes e desenvolvedores e obtém *feedback* sobre o andamento do processo. Neste caso, pode ser necessário modificar dinamicamente o modelo. Assim, as fases de modelagem, instanciação e validação podem ser realizadas em paralelo com a execução. Durante ou após a execução, o modelo de processo é avaliado na fase de **avaliação do processo** para gerar novos requisitos e o registro de todas as ocorrências da execução. Processos executados também são enviados à fase de **generalização de processos** que alimenta a base de processos para reutilização.

Algumas fases do ciclo de vida podem ser apoiadas por ferramentas de gerência de processos e de projetos, assim como por políticas definidas que ajudam a verificar propriedades do processo (políticas estáticas), instanciar atividades (políticas de instanciação) e tratar eventos durante a execução (políticas dinâmicas). Além disso, o ciclo de vida leva em consideração as informações sobre a organização que o está adotando, ou seja, agentes, grupos, cargos e recursos envolvidos com atividades do processo de software. A integração entre as fases propostas é obtida através do meta-modelo unificado já apresentado. Acredita-se que o ciclo de vida do APSEE contribua no estabelecimento de requisitos para PSEEs em geral, pois suas fases demandam o uso de técnicas e ferramentas diferenciadas, que não são normalmente encontradas em PSEEs ou ambientes de *Workflow*, tais como reutilização e validação por simulação. Deste modo, a inclusão dessas fases aumenta a possibilidade de aperfeiçoamento do processo a cada ciclo.

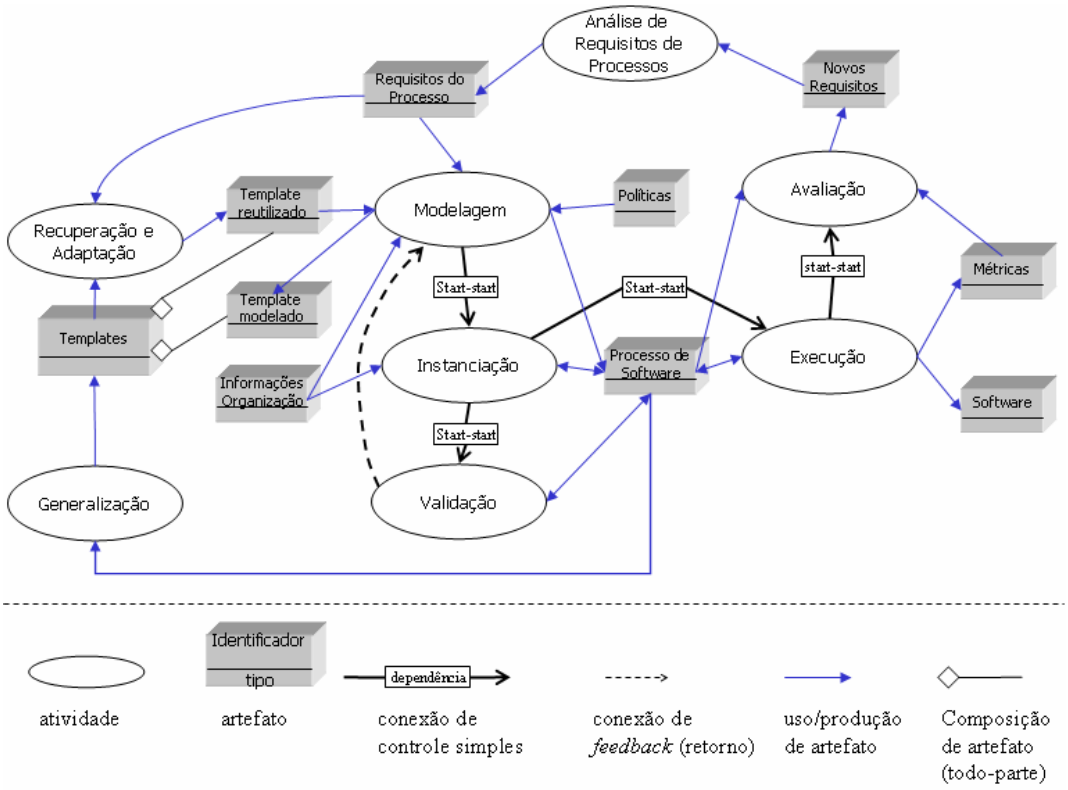


Figura 4.42: Ciclo-de-vida de processos de software no APSEE.

5 ESPECIFICAÇÃO DO MODELO PROPOSTO

Os capítulos anteriores descreveram informalmente o meta-modelo e mecanismos para gerência de processos de software propostos para o ambiente APSEE. O funcionamento do mecanismo de execução obedece à semântica do formalismo de modelagem, assim como o funcionamento do mecanismo de instanciação obedece à semântica de políticas de instanciação. Uma semântica formal é necessária para esses mecanismos devido às suas características de concorrência e devido à possibilidade de obter um modelo em alto nível de abstração para servir de apoio à implementação. Este capítulo apresenta a semântica desses mecanismos usando diferentes métodos de especificação formal.

5.1 Formalismos Utilizados

Especificação formal é a aplicação da matemática na especificação de software. Seu objetivo é superar a falta de precisão e a ambigüidade inerentes às especificações informais (por exemplo em linguagem natural) (NISSANKE, 1999). Vários métodos de especificação formal surgiram com diferentes abordagens. Por exemplo, na abordagem **operacional**, temos as máquinas de estados finitos e o CCS (MILNER, 1980) (neste último caso com o objetivo de especificar sistemas concorrentes e/ou paralelos). A abordagem **denotacional** é baseada na teoria dos domínios de Scott ((STOY, 1977), citado por (RIBEIRO, 2000)), onde o modelo do sistema é construído a partir de estruturas matemáticas conhecidas, como conjuntos e funções, e posteriormente são construídas as funções que modificam o estado do sistema. VDM (*Vienna Development Method* (BJØRNER; JONES, 1978)) é um exemplo de método nesta abordagem. Na abordagem **axiomática**, as propriedades de um sistema são dadas por um conjunto de asserções que ele deve satisfazer. A linguagem algébrica (WATT, 1991) e OBJ (GOGUEM et al., 1986) são exemplos de métodos na abordagem axiomática.

Segundo Dèharbe et al. (2000), métodos que seguem abordagem operacional são mais facilmente entendidos que os de abordagem denotacional, porém resultam em especificações mais complexas. A idéia de se combinar mais de uma abordagem de especificação para a descrição de diferentes aspectos de um sistema complexo surgiu para reunir as vantagens das abordagens. Gramáticas de grafos é um exemplo de método no qual os estados de um sistema são descritos por estruturas algébricas (grafos) e o comportamento do sistema é descrito de maneira operacional através de mudanças de estados (DEHARBE et al., 2000).

No caso da especificação de um sistema complexo, que inclui uma linguagem visual de modelagem de processos, aliada a um mecanismo de execução e instanciação flexível de processos, é importante combinar métodos de especificação que sejam adequados a cada aspecto modelado. As seções a seguir apresentam as características do problema, as justificativas para escolha dos formalismos e uma breve introdução aos formalismos adotados.

5.1.1 Características do Problema

O meta-modelo APSEE, descrito no capítulo 3 e os mecanismos de gerência de processos descritos no capítulo 4 envolvem diferentes características para atender os requisitos do trabalho.

A linguagem de modelagem de processos APSEE-PML permite definir graficamente o processo, sendo, portanto uma linguagem visual. Sua representação interna é especificada no meta-modelo de processos, porém o usuário utiliza símbolos visuais na modelagem. Linguagens de modelagem de processos são diferentes de outras linguagens computacionais porque a maioria dos fenômenos descritos devem ser realizados por pessoas e não por máquinas. Portanto, a escolha de mecanismo de abstração é de extrema importância para representar de forma adequada as informações descritas em um processo de software.

A especificação da sintaxe e semântica de linguagens visuais é um tópico importante de pesquisa e tem evoluído bastante nos últimos anos (BARDOHL, 1999; 2000). Já a semântica da execução de um modelo de processo de software envolve transições de estados que dependem de valores de atributos envolvidos em relacionamentos com componentes do processo. Processos de software podem ser vistos como grafos que envolvem aspectos de concorrência, porém a execução de cada atividade depende de informações diversas e inter-relacionadas. Em um trabalho anterior (LIMA REIS, 1998), foi utilizado o método algébrico para especificar como um processo executa. Porém, o meta-modelo proposto neste trabalho possui uma quantidade de construtores muito maior que o citado, pois trata vários tipos de conexão entre atividades e propõe diversos mecanismos diferenciados, além de fornecer uma integração mais ampla para diferentes etapas do meta-processo de software. Especificar sua dinâmica utilizando somente abordagem algébrica dificultaria o entendimento e não permitiria que aspectos de concorrência sejam tratados[18]. Portanto, as características do mecanismo de execução de processos de software levam à seleção de formalismos adequados. A literatura descreve várias experiências usando abordagens formais e semi-formais (ver por exemplo, (KRAPP, 1998)). Assim, semântica formal é considerada um importante requisito para especificar os elementos críticos de máquinas de execução de processos.

As Políticas de Instanciação, por sua vez, são definidas textualmente e necessitam de um interpretador para sua execução. Tal interpretador analisa os critérios e condições e gera sugestões para o usuário. Nesse caso uma especificação algébrica ajuda a definir de forma abstrata os resultados esperados das funções de interpretação, as quais definem como o estado das sugestões é modificado quando políticas são escolhidas para instanciação.

---

[18] Considerando-se métodos algébricos tradicionais, tal como a proposta de Watt (1991).

É objetivo desse trabalho implementar e avaliar o modelo proposto. Para isso, a especificação torna-se aliada no projeto do sistema, permitindo uma descrição abstrata, porém precisa do seu comportamento. Vale ressaltar que, embora o uso de métodos formais possibilite realização de provas e verificações matemáticas acerca de propriedades do modelo, assim como possibilita a derivação para implementação, tais características não constituem objetivo do trabalho aqui apresentado, sendo, entretanto, recursos importantes que podem ser aproveitados em trabalhos futuros.

5.1.2 Justificativas para escolha dos formalismos

A partir das características do problema citadas na seção anterior, foi realizado um estudo de métodos mais adequados para especificação do meta-modelo e dos mecanismos propostos. Considerando que o trabalho está inserido no grupo de pesquisa PROSOFT, coordenado pelo Prof. Dr. Daltro José Nunes, que propõe um ambiente de desenvolvimento formal de software de mesmo nome, foi avaliada a definição das propostas para funcionamento integrado ao ambiente.

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. Além disso, o ambiente é baseado em objetos e distribuído, o que facilita a integração de ferramentas que necessitam desse recurso. No PROSOFT, as especificações formais constituem uma base sólida que guia a implementação de protótipos ou sistemas de software completos. Nesse ambiente as ferramentas são construídas utilizando um paradigma algébrico, e há uma correspondência entre os componentes especificados algebricamente e a sua implementação. Assim, o formalismo algébrico mostra-se adequado para especificar os tipos de dados e operações básicas do meta-modelo aqui proposto, assim como o mecanismo de interpretação de políticas.

Para a especificação da linguagem de modelagem visual APSEE-PML e do mecanismo de execução foi escolhida a abordagem de gramática de grafos. A escolha foi inspirada nos trabalhos propostos por Bardohl (1999; 2000), onde se utiliza esse formalismo para especificar sintaxe e semântica de linguagens visuais. 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 complementar a especificação da semântica da execução de processos.

Gramáticas de Grafos (GGs) surgiram na década de 1970 como uma alternativa promissora ao uso de métodos formais textuais. GGs evoluíram com o surgimento de vários métodos e ferramentas para apoiar seu uso em vários domínios (EHRIG et al., 1999; ROZENBERG, 1997). Na tecnologia de processos, uma experiência a ser destacada foi descrita para o ambiente Dynamite (WESTFECHTEL, 1999), que aplicou a técnica de reescrita de grafos para especificar seu meta-modelo. A experiência no projeto do ambiente Dynamite fornece uma percepção da viabilidade do uso de GGs para definir semântica formal para mecanismos de gerência de processos sofisticados de forma gráfica e intuitiva. Além disso, o uso de abordagens baseadas em especificação algébrica e GGs pode ser justificado por um número de fatores descritos abaixo:

- Para obter um nível aceitável de integração dos serviços fornecidos por uma infra-estrutura de gerência de processos, um meta-modelo unificado é

requerido. O meta-modelo contém informações relacionadas sobre os processos, suas atividades e dependências e sobre a organização de desenvolvimento de software juntamente com seus recursos tecnológicos. Requisitos adicionais para mecanismos de execução modernos também contribuem para aumentar a complexidade do meta-modelo, tais como a necessidade de apoiar reutilização e simulação de modelos de processo;

- A execução de processos é realizada através de transformações de fina granulosidade nos estados dos processos, as quais afetam as informações associadas (por exemplo, recursos alocados, pessoas e agendas). Assim, as operações de execução devem garantir a consistência do modelo durante essas transformações e tratar o alto grau de paralelismo existente durante a execução de processos. O uso de um método formal é necessário para descrever esse comportamento de forma abstrata e precisa. Além disso, o uso de uma notação gráfica tem o potencial de facilitar implementação de protótipos;
- Alguns requisitos específicos de execução de processos também constituem fatores importantes para justificar o uso da abordagem de GGs. A gerência de descrições de processos incompletos e o apoio a modificações dinâmicas constituem requisitos desafiadores que devem ser satisfeitos simultaneamente pelo mecanismo de execução aqui proposto que, por outro lado, também deve garantir consistência. Assim, a modelagem de regras de transformação pode ajudar a definição de estados de consistência que são úteis para detectar problemas de consistência e gerência;
- A necessidade de integrar a execução de processos ao modelo organizacional também constitui uma questão chave no mecanismo de execução proposto. Já que ambos, processos de software e modelos organizacionais formam estruturas de dados de tamanho considerável, o paradigma de modelagem através de grafos pode ser útil para categorizar ocorrências especiais, encapsulando importantes eventos de execução que diretamente afetam os componentes organizacionais associados.

5.1.3 PROSOFT Algébrico

PROSOFT-Algébrico (descrito em (NUNES, 1992; 1994)) é um formalismo que permite a descrição de tipos abstratos de dados através de um paradigma algébrico baseado em objetos. Esse paradigma adota uma abordagem *data-driven* (NUNES, 1994) 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 (COHEN, 1986). Cada tipo de dados é instanciado a partir de um ATO - Ambiente de Tratamento de Objetos. A figura 5.1 apresenta o esquema gráfico de um sistema de software desenvolvido sob o paradigma do PROSOFT, sendo composto por um número de ATOs.

Cada ATO é dividido em três partes, como mostrado à direita na figura 5.1. 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 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.

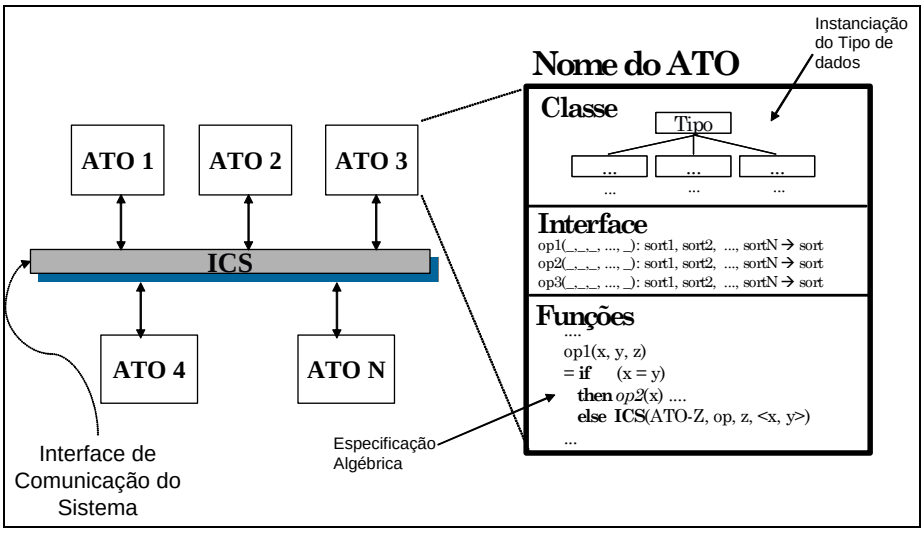


Figura 5.1: Composição de ATOs Algébricos na descrição de software no ambiente PROSOFT

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*>)[19]

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 dados. Qualquer termo desse tipo é chamado de objeto e, segundo Nunes (1994), não é similar com o conceito

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

de objeto das linguagens de programação orientadas a objetos[20]. As instâncias de uma classe (objetos) são entidades passivas, que não têm capacidade de responder a estímulos (mensagens) (DAUDT, 1992). 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.

Para definir as classes dos ATOs no PROSOFT-Algébrico é usada a linguagem gráfica para instanciação de tipos de dados derivada da notação de Jackson (1983), conforme exemplificado nas figuras a seguir. A definição da 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 associados a tipos primitivos ou definidos pelo usuário.

A figura 5.2 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, e permitem a manipulação dos itens de dados inseridos na lista instanciada.

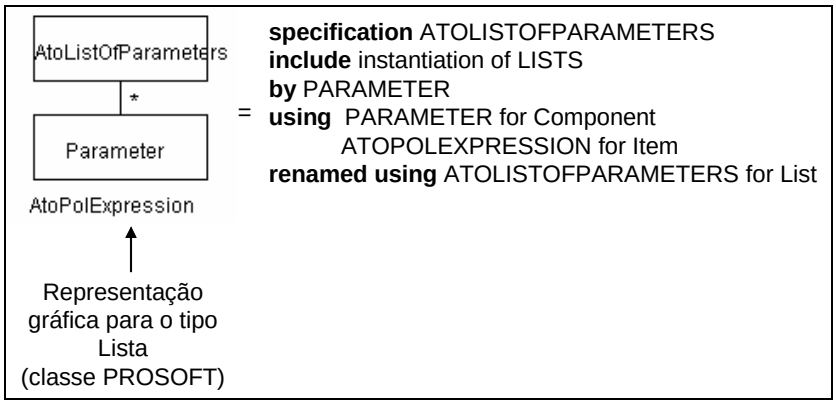


Figura 5.2: Exemplo da representação gráfica usada na composição de tipos de dados PROSOFT

Exemplos adicionais com os outros tipos de dados compostos (conjunto, mapeamento, união disjunta e registro) são apresentados na figura 5.3.

[20] 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 (1996), uma classe PROSOFT é uma estrutura de dados que corresponde ao que normalmente é chamado de atributos nas linguagens orientadas a objetos.

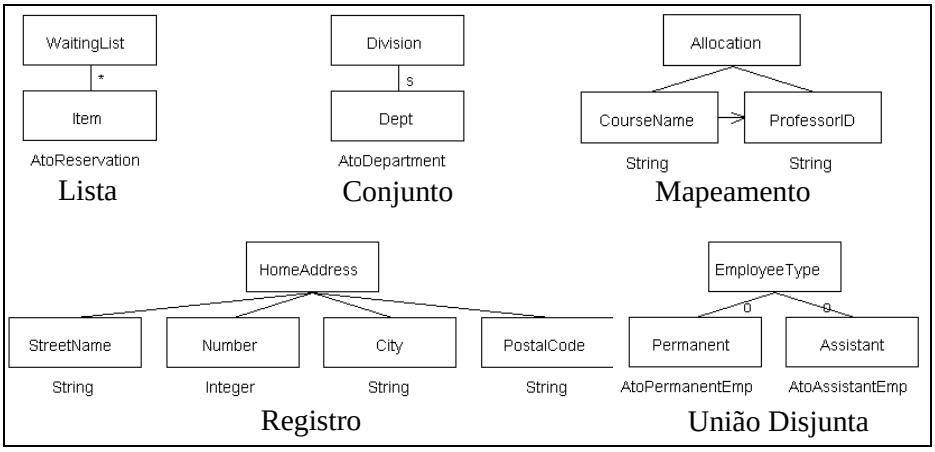


Figura 5.3: Notação gráfica para definição de tipos compostos

Comparando-se a especificação algébrica convencional (como o método descrito por Watt (1991)) 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, sendo que 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, onde os componentes são encapsulados e descrevem interfaces bem definidas.

A adoção do PROSOFT-Algébrico (para especificação dos tipos abstratos de dados) proporcionou um mapeamento para a implementação no ambiente PROSOFT-Java[21], visto que há uma correspondência para muitos elementos usados na especificação e implementação dos componentes de software descritos nesse paradigma.

5.1.4 Gramática de Grafos

Os métodos, técnicas e resultados na área de Gramáticas de Grafos já foram estudados e aplicados em uma grande variedade de campos da informática (DEHARBE et al., 2000). Gramáticas de Grafos generalizam gramáticas de Chomsky usando grafos ao invés de strings (RIBEIRO, 2000) e baseiam-se no processo de transformação que um grafo pode sofrer em função de um conjunto de regras previamente definidas. Um sistema 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 existe uma ocorrência (match em inglês) do lado esquerdo (L) da regra no grafo atual G. O lado direito (R) da regra define o grafo resultante da aplicação desta regra. A interpretação de uma regra r:L → R é feita da seguinte forma:

[21] PROSOFT-Java é a denominação para o ambiente escolhido para experimentação do modelo APSEE.

- Itens em L que não tem imagem em R são eliminados;
- Itens em L que são mapeados para R são preservados;
- Itens em R que não tem uma pré-imagem em L são criados.

Existem várias abordagens diferentes para Gramáticas de Grafos. Nesse trabalho será usada a abordagem algébrica através de mecanismos de "tipagem" nos grafos. Deste modo, será apresentado um grafo tipo representando os tipos de nodos e arcos do sistema. O grafo inicial e os grafos derivados das transformações devem ser compatíveis com o grafo-tipo. Em seguida, serão apresentadas algumas regras de derivação do grafo inicial.

A aplicação de uma regra a um grafo N (*passo de derivação)* resulta em:

1. Adicionar a N tudo o que for criado pela regra;
2. Excluir do grafo gerado pelo passo 1 tudo o que deve ser excluído pela regra (itens que fazem parte do lado esquerdo e não do lado direito);
3. Excluir arcos pendentes. Caso vértices tenham sido excluídos no passo anterior e se existirem vértices conectados a eles, estes devem ser excluídos também.

Existem diversas aplicações para gramáticas de grafos apontadas por Ribeiro (2000), como por exemplo, sistemas concorrentes, linguagens visuais, reescrita de termos, programação em lógica, reconhecimento e geração de imagens, biologia, dentre outros. Para mais definições formais de Gramáticas de Grafos e exemplos o leitor interessado pode consultar (EHRIG, 1979; 1997; 1999; ROZENBERG, 1997; ENGELS; SCHÜRR, 1995).

5.1.5 Especificação de Linguagens Visuais

Para desenvolver uma linguagem visual é necessária inicialmente uma especificação desta linguagem. Devido ao caráter multi-dimensional de linguagens visuais, suas especificações textuais são normalmente difíceis de escrever e entender. Por isso, especificações visuais de linguagens visuais têm sido cada vez mais utilizadas. Gramáticas de grafos, apresentadas na seção anterior, constituem um mecanismo adequado para definir uma linguagem visual.

Apesar da importância deste tópico, a maioria das PMLs não fornece sintaxe e semântica formais. Assim, aspectos críticos da execução de processos são descritos informalmente, baseando apenas em descrições textuais ou especificações semi-formais, ou estão descritos somente em linguagens de programação de baixo nível.

Uma técnica recente para especificar linguagens visuais foi proposta por Bardohl (1999; 2000). Nesta abordagem, chamada GENGED, uma linguagem visual é especificada por um alfabeto e uma gramática onde as sintaxes abstrata e concreta são distintas. De acordo com Bardohl, o uso de gramáticas de grafos provê um formalismo mais natural e visual para a especificação de linguagens visuais.

Segundo Bardohl (2000), existem dois níveis de descrição de linguagens visuais: o nível de sintaxe abstrata, que descreve o significado lógico das sentenças e o nível de sintaxe concreta, que é usado para descrever o *layout* das sentenças. A combinação dos

dois níveis é chamada Sintaxe Visual. A sintaxe abstrata geralmente corresponde um grafo tipo (ou *graph schema*) onde os vértices são os elementos do alfabeto da linguagem e os arcos denotam o relacionamento entre eles. Para complementar a sintaxe visual devem ser definidas regras para especificar como são construídas as sentenças visuais e, quando necessário, sua semântica.

5.2 Especificação dos Tipos de Dados do modelo APSEE

O meta-modelo APSEE estabelece um conjunto de serviços para a gerência automatizada de processos de software e foi especificado usando o formalismo algébrico (já introduzido na seção 5.1.3) do ambiente PROSOFT. Estes tipos correspondem aos tipos já apresentados informalmente no capítulo 3. Porém, aqui eles são representados através da notação do PROSOFT, onde as classes são criadas a partir da composição de outros tipos de dados.

A classe APSEE é a classe principal do meta-modelo e é apresentada na figura 5.4 através da composição de vários elementos envolvidos na gerência de processos de software componentes da arquitetura APSEE: 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;
- *APSEE-Types*: armazena as hierarquias de tipos para componentes APSEE;
- *Organization*: descreve os agentes, cargos e recursos existentes em uma organização específica;
- *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.

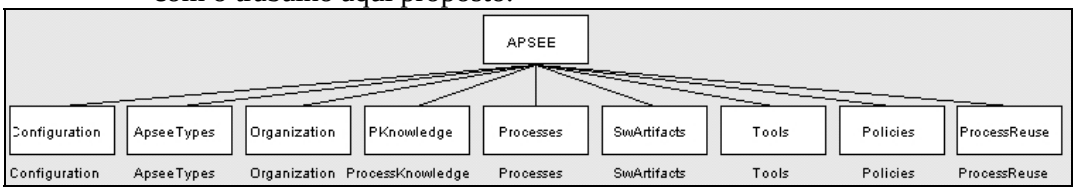


Figura 5.4: Classe APSEE no Prosoft algébrico.

Como se pode observar, com exceção do componente *Configuration*, que serve para guardar informações sobre o funcionamento do sistema, todos os componentes formam pacotes que já foram apresentados no capítulo 3, e para os quais já foi justificada a

necessidade de seus atributos e relacionamentos. Portanto, as classes correspondentes a estes pacotes são apresentadas detalhadamente no APÊNDICE A.

5.3 Especificação da Linguagem de Modelagem APSEE-PML

A especificação da linguagem de modelagem APSEE-PML foi inspirada na abordagem GENGED (BARDOHL, 2000) já citada na seção 5.1.5. Para a linguagem sendo proposta, optou-se por definir um grafo-tipo que define a sintaxe abstrata semelhante à sintaxe concreta, pois os vértices possuem sua representação final da linguagem, ou seja, usa os mesmos símbolos propostos na APSEE-PML. Isso facilitou a especificação uma vez que a proposta de Bardohl prevê o uso do ambiente GENGED para a especificação da sintaxe concreta e o nosso objetivo é a implementação no ambiente PROSOFT. Além disso, os tipos de dados propostos na linguagem são especificados algebricamente no paradigma PROSOFT, e já definem os seus relacionamentos. Com isso, houve uma combinação de métodos de especificação formal que facilitou o projeto da linguagem e do mecanismo de execução. Para complementar a sintaxe visual, foram construídas regras de gramática de grafos compatíveis com o grafo tipo e que serão usadas na definição (apresentadas nesta seção) e execução de processos no APSEE (a serem apresentadas na seção 5.4).

5.3.1 Alfabeto da APSEE-PML

Os símbolos visuais do alfabeto da APSEE-PML são mostrados na figura 5.5. Esses símbolos são modelados como vértices do grafo-tipo.

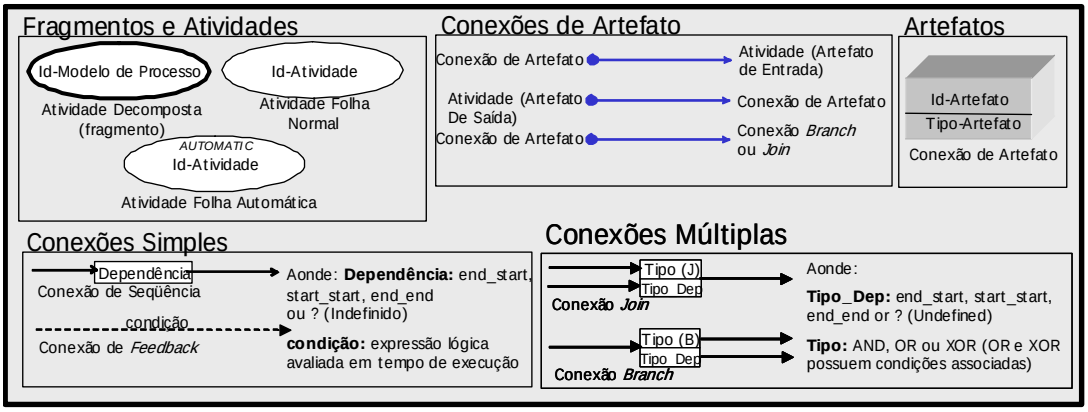


Figura 5.5: Símbolos visuais do alfabeto da APSEE-PML.

5.3.2 Grafo-Tipo

O grafo-tipo da APSEE-PML forma um diagrama que conecta vários componentes da linguagem. A figura 5.6 mostra o diagrama construído para definir a sintaxe da linguagem e em seguida serão mostradas e explicadas partes do mesmo diagrama para facilitar o entendimento. Os símbolos do grafo-tipo são os mesmos apresentados na figura 5.5. Deve-se observar que alguns tipos de dados que aparecem no grafo-tipo não ficam disponíveis visualmente para o usuário da linguagem, ou seja, não possuem

representação gráfica, tais como *Process*, *Process Model*, *Agent*, *Group*, *Agenda* e *Resources*. Estes vértices foram incluídos no grafo-tipo porque realizam importantes tarefas "de bastidores" relacionadas à execução de processos.

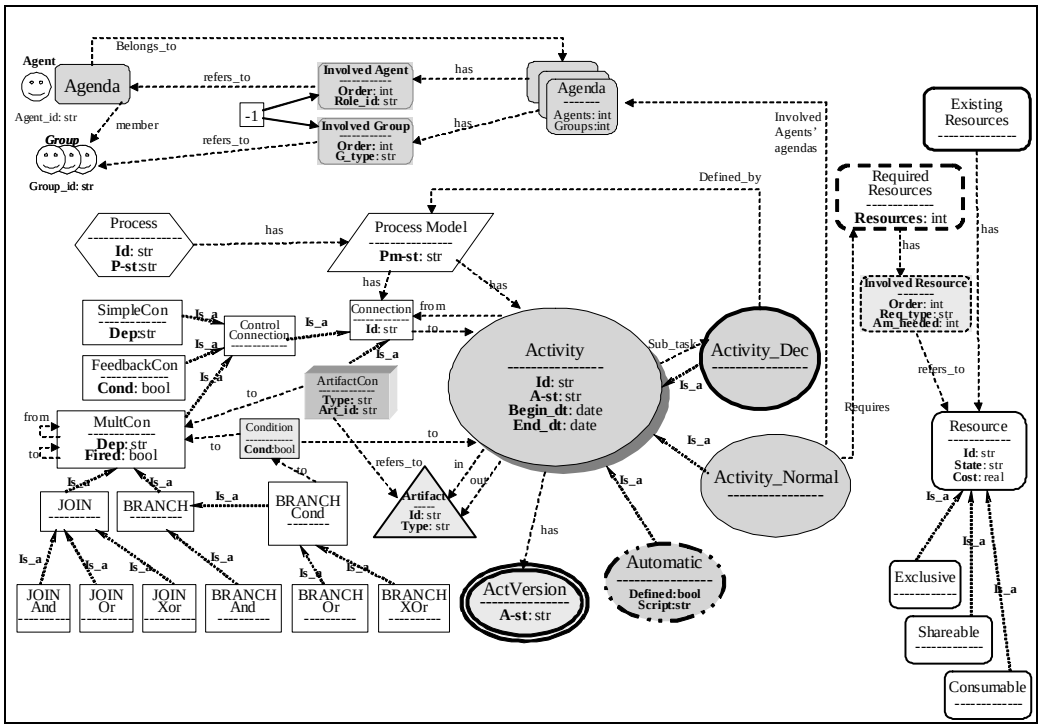


Figura 5.6: Grafo-tipo da APSEE-PML.

Cada símbolo está associado a um nome e um conjunto de atributos (separados por uma linha do nome do símbolo). Atributos de dados são elementos tipados usados para representar e raciocinar sobre o estado do processo. As setas do diagrama representam os arcos do grafo com significados específicos, a saber: setas pontilhadas representam relacionamentos[22], enquanto setas sólidas representam chamadas de mensagens (usadas na parte de interação do grafo apresentada posteriormente). Devido às várias possibilidades de chamadas de mensagens, as setas específicas com chamadas de mensagens foram substituídas por uma mais geral que as representa e que são detalhadas posteriormente.

A parte principal do grafo-tipo é apresentada na figura 5.7, onde um processo de software possui um modelo de processo composto por atividades e conexões. O atributo *P-st* corresponde ao estado da execução do processo e pode assumir os seguintes valores: *not-started* (não iniciado), *enacting* (executando) ou *finished* (concluído). O atributo *Pm-st* (do componente *Process Model*) está associado ao estado do processo no ciclo de vida de processos: *Null* (durante a fase de modelagem), *Requirements* (quando existe apenas uma descrição textual para o processo), *Abstract* (o processo não está

[22] Diferentemente da abordagem adotada por Bardohl (2000), a direção das setas aqui não indica dependência de um vértice sobre outro. Indica tão somente uma associação entre os dois tipos representados pelos seus vértices.

instanciado), *Instantiated* (o processo está pronto para execução porque possui componentes instanciados), *Enacting*, *Finished* ou *Mixed* (quando o processo é composto por atividades em diferente estados)[23]. Enquanto os estados possíveis para atividades (atributo *A-st*) são: *Null* (na fase de modelagem), *Waiting* (quando as dependencies não estão satisfeitas), *Ready* (pronta para executar), *Active* (em execução), *Paused*, *Finished*, *Cancelled* (a atividade foi cancelada antes de iniciar), e *Failed* (a atividade falhou após início da execução). As setas de relacionamento entre o tipo *Activity* e *Connection* indicam que uma conexão pode ter origem e destino em instâncias do tipo atividade.

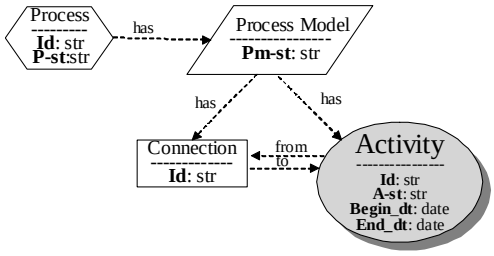


Figura 5.7: Parte do grafo-tipo relacionada a composição de processos de software.

A parte relacionada a conexões do grafo tipo é apresentada na figura 5.8, onde o relacionamento ***Is-A*** pode ser compreendido como herança, similar ao uso de herança em linguagens de programação orientadas a objetos. O relacionamento ***Is-A*** simplifica bastante o grafo-tipo e as regras de transformação correspondentes, já que atributos e relacionamentos são herdados aos subtipos. Essa abordagem é similar à usada por Gruner (2000) para meta-tipos em gramáticas de grafos. Por exemplo, quando o lado esquerdo de uma regra contiver uma referência para o tipo principal *Activity*, então essa referência pode ser substituída por quaisquer dos subtipos associados a *Activity*. Comportamento análogo acontece com o vértice que representa conexões múltiplas (*MultiCon*) e seus subtipos (*Join* e *Branch*)

Ainda na figura 5.8, os arcos **From** e **To** descrevem a origem e o destino da conexão. O relacionamento **To** entre conexão e atividade é redefinido para conexões múltiplas condicionais to tipo *Branch* (*Branch Or* e *Branch XOR*), já que essas conexões requerem uma condição lógica adicional para cada sucessor. Outros atributos importantes são: o tipo de dependência para conexões simples e múltiplas (*end-start*, *start-start*, *end-end*) e a condição lógica para conexões de *feedback* e *Branch-Cond*. É importante notar que conexões de controle podem ser: conexões simples (*SimpleCon*), conexões de *feedback* (*FeedbackCon*) e conexões múltiplas (*MultCon*).

[23] Estes estados já foram descritos informalmente na seção 3.7.1.

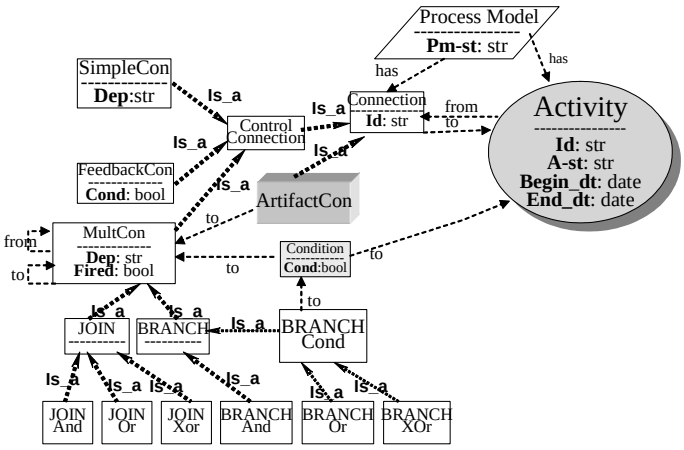


Figura 5.8: Parte do grafo-tipo relacionada a conexões.

A parte referente à definição de atividades no grafo-tipo é mostrada na figura 5.9. O relacionamento *Is-A* é usado para herdar atributos e relacionamentos do vértice *Activity*. No grafo-tipo, os símbolos que representam os tipos de atividades são diferenciados para facilitar o desenvolvimento das regras de gramática de grafos. A atividade decomposta (*Activity-Dec*) é definida por um modelo de processo que por sua vez possui conexões e atividades, porém esse relacionamento é refinado transitivamente pelo relacionamento *Sub-Task*. O vértice *ActVersion*, por sua vez, representa as versões de atividades criadas pelo sistema quando a atividade tem que ser re-executada ou o usuário solicita a criação da versão.

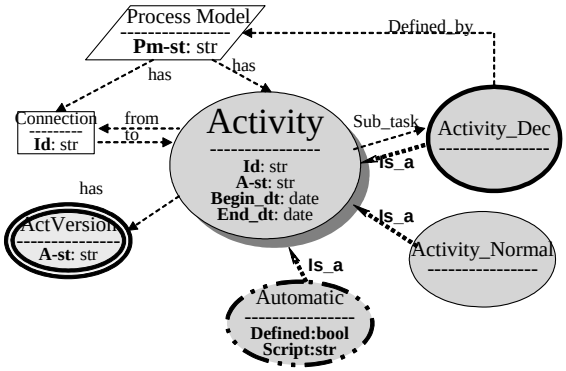


Figura 5.9: Parte do grafo-tipo relacionada a atividades.

Somente as atividades normais (*Activity_Normal*) possuem uma ligação com agentes e recursos. Assim, na figura 5.10 é apresentada a ligação de uma atividade normal com agentes. Cada atividade possui uma ligação com o vértice *Agenda*. Este, por sua vez, possui atributos que identificam a quantidade de pessoas e grupos trabalhando na atividade. A agenda referencia agentes e grupos. Através desses relacionamentos é possível identificar os agentes que trabalham para uma atividade.

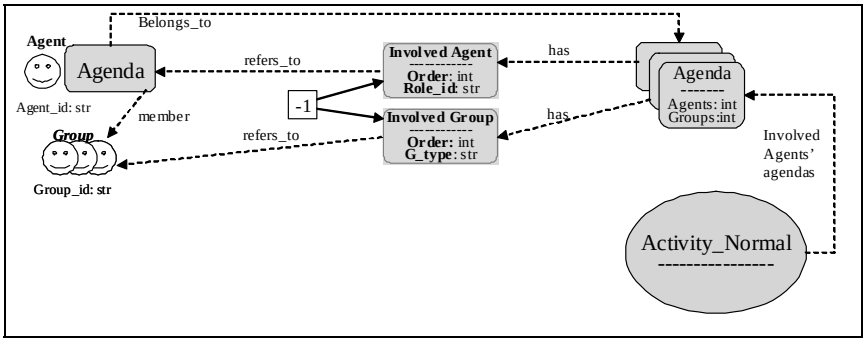


Figura 5.10: Parte do grafo-tipo relacionada a envolvimento com agentes.

A figura 5.11 mostra a parte do grafo-tipo (reposicionada) referente à ligação de atividades normais com recursos. Essa representação corresponde ao meta-modelo apresentado no capítulo 3. Uma atividade normal requer recursos que podem ser exclusivos, compartilháveis ou consumíveis. Essa representação não fica graficamente visível para o usuário da linguagem, mas é necessária para descrever a alocação de recursos.

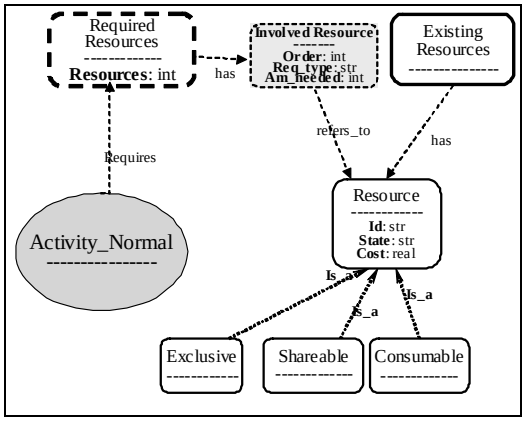


Figura 5.11: Parte do grafo-tipo relacionada a envolvimento com recursos.

5.3.3 Regras para definição de processos na APSEE-PML

Após a definição do grafo-tipo, é necessário declarar como criar sentenças válidas para a linguagem proposta. Na abordagem GENGED (BARDOHL, 2000) é proposta a criação de uma gramática visual para gerar as sentenças da linguagem e um conjunto de regras de comportamento para definir as transformações que modificam o sistema e os estados do modelo. A gramática visual da APSEE-PML é responsável por gerar instâncias de modelos de processos e também garantir sua consistência.

Foram descritas 114 regras para a gramática que gera sentenças da linguagem (outras regras para execução e modificação dinâmica serão mostradas nas próximas seções). Em muitos casos, foram usadas condições negativas de aplicação (NAC - *Negative Application Conditions*) para descrever condições que devem ser negativas para que a regra seja habilitada. Neste caso as partes cortadas no lado esquerdo da regra formam as condições negativas e não devem estar presentes na instância do grafo para

que a regra seja aplicada. Vale ressaltar também que as regras sempre apresentam componentes compatíveis com o grafo-tipo.

A figura figura 5.12 mostra a regra para inclusão de uma atividade em um processo. Dada a solicitação do usuário (*newActivity(new_id)*), o lado esquerdo da regra mostra a existência de um processo não concluído (*P-st = not_started* ou *enacting*) que possui um modelo de processo, que por sua vez, não pode conter uma atividade com o identificador igual ao que se quer criar. O lado direito da regra mostra que o processo e o modelo de processo se mantiveram, e a atividade foi criada com tipo *Activity_Normal* com o identificador *new_id*, assim como foram criadas ligações com agenda de agentes envolvidos e recursos requeridos.

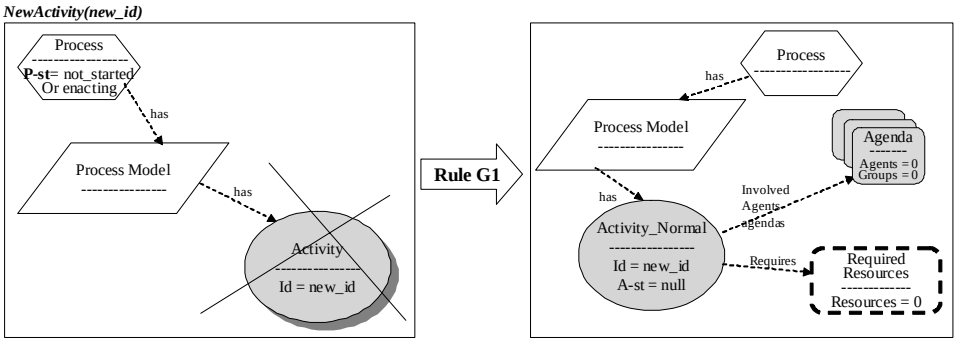


Figura 5.12: Regra para inclusão de nova atividade em um processo.

A figura 5.13 mostra um exemplo de regra para inserir uma conexão simples entre duas atividades (que podem ser automáticas, decompostas ou normais porque o símbolo *Activity*, mais geral, foi usado na regra). Na figura, o lado esquerdo da regra contém uma situação de um processo existente onde existem duas atividades *act_id1* e *act_id2* e acima da figura pode ser observada uma chamada que corresponde à solicitação do usuário. A regra da figura é disparada se o usuário solicita a inclusão de uma conexão simples entre duas atividades existentes, com uma determinada dependência. Para prevenir a inclusão de ciclos (que causariam *deadlock*), uma NAC define que a atividade destino não pode ter um fluxo de controle[24] direto ou indireto para a atividade origem, enquanto uma NAC adicional assume que não existe nenhuma conexão de controle entre a origem e o destino (que tornaria desnecessária a inclusão de uma nova). Como resultado, se a regra for aplicada, as atividades permanecem no mesmo estado e é acrescentada uma conexão simples entre as mesmas. Vale notar que essa regra se aplica a atividades no estado *null*, portanto, não estão executando ainda.

[24] Regras de detecção de fluxo de controle são detalhadas no Apêndice B. Informalmente, elas buscam a ocorrência de um caminho direto ou indireto entre atividades, através de conexões simples ou múltiplas.

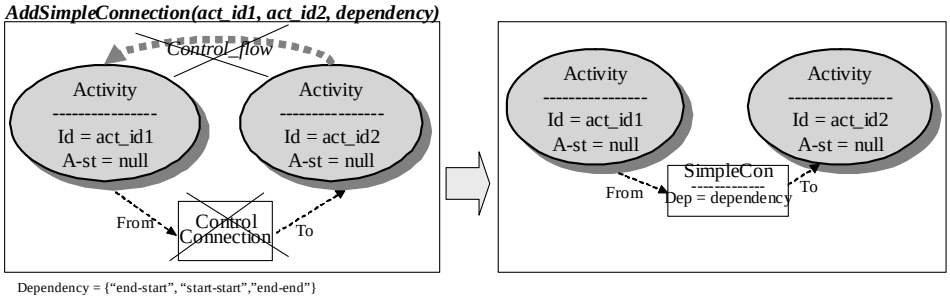


Figura 5.13: Regra para inclusão de uma conexão simples entre atividades.

Uma outra regra que ilustra a estratégia adotada no trabalho é apresentada na figura 5.14. O objetivo da regra é: dada uma atividade qualquer de um processo e uma conexão não disparada (*fired* = false) do tipo JOIN XOR, definir que a atividade é origem do Join. A regra demonstra visualmente que algumas condições são necessárias para que seja estabelecida essa ligação: a atividade não pode ter uma ligação do tipo From pré-existente com a conexão; não pode existir fluxo de controle da conexão para a atividade (para evitar ciclos) e a atividade não pode ser destino de uma conexão *Branch AND* (pois como mostrado na seção 4.1.3, isso causaria um conflito de *deadlock*). Além disso os estados da atividade e da conexão são determinados no lado esquerdo da regra (outras regras tratam a mesma solicitação para estados diferentes e estão apresentadas no apêndice b).

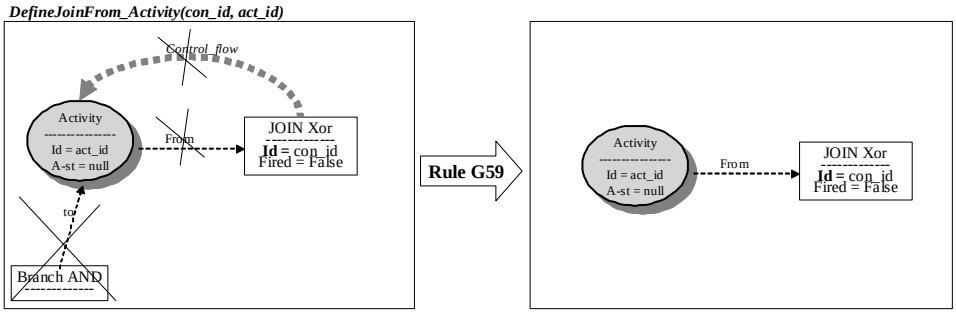


Figura 5.14: Regra para definir uma atividade como origem de conexão Join.

5.4 Semântica da Execução de Processos no APSEE

Nesta seção são apresentadas as regras desenvolvidas para dar semântica à execução de processos de software no APSEE.

5.4.1 Regras para definir a execução de processos

Foram criadas regras para definir o comportamento do processo de software durante a execução. Como se trata de um sistema interativo, as ações dos agentes e do gerente participam da semântica como geradores de eventos que desencadeiam transformações. Além disso, como a execução envolve transformações encadeadas, foi necessário utilizar nas regras alguns símbolos visuais sinalizadores que servem para identificar situações do processo. Esses símbolos adicionais foram acrescentados ao grafo-tipo, mas seu funcionamento é o mesmo de um atributo de um vértice, e não foi modelado

como tal para que pudéssemos distinguir os atributos essenciais do meta-modelo de atributos sinalizadores.

O grafo-tipo apresentado na seção anterior foi aperfeiçoado para conter todos os símbolos necessários às regras de transformação. A figura 5.15 mostra o grafo-tipo completo, onde podem ser identificados símbolos para representar o gerente (*manager*), o mecanismo de execução (*APSEE-Manager*), Eventos (*events*) e alguns símbolos ligados a atividades e conexões para identificar situações específicas da execução.

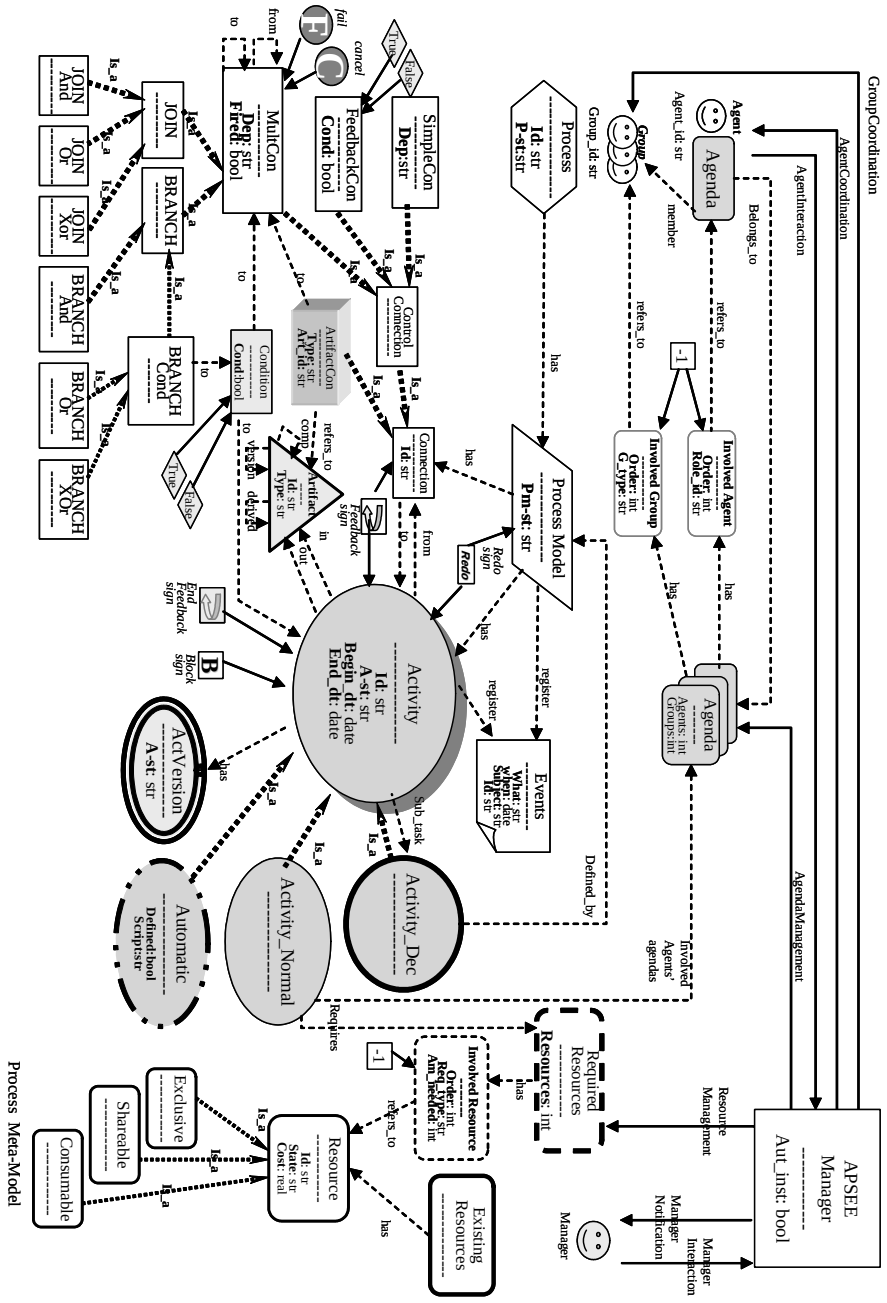


Figura 5.15: Grafo-tipo do modelo APSEE usado para semântica da execução.

A figura 5.16 mostra os tipos de cada atributo pertencente a vértices do grafo-tipo, enquanto a figura figura 5.17 mostra o significado de alguns símbolos adicionais do grafo-tipo.

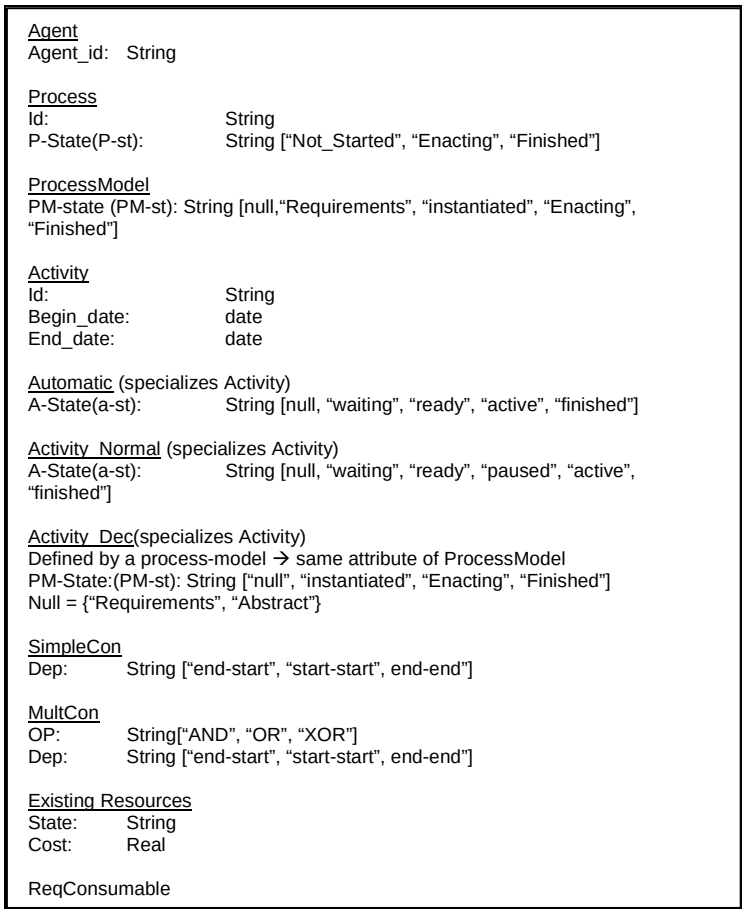


Figura 5.16: Detalhamento dos atributos do grafo-tipo.

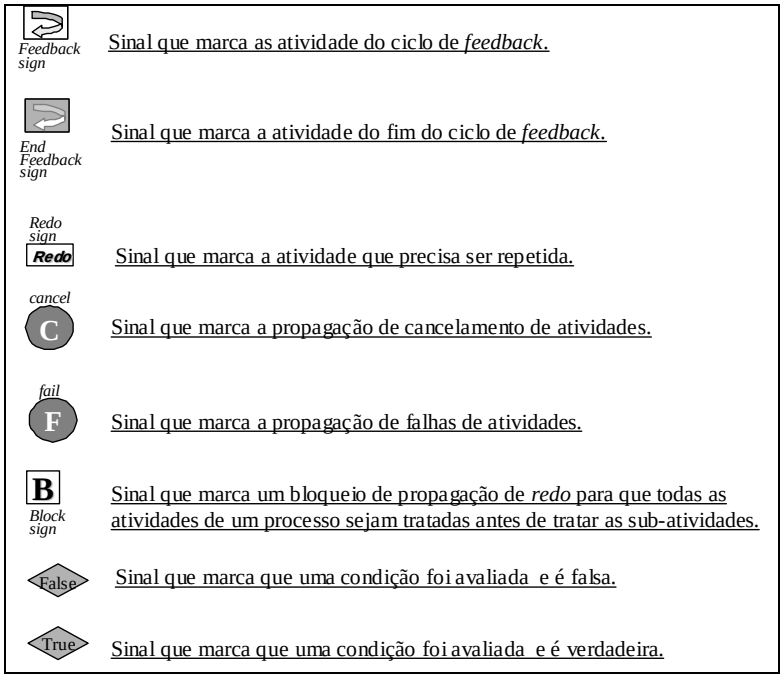


Figura 5.17: Símbolos adicionais do grafo-tipo e seus significados.

Do ponto de vista de sincronização de processos e atividades, a estratégia de execução leva em consideração: as atividades e suas sub-atividades; as dependências das conexões; a disponibilidade de recursos e as pré e pós condições de atividades. O início da execução de um processo é solicitado pelo gerente. A figura 5.18 mostra a regra correspondente. A partir desse evento, várias transformações são realizadas.

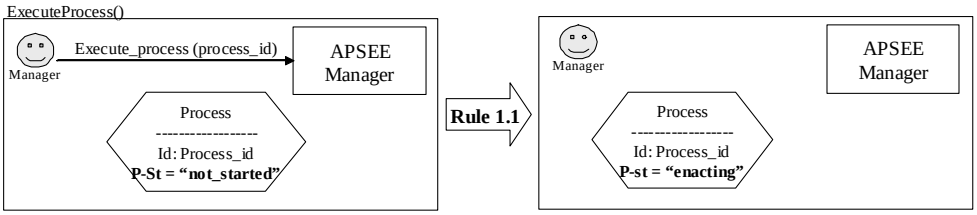


Figura 5.18: Início da execução de processos - alteração do estado do processo.

Existe um conjunto de regras para manter o estado do modelo de processo consistente com os seus componentes. Essas regras são executadas sem intervenção e seu objetivo é atualizar o estado do modelo de processo para que a execução possa continuar. Apenas quando o modelo de processo está instanciado ele pode ser executado, portanto é importante detectar essa situação. A figura 5.19 mostra algumas regras (o restante encontra-se no Apêndice C) que detectam essa situação. A primeira define que se o modelo de processo (*Process Model*) está em um estado diferente de *instantiated* e possui alguma atividade decomposta no estado *instantiated* e nenhuma atividade no estado *enacting*, então seu estado é *instantiated*. A segunda regra da figura indica que um processo está instanciado se existe pelo menos uma atividade com agente definido. Enquanto a terceira define que se existe alguma atividade automática definida, isso também leva o estado do processo para instanciado.

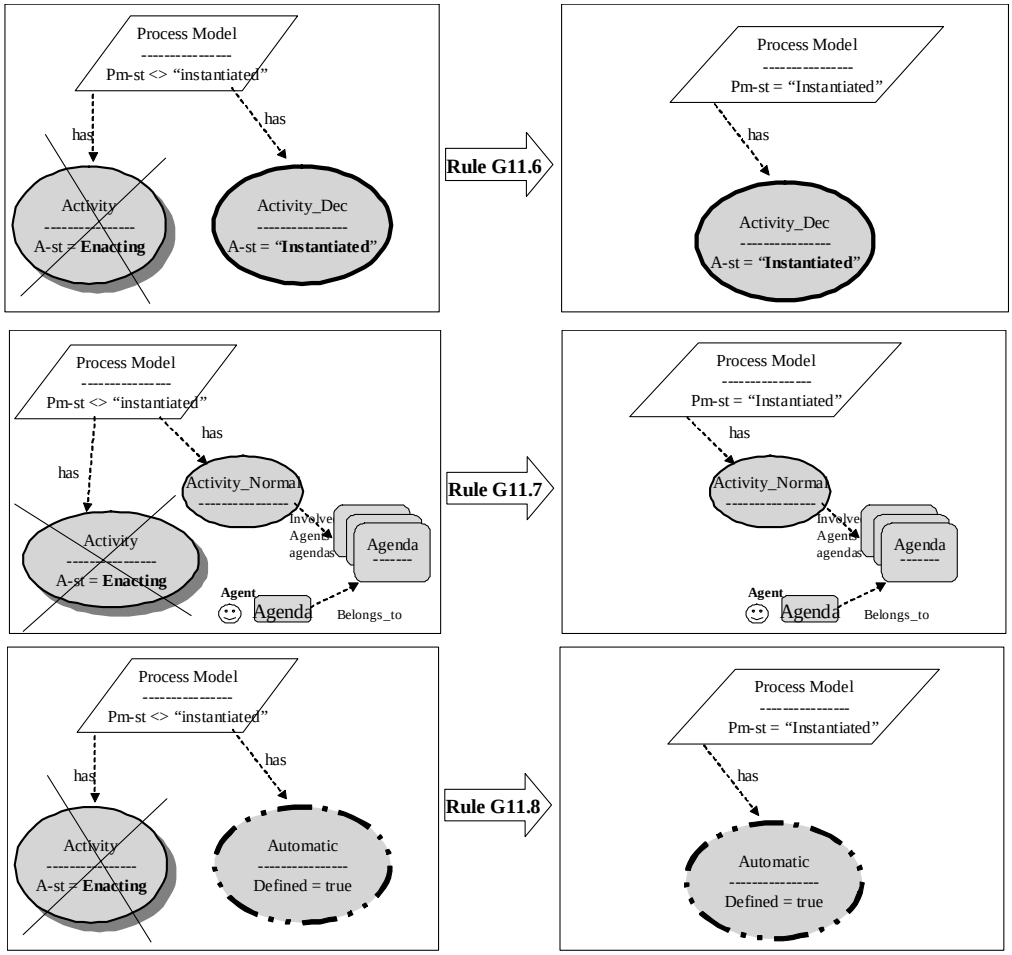


Figura 5.19: Regras para definição automática do estado do modelo de processo.

Devido às conseqüências das regras da figura 5.19 sobre modelos de processo, regras adicionais podem ser ativadas para transformar o estado das atividades. A figura 5.21 mostra duas das regras que definem esse estado. A primeira estabelece que se o processo está *enacting* e o modelo de processo está *instantiated* ou *enacting* e possui uma atividade normal sem estado definido que possua agentes (está instanciada), então a atividade torna-se *waiting* e a agenda dos agentes é atualizada pelo *APSEE-Manager*. O evento ocorrido é registrado através da criação do item *Events*. Deve-se notar que embaixo da seta com a identificação da regra foi acrescentada uma condição lógica de guarda para que houvesse a transformação (*Ag > 0 or GR > 0*, onde *Ag* é o número de agentes na agenda e *Gr* o número de grupos de agentes). A regra seguinte da mesma figura tem o mesmo objetivo, mas trabalha com as sub-atividades.

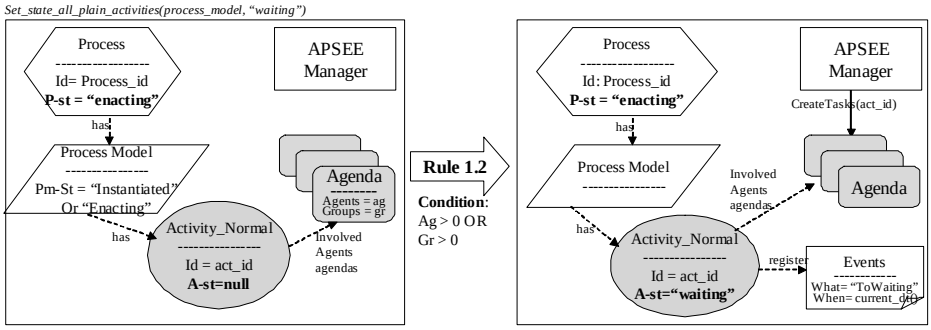


Figura 5.20: Regra do início da execução para definir estado das atividades.

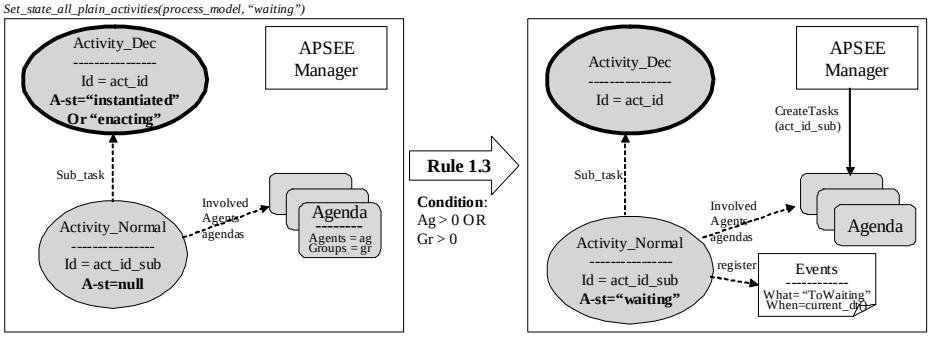


Figura 5.21: Regra do início da execução para definir estado das atividades.

A figura 5.22 e a figura 5.23 apresentam a regra que define como é feita a transição de *waiting* para *ready* em uma atividade normal considerando as dependências estabelecidas pelas 8 NACs[25] da figura 5.22. Por exemplo, para que a atividade mude para *ready* ela não pode depender com conexão simples *end-start* de uma atividade que ainda não terminou (NAC 08 da figura figura 5.22). O restante das regras de transição de estados está no Apêndice C.

[25] A apresentação das NACs em separado tem o mesmo significado de serem apresentadas cortadas no lado esquerdo da regra.

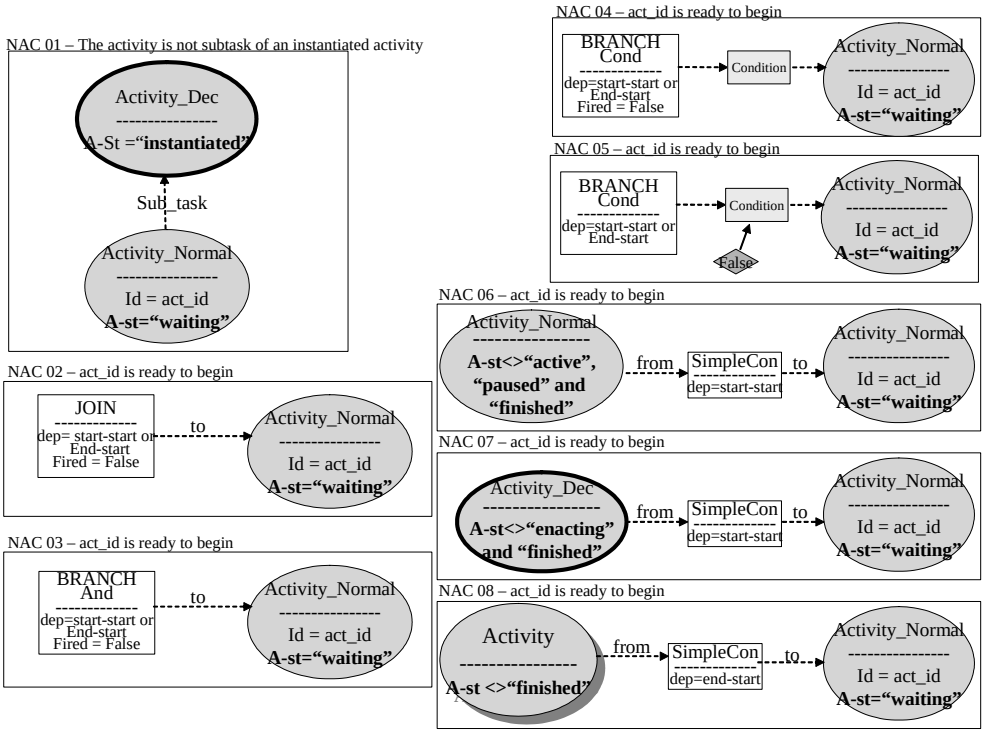


Figura 5.22: NACs para a regra *SearchForReadyActivities* da próxima figura.

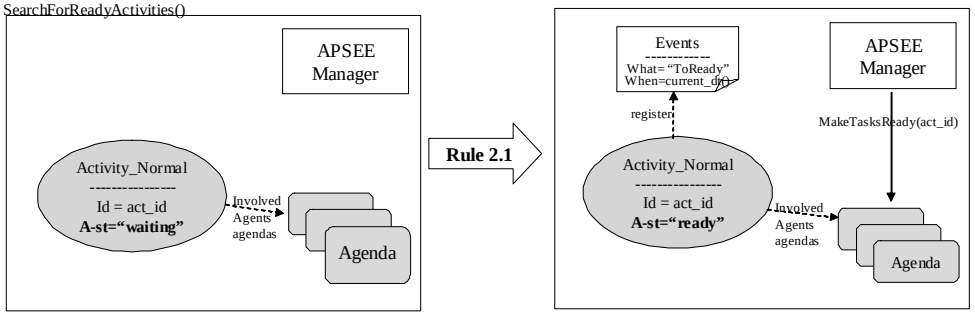


Figura 5.23: Regra para transição de estados de uma atividade normal (NACs na figura 5.22).

A partir do momento em que uma atividade normal passa para o estado *ready*, os agentes podem dar início às mesmas e são responsáveis por sua evolução. A cada evento o *APSEE-Manager* procura atualizar o estado de todas as atividades. No Apêndice C são apresentadas as regras que foram desenvolvidas para definir o comportamento do sistema durante a execução (excetuando as de modificação dinâmica). As regras foram divididas nos seguintes grupos principais:

- 1 - Regras do início da execução do processo;
- 2 - Regras que procuram atividades prontas para executar;
- 3 - Regras que executam a partir da interação com o usuário;
- 4 - Regras que ajustam o estado de atividades decompostas (para refletir o estado geral das atividades componentes);

- 5 - Regras que propagam falhas entre atividades. Quando o gerente interrompe uma atividade essas regras definem as conseqüências no processo;
- 6 - Regras que propagam cancelamento entre atividades;
- 7 - Regras que tratam conexões *Branch*. Essas regras definem basicamente como ficam as atividades/conexões posteriores a um *Branch* que teve suas dependências satisfeitas (*Fired = True*);
- 8 - Regras que tratam ativação de conexões múltiplas (definem quando o atributo *fired* será igual a *true* tanto em *Branchs* quanto em *Joins*);
- 9 - Regras que tratam propagação de *Feedback*. Essas regras definem como o processo repete atividades de acordo com a definição da conexão de *feedback*, criando novas versões para atividades a serem repetidas e redistribuindo essas atividades para os agentes;
- 10 - Regras para instanciação de processos durante a execução. Essas regras são ativadas automaticamente (se o usuário tiver configurado o sistema para tal) e chamam a ferramenta de instanciação para definir agentes e recursos adequados a atividades prestes a iniciar;
- 11 - Regras para manutenção do estado do modelo de processo. Essas regras foram citadas nesta seção e algumas delas foram apresentadas na figura Figura 5.19.

5.4.2 Regras para Modificação Dinâmica durante execução

A forma adotada para tratar a modificação dinâmica neste trabalho é baseada na manutenção da consistência do processo. Se a mudança for possível e não influenciar no andamento do processo então é realizada; porém se a mudança for possível mas indicar necessidade de outras ações automáticas para compensar seus efeitos, então a mudança é realizada e essas ações são realizadas. Finalmente, se a mudança causar problemas irreversíveis e não puder ser compensada então ela não é permitida. Por isso as regras para tratar mudanças são extensões às regras de sintaxe (criação de processo) levando em consideração o estado da execução e definindo quais ações devem ser realizadas automaticamente para compensar a mudança.

As regras da figura 5.24 a seguir são extensões à regra da figura 5.13 para inclusão de uma conexão simples entre duas atividades (existem mais regras com esse mesmo objetivo no Apêndice B). A diferença está no tratamento dado aos estados das atividades envolvidas. Nesse caso, só existe a possibilidade de inclusão da conexão se as atividades estiverem satisfazendo as condições de estados do lado esquerdo das regras.

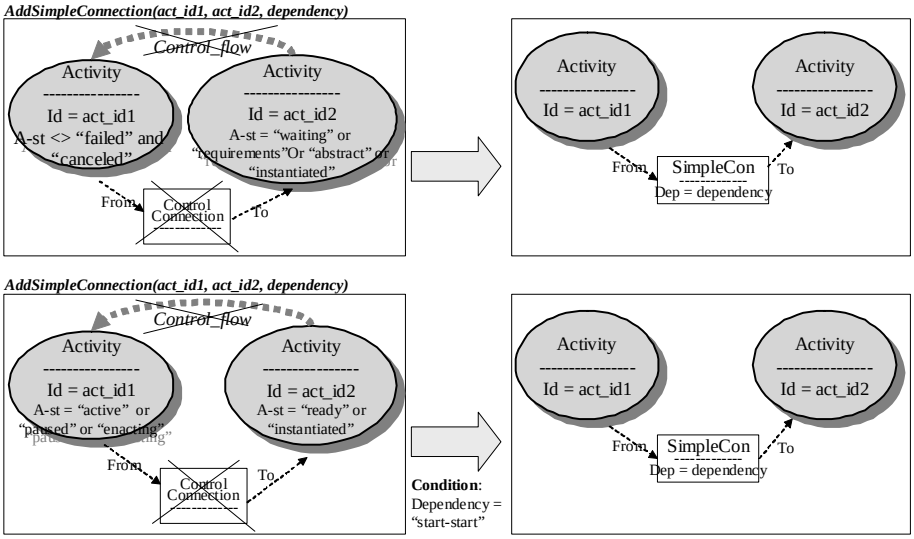


Figura 5.24: Regras adicionais para tratar a inclusão de conexões durante a execução.

Um exemplo de regra que permite mudança mas toma medidas adicionais é apresentada na figura 5.25. Essa regra é ativada quando o usuário deseja definir que uma atividade é destino de uma conexão *Join* (para a qual ainda não foi definido o destino), sendo que a conexão possui dependência *end-start* ou *start-start* ainda não satisfeita (*fired = false*) e a atividade está no estado *ready*. Como conseqüência, a atividade volta para o estado *waiting* e o mecanismo de execução altera a agenda dos agentes de acordo com essa mudança.

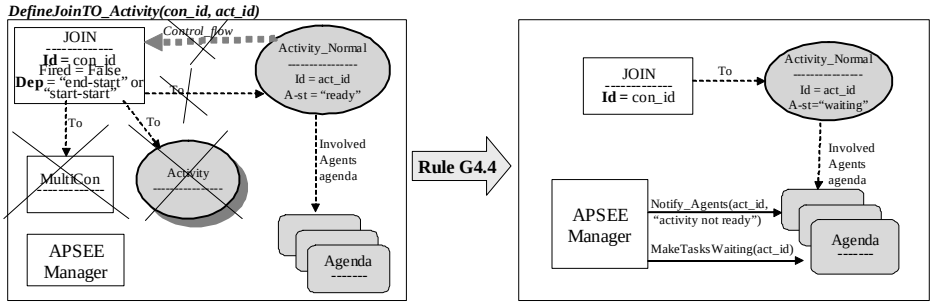


Figura 5.25: Regra para inclusão de atividade como destino de *Join* durante execução.

Assim como a regra mostrada, existem outras regras para atender as várias possibilidades de edição do processo durante a execução. Essas regras foram definidas em grupos de regras para lidar com a edição de processos. Todos os grupos são apresentados no Apêndice B e são listados abaixo:

- 1 - Regras para definição de atividades;
- 2 - Regras para definição de conexões de artefatos;
- 3 - Regras para definição de conexões de controle;
- 4 - Regras para definição de conexões *Join*;
- 5 - Regras para definição de conexões *Branch*;
- 6 - Regras para definição de cargos, agentes e grupos em atividades;

- 7 - Regras para definição de recursos em atividades;
- 8 - Regras para teste de fluxo de controle entre atividades e conexões múltiplas

5.5 Especificação da Interpretação de Políticas de Instanciação

Esta seção apresenta alguns dos principais tipos de dados criados para o armazenamento e processamento de Políticas de Instanciação, assim como as operações algébricas mais importantes para instanciação de recursos. A instanciação de agentes foi especificada de forma análoga, e por isso não é mostrada nesse texto.

5.5.1 Tipos de dados para representação de políticas de instanciação

A definição de Política de Instanciação de Recursos é realizada pelo usuário em um formulário apropriado no ambiente PROSOFT-APSEE. A partir das informações de entrada são gerados os objetos das classes PROSOFT descritas nesta seção.

As políticas envolvidas com gerência de processos no APSEE são definidas na classe *Policies*. Esta classe, por sua vez, compõe as políticas estáticas, dinâmicas e de instanciação (ver figura 5.26). A classe que contém as políticas de instanciação é a classe *InstantPolicies*, mostrada na mesma figura. As políticas para instanciação de recursos são armazenadas na classe *InstResources*, mostrada na figura 5.27.

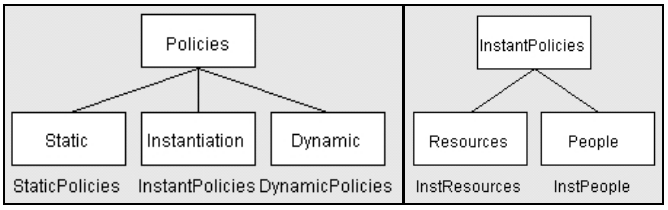


Figura 5.26: Classe *Policies* e Classe *InstantPolicies*.

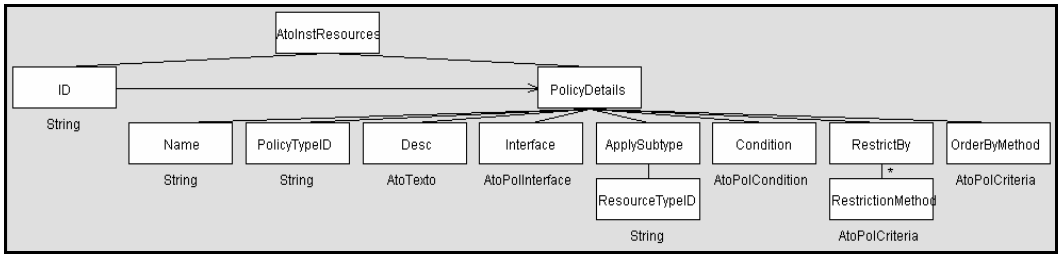


Figura 5.27: Classe *InstResources* – Instanciação de recursos

Os componentes da classe *InstResources* refletem a gramática mostrada na seção 4.2. Cada política de instanciação de recursos é um elemento do mapeamento com identificador único e detalhes. A interface da política de instanciação é definida da mesma forma que em políticas estáticas (REIS et al., 2002b; REIS, 2002f), através e um objeto da classe *PolInterface*, mostrada na figura 5.28.

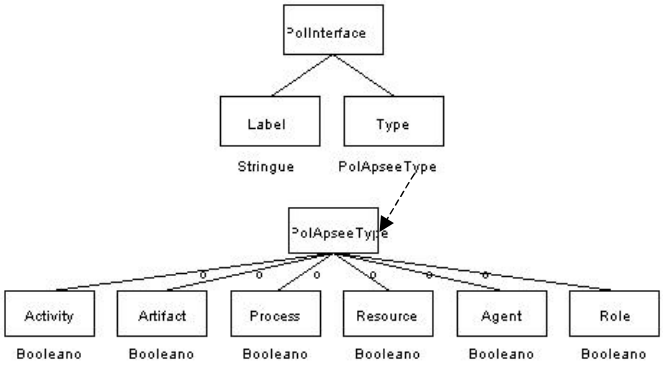


Figura 5.28: Classe *PolInterface* para definição da interface da política (extraída de (REIS, 2002f)).

5.5.2 Tipos de Dados para armazenar o resultado da instanciação

O resultado da avaliação das políticas habilitadas para um fragmento de processo é fornecido através da classe *ResourcePolicyLog* mostrada na figura 5.29, juntamente com seus componentes. Neste resultado, para cada fragmento de processo são armazenadas as sugestões e a escolha de recursos para cada atividade. A classe armazena informações sobre qual política foi usada para gerar sugestões e para cada item de sugestão, o usuário poderá saber qual o valor utilizado para ordenação (referente ao critério *OrderBy* da política). Esta informação adicional facilita a escolha de recursos por parte do usuário, quando os valores de ordenação são aproximados. As etapas para geração da instanciação mostradas anteriormente na seção 4.2.4 objetivam preencher apenas uma das sugestões de recurso de uma atividade-folha, ou seja, preenchem o objeto da classe *ResourceSuggestion*, assim como geram uma lista de sugestões – *List_Suggestion*. Para cada recurso requerido de uma atividade, as etapas devem ser repetidas até completar a lista mapeada pelo identificador da atividade em *ResourcePolicyLog*. Além disso o resultado de uma sugestão pode influenciar na próxima aplicação das etapas. Por exemplo, caso uma atividade requeira duas salas, o mecanismo de instanciação aplicará a mesma política para instanciá-las, porém as listas de sugestão para cada sala requerida serão diferentes. Esta característica evita problemas de conflito na sugestão de recursos.

Na figura 5.30 é mostrada a classe que armazena os possíveis usos de recursos. Ela é útil para guardar as primeiras sugestões de cada recurso da classe de resultados e permite o tratamento de conflitos de sugestões quando ocorre instanciação de um processo. Um recurso tem possibilidade de uso se ele foi o primeiro sugerido para alguma atividade, mesmo que ainda não tenha sido escolhido para a mesma.

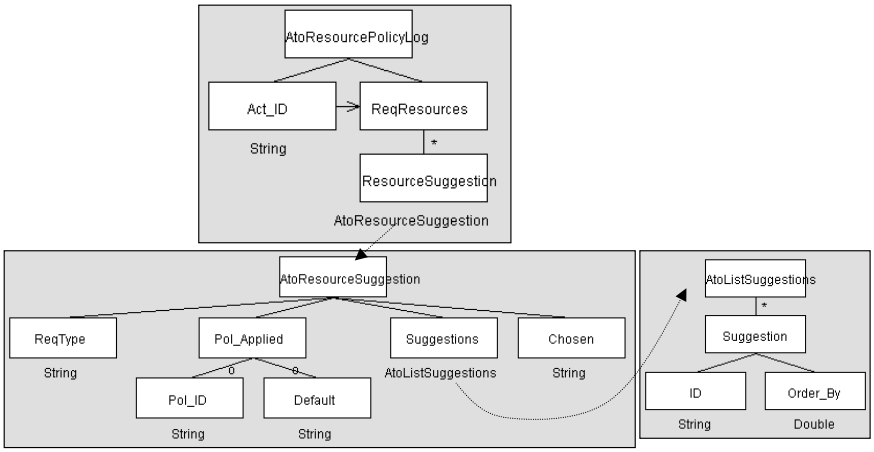


Figura 5.29: Classes *ResourcePolicyLog*, *ResourceSuggestion e ListSuggestions* – Resultado da avaliação das políticas de recursos.

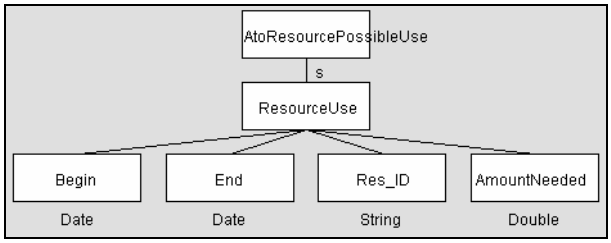


Figura 5.30: Classe *Resource_Possible_Use* - Possibilidades de uso de recursos.

Ambas as classes *ResourcePolicyLog* e *ResPossibleUse* estão integradas ao APSEE através do componente *Planner_Info* inserido na classe *Organization*. A classe *Planner_Info* é apresentada na figura 5.31.

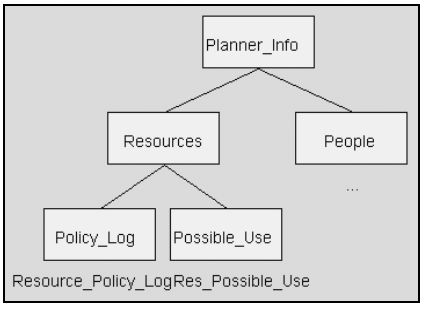


Figura 5.31: Classe *Planner_Info.*

A classe *Planner_Info* guarda as sugestões de instanciação e as possibilidades de uso dos recursos. Deve-se notar que foi omitida a parte de instanciação de pessoas nesta figura. Para que ocorra instanciação, é necessário que o usuário escolha um dos recursos sugeridos em *ResourceSuggestion* e atualize o processo de software com essa informação. Se o usuário preferir, a instanciação pode ser automática. O processo de obtenção das sugestões é o mesmo, gerando resultados na classe *ResourcePolicyLog*, mas um passo adicional é realizado no sentido de atualizar o processo/atividade com os recursos sugeridos durante a instanciação. Neste caso o primeiro recurso da lista de sugestões é escolhido.

5.5.3 Geração de Sugestões de Instanciação

No capítulo anterior, seção 4.2.4, foram apresentadas as etapas da geração de sugestões de instanciação. Esta seção tem como objetivo apresentar como foi especificada essa operação considerando os tipos de dados utilizados para armazenar esses resultados, mostrados na seção anterior.

Para o usuário final são disponibilizadas algumas operações principais de instanciação. Entretanto, a grande diversidade de informações existentes no meta-modelo APSEE e a necessidade de realizar instanciação a partir de políticas definidas pelo usuário levou à especificação de muitas operações auxiliares. Nesta seção serão apresentadas apenas as operações principais. Para completo entendimento dessas operações é necessário considerar as operações adicionais e auxiliares apresentadas no Apêndice D.

5.5.3.1 Instanciação de um Recurso

As etapas de geração de instanciação (seção 4.2.4) foram criadas para permitir a geração de uma sugestão de instanciação para um tipo de recurso de uma atividade particular. O objeto resultante da integração destas funções pertence à classe *ResourceSuggestion* mostrada na Figura 5.29. A operação que integra essas etapas e retorna um objeto do tipo *ResourceSuggestion* é especificada pela função *Generate_resource_suggestion*, apresentada na Figura 5.29. Antes de definir a sugestão é necessário definir qual política será utilizada para instanciar o recurso. A linha rotulada como {etapa 2} fornece a lista de políticas ordenadas e a função *select_policy* utiliza esta lista para testar a condição de cada política até que seja verdadeira, e definir essa política na classe *ResourceSuggestion*.

Ato APSEE

Generate_resource_suggestion: Apsee, String, String, String, Longue, Longue, Real → ResourceSuggestion
Generate_resource_suggestion(apsee, proc_id, act_id, req_type, begin, end, amount_needed)
= **ICS**(RESOURCESUGGESTION, adjust_conflicts,
ICS(RESOURCESUGGESTION, define_suggestion,
<*select_policy*(apsee, req_type,
get_ordered_instR_policies(apsee, req_type, proc_id, act_id), {etapa2}
proc_id, act_id),
apsee, proc_id, act_id,
initial_Resources_list(apsee, req_type, begin, end, amount_needed)>), {etapa1}
<begin, end, amount_needed, *get_ResPossibleUse*(apsee)>) {término do adjust_conflicts}
** a operação necessita do identificador da atividade (proc_id e act_id), do tipo requerido de recurso, do período da atividade (begin, end) e da quantidade necessária do recurso (caso consumível)

Select_policy: Apsee, String, listofString, String, String → ResourceSuggestion
Select_policy(apsee, req_type, list_pol, proc_id, act_id)
= **if** head(list_pol) <> "default"
then **if** *verify_conditions*(apsee, {etapa3}
get_condition_instRPolicy(apsee, head(list_pol)),
proc_id, act_id)
then **ICS**(RESOURCESUGGESTION, create_pol_applied,
<req_type, head(list_pol)>)
else*select_policy*(apsee, req_type, tail(list_pol), proc_id, act_id)
elseICS(RESOURCESUGGESTION, create_default, <req_type>)

** essa função recebe uma lista ordenada de políticas aplicáveis e verifica qual delas se aplica (condição verdadeira) a situação da atividade em questão. Se nenhuma política se aplicar, então é colocado o algoritmo default para ser aplicado depois. É retornado um objeto ResourceSuggestion com os atributos Req_Type e Pol_Applied preenchidos.

A operação *define_suggestion* recebe um objeto *ResourceSuggestion* com o *reqtype* e *Pol_applied* definidos e preenche o campo *ListSuggestions* através da aplicação do método de ordenação {etapa 5} sobre a restrição de recursos {etapa 4}.

Ato ResourceSuggestion

Define_Suggestion: ResourceSuggestion, Apsee, String, String, listofString → ResourceSuggestion

Define_suggestion ((ReqType req_type, Pol_Applied Pol_Id pol, _, _), Apsee, proc_id, act_id, initial_res_list)
= (ReqType req_type,
Pol_Applied Pol_Id pol,
Apply_orderBy_Method(apsee, {etapa5}
Restrict_resources(apsee, initial_res_list, {etapa4}
Get_restrictBy_methods_instRPolicy(apsee, pol)),
Get_orderBy_method_instRPolicy(apsee, pol)),_)

Define_suggestion ((ReqType req_type, Pol_Applied Default "default", _, _), apsee, proc_id, act_id, initial_res_list)
= (ReqType req_type,
Pol_Applied Default "default",
Apply_orderBy_Method(apsee, {etapa5}
Initial_res_list
ICS(POLCRITERIA, Low_Cost_method)),_)

** esta operação recebe um objeto do tipo ResourceSuggestion já existente e preenche apenas a lista de sugestões de acordo com o resultado da aplicação da política escolhida.
Se nenhuma política foi escolhida, então é aplicado o algoritmo default que simplesmente ordena os recursos disponíveis pelo critério de menor custo.

5.5.3.2 Instanciação de uma atividade folha - vários recursos

A instanciação de uma atividade folha parte da lista de recursos requeridos da atividade para gerar a lista de sugestões. Nesta fase todos os recursos de uma atividade folha terão sugestões no *ResourcePolicyLog*, com exceção dos recursos que o usuário já instanciou (já definiu o id do recurso a ser usado) ou não preencheu o tipo de recurso requerido. Para cada recurso requerido, são realizadas as operações mostradas nas seções anteriores para gerar uma lista de sugestões a partir da política aplicada. A cada passo desta função é necessário verificar se o primeiro recurso sendo sugerido está reservado (i.e., possível utilização) para o mesmo período. Por exemplo, se uma atividade requer dois computadores, ao gerar sugestões para o segundo computador, a sugestão feita para o primeiro é levada em consideração, para que não sejam feitas sugestões idênticas.

Para instanciar uma atividade folha, foi criada a operação principal do Ato APSEE chamada *Resource_instantiation_plain_activity*. O objeto resultante contém o *ResourcePolicyLog* atualizado assim como o *ResPossibleUse* (possibilidades de uso de recursos). Nenhuma alteração é feita na definição da atividade. O objetivo até este ponto é sugerir recursos. A operação principal chama a operação *detailed_instantiation* para inserir instanciações no *log* da atividade em cada posição de recurso requerido.

Ato APSEE

Resource_Instantiation_Plain_Activity: Apsee, String, String → Apsee

```
Resource_Instantiation_Plain_Activity (apsee, proc_id, act_id)
= detailed_instantiation(create_instR_Log_Activity(apsee,
        ICS(REQ_RESOURCES, length, get_req_resources(apsee, proc_id, act_id)),   {1}
        Proc_id, act_id,
        Get_begin_act(apsee, proc_id, act_id),
        Get_end_act(apsee, proc_id, act_id),
        Get_req_resources(apsee, proc_id, act_id),
        1)                                                                       {2}
```

{1} É acrescentado ao ResourcePolicyLog do Apsee uma ocorrência para o processo e atividade em questão e o número de sugestões a serem geradas é o tamanho do req_resources da atividade. O apsee enviado à função detailed_instantiation já inclui esta modificação.
{2} o número 1 enviado para a função detailed_instantiation inicializa a geração de sugestões para os recursos requeridos da atividade a partir da posição 1.

Detailed_Instantiation: apsee, String, String, longue, longue, Req_Resources, Inteiro → apsee

```
Detailed_instantiation(apsee, proc_id, act_id, begin, end, req_resources, position)
if    position > ICS(REQ_RESOURCES, length, req_resources)
then  apsee
else  if    ICS(REQ_RESOURCES, get_req_type, req_resources, <position>) = "" or
            ICS(REQ_RESOURCES, get_type_id, req_resources, <position>) <> ""
      Then  detailed_instantiation(apsee, proc_id, act_id,
            begin, end, req_resources, position+1)
      Else  detailed_instantiation(                                              {1}
              Update_ResPossibleUse(
                Update_resource_pol_log(
                  Apsee,
                  ICS(RESOURCEPOLICYLOG, Insert_res_suggestion,
                    get_resource_Pol_log(apsee),
                    <proc_id, act_id,
                    generate_resource_suggestion(apsee, proc_id, act_id,
                      ICS(REQ_RESOURCES,get_req_type,
                          req_resources, <position>),
                      Begin, end,
                      ICS(REQ_RESOURCES,get_amount, req_resources,<position>),
                      Position))>),          {fim de generate|insert|update_resource}
                Proc_id, act_id, begin, end,
                ICS(REQ_RESOURCES,get_amount, req_resources,<position>),
                Req_resources,
                Position),
              Proc_id, act_id, begin, end, req_resources, position+1)
```

{1} Chamada recursiva para a própria função enviando o apsee atualizado com relação à instanciação de um recurso e solicitando a instanciação da próxima posição (position+1) de req_resources.

5.5.3.3 Instanciação de um fragmento - várias atividades-folha

Um fragmento pode ser: um processo de software completo, existente na classe Processes do APSEE, indicado pelo seu proc_id; uma atividade decomposta de um processo; ou ainda um conjunto de identificadores de atividades-folha de um processo. Neste caso, o usuário pode requerer, por exemplo, a instanciação de apenas duas atividades de um fragmento que possui 5 atividades.

A operação utilizada para instanciar (sugerir instanciação) um fragmento no APSEE é *Resource_Instantiation_Fragment*. Existe uma definição da operação para cada tipo de fragmento que se deseja instanciar. Em todos os casos é gerado um conjunto que

contém os identificadores de atividades-folha que se quer instanciar e cada elemento desse conjunto é submetido à instanciação de atividade-folha mostrada na seção anterior.

Resource_Instantiation_Fragment: Apsee, String → Apsee
Resource_Instantiation_Fragment: Apsee, String, String → Apsee
Resource_Instantiation_Fragment: Apsee, String, SetofString → Apsee

Resource_Instantiation_fragment(apsee, proc_id)
= *Resource_Instantiation_fragment*(apsee, proc_id,
get_setOf_plain_acts(apsee, proc_id)) {1}

Resource_Instantiation_fragment(apsee, proc_id, act_id)
= **If** *is_plain_activity*(apsee, proc_id, act_id)
then *Resource_instantiation_plain_activity*(apsee, proc_id, act_id) {2}
else *Resource_Instantiation_fragment*(apsee, proc_id,
get_setOf_plain_acts(apsee, proc_id, act_id))

Resource_Instantiation_fragment(apsee, proc_id, add(activities_ids, act_id))
= *Resource_Instantiation_fragment*(
Resource_instantiation_plain_activity(apsee, proc_id, act_id),
proc_id,
activities_ids)
Resource_Instantiation_fragment(apsee, proc_id, empty_set) = apsee

{1} A chamada recursiva é adaptada para executar a terceira definição da operação (com 3 argumentos). Por isso, é obtido o conjunto dos identificadores de todas as atividades-folha do processo e a operação adequada é chamada para instanciar.
Para instanciar um processo completo, basta obter os ids das atividades folha do mesmo. Esses ids possuem, por definição no Apsee, o "endereço" da atividade no processo. O id de uma atividade, fornecido pelo usuário, é acrescido do id dos processos que a compõem, separados por pontos. Assim, a atividade x do sub-processo s do sub-processo p é identificada como p.s.x no Apsee. A partir do proc_id e do id de uma atividade qualquer é possível caminhar dentro do processo até encontrar a atividade.

{2} Se a operação recebe o identificador de uma atividade folha, então a função que instancia atividade folha é chamada. Se o act_id se refere a uma atividade decomposta, então são obtidos os ids de toda as atividades folha desse fragmento, e o resultado da função é obtido através do envio desses argumentos à terceira definição de resource_instantiation_fragment.

5.5.3.4 Instanciação Automática de Recursos

Para instanciar automaticamente uma atividade ou fragmento de processo, primeiramente é realizada a instanciação com os resultados sendo atualizados no *ResourcePolicyLog* e depois é feita a atualização nas atividades do processo, atribuindo os identificadores (ids) dos recursos a partir dos primeiros elementos das listas de sugestão de recurso. Portanto, a instanciação automática insere na definição de recursos requeridos da atividade os resultados da instanciação normal.

A especificação formal desse passo não é apresentada aqui por ser uma operação direta de manipulação dos tipos de dados, tendo sido implementada no protótipo.

6 PROTÓTIPO DO MODELO APSEE EM PROSOFT-JAVA

A especificação apresentada no capítulo anterior usando o PROSOFT-Algébrico e Gramáticas de Grafos serviu de base para a implementação de protótipos no ambiente PROSOFT-Java. O objetivo dessa implementação foi permitir uma avaliação da utilização dos componentes do modelo proposto.

As seções a seguir descrevem o ambiente PROSOFT-Java, o ambiente APSEE, o editor de processos de software, o mecanismo de execução de processos, e o editor e interpretador de políticas de instanciação.

6.1 O ambiente PROSOFT-Java

Esta seção fornece uma descrição geral do ambiente PROSOFT-Java de forma análoga ao encontrado em outros trabalhos recentes do mesmo grupo, que envolvem o mesmo ambiente (como por exemplo (REIS, 202f)).

PROSOFT-Java é a denominação da mais recente[26] implementação do paradigma PROSOFT, na forma de um ambiente homogêneo e integrado de desenvolvimento de software escrito na linguagem Java (SCHLEBBE,1997)). 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 principal objetivo do ambiente é estabelecer uma infra-estrutura que apóie o desenvolvimento de software de alta complexidade através da integração de ferramentas escritas em um paradigma próprio. Atualmente, o PROSOFT-Java é um ambiente portável[27], distribuído[28] e cooperativo[29] que integra ferramentas CASE para auxiliar diferentes fases do processo de software.

---

[26] A primeira versão monousuária do PROSOFT foi desenvolvida em Solaris-Pascal, como descrito por Nunes (1992). Posteriormente, o PROSOFT-Distribuído foi desenvolvido em Pascal e C, segundo descrito por Granville e Schlebbe (1996) e Schlebbe (1994). Finalmente, segundo Schlebbe (1997), em 1997 foi selecionada Java como nova linguagem hospedeira para o ambiente.

[27] Portável no sentido em que o sistema é executável nas plataformas para as quais estão disponíveis o *Java Runtime Environment* (SUN, 2002).

[28] 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 (SUN, 2002).

Há uma correspondência entre ATOs Algébricos e ATOs Java, conforme ilustrado pelo exemplo da figura 6.1. 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 das funções algébricas (axiomas) correspondentes.

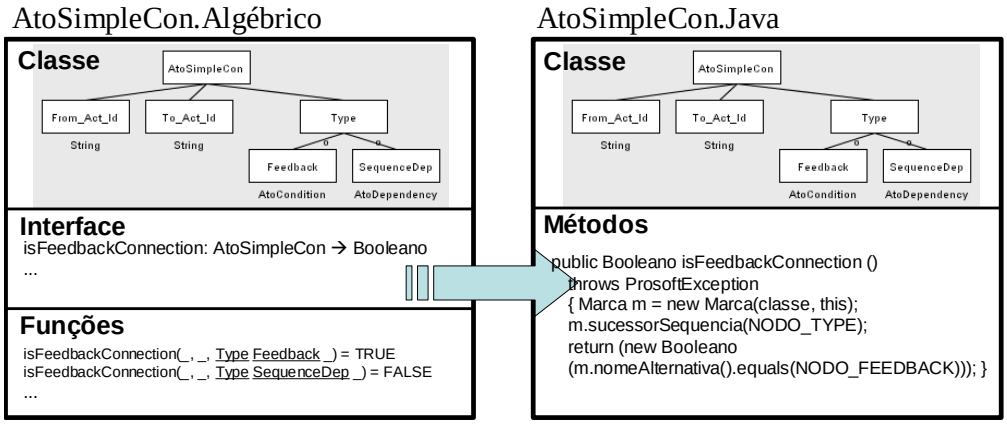


Figura 6.1: Exemplo esquemático de derivação manual de ATOs Java a partir de ATOs Algébricos

A ferramenta ATO-Classe está disponível para edição de classes PROSOFT, conforme ilustrado na figura 6.2, 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 (procedimentos rotulados 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 associados às opções de menus (denominada de "parte semântica" do ATO). Os esqueletos dos dois arquivos são gerados automaticamente pela operação "Generate Ato and Semantical Sources" 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., 1996), podendo utilizar os construtores disponíveis pelo pacote *prosoft.kernel* (uma descrição completa acerca do desenvolvimento de ATOs-Java foi disponibilizada por Schlebbe (1997; 2003).

6.2 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.

[29] Funcionalidade para trabalho cooperativo está disponível a partir da extensão denominada PROSOFT-Cooperativo, descrita por (REIS, 1998a; REIS et al., 1998b).

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.

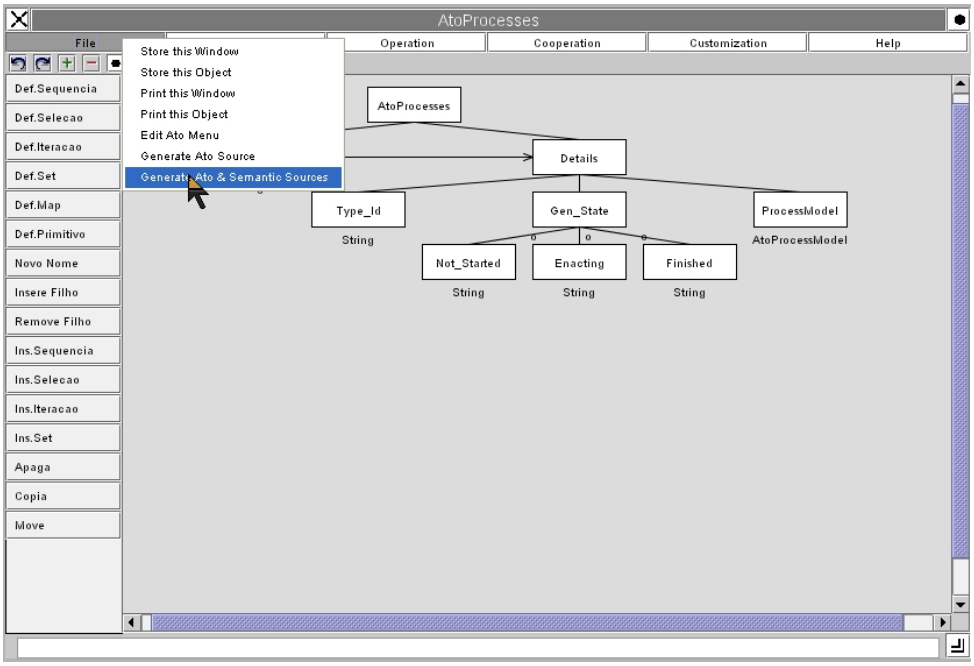


Figura 6.2: Tela do Editor ATO Classe.

6.3 O Editor de Processos de Software

Um editor para processos de software foi desenvolvido em PROSOFT-Java e integrado ao sistema APSEE. O editor implementa os tipos de dados, funções e regras descritas no capítulo 5. A implementação do editor de processos foi realizada com o auxílio dos Srs. Heribert Schlebbe e Marcelo Abreu (pesquisadores associados do projeto).

A figura 6.3 apresenta uma tela com o editor de processos, apresentando no destaque a classe *AtoProcessModel* associada ao 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 (SCHLEBBE, 1997). Assim, a tela na figura 6.3 apresenta o resultado da função “exibe” para o objeto armazenado no sistema no momento.

Alguns atributos dos ATOs especificados para o meta-modelo APSEE armazenam informações essencialmente textuais, e por isso a implementação atual do *exibe* está associada a formulários a serem manipulados pelo usuário. A figura 6.4 apresenta um exemplo de formulário, exibido para descrever os detalhes de uma atividade (i.e., objeto

da classe *Activities*). O formulário permite a descrição dos atributos *ID*, *Name*, *TypeID*, *Description e Policies* do objeto, tal como ilustrado pelas setas inseridas no diagrama.

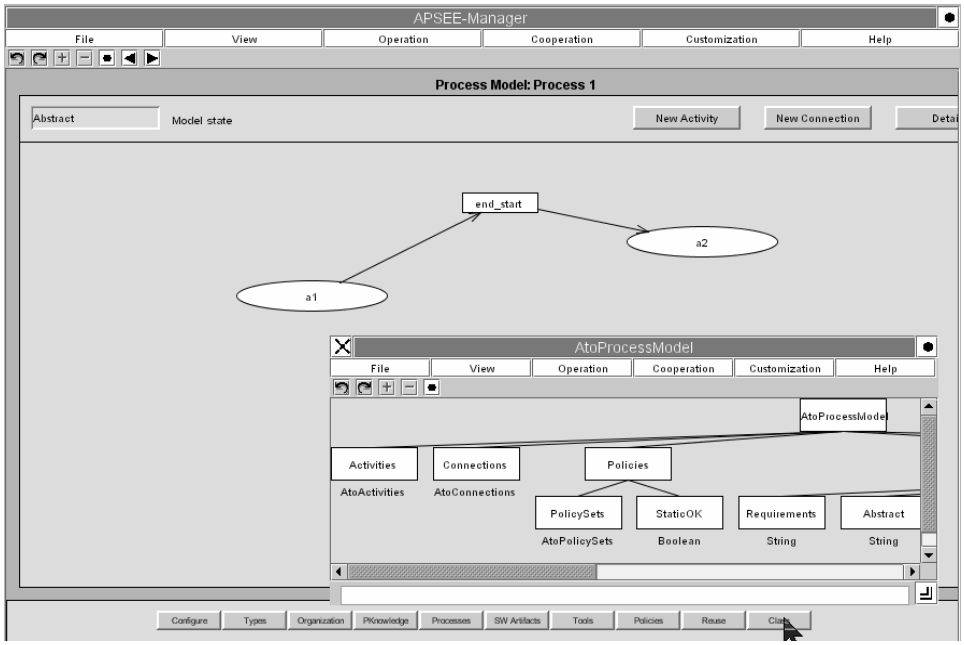


Figura 6.3: Tela do editor de processos, apresentando no destaque a classe *AtoProcessModel* associada

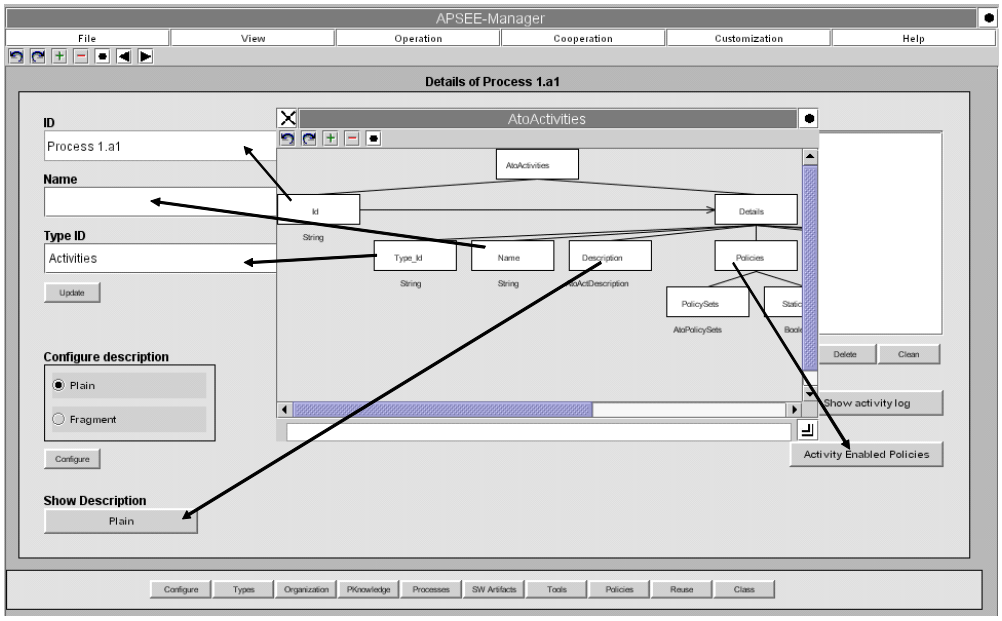


Figura 6.4: Formulário para definição de detalhes de uma atividade.

O editor de processos implementa as regras descritas nos apêndices desse trabalho. No caso das regras sintáticas, foram implementados métodos em PROSOFT-Java tendo por base as regras definidas. Por exemplo, as regras que definem a adição de uma conexão simples entre atividades, G3.1 a G3.6, apresentadas no Apêndice B, seção B.3,

foram implementadas em conjunto porque tratam casos diferentes para a mesma função. O método apresentado na figura 6.5 foi inspirados nessas regras e permite a inclusão de uma conexão simples entre atividades, verificando se as condições das regras são satisfeitas (chamada do método *verifySimpleDep*, destacado na figura). Em seguida a figura 6.6 mostra uma tela que ilustra o comportamento da regra implementada no momento em que o usuário tenta criar uma conexão que forma um ciclo no processo.

```
public void semDefineSimpleConFromActivity
(Ato processmodel, Stringue id_con, Stringue id_activ) throws ProsoftException {
Dialogo d = getDialogo();
d.message ("Click an activity to connect from");
Objeto from = d.clickObjeto ();
int clicked = getClicked (processmodel, from);
switch (clicked) {
case IS_ACT:
   Stringue from_id = (Stringue)from;
   Ato connections = (Ato)processmodel.ics ("getConnections");
   Ato connection = (Ato)connections.ics ("getConnection", id_con);
   Ato dep = (Ato)connection.ics ("getSequenceDep");
   Stringue to_id = (Stringue)connection.ics ("getTo_Act_Id");
   if (! to_id.valor().equals (from_id.valor())) {
 if (! ((Booleano)processmodel.ics ("verifySimpleDep",  connection, dep, from_id, to_id)).valor())
 {d.confirmOk ("Connection not possible."); return;}
connection.ics ("setFrom_Act_Id", from_id);
processmodel.ics ("exibeQuadro");  }
   else d.confirmOk ("Input / output activities are the same");    break;
default: d.confirmOk ("No input activity clicked for simple connection");
} }
```

Figura 6.5: Método em PROSOFT-JAVA para definição de conexão simples entre atividades.

Nos detalhes de uma atividade normal é possível definir através de formulários quais os agentes ou grupos serão envolvidos, assim como recursos necessários. A figura 6.7 mostra o formulário de definição de pessoas para uma atividade. Pode-se observar na figura que há uma integração com o mecanismo de instanciação através do botão "*Show Suggestion*".

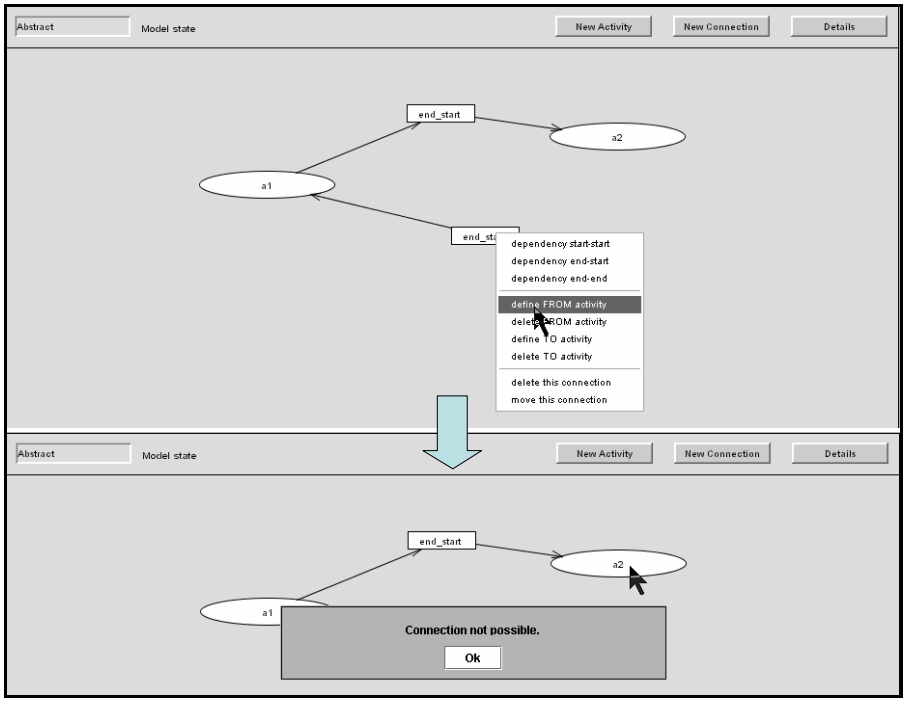


Figura 6.6: Comportamento da regra de adição de conexão simples quando a tentativa de criar um ciclo.

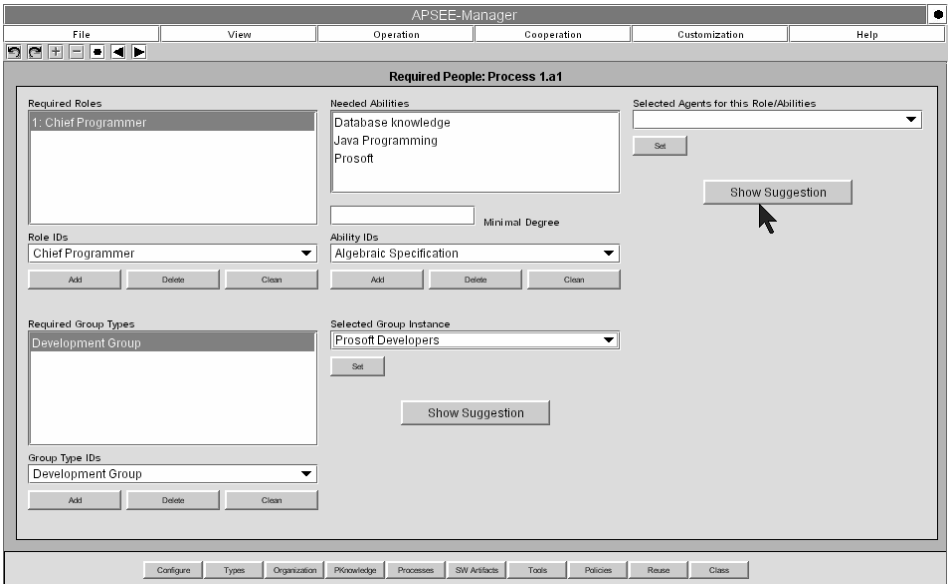


Figura 6.7: Definição de pessoas requeridas para atividade normal.

6.4 O Mecanismo de Execução de Processos do APSEE

Assim como o editor para processos, o mecanismo de execução também foi desenvolvido em PROSOFT-Java e integrado ao sistema APSEE. A figura 6.8 mostra a tela onde o usuário pode selecionar um processo para execução.

Após o início da execução o gerente pode acompanhar o andamento do processo usando a mesma ferramenta de edição de processos. A partir desse momento o editor possui todas as suas funcionalidades originais, fornecendo ainda informações sobre a dinâmica de execução do processo. Uma das maneiras de informar o gerente sobre o andamento da execução é através da mudança de cor das atividades de acordo com o seu estado. Por exemplo, a figura 6.9 mostra um processo simplificado no qual a primeira atividade está pronta para executar (*ready)*. Além disso, através dos detalhes de cada atividade é possível obter as informações sobre a execução como data de início e fim de cada atividade e eventos ocorridos.

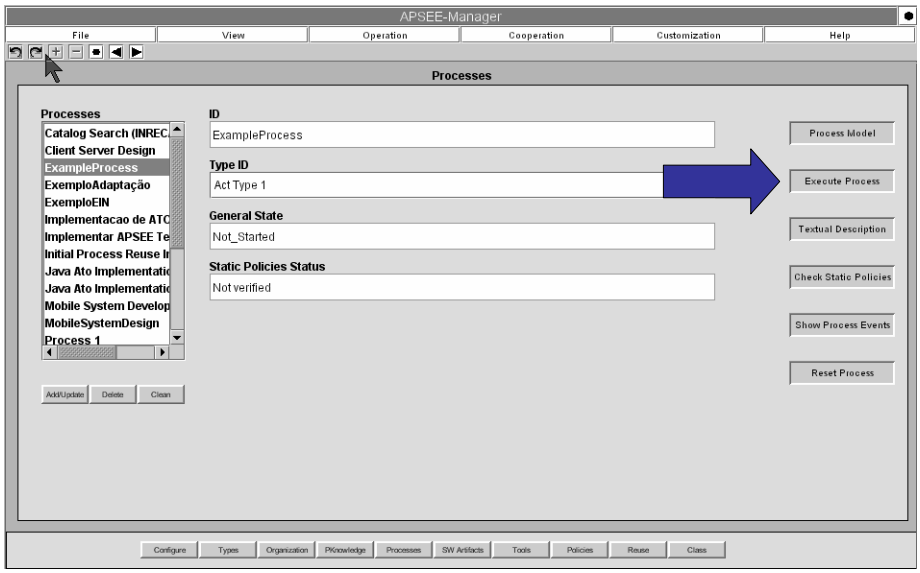


Figura 6.8: Tela de seleção de processos para início da execução.

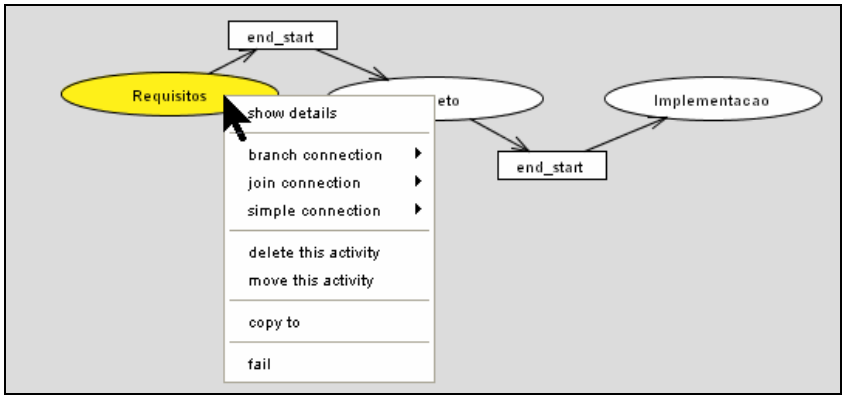


Figura 6.9: Exemplo de processo em execução.

Do ponto de vista do desenvolvedor, o mecanismo de execução fornece informações através da agenda de atividades. Nessa agenda o usuário pode visualizar o estado da atividade, pode interagir solicitando mudanças de estado e pode visualizar os detalhes da atividade.

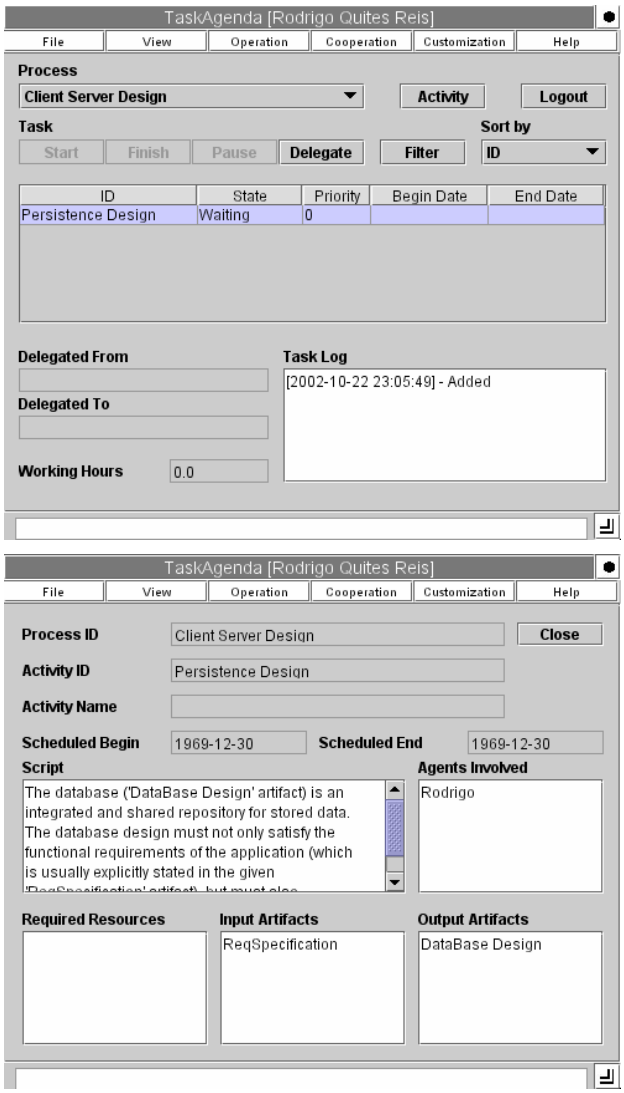


Figura 6.10: Agenda do desenvolvedor.

6.5 O Editor e o Interpretador de Políticas de Instanciação

A partir da especificação algébrica apresentada no capítulo 5 foi construído um protótipo para o interpretador de políticas de instanciação. Essa implementação contou com o auxílio do Sr. Heribert Schlebbe.

A figura 6.11 ilustra o mapeamento da função algébrica *getInitialResourcesList* da primeira etapa da instanciação de recursos para método Java correspondente. 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.

initial_resources_list: APSEE, String, longue, longue, real → ListofString

initial_resources_list(Apsee, req_type, begin, end, amount_needed) =
= ICS(RESOURCES, get_needed_resources, **get_resources*(apsee)*,**
<*get_apseetypes*(apsee), req_type, begin, end, amount_needed>)

```
public Ato getInitialResourcesList (Ato apsee, Stringue req_type, Longue begin, Longue end, Real
amount_needed)  throws ProsoftException {
                    if (TRACE_POLICIES)
                    System.out.println ("+ getInitialResourcesList: "+ "req_type: "+req_type.valor());
                    Ato resources  = (Ato)apsee.ics ("getResources");
                    Ato apseetypes = (Ato)apsee.ics ("getApseeTypes");
                    Ato needed_resources = getNeededResources (resources, apseetypes, req_type,
                                                        begin, end, amount_needed);
                     if (TRACE_POLICIES)System.out.println ("- getInitialResourcesList: "+   "req_type:
                                "+req_type.valor()+"\n  -> "+toString (needed_resources));
                       return (needed_resources);          }
```

Figura 6.11: Especificação algébrica e implementação da função *GetInitialResourcesList.*

Existe um editor para políticas de instanciação implementado para que o usuário crie políticas através de um formulário apropriado, que contém todas as operações básicas da linguagem de políticas. No componente *Policies* do ambiente APSEE, é possível visualizar e manipular as políticas de instanciação existentes através da verificação dos seus detalhes como mostra a figura 6.12. Em seguida a figura 6.13 mostra os detalhes da política "*Consumable*". Cada componente da política pode ser editado a partir desse formulário.

Figura 6.12: Componente *Resource Instantiation Policies* do APSEE.

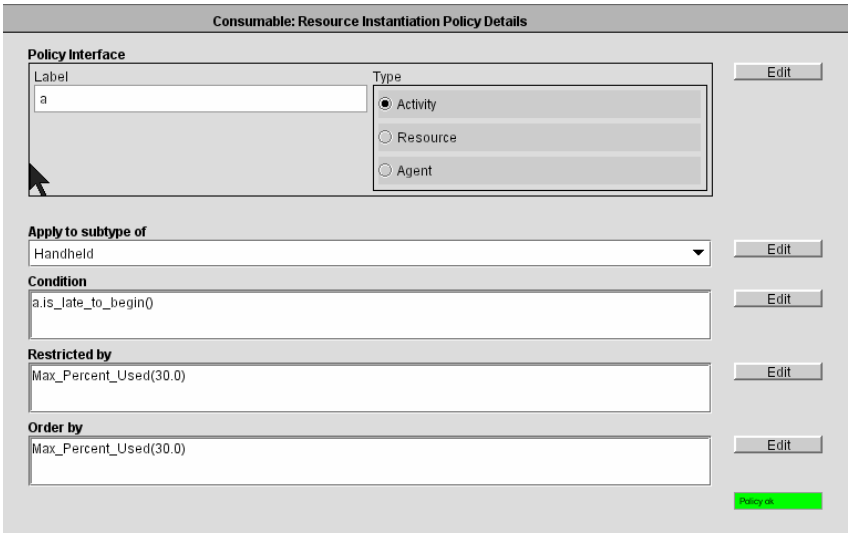


Figura 6.13: Detalhes da política "*Consumable*"

A seguir são apresentados alguns formulários do editor de políticas de instanciação. Na figura figura 6.14 é apresentado o formulário para definição do critério de restrição da política (a definição do critério de ordenação é análoga). A figura 6.15 mostra o formulário que permite a escolha do método de restrição dentre os métodos possíveis da linguagem. Finalmente a figura 6.16 mostra o editor de condições lógicas.

Quando uma atividade habilita uma política de instanciação, então o usuário pode chamar o interpretador de políticas através do formulário de edição da atividade (*Required People* ou *Required Resource)* através do botão Show Suggestion. Além disso, caso a instanciação automática esteja ligada no ambiente, são executadas as regras do apêndice A, seção A.3.10, que chamam o interpretador automaticamente durante a execução.

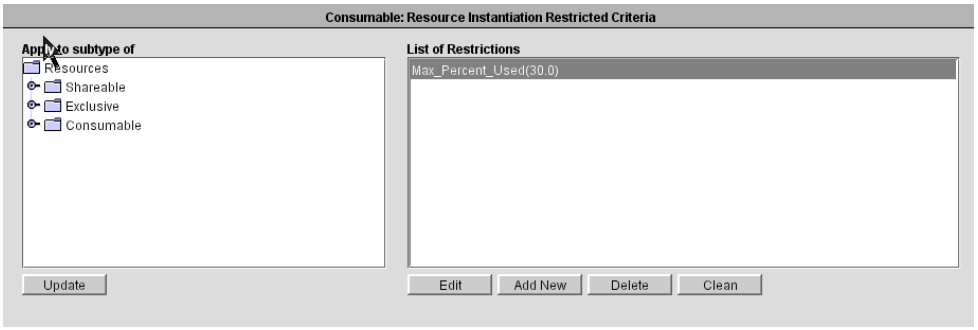


Figura 6.14: Definição do critério de restrição da política.

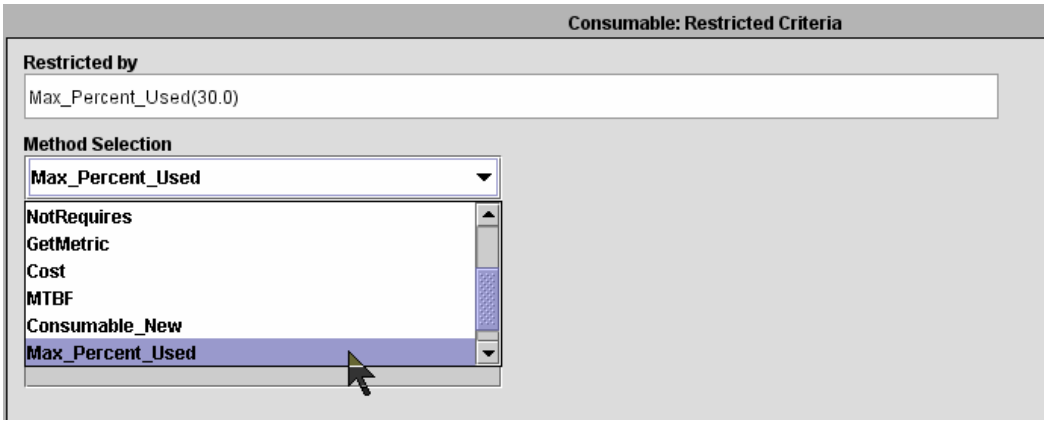


Figura 6.15: Escolha do método de restrição para política.

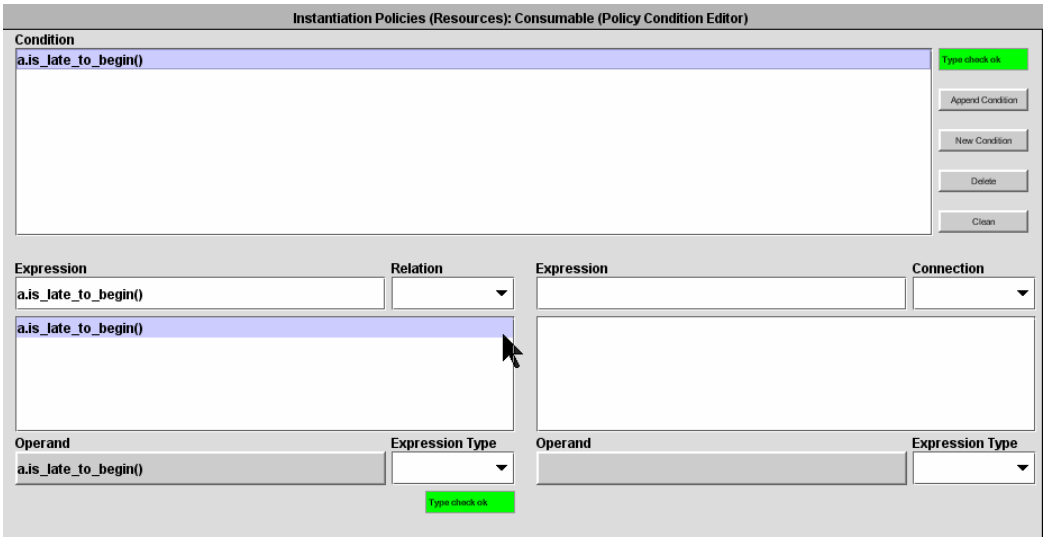


Figura 6.16: Editor de condições lógicas.

7 ESTUDOS DE CASO

Esse capítulo descreve o uso prático do modelo proposto em situações que envolvem a construção e execução de processos de software 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, execução e instanciação de processos. Em (REIS, 2002f) foram apresentados vários processos abstratos (templates), prontos para reutilização, construídos com o modelo APSEE-Reuse. Dentre os templates apresentados em (REIS, 2002f), foram escolhidos o template para o desenvolvimento no PROSOFT-Java e para Sistemas Móveis. Esses templates são instanciados e executados neste capítulo com o objetivo de demonstrar o funcionamento do modelo proposto.

7.1 O processo para desenvolvimento no PROSOFT-Java

Em (REIS, 2002f) foi apresentado o *template* para desenvolvimento de software no PROSOFT-Java. O principal objetivo da criação e disponibilização desse processo abstrato foi descrever de forma genérica a metodologia a ser adotada para a construção de ATOs no PROSOFT. 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* (SCHLEBBE, 2003)). Assim, a especificação do *template* denominado *JavaAtoImplementation* forneceu uma descrição genérica que está sendo reutilizada para este estudo de caso.

Nesse texto o *template* mencionado será utilizado para descrever a execução de um processo real com alternativas de fluxo de execução (conexões múltiplas do tipo *Branch* e *Join*). Além disso, como mostrado nas sub-seções a seguir, o estudo de caso serve para descrever o comportamento do modelo quando uma nova atividade e conexão são inseridas em um processo em execução.

7.1.1 Modelagem

A figura 7.1 apresenta a descrição do processo PROSOFT-Java. Assim, de acordo com as ferramentas disponibilizadas pelo PROSOFT-Java, o desenvolvimento de ATOs Java é realizado por uma seqüência de atividades listadas abaixo (REIS, 2002f):

- A atividade normal *Desenhar Classe Prosoft* é responsável por definir uma classe PROSOFT usando o editor fornecido no ATO Classe;
- A atividade automática *Gerar Ato* gera o esqueleto de código-fonte correspondente à classe PROSOFT fornecida, utilizando uma função disponibilizada pelo ATO Classe;
- A atividade decomposta *Adicionar Funcionalidade ao ATO* 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, essa atividade decomposta é opcional, podendo ser suprimida para ATOs simples que não requeiram métodos adicionais;
- A compilação do ATO (atividade automática *Compilar Código*) 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 *Integrar Ato ao PROSOFT*. Segundo (SCHLEBBE, 2003), a integração é importante por definir os recursos do ATO que estarão disponíveis para acesso remoto por outros ATOs;
- Finalmente, a atividade *Testar o Ato* descreve o teste final realizado no ATO recém implementado confrontando-o com os requisitos da aplicação desenvolvida.

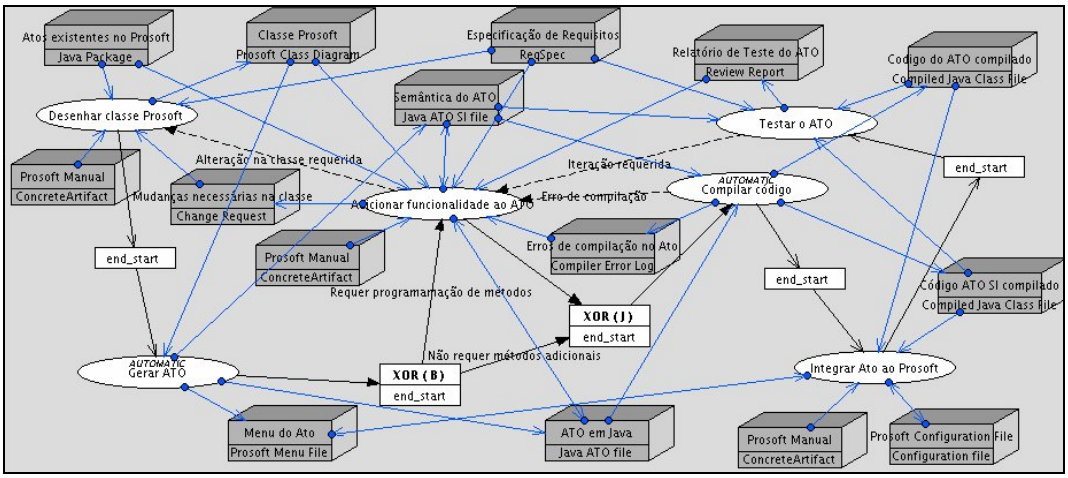


Figura 7.1: Descrição do processo que descreve o desenvolvimento de software no PROSOFT-Java

Para cada atividade definida, foram especificados detalhes adicionais os quais são fornecidos na versão *online* do manual do PROSOFT (SCHLEBBE, 2003), e que acompanha a distribuição pública do pacote PROSOFT-Java atual.

7.1.2 Execução

Antes do início da execução do processo do PROSOFT-Java todas as atividades foram instanciadas manualmente. Além disso, alguns ajustes adicionais foram necessários, pois existem conexões no processo que dependem da avaliação de

condições lógicas para decidir o caminho a seguir (*XOR Branch* e conexões de *feedback*). Apesar de um editor e um interpretador de condições lógicas estarem disponíveis no ambiente para uso pelas políticas de instanciação, na versão testada esse recurso ainda não havia sido integrado ao mecanismo de execução. Por isso, houve necessidade de substituir as condições apresentadas no modelo pelos valores *true* e *false.* A figura 7.2 mostra o processo no início da execução com a atividade *Desenhar Classe Prosoft* pronta para ser executada. A figura 7.3, por sua vez, mostra a situação nas agendas dos agentes envolvidos no processo.

A figura 7.4 mostra o início da execução da atividade *Desenhar Classe Prosoft* pelo agente enquanto a figura 7.5 mostra a situação das agendas dos agentes após o fim dessa atividade. Deve-se notar que a atividade *Adicionar Funcionalidade ao Ato* é cancelada automaticamente devido à avaliação da condição do *XOR Branch*. Os eventos ocorridos no processo e nas atividades são registrados e apresentados como mostra a figura 7.6.

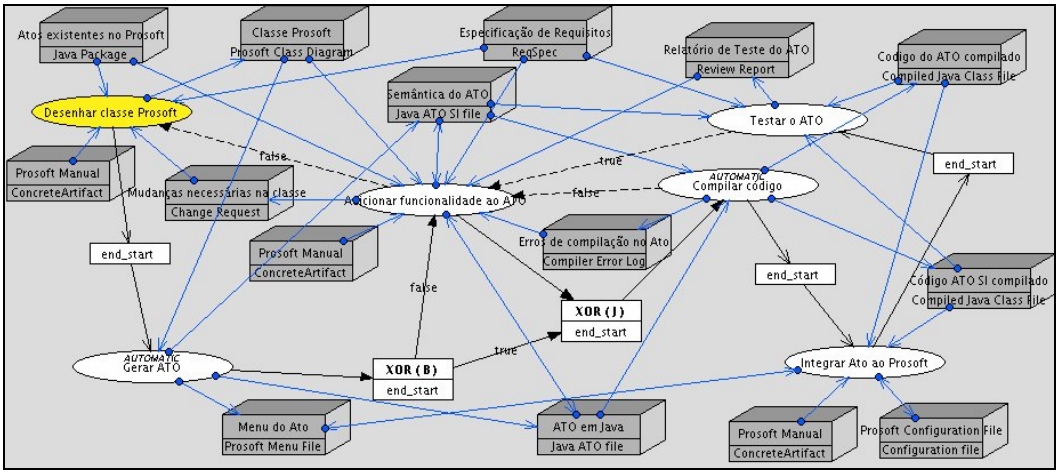


Figura 7.2: Processo PROSOFT-Java no início da execução.

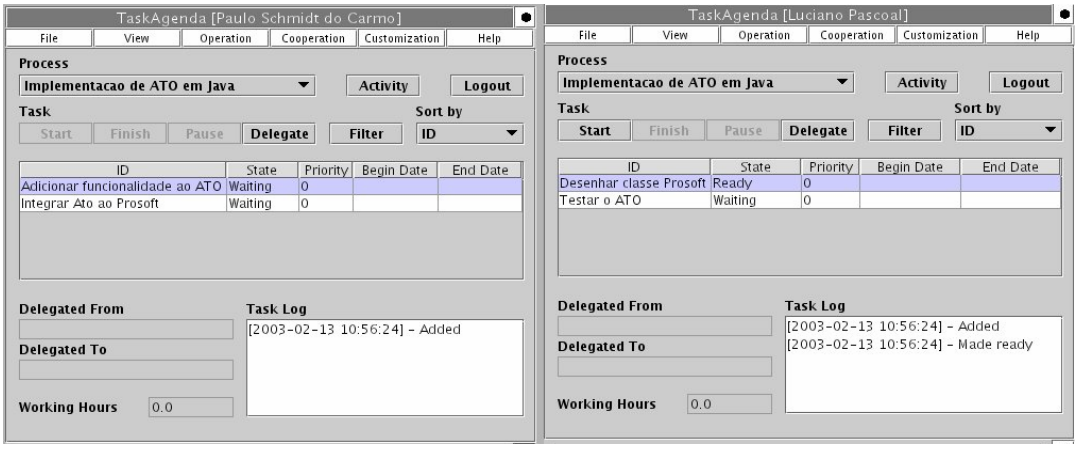


Figura 7.3: Agendas dos agentes envolvidos no processo.

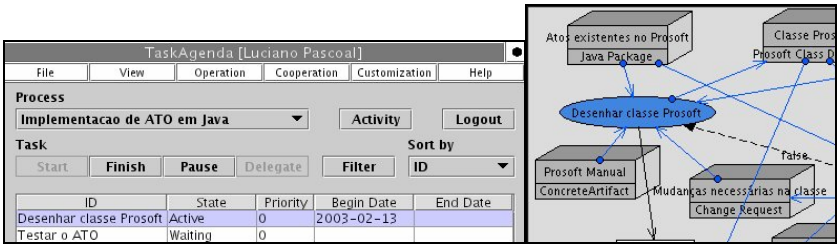


Figura 7.4: Início da execução da atividade *Desenhar Classe Prosoft* pelo agente.

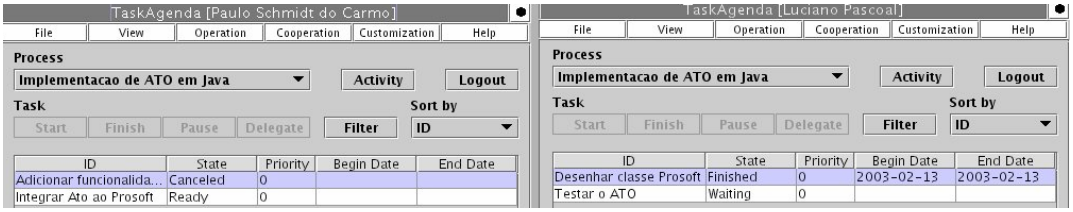


Figura 7.5: Situação nas agendas após fim da atividade *Desenhar Classe Prosoft*.

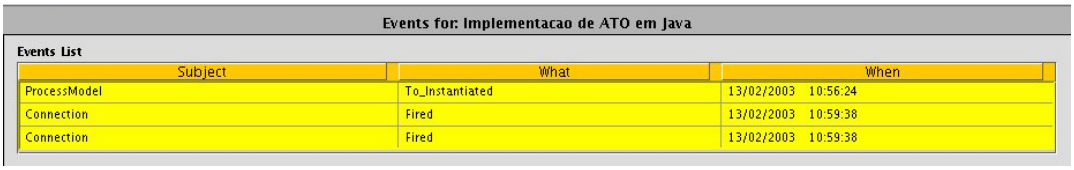


Figura 7.6: Lista de eventos ocorridos no processo até o momento.

A figura 7.7 mostra em seqüência a criação de uma nova atividade no processo chamada *Documentar Ato*. Logo após sua criação, ela é tornada pronta (*ready*) e em seguida está ativa (o agente responsável iniciou a sua execução). Enquanto isso, na parte inferior da figura é mostrada a tentativa de criar uma dependência em relação à atividade *Integrar Ato ao Prosoft*. Porém, como a atividade destino da conexão já iniciou, não é possível condicionar seu início ao fim de outra. Na última parte da figura é mostrada uma conexão possível com dependência *end-end*.

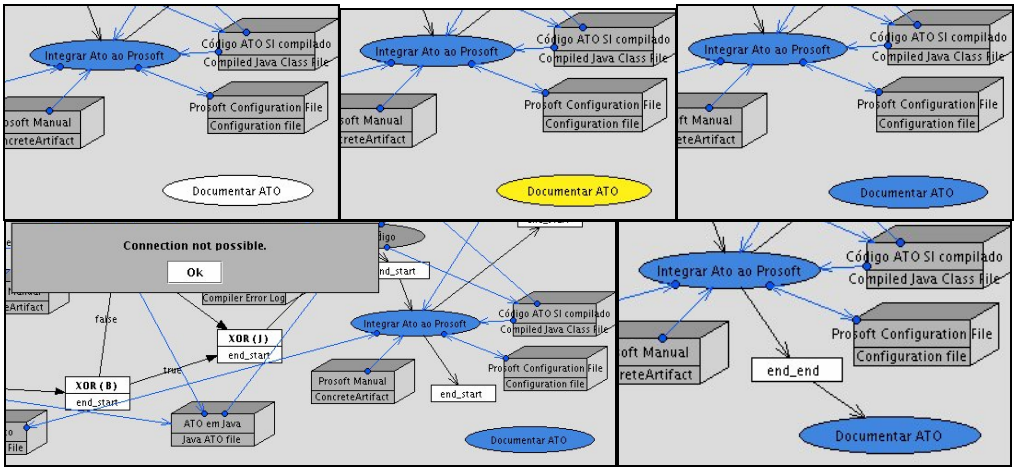


Figura 7.7: Modificação dinâmica no processo: criação de uma nova atividade.

7.2 O Processo para Sistemas Móveis

O *template* para desenvolvimento de sistemas móveis (REIS, 2002f) foi 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 a autora desse trabalho e os Srs. Adenauer Yamin, Rodrigo 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 *template* foi desenvolvido e apresentado em (REIS, 2002f) visando apoiar a infra-estrutura para desenvolvimento de Sistemas Móveis - denominada ISAM - que permite a mobilidade de agentes para a resolução de problemas que demandam mobilidade física e lógica de software (YAMIN et al., 2001). Da mesma forma que o estudo de caso anterior, o *template* foi instanciado e várias situações de execução foram simuladas - entre elas, a falha da execução de uma atividade (acionada pelo gerente) - para demonstrar o funcionamento do protótipo.

7.2.1 Modelagem

A figura 7.8 apresenta de forma compacta a hierarquia[30] que descreve a composição de atividades do processo *Mobile System Development*. A figura 7.9, por sua vez, descreve em detalhe o ordenamento das atividades (todas decompostas) e os artefatos de software envolvidos. No processo de projeto do sistema móvel mostrado na figura 7.10 (atividade decomposta *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*). Deve-se notar que há uma consistência entre os artefatos envolvidos nos níveis inferior e superior.

[30] A decomposição hierárquica das atividades que compõem um modelo de processo de APSEE é uma visão fornecida pela ferramenta APSEE-Monitor (SOUZA, 2003).

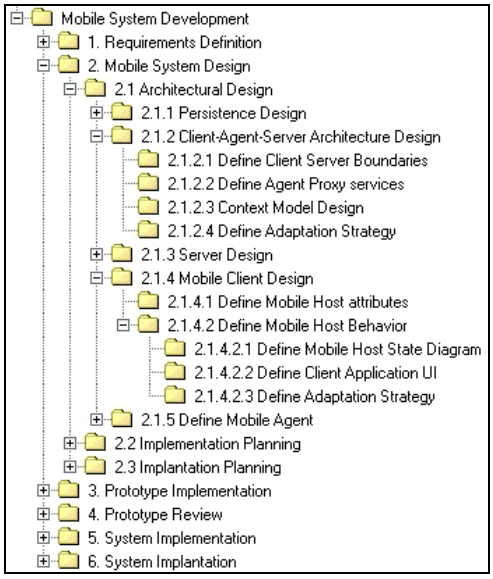


Figura 7.8: Hierarquia de atividades para o *template Mobile System Development*

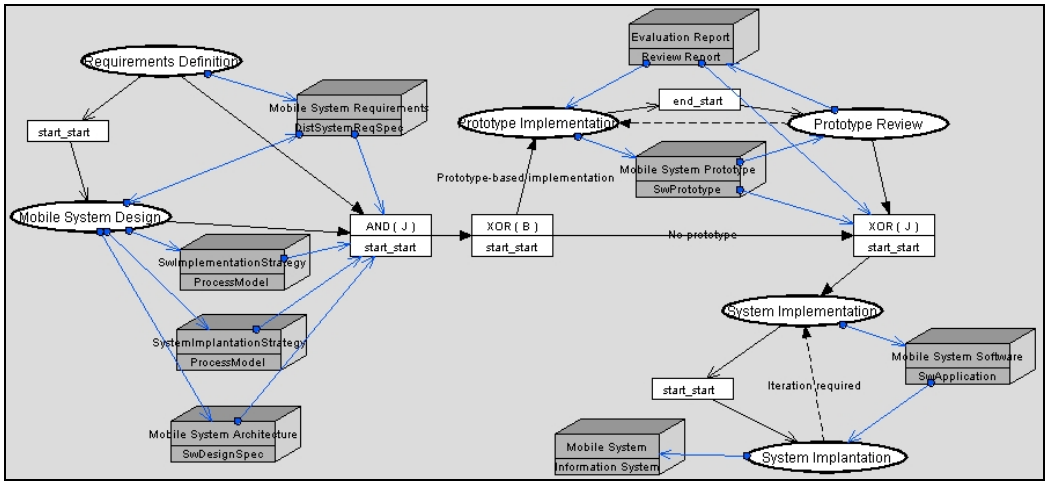


Figura 7.9: O processo de alto nível para *Mobile System Development*

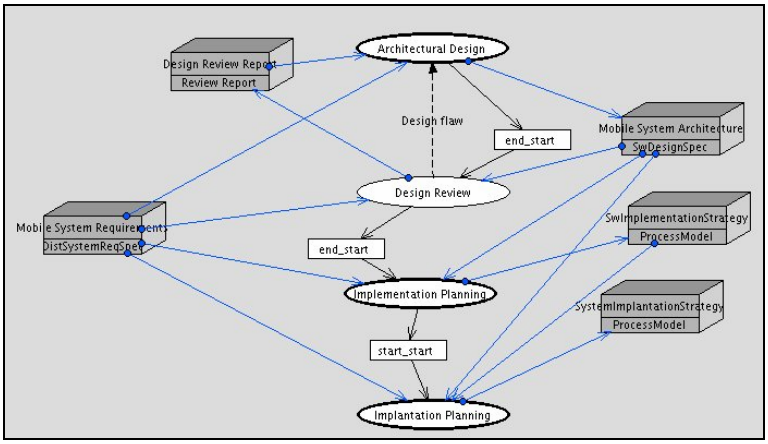


Figura 7.10: Detalhamento da atividade decomposta *Mobile System Design.*

A figura 7.11 fornece o detalhamento para a atividade decomposta *Architectural Design* (acima) e a decomposição *Client-Agent-Server Architecture Design*. A figura 7.12 mostra o conteúdo da atividade decomposta *Mobile System Design* que também compõe a atividade decomposta *Architectural Design*.

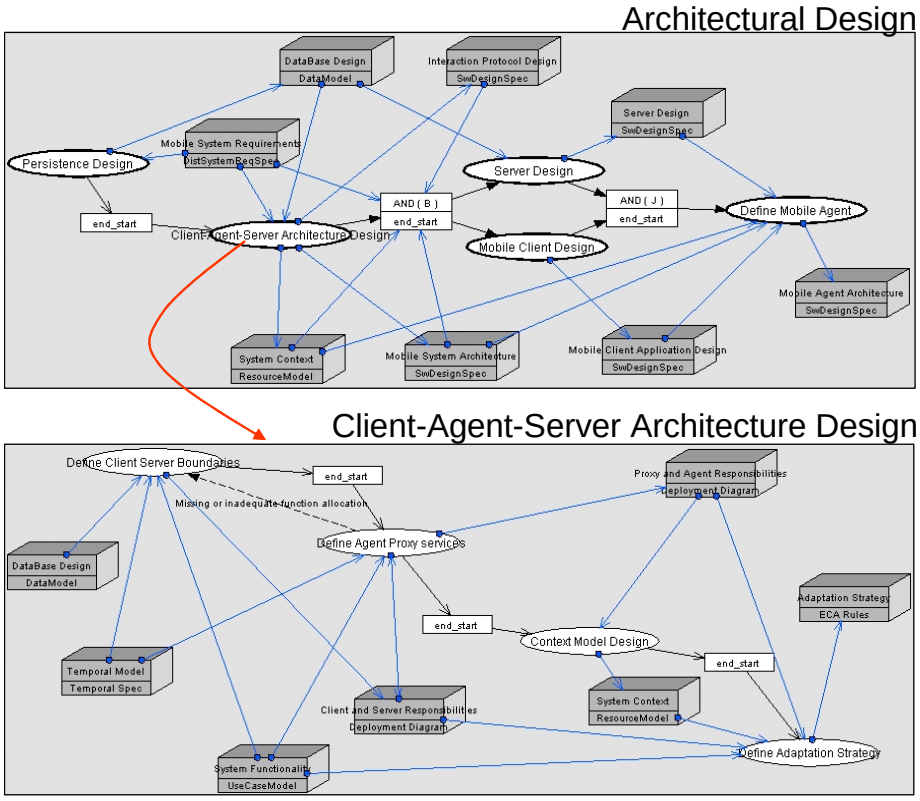


Figura 7.11: Atividades decompostas *Architectural Design* e *Client-Agent-Server Architecture Design*

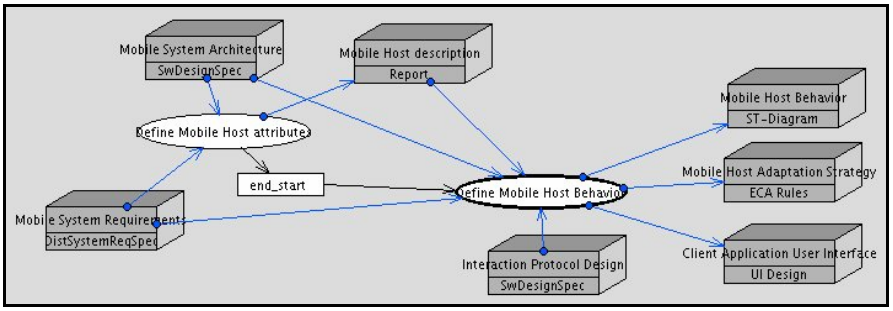


Figura 7.12: Atividades decompostas *Mobile Client Design* (detalhamento do *Architectural Design*).

7.2.2 Execução

Para executar o processo, foi também necessário atribuir *true* e *false* para destinos de conexões XOR, assim como no estudo de caso anterior. A figura 7.13 mostra o processo principal após início da execução com as atividades *Requirements Definition* e *Mobile System Design* ativas (azuis), enquanto que a atividade *System Implementation* foi cancelada (vermelho-claro), o que gerou o cancelamento de *System Implantation* por propagação.

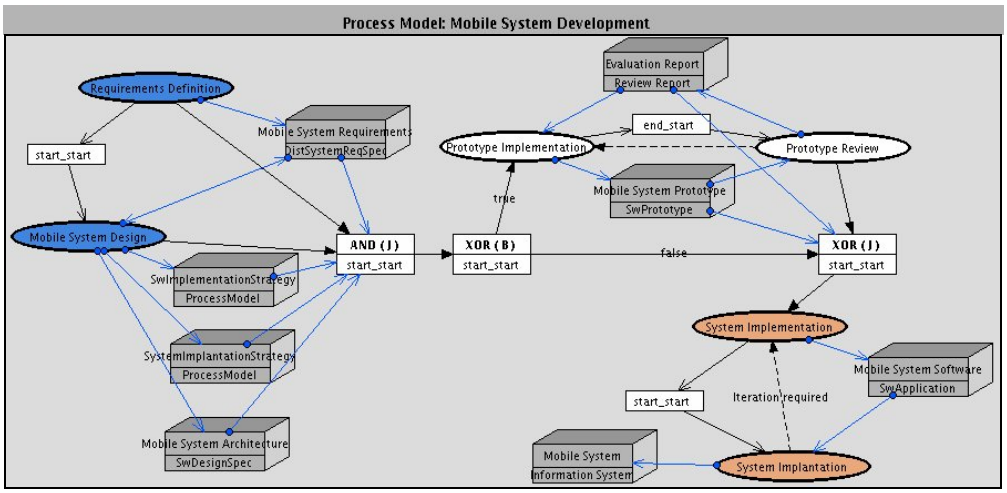


Figura 7.13: Processo de Sistemas Móveis após início e cancelamento de algumas atividades.

A figura 7.14 mostra o estado inicial das agendas dos agentes envolvidos.

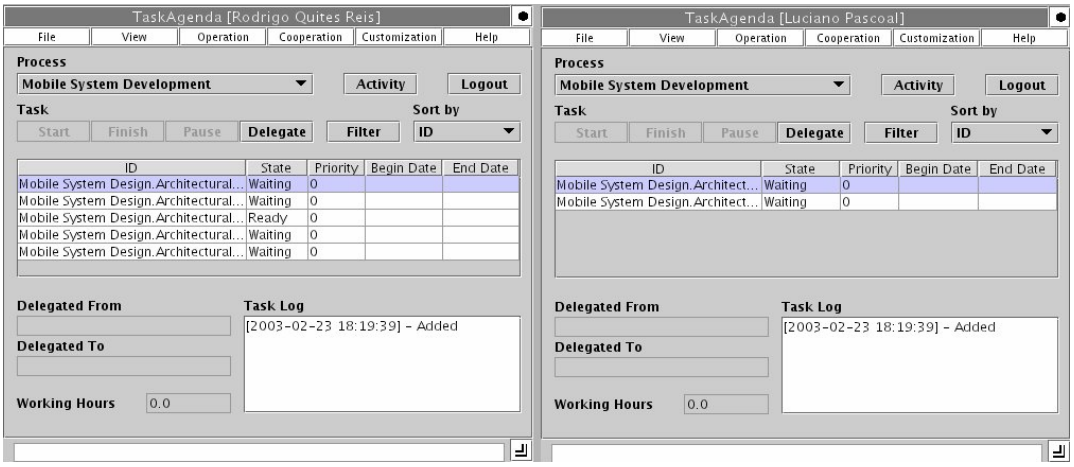
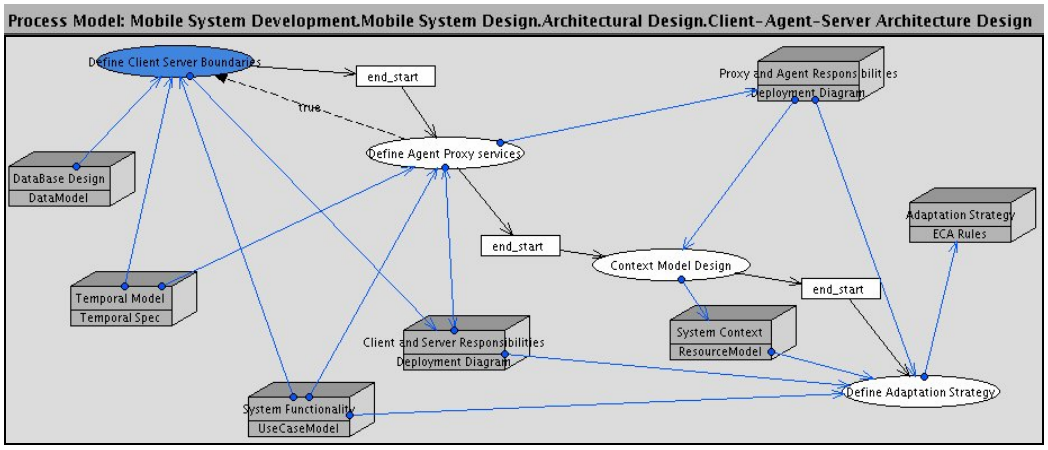


Figura 7.14: Estado inicial das agendas no processo de Sistemas Móveis.

A figura 7.15 mostram uma seqüência com a visão da execução de uma das atividades decompostas do processo principal (*Mobile System Development.Mobile System Design.Architectural Design.Client-Agent-Server Architecture Design*). Para este modelo de processo serão mostrados os casos de ativação de *feedback*, falhas em atividades e alocação de recursos. A primeira parte da figura 7.15 mostra a atividade "*Define Client Server Boundaries*" no estado ativa (em azul) enquanto as outras estão *waiting* (brancas). A segunda parte da figura mostra a atividade citada no estado de concluída e a atividade seguinte, *Define Agent Proxy Service*, no estado *ready* (pronta, em amarelo). Finalmente a terceira parte da figura mostra a atividade "*Define Agent...*" que estava pronta, no estado ativa (azul).



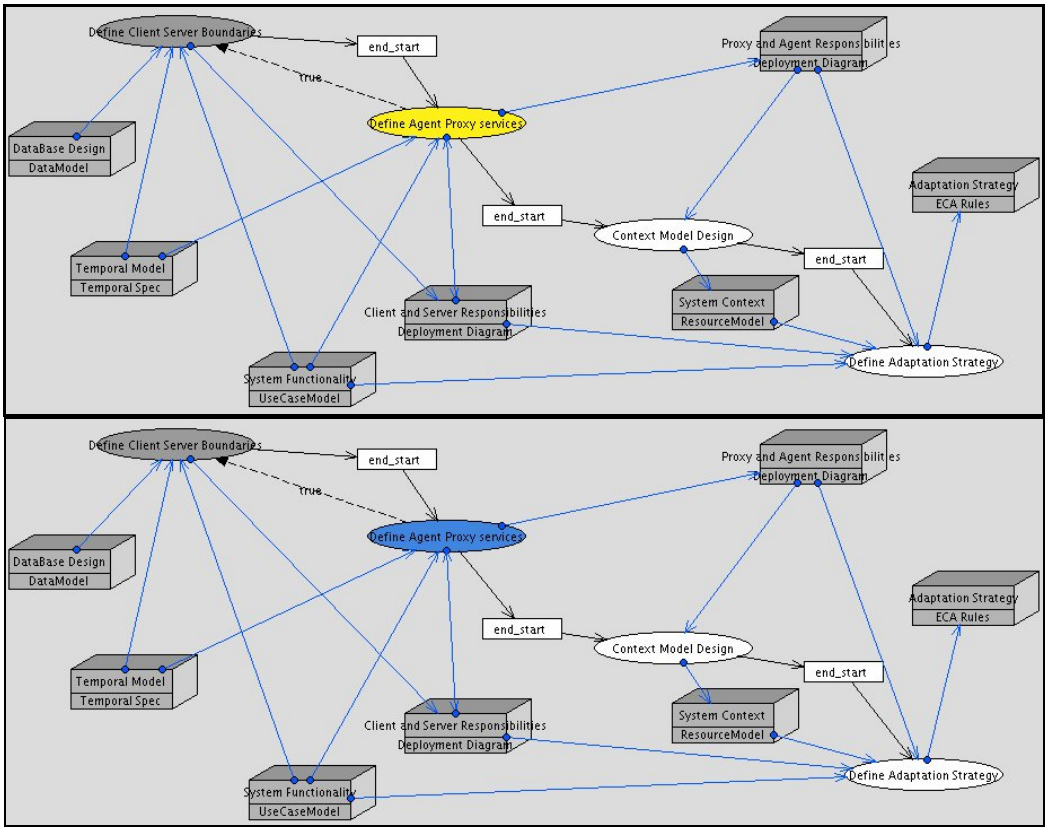


Figura 7.15: Evolução da execução na atividade decomposta *Client-Agent-Server Architecture Design.*

Como existe uma conexão de *feedback* a ser ativada após o término da atividade *Define Agent Proxy service*, a continuação da execução leva para a situação mostrada na figura 7.16. Nesse momento a primeira atividade (*Define Client Server Boundaries*) torna-se pronta (amarela) para iniciar (na verdade, foi criada uma nova versão e a versão anterior foi armazenada) enquanto a atividade que originou o *feedback* aguarda novamente o seu término (estado *waiting*) para reiniciar.

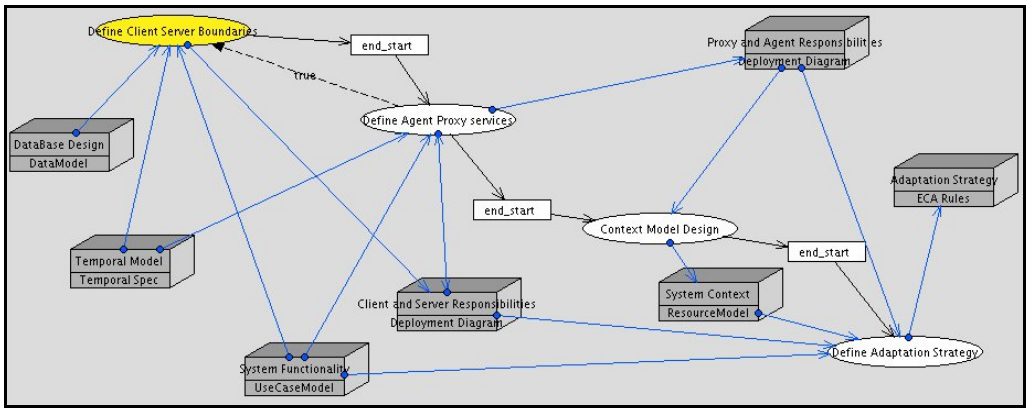


Figura 7.16: Processo após ativação da conexão de *feedback.*

Na seqüência, foram simuladas as seguintes situações: a) foi alterado o valor da condição de *feedback* para *false* (parte superior da figura 7.17). Com isso, após a atividade *Define Agent Proxy Service*, o processo continua sua execução para as atividades subseqüentes (notar que isso pode ocorrer durante a execução mesmo com uma condição lógica mais complexa, pois a condição somente é verificada no momento do *feedback*); b) foi requisitada a falha da atividade *Define Agent Proxy Service* durante sua execução (parte inferior da figura 7.17). Com isso, todas as atividades restantes falharam em resposta às regras de propagação de falhas do sistema (atividades falhadas estão em vermelho).

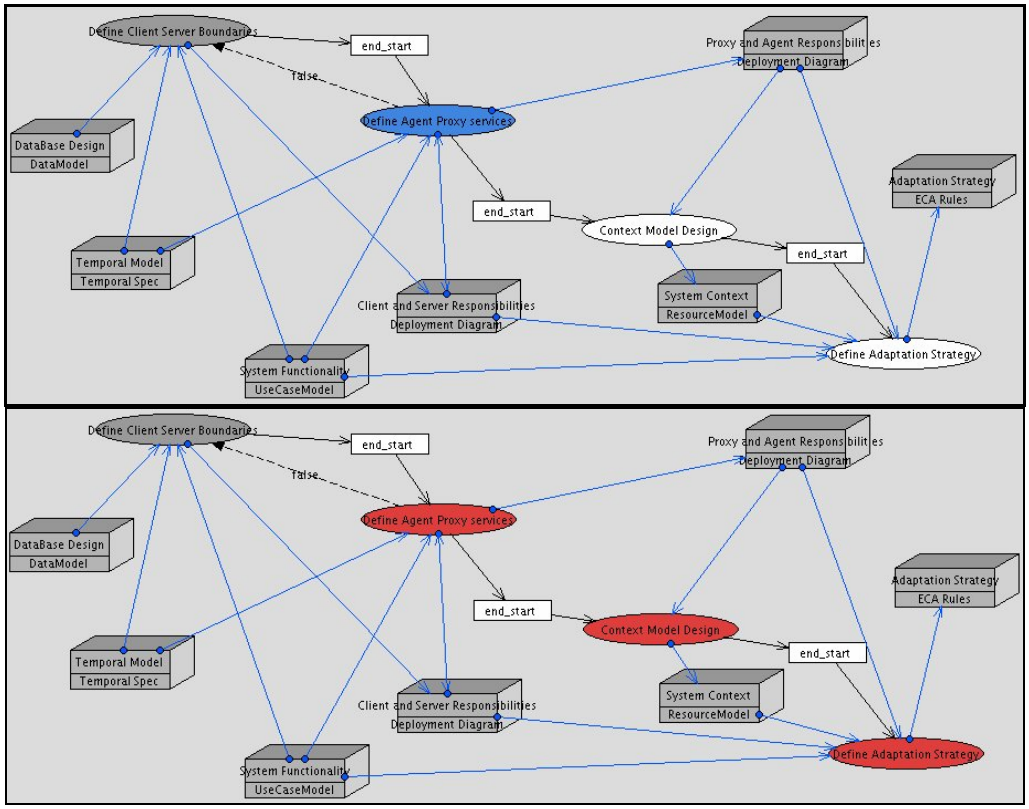


Figura 7.17: Continuação da execução da atividade decomposta sem *feedback* e com propagação de falhas.

Com respeito à ativação de conexões de *feedback* e criação de versões de atividades, é necessário garantir a alocação de recursos para que as atividades sejam executadas. A figura 7.18 mostra que o recurso *BlueAuditorium* utilizado pela atividade *Define Client Server Boundaries*, foi alocado e liberado duas vezes para a mesma atividade e seu uso fica registrado no *log* de eventos do recurso. Além disso, a figura 7.19 mostra a evolução da execução da atividade decomposta sendo mostrada através do seu *log* de eventos.

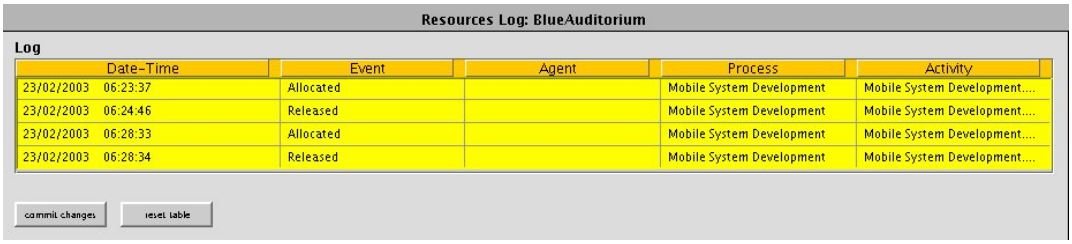


Figura 7.18: Eventos do recurso *BlueAuditorium*.

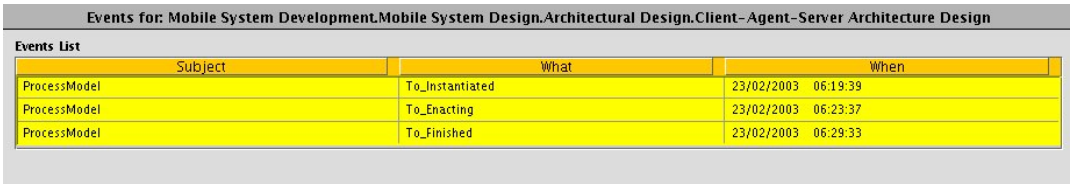


Figura 7.19: Eventos da atividade decomposta *Client-Agent-Server Architecture Design.*

A seguir é mostrada a execução de uma outra atividade decomposta do mesmo processo, visando apresentar o comportamento do sistema para modificação dinâmica de processos. A figura 7.20 mostra a atividade decomposta *Mobile System development.Mobile System Design.Architectural Design.Mobile Client Design* na qual a atividade *Define Mobile Host Attribute* está executando (em azul) enquanto uma nova atividade *New Activity* é inserida dinamicamente (ainda não instanciada).

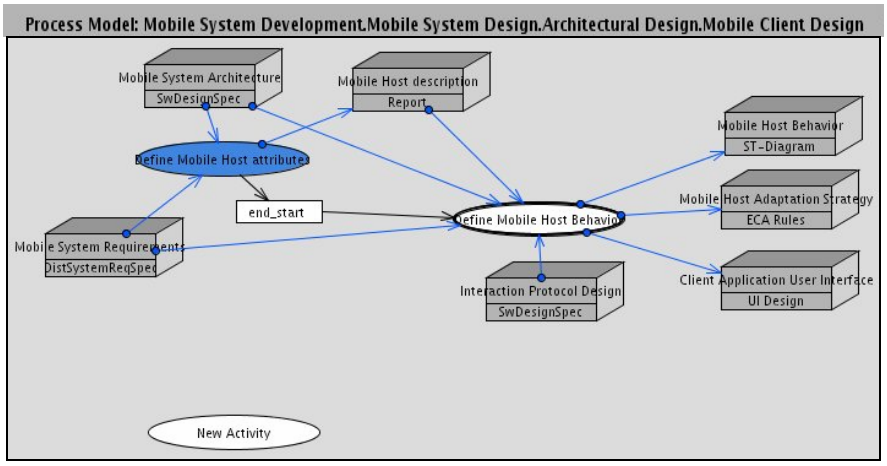


Figura 7.20: Atividade decomposta *Mobile Client Design* sendo executada com inclusão de nova atividade.

A figura 7.21 mostra o passo seguinte onde a nova atividade foi instanciada e por isso tem seu estado alterado para *ready* (por não depender de nenhuma outra) e é inserida uma conexão simples *end-end* com destino na atividade que está ativa (parte superior da figura). A segunda situação da figura mostra a criação de uma conexão com destino na nova atividade e com dependência indefinida. A partir do momento em que é definida a dependência, o sistema verifica a possibilidade da conexão e conclui que a mesma forma um ciclo, impedindo a sua criação (parte inferior da figura), garantindo, assim, a consistência do processo em execução.

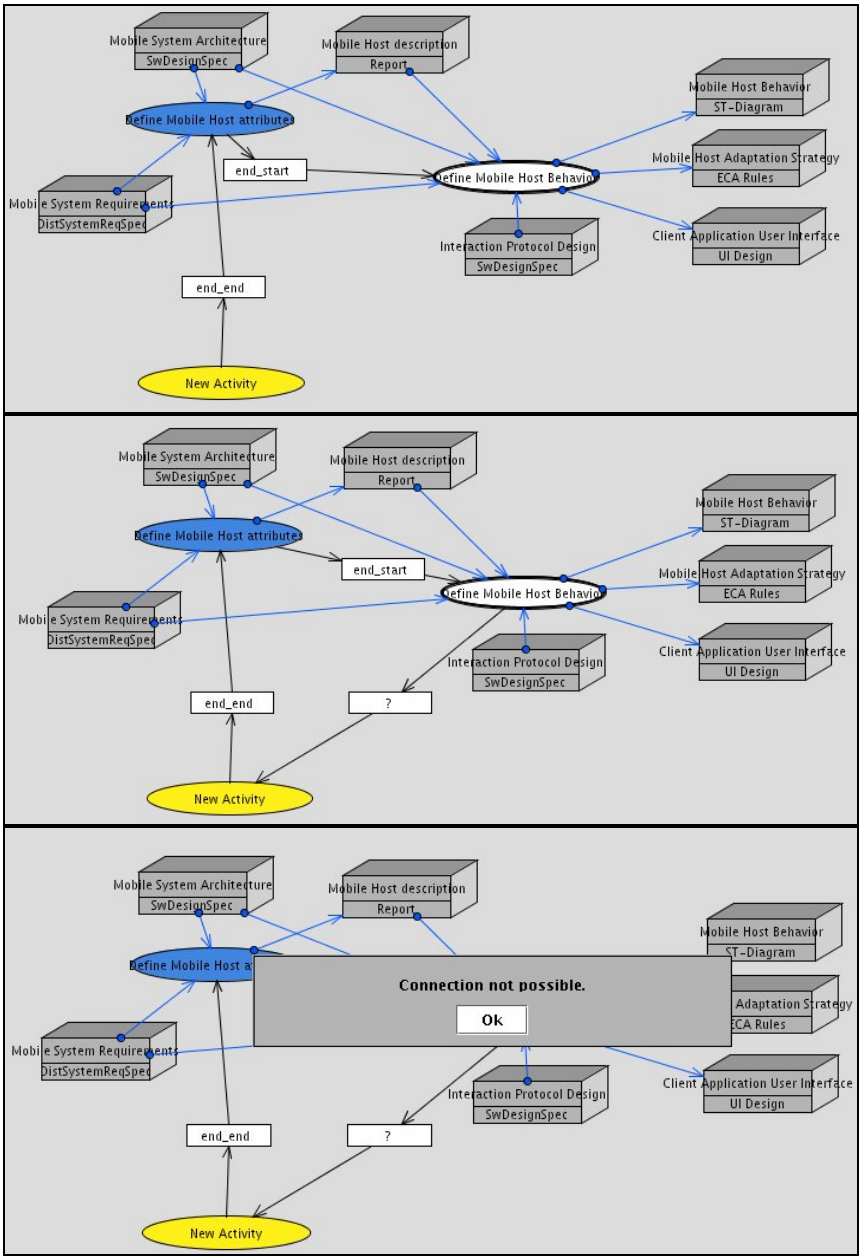


Figura 7.21: Seqüência de execução com criação de conexões de controle dinamicamente.

7.3 Exemplo de Instanciação de um recurso

Para ilustrar a instanciação de um recurso foram construídas algumas políticas e foram inseridas informações sobre recursos existentes e atividades de um processo exemplo. A política de instanciação *Pol_Inst3* para recursos consumíveis foi criada no APSEE e sua definição é ilustrada pela figura 7.22 a seguir.

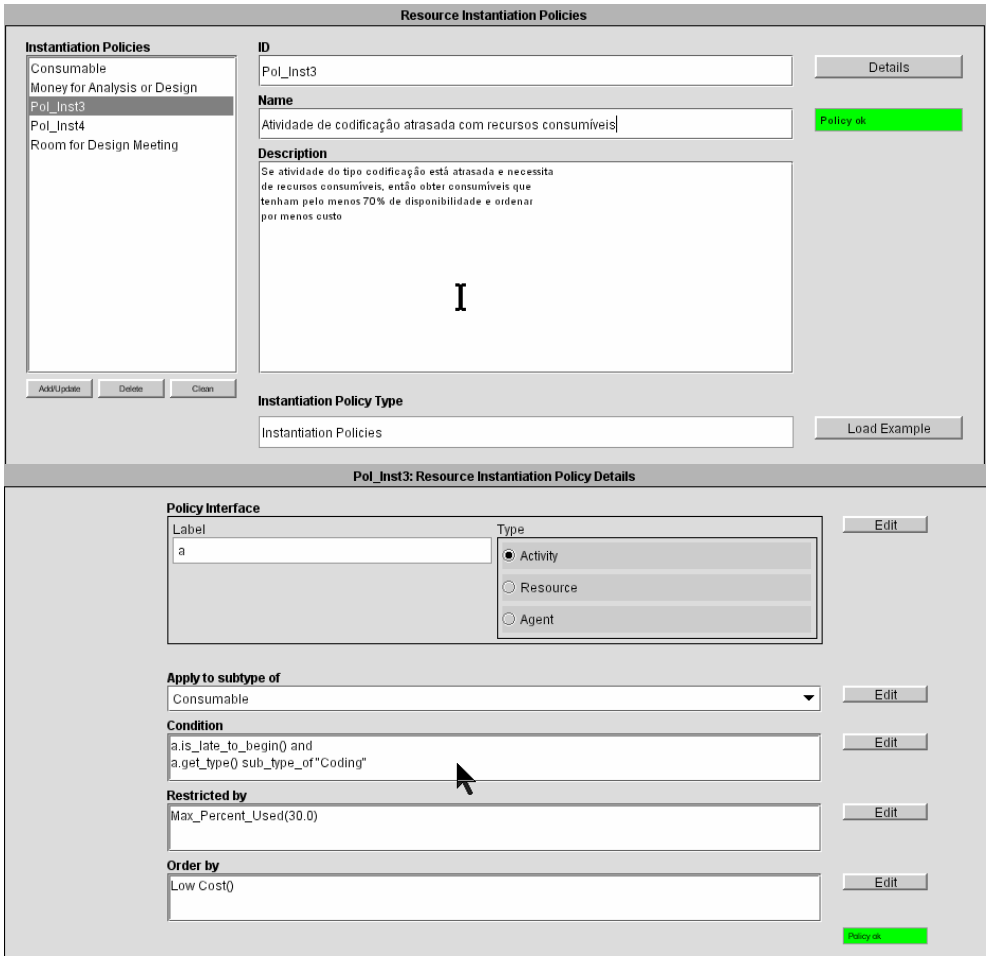


Figura 7.22: Política de instanciação para recursos consumíveis.

Para ilustrar o uso dessa política, foi criada uma atividade em um processo chamada *ActPolInst3* do tipo *Coding* (como requer a política) e foi habilitada essa política na atividade. A atividade requer três tipos de recurso: *Money*, *Paper e Printer*, como mostrado na figura 7.23 e seu início está atrasado. Será testada a instanciação do recurso do tipo Papel (*paper*), por ser consumível.

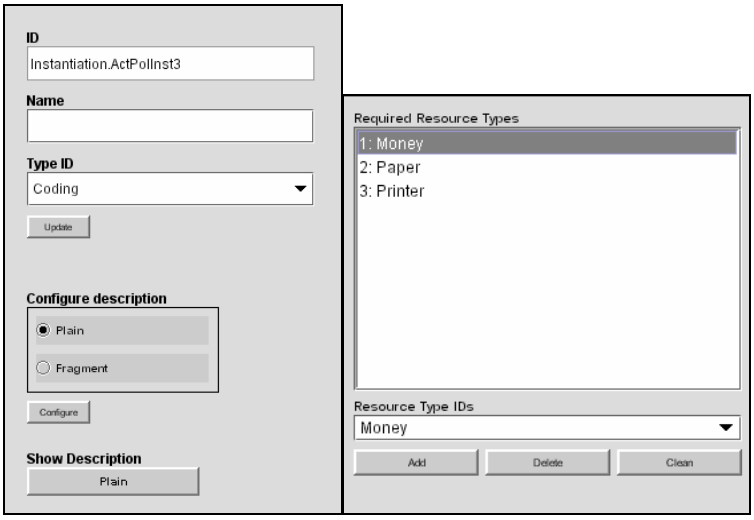


Figura 7.23: Atividade com política *PolInst3* habilitada e seus recursos requeridos.

A situação dos recursos na organização é mostrada na figura 7.24.

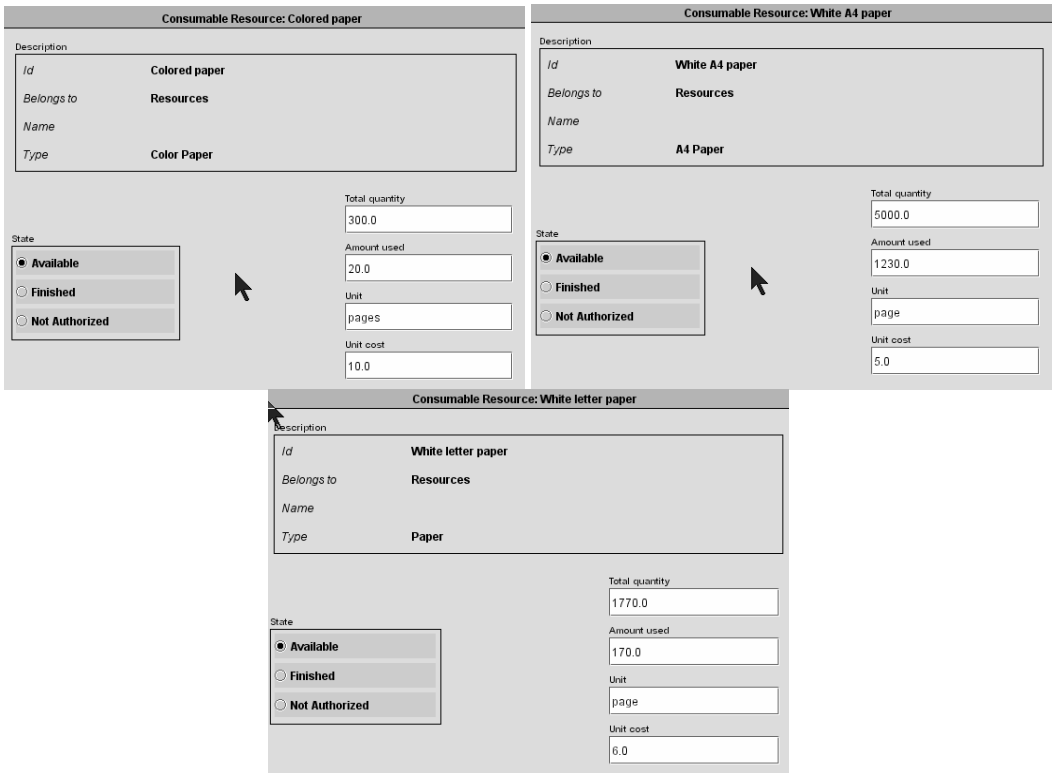


Figura 7.24: Situação dos recursos do tipo papel na organização.

A figura 7.25 mostra a geração de sugestões para o recurso Paper na atividade criada, onde a quantidade requerida do recurso é 100 (*amount needed*). É importante notar que foram sugeridas as instâncias de papel existentes ordenadas pelo seu preço (o preço usado para ordenar é mostrado entre parênteses após o nome do recurso no quadro

de sugestões). Caso fosse solicitada uma quantidade de papel maior do que a existente, não haveria sugestão.

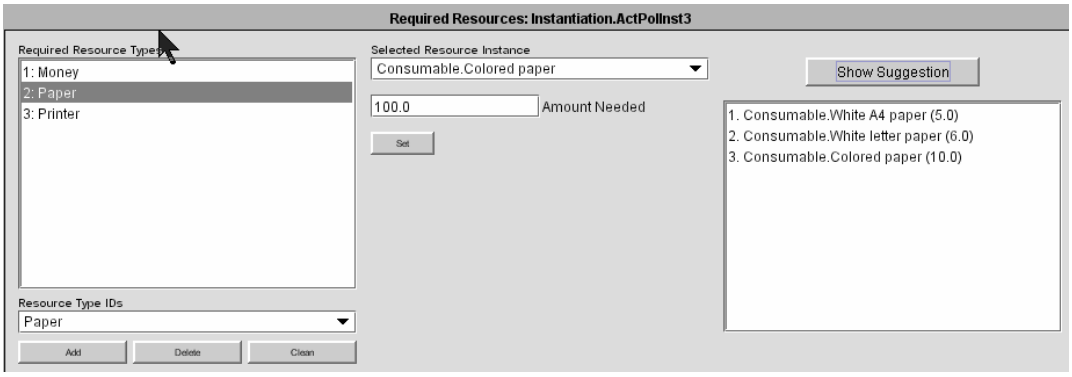


Figura 7.25: Geração de sugestões para recursos consumíveis do tipo papel na atividade exemplo.

Outro exemplo de instanciação será mostrado para um recurso do tipo exclusivo. A política de instanciação "*Room for design meeting"* é mostrada na figura 7.26 e serve para instanciar salas adequadas para atividades do tipo reunião de projeto. A sala adequada, segundo a política, deve ter mais de 10m^2 e deve ter como componente um computador. A ordenação deve ser pela métrica Conforto - Room Confort (a métrica deve existir na base de métricas). É importante ressaltar que a métrica "*Room Confort*" foi criada pelo usuário e qualquer métrica criada pode ser usada para restringir ou ordenar sugestões de instanciação.

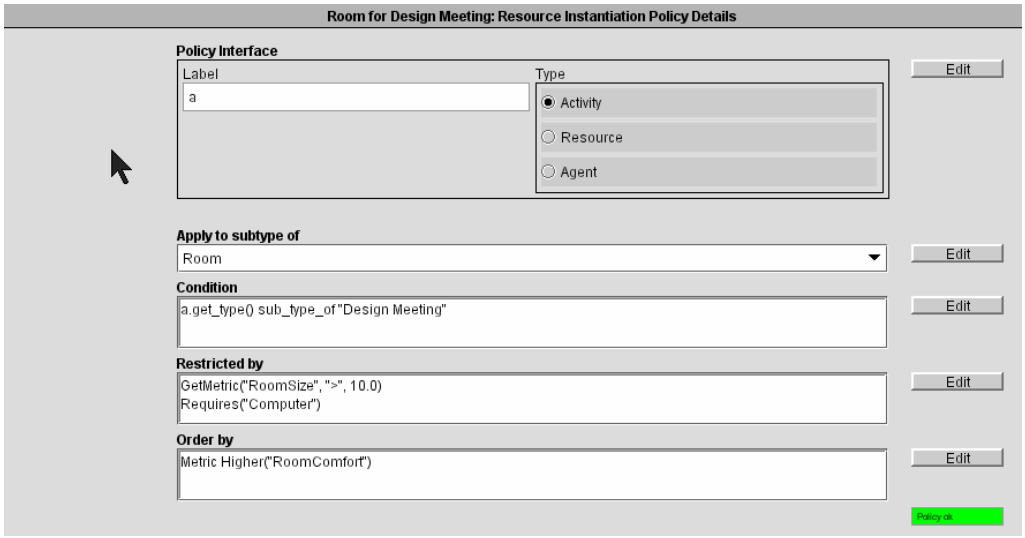


Figura 7.26: Descrição da política "*Room for Design Meeting*".

A atividade criada requer uma sala e é do tipo "*design meeting*". A geração de sugestões para a sala requerida é mostrada na figura 7.27 a seguir. A única sala sugerida é *Room 1* que possui nível de conforto igual a 6. O motivo pelo qual a sala *Room 2* não foi sugerida, mesmo em segundo lugar, foi o uso da restrição *GetMetric("RoomSize",>, 10.0)* na política pois *Room 2* possui somente 9 m^2.

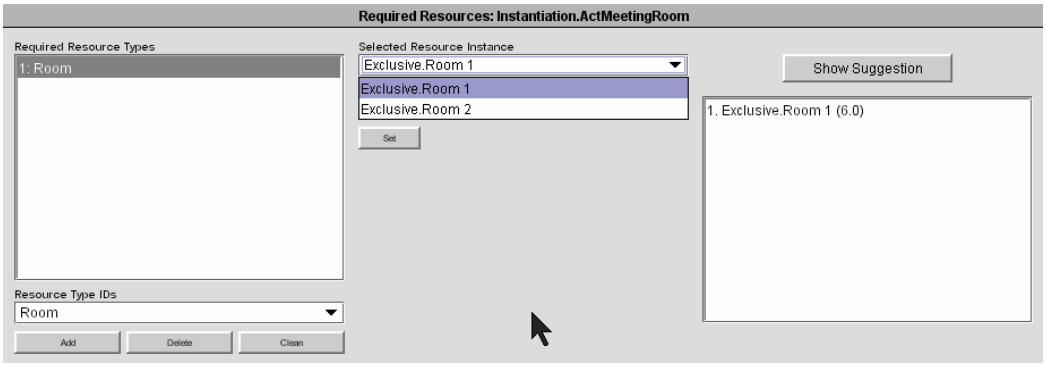


Figura 7.27: Geração de sugestões para sala com a política "*Room for Design Meeting*".

8 CONCLUSÕES

Este trabalho apresentou uma proposta de mecanismos de gerência de processos de software e meta-modelo associado que têm como objetivo principal o aumento da flexibilidade e do nível de automação na execução de processos. A solução fornecida para o problema investigado é um primeiro passo em direção ao aumento da qualidade na gerência de processos de software. Porém, acredita-se que o meta-modelo proposto juntamente com os componentes da sua arquitetura constituem uma contribuição importante para o estado da arte tecnológico atual.

Foi proposta uma linguagem de modelagem associada a políticas de instanciação e um mecanismo de execução de processos, sendo que todos os mecanismos foram especificados formalmente. A partir da especificação foi construída a implementação de um protótipo, o qual permitiu que fossem testados e avaliados alguns estudos de caso. As seções a seguir aprofundam a discussão das contribuições, dos resultados da pesquisa, dos trabalhos relacionados e das questões em aberto e trabalhos futuros.

8.1 Resumo das Contribuições

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

- O texto propõe um conjunto de requisitos para a execução de processos de software, apresentados no capítulo 2. Acredita-se que tais 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 e Sistemas de *Workflow*. Tais requisitos foram levantados em vários trabalhos realizados durante o doutorado e publicados como relatórios de pesquisa publicados em (LIMA REIS, 2000) e (SOUZA et al., 2001);
- Foi proposto um meta-modelo unificado, chamado APSEE, que integra diferentes componentes para atender os serviços e ferramentas relacionados com a gerência de processos. O meta-modelo apresentado no capítulo 3 avança em relação aos existentes principalmente por seu nível de detalhamento e abrangência. Além disso, 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;

- Foi proposta uma linguagem de modelagem visual chamada APSEE-PML baseada no meta-modelo unificado, que permite a definição de processos em diferentes níveis de abstração e integra a habilitação de Políticas. A linguagem foi apresentada no capítulo 4 através de seus construtores gráficos e foi especificada formalmente usando a abordagem proposta por Bardohl (2000) (através de gramáticas de grafos). A especificação define a sintaxe da linguagem e inclui regras para a detecção de conflitos em tempo de modelagem. A linguagem de modelagem foi projetada para atender os requisitos de flexibilidade do mecanismo de execução e permitir a integração com diferentes componentes relacionados com etapas específicas do meta-processo de software (destacando-se a integração com os componentes voltados para reutilização e visualização de processos - APSEE-REUSE (REIS, 2002f) e APSEE-Monitor (SOUZA, 2003), respectivamente). Diferentes aspectos da linguagem de modelagem foram apresentados em (LIMA REIS et al., 2001c; 2002a; 2002d; 2002e);
- No meta-modelo proposto foram definidas de forma precisa as propriedades dos recursos e agentes. Esse detalhamento de informações foi útil para distinguir tais componentes tanto na linguagem de modelagem quanto na proposta de políticas de instanciação, pois na maioria dos PSEEs esses componentes são tratados da mesma forma;
- Foi proposto um mecanismo para auxiliar a instanciação de processos de software através de políticas definidas pelo usuário. Este mecanismo foi desenvolvido para auxiliar no aumento da flexibilidade da execução de processos. Utilizando a definição das políticas e as informações acerca da organização (recursos, pessoas) e das necessidades do processo, o mecanismo gera sugestões de instanciação adequadas a cada atividade e situação. A definição de Políticas de Instanciação foi feita de forma genérica com o objetivo de facilitar sua reutilização em diferentes modelos de processo. Além disso, as políticas constituem um componente de primeira ordem na modelagem de processos no APSEE, estendendo a APSEE-PML com construtores para armazenar informações úteis sobre gerência de processos e podendo ser habilitadas em diferentes níveis do processo. A linguagem de políticas proposta possui semântica algébrica, a qual foi usada para construir a implementação, e que tem o potencial de permitir, no futuro, o desenvolvimento de extensões e adaptação a outros ambientes. A principal contribuição das Políticas de Instanciação é a flexibilidade provida para o gerente que está encarregado de realizar instanciação em processos complexos, pois o mecanismo provê uma solução automatizada de apoio a uma tarefa que consome tempo. Políticas de Instanciação e o Gerenciamento de Recursos no sistema APSEE foram publicados em (LIMA REIS et al., 2001a; 2002b; 2002c);
- Foi proposto um mecanismo de execução de processos baseado no meta-modelo unificado e que teve como principal objetivo aumentar o grau de

flexibilidade na execução dos processos. Esse mecanismo obedece a semântica para a APSEE-PML. Tal semântica foi especificada através do uso do paradigma PROSOFT-Algébrico combinado com Gramática de Grafos. A especificação formal permitiu trabalhar em alto nível de abstração e permitiu definir as conseqüências das transições de estados de forma compacta. Além disso, a prototipação do mecanismo de execução foi realizada a partir da especificação formal e demonstrou vantagens nessa abordagem. Nessa contribuição destaca-se a importância dada aos requisitos resultantes do envolvimento humano no processo, que é refletida na capacidade de executar processos incompletos, permitir execução de processos colaborativos (por exemplo, com dependência *start-start*), apoiar a modificação dinâmica de processos, além de fornecer um formalismo gráfico e executável que permite a modelagem e acompanhamento de processos. O mecanismo proposto permite definir *a priori* as alternativas para o fluxo de execução dependendo da situação corrente (conexão condicional *Branch*) ou de informações sobre processos executados anteriormente. Além disso, é possível definir condições para controlar a repetição de atividades (conexão de *feedback*), dependendo da situação corrente ou histórico do processo. As conexões de *feedback* também podem ser usadas para modificação dinâmica de processos quando essa mudança ocorre em porções do processo (fragmentos) que estão sendo executados: a abordagem de reexecução do fragmento (Solução C apresentada na seção 2.4.2, página 41) pode ser usada através da ativação de um *feedback* para o início do processo. Essa contribuição foi discutida nos artigos publicados em (LIMA REIS et al., 2001c; 2002a; 2002d; 2002e);

- Um protótipo que implementa o meta-modelo, a linguagem de modelagem, as políticas de instanciação e o mecanismo de execução foi construído para validar as contribuições do trabalho. *Templates* propostos por Reis (2002f), onde são modelados processos reais, foram instanciados e executados no protótipo visando avaliar as características do mecanismo de execução e demonstrar a exeqüibilidade do modelo.

8.2 Análise do Modelo Proposto

8.2.1 Considerações sobre os Formalismos Utilizados

O modelo APSEE foi apresentado através da combinação de diferentes formalismos. No capítulo 3, foram utilizadas as notações para pacotes e classes fornecidas pela UML, tendo como objetivo auxiliar no entendimento da estrutura principal do meta-modelo, por ser uma notação bastante conhecida. O capítulo 5, por sua vez, descreveu os detalhes do modelo proposto usando o PROSOFT-Algébrico e Gramáticas de Grafos, sendo que estas foram usadas para descrever a sintaxe e a semântica da APSEE-PML. Diferentes formalismos foram usados em diferentes partes do modelo para que fosse possível explorar a descrição da arquitetura do modelo em alto nível, sem preocupação com detalhes específicos das especificações formais e da implementação dos protótipos desenvolvidos.

Os formalismos adotados foram utilizados praticamente sem apoio automatizado, ou seja, não estava disponível um editor de especificações do PROSOFT Algébrico (apenas de classes), nem tampouco uma ferramenta de gramática de grafos que suportasse o modelo proposto. Atualmente já está disponível uma ferramenta proposta por Rangel (2003) que permite editar especificações algébricas no PROSOFT e traduzi-las para OBJ a fim de testar seu funcionamento antes da implementação.

Vale ressaltar a experiência do trabalho com relação ao uso de Gramáticas de Grafos, usada para especificar a sintaxe, verificação de consistência e execução de processos no APSEE. Foi possível verificar a facilidade de entendimento do modelo com vistas à implementação, principalmente considerando a complexidade da especificação do meta-modelo e das transições de estado do processo. Além disso, a adoção de tal formalismo permitiu a realização de algumas simulações para observar o comportamento de algumas regras. Tais simulações foram testadas na ferramenta AGG descrita em (AGG, 2002; EHRIG, 1999), porém devido ao grande número de regras a testar (aproximadamente 400), e também considerando que o objetivo da adoção do formalismo no trabalho era o de desenvolver a implementação do protótipo, não foi usada nenhuma ferramenta automatizada de interpretação de regras de gramáticas de grafos. Entrentanto, após a construção do protótipo totalmente baseado nas regras especificadas, foi possível realizar alguns estudos de caso de forma satisfatória.

Algumas dificuldades e situações específicas foram encontradas no uso de gramáticas de grafos, como por exemplo:

- A falta de padronização nas linguagens para a construção de grafos-tipo e regras. Apesar dos avanços na área de transformações de grafos, é difícil encontrar abordagens que utilizem uma notação uniforme. Cada autor/grupo de pesquisa utiliza a notação que acha mais apropriada para o problema sendo modelado. Para construir o grafo-tipo e as regras, o trabalho inspirou-se nas propostas de Bardohl (2000; 1999) e Gruner (2000), dentre outros, e optou-se por utilizar os mesmos símbolos da linguagem APSEE-PML para facilitar o entendimento das regras;
- As principais dificuldades com relação às regras foram com relação aos efeitos de regras de transformação de estados em conexões múltiplas (*Branch* e *Join*). Quando as transformações deveriam atingir todos os componentes subseqüentes e não era conhecido o número de componentes, foi necessário acrescentar símbolos que representassem sinais marcadores para outras regras serem ativadas. Tais sinais não fazem parte da linguagem mas tornaram possível sua construção, são eles: *redo*, bloqueio, início de ciclo de *feedback*, fim de *feedback*, *true* e *false*;
- Outra dificuldade foi com o uso de NACs (condições negativas de aplicação), que permitem especificar o que não deve estar presente no modelo sendo interpretado para que a regra seja aplicada. Por exemplo, foi difícil expressar com NACs situações como: "*Não existe atividade que não tenha o sinal de redo*", ou de outra forma, *todas as atividades têm o sinal de redo.* Entretanto, não existia um símbolo que representasse todas as atividades. A solução foi o uso de *set nodes*, também usado na ferramenta PROGRES (SCHÜRR et al., 1995), que permite que sejam representadas

todas as instâncias existentes de um determinado tipo. Outro cuidado com a especificação das regras que pode causar confusão no entendimento ocorre com relação a diferença entre definir várias NACs separadas para a mesma regra, ou juntar todas as negações em uma só NAC. No primeiro caso, nenhuma das situações especificadas nas NACs pode ocorrer (trata-se de conjunção) e no segundo caso, a interpretação é de que as condições negativas não podem ocorrer ao mesmo tempo (disjunção);

- A falta de um mecanismo de herança na literatura para a construção de grafos-tipo foi superada pela criação do relacionamento *Is-a*. Uma proposta inicial para o uso de herança foi apresentada em (GRUNER, 2000) e a confirmação de que o uso de herança ainda não foi completamente estabelecido na área de gramática de grafos veio de Heckel (2002). A herança simplificou sobremaneira o grafo-tipo e principalmente diminuiu a quantidade de regras. Foi necessário, entretanto, que definíssemos a prioridade de avaliação das regras caso duas ou mais regras diferentes pudessem ser ativadas para a mesma situação por causa da herança: a prioridade foi dada para o caso mais específico (por exemplo, uma regra que contém *Activity_Normal* é avaliada antes de uma com o tipo *Activity*, mais geral).

Definir regras de transformação livres de conflito é uma tarefa desafiadora, principalmente considerando-se o grande número de estados possíveis para um modelo de processo. Entretanto, as NACs ajudaram a tratar o problema pois permitiram descrever soluções compactas para os casos encontrados. Além disso, os sinais marcadores foram usados para sinalizar quais conjuntos de regras são mais apropriados para cada situação. Com isso, o entendimento das regras foi melhorado, e a implementação das mesmas no protótipo propiciou importante *feedback* sobre a sua construção. Como resultado, foi possível observar o comportamento das regras nos vários estudos de caso testados no protótipo.

8.2.2 Escalabilidade

A análise da escalabilidade em PSEEs é baseada na possibilidade de prover assistência automatizada para gerenciar processos em organizações de grande porte. Apesar disso, organizações de pequeno e médio porte podem se beneficiar dessa tecnologia.

A construção do meta-modelo APSEE foi influenciada por tendências recentes na área de linguagens de programação, pois é fornecido apoio à separação explícita de detalhes. Assim, para modelar processos os usuários trabalham com modelos simplificados que podem ser detalhados separadamente, como por exemplo, na habilitação de Políticas, assim como os processos podem ser decompostos em vários níveis.

O uso de hierarquias de tipos é um aspecto importante do modelo que contribui para a sua escalabilidade. Políticas de instanciação podem ser reutilizadas em diferentes processos, pois referenciam tipos ao invés de instâncias, portanto, mesmo que o número de processos aumente, as mesmas políticas continuam podendo ser usadas. Entretanto,

como observado por Reis (2002f), as hierarquias de tipos podem ser de manutenção complexa para organizações e/ou processos grandes porque a profundidade e a largura das árvores de tipos tendem a aumentar à medida que o tamanho e quantidade de processos modelados aumentem.

O modelo proposto neste trabalho está preparado para atender organizações de grande porte onde são executados vários processos simultâneos que concorrem pelos mesmos recursos. A definição rigorosa dos recursos e agentes torna possível controlar sua alocação e o mecanismo de instanciação fornece o apoio adequado para auxiliar a seleção desses componentes nos processos. Considerando que o aumento do número de processos concorrentes torna a tarefa de instanciação difícil para um gerente, o modelo proposto fornece um importante auxílio nesse sentido, facilitando a gerência de grande quantidade de processos simultâneos.

8.2.3 Adaptabilidade

Em princípio, os construtores do meta-modelo proposto podem ser adaptados para diferentes PSEEs. Isso pode ser facilitado devido ao fato de que é fornecida semântica formal, que provê descrição precisa, em alto nível de abstração do modelo e independente de linguagem de implementação. Entretanto, o modelo de processos adequa-se mais a ambientes e linguagens que adotam o paradigma baseado em redes de atividades. Investigação adicional deve ser realizada para avaliar a adaptação para linguagens que adotam diferentes paradigmas de modelagem.

Outra possibilidade de adaptação da solução proposta envolve sua aplicação em problemas diferentes, não somente para processos de software, como por exemplo processos de negócio e processos de cursos (aplicação na Educação). De fato, processos de negócio são bastante próximos de processos de software e possuem problemas parecidos tratados no modelo proposto. Uma diferença nesse caso está na adoção de mudanças dinâmicas *ad-hoc*, discutidas na seção 2.4.2, não serem suficientes para tratar todos os tipos de mudanças dinâmicas de processos de negócio: enquanto processos de software são únicos e suas mudanças não necessariamente afetam outros processos em execução, os processos de negócio ocorrem para vários casos e quando há mudança, essa mudança tem que lidar com os casos atuais e os que ainda não executaram. Entretanto, os mecanismos de gerência de processos juntamente com a abordagem de mudanças dinâmicas da solução proposta podem apoiar processos de negócio.

Quanto a processos de cursos, podem ser criadas analogias a partir dos conceitos de gerência de processos para gerência de cursos. No grupo PROSOFT, existe uma proposta de criar essas analogias e investigar essa possibilidade (DAHMER, 2000). Além disso, existem ambientes que utilizam workflow para gerência de cursos, como por exemplo o Flex-el (FLEX-EL, 2002). Sem deixar de considerar a necessidade de investigação, podem ser visualizadas analogias compatíveis para o modelo proposto neste trabalho. Por exemplo, o ambiente pode auxiliar a gerência de cursos presenciais e a distância por conter elementos que podem ser mapeados diretamente para o domínio de cursos, tais como: agentes para alunos e professores, atividades de processo para atividades de curso, processos de software para processos de curso. Uma interface específica para o problema poderia ser criada como uma camada sobre o modelo de

gerência de processos respeitando as necessidades diferenciadas de interação dos usuários desse domínio.

8.3 Trabalhos Relacionados

De acordo com Garg e Jazayeri (1996), devido à falta de um padrão aceito para o projeto de arquitetura de PSEEs, é difícil comparar e avaliar experiências apresentadas na literatura. A maioria dos PSEEs existentes tende a concentrar-se somente nas fases de modelagem e execução de processos. Isso constitui um obstáculo para o fornecimento de suporte integrado para gerência de processos. Essa seção faz um levantamento geral de como o trabalho proposto está relacionado com outras soluções encontradas no contexto da execução flexível de processos e do auxílio à instanciação.

8.3.1 Trabalhos relacionados sobre Execução de Processos

Acredita-se que as principais diferenças entre o modelo de execução proposto e outros ambientes encontrados na literatura são originadas no meta-modelo adotado. O meta-modelo do APSEE foi construído para permitir a integração de vários serviços de gerência de processos, incluindo modelagem, execução, reutilização, simulação, visualização, descoberta de conhecimento sobre processos, coleta automática de métricas, verificação estática dos processos através de políticas, instanciação e resposta a eventos da execução (políticas dinâmicas). Apesar de nem todos os serviços estarem disponíveis atualmente, o meta-modelo já foi construído tendo-os como alvo. Com isso, a linguagem de modelagem foi proposta a partir da reunião de vários construtores encontrados na literatura, sendo que alguns foram aperfeiçoados para aumentar a flexibilidade da linguagem, como é o caso do *feedback* e das conexões múltiplas com condições lógicas associadas.

Do ponto de vista dos construtores da linguagem de modelagem que permitem essa flexibilidade é importante mencionar como a abordagem proposta nesse trabalho se diferencia da adotada pelo ambiente Trigsflow (KAPPEL et al., 1998). Trigsflow provê uma linguagem de modelagem visual que suporta dependências *end-start*, *start-start* e *end-end*, através de conectores *Branch* e *Join*. Esses construtores são semanticamente mapeados para regras ECA (Evento-Condição-Ação) textuais que são interpretadas pelo mecanismo de execução do ambiente. Entretanto, o Trigsflow não permite a inclusão e avaliação de condições lógicas nas conexões (essas condições podem definir o caminho do processo baseado na situação corrente). Além disso, conforme apresentado em (KAPPEL et al., 1998), conexões de *feedback* não estão disponíveis.

O ambiente Dynamite, já citado neste trabalho, foi desenvolvido como uma aplicação de técnicas de reescrita de grafos através do uso da linguagem PROGRES (KRAPP, 1998; SCHÜRR et al., 1995). Entretanto, essa especificação não detalha interação com o usuário do sistema, focalizando apenas a consistência interna do modelo através da observação dos fluxos de dados e de controle. O meta-modelo adotado no Dynamite não inclui detalhes da organização, e a única conexão de controle suportada é a equivalente a conexão simples *end-start* do modelo proposto neste trabalho, ou seja, as conexões múltiplas e as dependências *end-end* e *start-start* não são suportadas pelo modelo. Devido à simplicidade do meta-modelo, as regras relacionadas à execução de processos no Dynamite, apresentadas em (KRAPP, 1998), restringem-se

à definição do uso de artefatos por instâncias de atividades e pouco suporte é fornecido para mudanças dinâmicas no processo. O APSEE avança em relação ao Dynamite em vários aspectos. O APSEE possui um conjunto mais rico e abrangente de construtores, o que aumenta também a complexidade da semântica associada. Além disso, o *feedback* no APSEE é automatizado permitindo que o usuário especifique a condição de retorno, enquanto no Dynamite o usuário tem que inserir uma conexão de *feedback* e para que ele seja ativado, deve informar explicitamente ao sistema quais atividades devem ser refeitas.

Uma PML que propõe os vários tipos de conexões propostos no APSEE é a linguagem Promenade (RIBÓ; FRANCH, 2000; 2001). Porém, sua conexão de *feedback* é similar a do Dynamite (não possui condição associada) e suas conexões múltiplas não permitem escolher caminhos alternativos a partir de condições associadas a cada escolha (como é o caso do *Branch OR/XOR*). A semântica dos fluxos de controle de Promenade não foi encontrada na literatura nem citada pelos autores. Além disso, não há mecanismo de execução para a linguagem.

Quanto ao suporte a modificações dinâmicas (ou suporte à evolução), a abordagem usada no APSEE permite modificação em partes do processo que ainda não executaram, nas que já executaram e nas que estão executando. Geralmente, os problemas de consistência ocorrem mais no segundo e terceiro casos. As regras especificadas para o mecanismo de execução permitem alguma mudança nas partes que estão executando desde que não afetem a consistência do modelo ou possam ser tratadas (por exemplo, adicionar uma conexão entre atividades e adicionar um agente à atividade) mas não permitem modificações em partes já executadas, porque isso tornaria o processo inconsistente. A solução proposta neste trabalho para tratar esses casos é criar uma conexão de *feedback* que volte ao início do processo e solicitar falha na atividade que está em execução (a falha ativa o *feedback*). Nesse caso o fragmento de processo atingido é reiniciado e assim, pode sofrer mudanças. Essa abordagem atende às soluções B e C propostas por Arbaoui et al (2002) citadas na seção 2.4.2 sobre Modificações Dinâmicas. Os ambientes analisados em (ARBAOUI et al., 2002), incluindo ADELE/TEMPO/APEL, LITTLE JIL, PROCESS Wise, LEU, PEACE/PEACE+, PIE e OZ, somente suportam a solução A (mudança apenas em partes ainda não executadas). Portanto, o suporte a modificações dinâmicas proposto representa um avanço em relação à PSEEs existentes encontrados na literatura.

8.3.2 Trabalhos Relacionados sobre Políticas de Instanciação

Políticas de Instanciação foram propostas neste trabalho para dar auxílio no momento da escolha de agentes e recursos para atividades. Apesar do conceito de políticas não ser novo no campo de automação de processos (FEILER; HUMPHREY, 1993), a maioria dos trabalhos neste tópico focaliza a verificação de propriedades dinâmicas dos eventos ocorridos nos processos (KAPPEL, 1998). A maioria das abordagens que tratam de gerência de atividades (WfMS e PSEEs) não distingue entre agentes e recursos (PODOROZHNY, 1999). Agentes são normalmente tratados como recursos de apoio, e são alocados para atividades, enquanto que os recursos de apoio nem sempre possuem informação detalhada sobre seu estado, impedindo verificação automática e adoção de estratégias para alocação. Mesmo quando recursos e agentes são

definidos de forma independente, seus modelos não são descritos em nível de detalhe suficiente para permitir raciocínio adequado acerca de estratégias de alocação.

Plekanova (1997; 1998; 1999) propõe alocação de recursos humanos através de programação linear usando informações sobre recursos disponíveis e suas capacidades (pessoas e habilidades) e recursos requeridos. Contudo, para aplicar essa abordagem é necessário definir o processo completamente antes da execução, o que inibe a adoção da abordagem em PSEEs com mudanças dinâmicas. Na área de processos de software, alguns trabalhos tratam a necessidade de modelar e gerenciar recursos. Dois exemplos são o MVP-L (ROMBACH; VERLAGE, 1993) e o APEL (ESTUBLIER et al., 1997). MVP-L permite modelagem de recursos mas não controla o acesso aos mesmos. Da mesma forma, APEL trabalha com aspectos organizacionais de forma ortogonal às questões de processo (com uma abordagem similar a proposta deste trabalho), porém não incorpora aspectos de escalonamento. Convém destacar que essas e outras abordagens tratam o conceito de recursos de forma generalizada, incluindo modelagem e gerência de pessoas e artefatos. Em uma avaliação de PSEEs apresentada em (FINKELSTEIN, et al., 1994), Lonchamp aponta que o único ambiente avaliado a tratar o conceito de recursos sem denotar pessoas ou dados é o ambiente E3 (JACCHERI et al.,,1995; JACCHERI, 1996; 1998), e mesmo assim, apenas trata o conceito de alocação de tempo e máquinas para atividades.

Um trabalho que se preocupa com todos os tipos de recursos tratados nesta tese é o do grupo do Prof. Leon Osterweil (LERNER et al., 2000; PODOROZHNY et al., 1999) na *Massachussets University*. Apesar de adotar o conceito genérico de recursos, tratando da mesma forma pessoas, artefatos e recursos de apoio, compartilham conceitos adotados no modelo de recursos, como tipos e instâncias de recursos, reservas para quaisquer recursos, além de relacionamentos de composição (*belongs_to*) e requisitos de recursos (*requires*). Uma desvantagem da proposta de Osterweil é que, além de não permitir instanciação automática, devido ao tratamento igual dado a tipos de recursos diferentes, o modelo permite que um recurso humano seja composto de uma máquina por exemplo e, para contornar esse problema, são propostos critérios de participação em cada grupo de recurso. Esses critérios são regras que devem ser satisfeitas para que o recurso seja de um determinado tipo ou tenha um determinado relacionamento, o qual adiciona maior complexidade ao uso do modelo.

Um conjunto genérico de interdependências entre atividades e mecanismos de coordenação de atividades (modelados com Redes de Petri) foram propostos no trabalho de Raposo em (RAPOSO, 2000b; RAPOSO et al., 2000a). Tais mecanismos tratam dependências temporais e de gerenciamento de recursos entre atividades e podem ser utilizados em ambientes de coordenação de atividades. A abordagem utilizada no presente trabalho é similar quanto à separação entre tarefas e interdependências. Entretanto, para Raposo, o conceito de recurso é usado de forma mais ampla, denotando artefatos, recursos de apoio e pessoas. Os tipos de dependência para esses recursos propostos por Raposo são: compartilhamento (por exemplo, edição cooperativa de um documento), simultaneidade de uso (por exemplo, canais de comunicação e máquinas) e volatilidade de recursos (equivalente aos recursos consumíveis propostos neste trabalho). Portanto, o presente trabalho propõe dependências similares às de Raposo, porém trata de forma distinta pessoas e recursos de apoio, fornecendo para estes últimos uma classificação em: consumíveis, exclusivos ou compartilhados.

Huang e Shan (1999) dos Laboratórios Hewlett-Packard propõem uma abordagem conceitualmente similar às Políticas de Instanciação provendo uma linguagem para automatizar alocação de pessoas em projetos de *workflow*. Entretanto, sua linguagem restringe na própria política para qual atividade ela é válida, inibindo seu potencial de reutilização. Existem também políticas que expressam requisitos baseados em avaliação de condições lógicas para selecionar recursos. Entretanto, não são providas funções para extrair informações a partir de métricas, nem existem critérios de ordenação. Outra desvantagem é que a abordagem de políticas proposta por Huang e Shan somente se aplica a pessoas, não sendo útil para instanciar recursos de apoio.

No país, algumas propostas foram desenvolvidas no âmbito de ambientes de modelagem e gerência de processos de software. Dentre elas, o ambiente ExPSEE (GIMENES, 2000) que permite definição precisa e instanciação de atividades do processo de software em relação a artefatos, atores (agentes) e ferramentas utilizadas por atividades e a Estação TABA (TRAVASSOS, 1994), que possui um modelo recente para definição de processos que permite definir recursos humanos e ferramentas, dentre outros conceitos (MACHADO, 2000).

8.4 Questões em Aberto e Trabalhos Futuros

O presente trabalho representa um passo no sentido de fornecer soluções para aumentar a flexibilidade da execução de processos. Várias questões ainda estão em aberto, definindo pontos que devem ser explorados em outros trabalhos que sigam a mesma linha.

Um dos tópicos que necessita de exploração é a avaliação empírica da influência do modelo proposto no aumento da qualidade do processo e do produto. 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, uma importante atividade a ser realizada é 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.

O conjunto de dependências entre atividades proposto (conexões simples, de *feedback* e múltiplas com dependências *end-start, start-start* e *end-end*) não pretende ser completo. Outros tipos de dependência podem ser incorporados ao modelo como trabalhos futuros. Por exemplo, o trabalho de Raposo (2000a; 2000b) propôs dependências temporais entre atividades que ainda não são fornecidas pelo APSEE, como por exemplo: a atividade A deve terminar junto com a atividade B; A deve ser executada enquanto B estiver executando; A e B devem ter algum instante de intersecção em sua execução.

A decisão de especificar o modelo para um ambiente existente (no caso, o PROSOFT) trouxe benefícios, como por exemplo: é possível editar objetos (artefatos) cooperativamente, o ambiente é distribuído, a integração com ferramentas existentes no ambiente é imediata, dentre outros. Porém, a atual limitação da interoperabilidade do sistema com ferramentas externas é um tópico a ser tratado em trabalhos futuros. Propostas de *middleware* para processos de software a fim de aumentar a

interoperabilidade entre PSEEs estão sendo apontadas como soluções para a próxima geração de PSEEs (GRUHN, 2002).

A edição cooperativa de processos, como ocorre no ambiente Serendipity II (GRUNDY, 1998), não é tratada no presente trabalho. Apesar da execução do processo ser distribuída com os agentes atuando remotamente, a coordenação do processo é centralizada. Investigações adicionais são necessárias para definir como o processo pode ser executado de forma distribuída, isto é, reconciliando as modificações introduzidas nas várias réplicas de um processo.

Atualmente, políticas de instanciação contraditórias podem ser habilitadas no mesmo processo. 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. O mecanismo de interpretação de políticas resolve o conflito através da escolha de uma política a ser aplicada em cada situação através da análise da compatibilidade entre o tipo requerido (de recursos ou de agentes) e o tipo da política. E mesmo que duas ou mais políticas sejam igualmente compatíveis, a prioridade é dada a políticas mais locais (habilitadas em atividades normais, por exemplo, em detrimento das habilitadas no processo). 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 (MOFFETT; McDERMID, 1994; COLE et al., 2001). 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 de conflitos e análise de consistência (Steem citado por Cole et al. (2001)). Em virtude da base semântica algébrica aqui fornecida para a definição e interpretação de Políticas de Instanciação, vislumbra-se como trabalho futuro a realização de investigações adicionais acerca da análise e prevenção automática de conflitos.

O modelo proposto registra eventos ocorridos no processo e nos seus componentes. Esse registro de eventos é um requisito básico da execução de processos e facilita a criação de diversos mecanismos, não propostos neste trabalho, tais como:

- **Descoberta de conhecimento após execução**: Pode ser construído um mecanismo de coleta de informações sobre o processo usando técnicas de descoberta de conhecimento e mineração de dados (FAYYAD, 1996a; 1996b; 1996c; WANG, 1999). O conhecimento sobre processos de engenharia de software executados é complexo, porém é possível enumerar *a priori* qual o conhecimento útil a ser obtido a partir dessa base de informações. A tecnologia de mineração de dados é útil principalmente quando se quer obter conhecimento que provavelmente não foi previsto antes da consulta. Tal mecanismo poderia trazer como benefícios a geração de métricas sobre o processo e de regras de comportamento dos agentes envolvidos com o processo. Essas informações são de extrema valia para simulação de processos e para planejamento das atividades;

- **Execução de Políticas Dinâmicas**: De forma análoga a Políticas de Instanciação, Políticas Dinâmicas poderiam ser habilitadas em pontos do processo e levar em consideração os eventos ocorridos durante execução. Ao utilizar uma estrutura de regras ECA (Evento-Condição-Ação), essas políticas seriam ativadas somente se estiverem habilitadas para uma atividade e se o evento que é tratado pela política ocorre e é inserido na lista de eventos da atividade. Os benefícios desse mecanismo são a possibilidade de tratar exceções imediatamente no processo e descrever estratégias do usuário para tratar todos os tipos de eventos. A ação tomada por esse mecanismo pode tanto modificar o processo quanto simplesmente avisar o gerente da ocorrência do evento. Políticas dinâmicas também poderiam ser habilitadas em artefatos. Um exemplo de política dinâmica pode estabelecer que se houver falha em uma atividade que deveria produzir um determinado artefato, então seja criada uma conexão de *feedback* para a atividade que produz o artefato de entrada desta. Regras ECA têm sido usadas em ambientes de workflow e PSEEs como paradigma de execução combinado ao tratamento de eventos (KAPPEL et al., 1998; JOERIS; HERZOG, 1999).

Outro trabalho futuro diz respeito ao requisito R15, citado na seção 2.5.3, que diz que o mecanismo de execução deve adaptar sua interação de acordo com o perfil do usuário. Nesse caso o tipo de orientação oferecida pode ser ativa ou passiva, dependendo do agente, das suas habilidades no momento e da tarefa a ser realizada.

Tanto o meta-modelo quanto os mecanismos de gerência propostos foram especificados formalmente com o objetivo de fornecer descrição precisa em alto nível de abstração e, a partir dessa descrição, construir um protótipo para avaliar o modelo. Porém, o poder de descrição dos formalismos tornou possível raciocinar sobre os problemas tratados de forma clara e intuitiva. Apesar disso, não foi objetivo deste trabalho realizar verificações formais nas especificações, sendo este também um tópico relevante para trabalhos futuros nessa área.

8.5 Considerações Finais

Esse trabalho foi proposto com o objetivo de contribuir com a evolução da Tecnologia de Processos de Software. Foi construída uma arquitetura que serve de plataforma de integração para vários serviços de gerência de processos e que contribui para o aumento da flexibilidade na execução de processos. Um resumo das contribuições do trabalho, análise do modelo assim como limitações e questões em aberto foram apontadas neste capítulo.

Apesar da crescente evolução da área de processos de software, ainda são encontradas dificuldades para obter em um mesmo ambiente diferentes serviços de gerência, flexibilidade, linguagem de modelagem gráfica e executável, distribuição, dentre outros requisitos de PSEEs. A abrangência do meta-modelo proposto por um lado beneficia a tarefa de modelar e executar processos, mas por outro lado aumenta a dificuldade da especificação da linguagem e sua semântica. O uso de uma combinação de métodos formais para especificar o trabalho trouxe muitos benefícios quando se

considera a obtenção de uma especificação independente de implementação, tendo também o potencial de permitir que as idéias do trabalho possam ser implementadas em outros ambientes.

As visitas científicas realizadas na Alemanha durante a realização desta tese de doutorado (LIMA REIS, 2001b) levaram a autora a conhecer grupos de pesquisa e discutir as idéias do trabalho com os desenvolvedores de outros ambientes existentes. Dentre os grupos visitados estão os liderados pelos Professores Jochen Ludewig da *Uni-Stuttgart*, Dieter Rombach da *Uni-Kaiserslautern*, Hartmut Ehrig da *TU-Berlin* e Volker Gruhn da *Universität Dortmund*. Através do projeto de cooperação internacional Brasil-Alemanha foi possível contar com o auxílio de pesquisadores que muito contribuíram para a realização do trabalho, dentre eles o pesquisador Heribert Schlebbe da *Universität Stuttgart*, e a Dra. Roswitha Bardohl da *Techniche Universität Berlin*.

Este trabalho, além das contribuições científicas, favoreceu a combinação e a exploração de várias tecnologias que vinham sendo utilizadas de forma separada em projetos existentes. Várias extensões têm sido propostas para integrar a arquitetura do APSEE, dentre elas estão: um simulador de processos de software, uma ferramenta de visualização e monitoração de processos (SOUZA, 2003) e um meta-modelo para reutilização de processos (REIS, 2002f; REIS et al., 2002c; 2002d; 2002e). Assim, acredita-se 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. Por fim, os resultados acima apresentados e as publicações obtidas com essa tese constituem-se num indicador importante das possibilidades dessa arquitetura.

REFERÊNCIAS

AALST, W. M. P. van der. Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information? In: INTERNATIONAL CONFERENCE ON COOPERATIVE INFORMATION SYSTEMS, COOPIS, 4., 1999, Edinburgh, Scotland. **Proceedings…** Edinburgh: IEEE Computer Society Press, 1999.

ABEL, M. **Introdução aos Sistemas Especialistas**. Porto Alegre: UFRGS. Instituto de Informatica, 1994.

AMBLER, S. **The Elements of UML Style**. Cambridge: Cambridge University Press, 2002.

AGG: The Attributed Graph Grammar System. Berlin: Technischen Universität Berlin. Disponível em: <http://tfs.cs.tu-berlin.de/agg/>. Acesso em: abr. 2002.

AMBRIOLA, V. et al. Assessing process-centered software engineering environments. **ACM Transactions on Software Engineering and Methodology**, New York, v. 6, n.3, p. 283-328, July 1997.

ARBAOUI, S.; OQUENDO, F. PEACE: Goal-Oriented Logic-Based Formalism for Process Modelling. In: FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology**. Taunton: Research Studies Press, 1994.

ARBAOUI, S.; DERNIAME, J.; OQUENDO, F.; VERJUS, H. A comparative review of Process-Centered Software Engineering Environments. **Annals of Software Engineering**, The Netherlands, v. 14, p. 311-340, 2002.

ARMENISE, P. et al. Software Processes Representation Languages: Survey and Assessment. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 4., 1992, Capri, Italy. **Proceedings…** Los Alamitos: IEEE PRESS, 1992.

AVRILIONIS, D.; CUNIN, P.; FERNSTRÖM, C. OPSIS: A View Mechanism for Software Processes which Supports Their Evolution and Reuse. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 18., 1996, Berlin, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p. 25-29.

BANDINELLI, S. et al. Process Enactment in SPADE. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 2., 1992, Trondheim, Norway. **Proceedings...** Heildelberg: Springer, 1992. (Lecture Notes in Computer Science, v. 635).

BANDINELLI, S.; DI NITTO, E.; FUGGETTA, A.; LAVAZZA, L. Coupled vs Decoupled User Interaction Environments in PSEEs. In: INTERNATIONAL SOFTWARE PROCESS WORKSHOP, ISPW, 9. Airlie, USA. **Proceedings…** Airlie: IEEE Computer Society, 1994.

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.

BANDINELLI, S.; FUGGETTA, A.; LAVAZZA, L.; LOI, M.; PICCO, G. P. Modeling and improving an industrial software process. **IEEE Transactions on Software Engineering**, New York, v.21, n.5, p. 440-454, May 1995.

BANDINELLI, S.; DI NITTO, E.; FUGGETTA, A. Supporting cooperation in the SPADE-1 environment. **IEEE Transactions on Software Engineering**, New York, v. 22, n. 12, Dec. 1996.

BARDOHL, R. et al. Application of Graph Transformation to Visual Languages. In: EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation:** Applications, Languages and Tools. Singapore: World Scientific, 1999.

BARDOHL, R. **GenGED - Visual Definition of Visual Languages based on Algebraic Graph Transformation**. Hamburg: Kovac Verlag, 2000.

BECK, K. **Extreme Programming Explained**: Embrace Change. 2nd ed. [S.l.]: Addison-Wesley, 2000.

BELKHATIR, N.; ESTUBLIER, J.; MELO, W.; ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In: FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology.** Tauton: Research Studies Press, 1994. p. 187-222.

BIDER, I.; KHOMYAKOV, M. Is it possible to make workflow management systems flexible? Dynamical systems approach to business processes. In: INTERNATIONAL WORKSHOP ON GROUPWARE, CRIWG, 6., 2000, Madeira, Portugal. **Proceedings…** New York: IEEE Computer Society, 2000.

BJØRNER, D.; JONES, C.B. **The Vienna Development Method:** the meta-language. Berlin: Springer-Verlag, 1978. (Lecture Notes in Computer Science, v. 61).

BOGIA, D.; KAPLAN, S. Flexibility and Control for Dynamic Workflows in worlds Environment. In: CONFERENCE ON ORGANIZATIONAL COMPUTING SYSTEMS, COOCS, 1995, Milpitas, CA, USA. **Proceedings…** New York: ACM Press, 1995.

BOLOGNESI, T.; BRINKSMA, E. Introduction to the ISO specification language LOTOS. **Computer Networks ans ISDN Systems**, [S.l.], v.14, n.1, 1987.

BOOCH, G. et al. **The Unified Modeling Language User Guide**. [S.l.]: Addison-Wesley, 1998.

CASS, A. G. et al. Little-JIL/Juliette: A Process Definition Language and Interpreter. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2000, Limerick, Ireland. **Proceedings…** Los Alamitos: IEEE Press, 2000.

CERN. **ProcessWeaver Evaluation.** Disponível em: <http://sdt.cern.ch/ProcessWeaver/>. Acesso em: set. 1996.

CHANG, C.K.; CHRISTENSEN, M.J.; ZHANG, T. Genetic Algorithms for Project Management. **Annals of Software Engineering**, [S.l.], v.11, n.1, p. 107-139, Nov. 2001.

CHRISTIE, A. **Software Process Automation**: The Technology and its adoption. Berlin: Springer Verlag, 1995.

COHEN, B.; HARWOOD,W.T; JACKSON, M.I. **The Specification of Complex Systems.** Wokinghan: Addison-Wesley, 1986.

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**. Berlin: Springer-Verlag, 2001, p.1-18. (Lecture Notes in Computer Science, v. 1995).

CONRADI, R. et al. EPOS: Object-Oriented Cooperative Process Modelling. In: FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology**. Taunton: Research Studies Press, 1994. p. 33-70.

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).

CUGOLA, G. Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models. **IEEE Transactions on Software Engineering**, [S.l.], v. 24, n. 11, Nov. 1998.

CURTIS, B. et al. Process Modelling. **Communications of the ACM**, New York, v.35, n.9, Sept. 1992.

DAHMER, A. **Uma Abstração do Modelo APSEE de Processo de Software para Criação e Execução de Cursos a Distância**. 2000. Plano de Doutorado (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DAUDT, R. **Uma Proposta de Extensão do PROSOFT Básico.** 1992. Trabalho de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DAVIS, A.M. **201 Principles of Software Development**. New York: McGraw-Hill, 1995.

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 (RITA)**, Porto Alegre, v. 7, n. 1, set. 2000.

DERNIAME, J. C.; KABA, B. A.; WASTELL, D. (Ed.). **Software Process:** Principles, Methodology, and Technology. Berlin: Springer-Verlag, 1999. (Lecture Notes in Computer Science, 1500).

DOWSON, M.; NEJMEH, B.; RIDDLE, W. Fundamental Software Process Concepts. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 1., 1991, Milan. **Proceedings...** Milan: Italian Society of Computer Science, 1991.

DOWSON, M. Towards requirements for enactment mechanisms. In: INTERNATIONAL CONFERENCE ON SOFTWARE PROCESS, ICSP, 2., 1993. **Proceedings...** Berlin, Germany: IEEE Computer Society Press, 1993.

DOWSON, M.; FERNSTRÖM, C. Towards Requirements for Enactment Mechanisms. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, 3., 1994, Villard de Lans, France. **Proceedings...** Berlin: Springer-Verlag, 1994. (Lecture Notes in Computer Science, v. 1994).

DUITSHOF, M. **Workflow Automation in Three Administrative Organizations.** 1995. Dissertação (M.Sc. Thesis) - Departament of Computer Science, University of Twente, The Netherlands.

D'SOUZA, D.F.; WILLS, A.C. **Objects, Components, and Frameworks with UML**: the Catalysis approach. [S.l.]: Addison-Wesley, 2000.

EHRIG, H. Introduction to the Algebraic Theory of graph grammars**.** In: GRAPH GRAMMAR WORKSHOP, 1., 1978, Honnef. **Proceedings…** Berlin: Springer Verlag, 1979. p. 1-69. (Lecture Notes in Computer Science, v. 73).

EHRIG, H; HECKEL, R.; KORFF, M.; LÖWE, M.; RIBEIRO, L. WAGNER, A.; CORRADINI, A. Algebraic Approaches to graph transformation II: Single Pushout Approach and comparison with double pushout approach. In: G. ROZENBERG (Ed.). **Handbook of graph grammars and computing by graph transformation**. Singapore: World Scientific Publishing, 1997. v.1.

EHRIG, H.; ENGELS, G.; KREOWSKI, H. J.; ROZENBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1999. v.2.

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, 1998.

ENDEAVORS home-page. Disponível em: <http://www.ics.uci.edu/pub/endeavors/>. Acesso em: nov. 2000.

ENGELS, G.; SCHÜRR, A. **Encapsulated Hierarquical Graphs, Graph Types and Meta Types**. 1995. Technical Report - Leiden University, The Netherlands.

ESTUBLIER, J.; CASALLAS, R. The Adele Configuration Manager. In: TICHY, W. (Ed.). **Configuration Management**. New York: John Wiley and Sons, 1994.

ESTUBLIER, J. et al. APEL: A graphical yet executable formalism for process modeling. **Automated Software Engineering**, The Netherlands, v.5, n.1, 1997.

FAYYAD, U. et al. **Advances in Knowledge Discovery in Databases**. Cambridge: MIT Press, 1996.

FAYYAD, U.; HAUSSLER, D.; STOLORZ, P. Mining Scientific Data. **Communications of the ACM**, New York, v. 39, n. 11, p. 51-57, Nov. 1996.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. The KDD Process for Extracting Useful Knowledge from Volumes of Data. **Communications of the ACM**, New York, v. 39, n. 11, p. 27-34, Nov. 1996.

FEILER, P.; HUMPHREY, W. Software Process Development and Enactment: Concepts and Definitions. In: INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS, ICSP, 2., 1993, Berlin. **Proceedings…** Berlin, Germany: IEEE Computer Society Press, 1993.

FERNSTRÖM, C. PROCESSWEAVER: Adding Process Support to UNIX. In: INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS, ICSP, 2., 1993, Berlin. **Proceedings…** Berlin, Germany: IEEE Computer Society Press, 1993.

FERNANDES, A. A. **Gerência de Software através de métricas**: Garantindo a qualidade do projeto, processo e produto. São Paulo: Atlas, 1995.

FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology**. Taunton: Research Studies Press, 1994.

Flex-el (Flexible e-learning). Disponível em: <http://flex-el.com/>. Acesso em: jun. 2002.

FLORAC, W. A.; CARLETON, A. D. **Measuring the Software Process**: Statistical Process Control for Software Process Improvement. [S.l.]: Addison Wesley, 1999.

FRANCA, L. P. A.; STAA, A.; LUCENA, C. J. P. DE. Medição de Software para Pequenas Empresas: Uma solução baseada na Web. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 12., 1998, Maringá. **Anais...** Maringá: SBC, 1998.

FROEHLICH, G. **Process Modelling Support in Metaview**. 1994. Dissertação (Master of Science Thesis) - University of Sakatchewan, Canada.

FUGGETTA, A.; WOLF, A. (Ed.). **Software Process**. Chichester: John Wiley & Sons, 1996.

FUGGETTA, A. Software Process: A Roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2000, Limerick, Ireland. **Proceedings…** New York: ACM Press, 2000.

GARG, P. K.; JAZAYERI, M. **Process-Centered Software Engineering Environments.** Los Alamitos: IEEE Computer Society Press, 1996.

GEORGAKOPOULOS, D. et al. **An Overview of Workflow Management:** From Process Modeling to Workflow Automation Infrastructure**.** Disponível em: <http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/g/Georgakopoulos:Dimitrios.html>. Acesso em: dez. 1994

GIMENES, I. M. S. **ExPSEE:** Um Ambiente Experimental de Engenharia de Software Orientado a Processos. 2000. Relatório de Projeto. Universidade Estadual de Maringá, Depto. de Informática, Maringá.

GIMENES, I. M. S. **Uma Introdução ao Processo de Engenharia de Software:** ambientes e formalismos. Caxambu, MG: SBC, 1997. Trabalho apresentado na 13. Jornada de Atualização em Informática, 1994, Caxambu.

GODART, C.; CHAROY, F. **Databases for Software Engineering**. London: Prentice Hall International, 1994.

GOGUEM, J. A.; THATCHER, J. W.; WAGNER, E. G. **An initial Algebra approach to the specification, correctness and implementation of abstract data types**. 1977. Relatório (Technical Report RC 6487 – 26817) - IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

GOSLING, J.; JOY, B.; STEELE, G. **The Java Language Specification**. Disponível em: <http://java.sun.com/doc/books/jls/index.html>. Acesso em: mar. 2002.

GRANVILLE, L.Z.; SCHLEBBE, H. **Distributed PROSOFT:** Management of Tools and Memory. Porto Alegre: UFRGS e Uni-Stuttgart, 1996. Relatório Técnico.

GREENWOOD, R. M. Coordination Theory and Software Process Technology. In: SCHÄFER, W. (Ed.). EUROPEAN WORKSHOP ON THE SOFTWARE PROCESS TECHNOLOGY, EWSPT, 4., Noordwijkerhout, the Netherlands. **Proceedings…**Berlin: Springer Verlag, 1995. (Lecture Notes in Computer Science, v. 913).

GRUNDY., J. A Decentralized Architecture for Software Process Modeling and Enactment. **IEEE Internet Computing**, New York, v. 2, n. 5, p. 53-62, 1998.

GRUNER, S. Meta Typing is Compatible to the Typed SPO Approach. In: JOINT APPLIGRAPH/GETGRATS WORKSHOP ON GRAPH TRANSFORMATION SYSTEMS, GRATRA, 2000, Berlin, Germany. **Proceedings…** Disponível em: <http://tfs.cs.tu-berlin.de/gratra2000>. Acesso em: nov. 2001.

GRUHN, V. Process-Centered Software Engineering Environments: A brief history and future Chalenges. **Annals of Software Engineering**, The Netherlands, v. 14, p. 363-382, 2002.

HAGEN, C.; ALONSO, G. Exception Handling in Workflow Management Systems. **IEEE Transactions on Software Engineering**, New York, v. 26, n. 10, Oct. 2000.

HECKEL, R. **O uso de herança em gramáticas de grafos** [mensagem pessoal]. Mensagem recebida por <reiko@uni-paderborn.de> em 28 jan. 2002

HEIMAN, P. et al. An Environment for Managing Software Development Processes. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ENVIRONMENTS, SEE, 1997, Cottbus. **Proceedings...** New York: IEEE Press, 1997. Disponível em: <http://www-i3.informatik.rwth-aachen.de/research/dynamite/>. Acesso em: nov. 2000.

HEINL, P. et al. A comprehensive approach to flexibility in workflow management systems. **Software Engineering Notes**, New York, v.24, n.2, p. 79-88. Trabalho apresentado no International Conference on Work Activities Coordination and Collaboration, 1999.

HOARE, C. A. R. **Communicating Sequential Processes**. London: Prentice Hall, 1985.

HUANG, Y.; SHAN, M. Policies in a Resource Manager of Workflow Systems: Modeling, Enforcement and Management. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 5., Australia, 1999. **Proceedings…** Los Alamitos: IEEE Press, 1999.

HUFF, K. Software Process Modeling. In: FUGGETTA, A.; WOLF, A. (Ed.). **Software Process**. [S.l.]: John Wiley & Sons, 1996.

HUMPHREY, W. S. **Managing the Software Process.** [S.l.]: Addison-Wesley, 1989.

HUMPHREY, W. S. **A discipline for Software Engineering**. [S.l.]: Addison Wesley, 1995. (SEI Series in Software Engineering).

HUMPHREY, W. S. **Introduction to the Personal Software Process**. [S.l.]: Addison Wesley, 1997. (SEI Series in Software Engineering).

INTERNATIONAL PROCESS TECHNOLOGY WORKSHOP**,** IPTW, 1999. **Proceedings…** Disponível em: <http://www-adele.imag.fr/IPTW/>. Acesso em: maio 2000.

JACOBSON, I.; RUMBAUGH, J.; BOOCH, G. **The Unified Software Development Process**. [S.l.]: Addison-Wesley, 1999.

JACKSON, M. **System Development.** New York: Prentice-Hall International, 1993.

JACCHERI, M.; AMERIO, E.; MALNATI, G. SENESI, P. **E3 p-draw:** a tool to produce and reuse software process models. 2002. Relatório técnico - Dipartamento de Automatica e Informatica, Politecnico de Torino, Itália. Disponível em: <http://www2.umassd.edu/SWPI/e3/tool.pdf>. Acesso em: maio 2002.

JACCHERI, M. Reusing Software Process Models in E3. In: INTERNATIONAL SOFTWARE PROCESS WORKSHOP, ISPW, 10., 1996, Ventron, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p. 100-102.

JACCHERI, M. L.; LAGO, P.; PICCO, G.P. Eliciting Software Process Models with the E^3 Language. **ACM Transactions on Software Engineering and Methodology**, New York, v.7, n.4, p. 368-410, 1998.

JOERIS, G.; HERZOG, G. Towards Flexible and High-Level Modeling and Enacting of Processes. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, CAiSE, 11., 1999, Heidelberg. **Proceedings...** Berlin: Springer Verlag, 1999. (Lecture Notes in Computer Sciece, v.1626).

JONES, C. **Applied Software Measurement**: Assuring Productivity and Quality. 2nd ed., [S.l.]: McGraw Hill, 1996.

JØRGENSEN, H. Interaction as a Framework for Flexible Workflow Modeling. In: INTERNATIONAL ACM SIGGROUP CONFERENCE ON SUPPORTING GROUP WORK, GROUP, 2001, Boulder, Colorado, USA. **Proceedings…** New York: ACM Press, 2001.

JUNKERMANN, G. et al. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment**.** In: FINKELSTEIN, A. et al. (Ed.). **Software Process Modelling and Technology**. Tauton: Research Studies Press, 1994. p. 103-130.

KAISER, G.; FELLER, P. H.; POPOVICH, S. S. Intelligent Assistance for Software Development and Maintenance. **IEEE Software**, Los Alamitos, v. 5, n. 3, May 1988.

KAISER, G.E.; BARGHOUTI, N.S.; SOKOLSKY, M.H. Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCE, 23., Kona. **Proceedings...** [S.l.: s.n.], 1990. p. 131-140.

KAPPEL, G. et al. Coordination in workflow management systems: A rule based approach. In: CONEN, W.; NEUMAN, G. (Ed.). **Coordination technology for collaborative applications**. Berlin: Springer Verlag, 1998. (Lecture Notes in Computer Science, v. 1364).

KLEIN, M. Coordination Science: Challenges and Directions. In: WORKSHOP ON COORDINATION TECHNOLOGY FOR COLLABORATIVE APPLICATIONS: ORGANIZATIONS, PROCESSES AND AGENTS, ASIAN, 1996, Singapore. **Proceedings...** Berlin: Springer Verlag, 1996. (Lecture Notes in Computer Science, v. 1364).

KOBIALKA, H.; LEWERENTZ, C. User Interfaces Supporting the Software Process. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 6., Weybridge, UK. **Proceedings...** Berlin: Springer Verlag, 1998. (Lecture Notes in Computer Science, v. 1487).

KÖRBES, F. **Implementação de um Mecanismo de Herança no PROSOFT**. 1996. 39f. Trabalho de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

KRAPP, C. A. **A process-management environment for the development of complex products**. 1997. Disponível em: <http://www-i3.informatik.rwthaachen.de/research/dynamite/>. Acesso em: dez. 1997.

KRAPP, C.A. **An Adaptable Environment for the Management of Development Processes**. 1998. Tese (Tese de Doutorado) - Aachen University, Aachen, Alemanha.

KRUKE, V. **Reuse in Workflow Modeling.** 1996. Diploma thesis - Norwegian University of Science and Technology, Trondheim, Norway. Disponível em: <http://www.pvv.ntnu.no/~crukis>. Acesso em: jan. 2000.

LERNER, B. S. et al. **Modeling and Managing Resource Utilization in Process, Workflow and Activity Coordination**. New England: Department of Computer Science, University of Massachusetts, 2000. Technical Report.

LIMA REIS, C.A. **Um Gerenciador de Processos de Software para o Ambiente PROSOFT.** 1998. 197f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

LIMA REIS, C. A. **Estudo da utilização de técnicas de inteligência artificial na tecnologia de processos de software**. 1999. Trabalho Individual (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

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, UFRGS, Porto Alegre.

LIMA REIS, C. A. et al. 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, Rio de Janeiro. **Anais...** Rio de Janeiro: IME/SBC, 2001.

LIMA REIS, C.A. **Visit Report of a Research Mission in Germany:** from January, 15th to March 23th, 2001**.** 2001. Relatório de Pesquisa – Instituto de Informática, UFRGS/CNPq/DLR, Porto Alegre.

LIMA REIS, C. A.; REIS, R. Q.; NUNES, D. J. APSEE: Uma Abordagem Integrada para Automação de Processos de Software. In: SIMPOSIO BRASILEIRO DE AUTOMAÇÃO INTELIGENTE, SBAI, 5., 2001, Canela - RS. **Anais...** Canela: SBA/UFRGS, 2001.

LIMA REIS, C.A. **Modelagem, Gerência e Instanciação de Recursos no Ambiente APSEE**. 2001. 78f. Relatório técnico (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre. Disponível em: <http://www.cultura.ufpa.br/clima/2001/resources_report.zip>. Acesso em: jul. 2001.

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, Havana. **Memórias...** Havana, Cuba: CYTED, 2002.

LIMA REIS, C. A.; REIS, R.Q.; SCHLEBBE, H.; NUNES, D.J. A Policy-based Resource Instantiation Mechanism to Automate Software Process Management. In: WORKSHOP ON SOFTWARE ENGINEERING DECISION SUPPORT, SEDECS, 2002, Ischia, Itália. **Proceedings…** New York: ACM Press, 2002.

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. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2002.

LIMA REIS, C. A.; REIS, R. Q.; ABREU, M. M.; SCHLEBBE, H.; NUNES, D. J. Flexible Software Process Enactment Support in the APSEE Model. In: IEEE CONFERENCE SERIES ON HUMAN-CENTRIC COMPUTING LANGUAGES AND ENVIRONMENTS, HCC, 2002, Arlington. **Proceedings…** Los Alamitos: IEEE Press, 2002.

LIMA REIS, C. A.; REIS, R. Q.; ABREU, M. M.; SCHLEBBE, H.; NUNES, D. J. Using Graph Transformation as the Semantical Model for Software Process Execution in the APSEE Environment. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, ICGT, 1., 2002, Barcelona. **Proceedings…** Berlin: Springer Verlag, 2002. (Lecture Notes in Computer Sciece, v. 2505).

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 Press, 1993.

MACHADO, L. F. C. **Modelo para Definição de Processos de Software na Estação TABA**. 2000. Dissertação (Mestrado em Ciência da Computação) - COPPE/UFRJ, Rio de Janeiro.

MALONE, T.; CROWSTON, K. The Interdisciplinary Study of Coordination. **ACM Computing Surveys**, New York, v. 26, n. 1, Mar. 1994.

MARVEL Environment Home-Page. Disponível em: <http://www.psl.cs.columbia.edu/marvel.html>. Acesso em: nov. 2000.

MILNER, R. **A Calculus of communicating systems**. Berlin: Springer Verlag, 1980. (Lecture Notes in Computer Science, v. 92).

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, Toulouse. **Proceedings…** [S.l.]: IEEE/IFIP, 1994. Disponível em: <http://www.cs.york.ac.uk/hise/bib.html>. Acesso em: nov. 2001.

MOITRA, D. India's Software Industry: the software superpower. **IEEE Software**, [S.l.], v.18, n.1, p. 77-80, Jan. 2001.

MOLINA, S. Objetos e Tarefas Distribuídas: um Enfoque ao BPA (Business Process Automation). **Developers' Magazine**, [S.l.], v.5, n. 53, p.12-13, Jan. 2001.

NGUYEN, M.; CONRADI, R. Classification of Meta-processes and their Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE PROCESS, ICSP, 3., 1994, Washington. **Proceedings...** Washington: IEEE Computer Society Press, 1994.

NGUYEN, M.; WANG, A. **Total Software Process Model in Epos**. Disponível em: <http://www.idt.unit.no/~epos/>. Acesso em: nov. 2000. (Relatório técnico).

NISSANKE, N. **Formal Specification:** Techniques and Applications. Berlin: Springer Verlag, 1999.

NUNES, D. J. Estratégia Data-Driven no Desenvolvimento de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 6., 1992, Gramado. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1992. v.1, p. 81-95.

NUNES, D.J. **PROSOFT**: Um Ambiente de Desenvolvimento de Software baseado no Método Algébrico**.** 1994. Relatório Técnico – Instituto de Informática, UFRGS, Porto Alegre. Disponível em: <http://www.inf.ufrgs.br/prosoft>. Acesso em: dez. 1999.

OCAMPO, C.; BOTELLA, P. **Some Reflections on applying Workflow Technology to Software Process.** 1998. Relatório Técnico LSI-98-5-R - Department de Lenguages i Sistemes Informàtics, UPC, Barcelona. Disponível em: <http://www.lsi.upc.ed/~ocampo>. Acesso em: dez. 1999.

OSTERWEIL, L. Software Processes are Software Too. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 1987, Monterey, California, USA. **Proceedings...** Los Alamitos: IEEE Press, 1987. p. 2-13.

PAULK, M.; WEBER, C.; CURTIS, B. **The Capability Maturity Model:** Guidelines for Improving the Software Process. [S.l.]: Addison-Wesley, 1994. (SEI Series in Software Engineering).

PLEKHANOVA, V.; OFFEN, R. Managing the Human-Software Environment. In: INTERNATIONAL WORKSHOP ON SOFTWARE TECHNOLOGY AND ENGINEERING PRACTICE, STEP, 8., 1997, London. **Proceedings…** Los Alamitos: IEEE Press, 1997.

PLEKHANOVA, V. On Project Management Scheduling where Human Resource is a Critical Variable. In: EUROPEAN WORKSHOP ON SOFTWARE PROCESS TECHNOLOGY, EWSPT, 6., 1998, Weybridge. **Proceedings…** Berlin: Springer Verlag, 1998. (Lecture Notes in Computer Science v. 1487).

PLEKHANOVA, V. Capability and Compatibility Measurement in Software Process Improvement. In: EUROPEAN SOFTWARE MEASUREMENT CONFERENCE, FESMA, 2., 1999, Amsterdam, The Netherlands. **Proceedings…** [S.l.:s.n.], 1999.

PODOROZHNY, R. et al. Modeling Resources for Activity Coordination and Scheduling. In: INTERNATIONAL CONFERENCE ON COORDINATION MODELS AND LANGUAGES, COORD, 3., Amsterdam. **Proceedings…** Berlin: Springer Verlag, 1999. (Lecture Notes in Computer Science, v. 1594).

POHL, K. et al. Prime – Toward Process-Integrated Modeling Environments. **ACM Transactions on Software Engineering and Methodology**, New York, v. 8, n. 4, Oct. 1999.

PRESSMAN, R.S. **Software Engineering:** A Practitioner's Approach. 5th ed., London: McGraw-Hill, 2001.

RAPOSO, A.B.; MAGALHÃES, L.P.; RICARTE, I.L.M. Petri Nets Based Coordination Mechanisms for Multi-Workflow Environments. **International Journal of Computer Systems Science & Engineering**, Leicester, v.15, n.5, Sept. 2000.

RAPOSO, A. B. **Coordenação em Ambientes Colaborativos Usando Redes de Petri**. 2000. Tese (Doutorado em Engenharia Elétrica) - Universidade Estadual de Campinas, Campinas – SP.

RANGEL, G. S. **ProTool**: Uma Ferramenta de Prototipação de Software para o Ambiente PROSOFT. 2003. 223 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

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, UFRGS, Porto Alegre.

REIS, R.; LIMA REIS, C.A.; NUNES, D. J. Gerenciamento do Processo de Desenvolvimento Cooperativo de Software no Ambiente PROSOFT. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 12., 1998, Maringá. **Anais**... Maringá: SBC, 1998. p. 221-236.

REIS, R.Q. **Uma avaliação dos paradigmas de linguagens de processo de software.** 1999. Trabalho Individual (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

REIS, R.Q.; LIMA REIS, C.A.; SILVA, F.; NUNES, D.J. SimAgentProcess: Uma Ferramenta para Simulação de Processos de Software Baseada em Conhecimento. In: WORKSHOP IBERO AMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, IDEAS, 3., Cancun, México, Abril, 2000. **Proceedings...** Cancún: CYTED, 2000.

REIS, R. Q.; LIMA REIS, C. A.; NUNES, D. J. Evolução do Ambiente PROSOFT para apoiar aspectos de Distribuição, Cooperação e Automação do Processo de Desenvolvimento de software. In: WORKSHOP IBERO AMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, IDEAS, 3., Cancun, México, 2000. **Proceedings...** Cancún: CYTED, 2000.

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., Germany. **Proceedings…**Los Alamitos: IEEE CS Press, 2001.

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., 2001. **Anais**... Rio de Janeiro: SBC, 2001.

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., 2002. **Proceedings...** Pasadena: Society for Design and Process Science, 2002.

REIS, R.Q.; LIMA REIS, C.A.; SCHLEBBE, H.; NUNES, D.J. Automatic Verification of Static Policies on Software Process Models. **Annals of Software Engineering,** The Netherlands, v. 14, p.197-234, Dec. 2002.

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, Havana. **Memórias...** Havana: CYTED, 2002.

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, 1., 2002, Enschede. **Proceedings…** New York: ACM Press, 2002.

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: IEEE ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 26., 2002, Oxford. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2002.

REIS, R. Q. **APSEE-REUSE**: Um Meta-modelo para apoiar a reutilização de processos de software. 2002. 213f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RIBEIRO, L. Métodos Formais de Especificação: gramáticas de Grafos. In: ESCOLA DE INFORMÃTICA DA SBC-SUL, 8., 2000. **Livro Texto**. Santa Maria: UFMS, 2000.

RIBÓ, J.M.; FRANCH, X. Searching for Expressiveness, Modularity, Flexibility and Standardization in Software Process Modelling. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 14., 2000, João Pessoa. **Anais…** João Pessoa: SBC, 2000.

RIBÓ, J.M.; FRANCH, X. Building Expressive and Flexible Process Models using a UML-based Approach. In: EUROPEAN WORKSHOP IN SOFTWARE PROCESS TECHNOLOGY, EWSPT, 8., 2001, Witten, Germany. **Proceedings...** Berlin: Springer-Verlag, 2001. (Lecture Notes in Computer Science v. 2077).

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de Software**: Teoria e Prática. São Paulo: Prentice Hall, 2001.

ROMBACH, D.; VERLAGE, M. How to assess a software process modeling formalism from a project member's point of view. In: INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS, ICSP, 2., 1993. **Proceedings…** Los Alamitos: IEEE Press, 1993.

ROZENBERG, G. (Ed.). **Handbook on Graph Grammars**. Singapore: World Scientific, 1997. v.1.

SADIQ, S. W.; ORLOWSKA, M Architectural Considerations in Systems Supporting Dynamic Workflow Modification. In: WORKSHOP ON SOFTWARE ARCHITECTURES FOR BUSINESS PROCESS MANAGEMENT, SABPM (CaiSE), 1999, Heildelberg, Germany. **Proceedings…**Berlin: Springer Verlag, 1999. (Lecture Notes in Computer Science, v. 1626).

SADIQ, S. W.; MARJANOVIC, O.; ORLOWSKA, M. Managing Change and Time in Dynamic Workflow Processes. **International Journal of Cooperative Information Systems**, [S.l.], v. 9, n. 1 e n.2, Mar./June 2000.

SADIQ, S. W.; ORLOWSKA, M. E. Analyzing Process Models using Graph Reduction Techniques. **Information Systems**, [S.l.], v. 25, n. 2, 2000.

SADIQ, S. W. Handling Dynamic Schema Change in Process Models. In: AUSTRALIAN DATABASE CONFERENCE, 11., Canberra, Australia. **Proceedings…** Los Alamitos: IEEE Press, 2000.

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. Relatório Técnico. Universität Stuttgart. Fakultät Informatik, Stuttgart.

SCHÜRR, A.; WINTER, A.J.; ZÜNDORF, A. Graph Grammar Engineering with PROGRES. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE, 5., 1995. **Proceedings…** Berlin: Springer Verlag, 1995. (Lecture Notes in Computer Science, v. 989).

SCHLEBBE, H.; SCHIMPF, S. **Reenginering of PROSOFT in Java**. 1997. Relatório Técnico. Stuttgart University (Alemanha) / UFRGS (Brasil).

SCHÄL, T. **Workflow Management Systems for Process Organizations**. 2nd ed. Berlin: Springer, 1998. (Lecture Notes in Computer Science, v. 1096).

SCHLEBBE, H. **Java-PROSOFT Manual.** Fakultät Informatik, Universität Stuttgart, 2002. Disponível em: <http://www.informatik.uni-stuttgart.de/ifi/bs/schlebbe/prosoft_doc/guide>. Acesso em: nov. 2000.

SCHWABER, K.; BEEDLE, M. **Agile Software Development with SCRUM**. London: Prentice-Hall, 2001.

SILVA, F.A.D. **Um Modelo de Simulação de Processos de Software Baseado em Conhecimento para o Ambiente PROSOFT**. 2001. 150f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SILVA, F.; REIS, R.Q.; LIMA REIS, C.; NUNES, D. Um Modelo de Simulação de Processos de Software baseado em Agentes Cooperativos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 13., 1999, Florianópolis. **Anais...** Florianópolis: UFSC/SBC, 1999.

SLISKI, T.; BILLMERS, M.; CLARKE, L.; OSTERWEIL, L. An architecture for flexible, evolvable process-driven user-guidance environments. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE/ACM SIGSOFT SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE, 8., 2001, Viena, Austria, 2001. **Proceedings...** New York: ACM Press, 2001.

SOUSA, A.L.R. **Uma Ferramenta de Apoio à Visualização e Representação de Modelos de Processos de Software**. 2001. Plano de Estudos e Pesquisa (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

SOUSA, A. L. R.; LIMA REIS, C. A. ; REIS, R. Q.; NUNES, D. J.; PIMENTA, M. S. Analisando a Interação de Gerentes e Desenvolvedores em ambientes de processos de software. In: WORKSHOP CHILENO DE ENGENHARIA DE SOFTWARE, 1., 2001, Punta Arenas, 2001. **Proceedings...** [S.l.:s.n.], 2001.

SOUSA, A. L. R. **APSEE-Monitor**: Um Mecanismo de Apoio à Visualização de Modelos de Processos de Software. 2003. 112 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SOUSA, A.L.R.; LIMA REIS, C. A.; REIS, R. Q. **Interação Humana durante execução de processos de software**: Classificação e Exemplos. Porto Alegre: UFRGS, jun. 2001. Relatório técnico. Disponível em: <http://www.inf.ufrgs.br/~prosoft>. Acesso em: dez. 2001.

SUN MICROSYSTEMS. **Java Runtime Environment.** Disponível em: <http://www.java.sun.com> . Acesso em: mar. 2002.

STOY, J. E. **Denotational Semantics**: the Scott-strachey approach to programming language theory. Cambridge: MIT Press, 1977.

SUTTON, S. M.; HEIMBIGNER, D.; OSTERWEIL, L. J. Language Constructs for Managing Change in Process-Centered Environments. In: ACM SIGSOFT/SIGPLAN SYMPOSIUM ON PRACTICAL SOFTWARE DEVELOPMENT ENVIRONMENTS, 4., New York, USA. **Proceedings...** New York: ACM Press, 1990.

TAYLOR, R. N.; BOLCER, G. A. Endeavors: A Process System Integration Infrastructure. In: INTERNATIONAL CONFERENCE ON SOFTWARE PROCESS, ICSP, 4., 1996, Brighton, UK. **Proceedings...** Los Alamitos: IEEE Press, 1996.

TONG, A.; KAISER, G.; POPOVICH, S. A flexible Rule-Chaining Engine for Process-Based Software Engineering. In: KNOWLEDGE BASED SOFTWARE ENGINEERING CONFERENCE, 9., Monterey, CA, 1994. **Proceedings...** [S.l.]: University of California Press, 1994. p. 79-88.

TRAVASSOS, G. H. **O Modelo de Integração de Ferramentas da Estação TABA.** 1994. Tese (Doutorado em Ciência da Computação) – Instituto Alberto Luiz Coimbra (COPPE), UFRJ, Rio de Janeiro.

WANG, X. Z. **Data Mining and Knowledge Discovery for Process Monitoring and Control**. Berlin: Springer, 1999.

WANG, Y.; KING, G. **Software Engineering Processes**: Principles and Applications. [S.l.]: CRC Press, 2000.

WATT, D. **Programming Language Syntax and Semantics**. London: Prentice-Hall, 1991.

WEBER, K.; DE LUCA, J.C.; ROCHA, A.R. (Ed.). **Qualidade e Produtividade em Software.** 2.ed. São Paulo: Makron Books, 1995.

WESTFECHTEL, B. **Models and Tools for Managing Development Processes**. Berlin: Springer, 1999. (Lecture Notes in Computer Science, v. 1646).

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. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2001. p.1–8.

YOUNG, P**. Customizable Process Specification and Enactment for Technical and Non-Technical Users**. 1994. Tese (Doutorado) - University of California, Irvine, USA, 1994. Disponível em: < http://www.ics.uci.edu/pub/endeavors>. Acesso em: nov. 1998.

APÊNDICE A Tipos de dados do APSEE no PROSOFT

A.1. APSEE Types

A FIGURA A 1 apresenta as classes *APSEE-Types* e *Types*, usadas para armazenar os nodos de hierarquias de tipos.

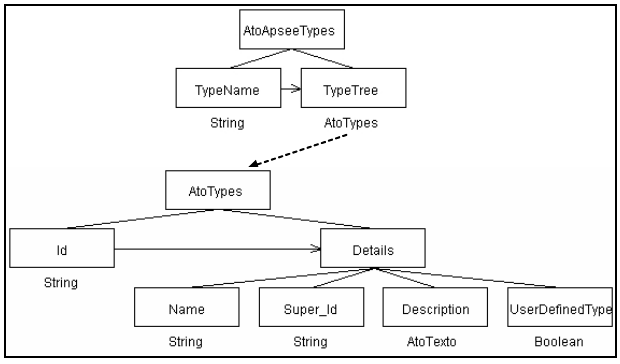


FIGURA A 1 - As classes *APSEE-Types* e *Types*

A.2. Organization

A seguir são apresentados a classe *Organization* e suas principais classes componentes.

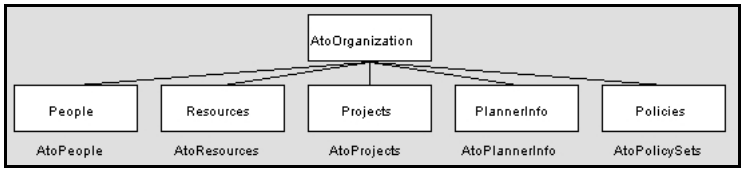


FIGURA A 2 - Classe Organization

A.2.1 People

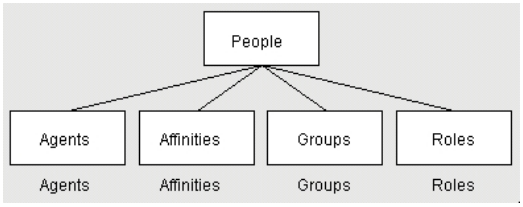


FIGURA A 3 - Classe People.

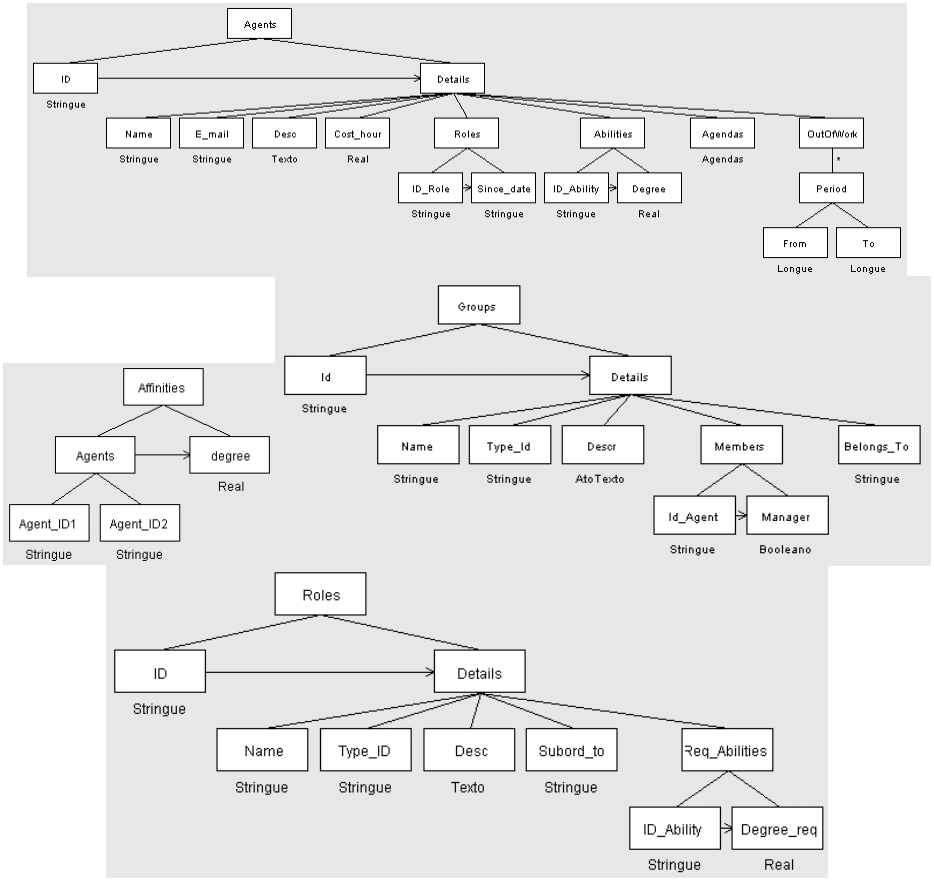


FIGURA A 4 - Classes *Agents*, *Affinities*, *Groups* e *Roles* do pacote *People*.

A.2.2 Resources

A seguir são apresentadas as classes principais do pacote *Resources*.

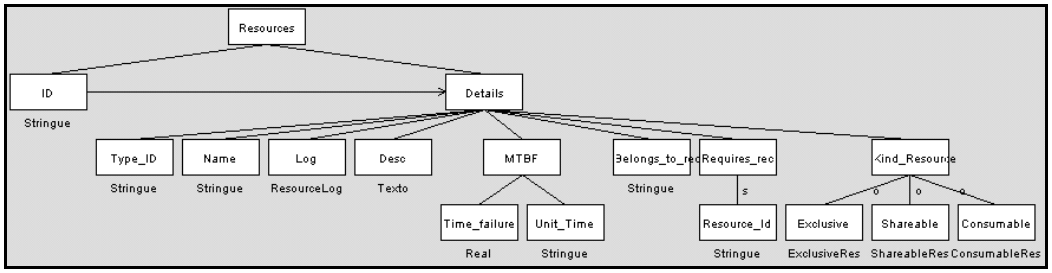


FIGURA A 5 - Classe Resources.

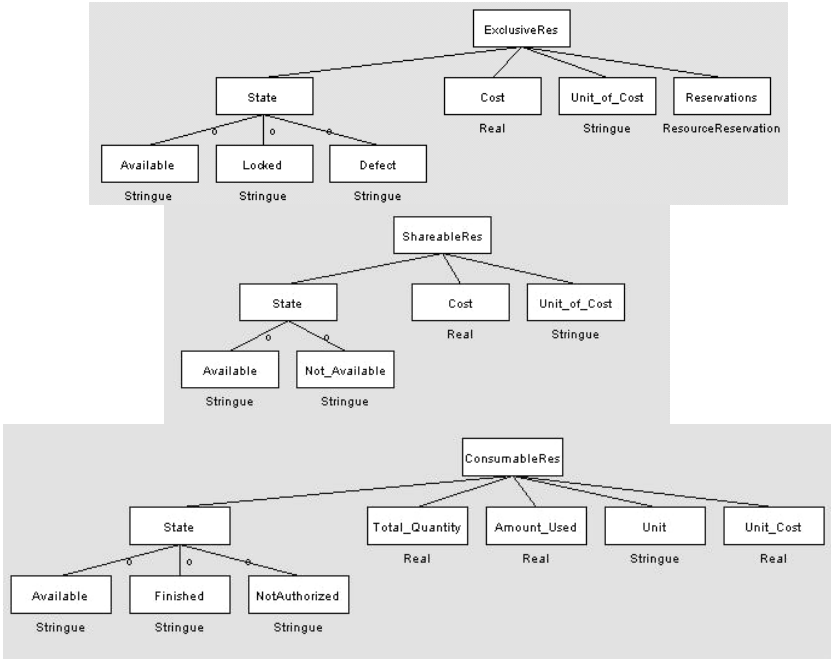


FIGURA A 6 - Classes *ExclusiveRes*, *ShareableRes* e *ConsumableRes*

A.3. ProcessKnowledge

A seguir são apresentadas as principais classes do pacote ProcessKnowledge.

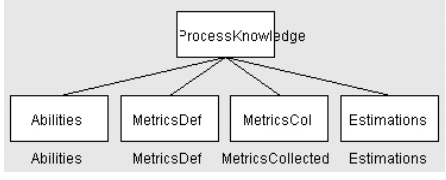


FIGURA A 7 - Classe *ProcessKnowledge*.

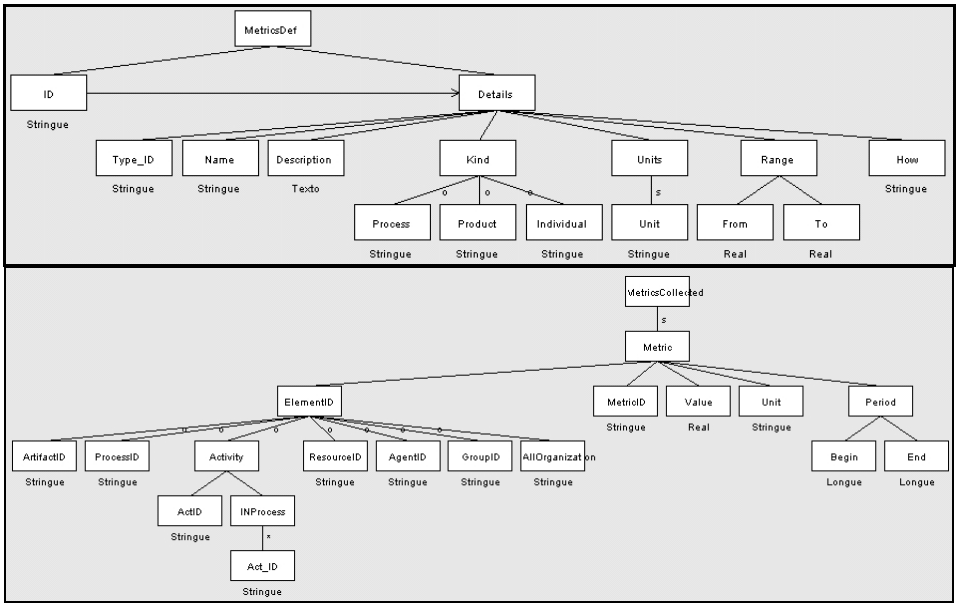


FIGURA A 8 - Classes *MetricsDef* e *MetricsCollected*.

A.4. Processes

A seguir são apresentadas as classes do pacote *Processes*.

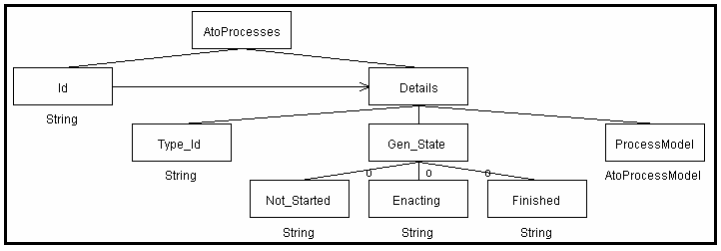


Figura A 9 A classe *Processes*

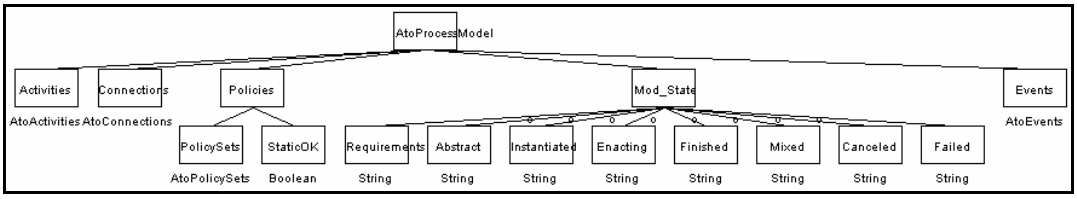


Figura A 10 A classe *ProcessModel*

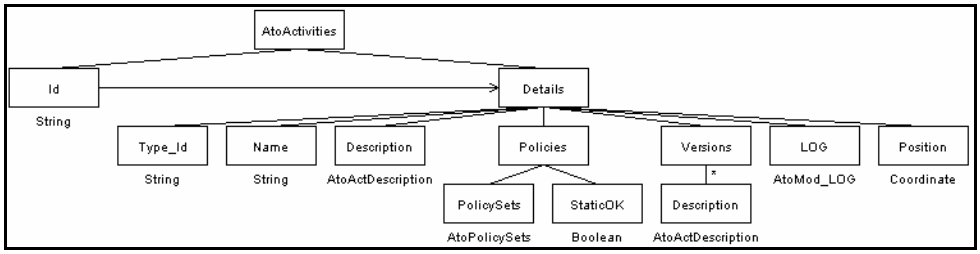


Figura A 11 A classe *Activities*

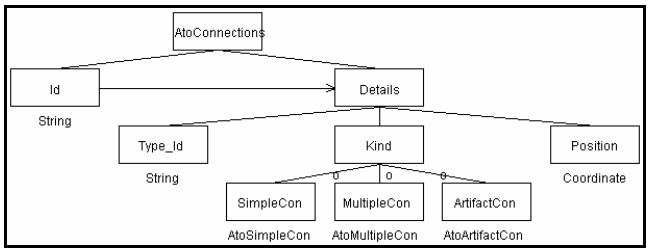


Figura A 12 A classe *Connections*

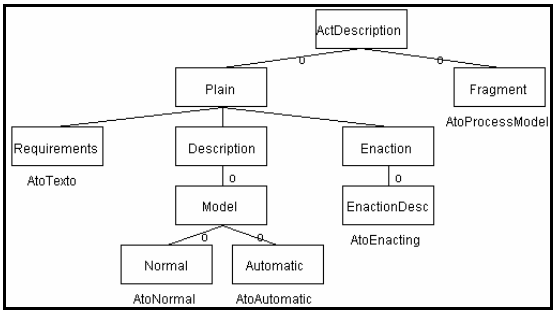


Figura A 13 A classe *ActDescription*

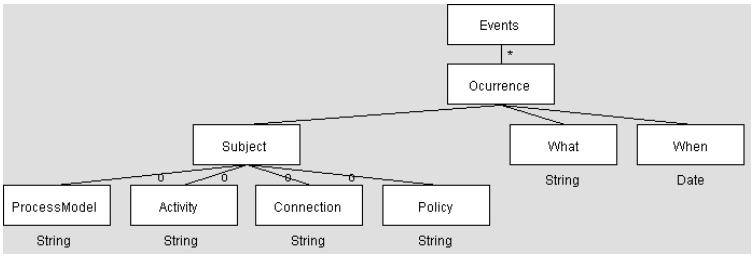


Figura A 14 A classe *Events*

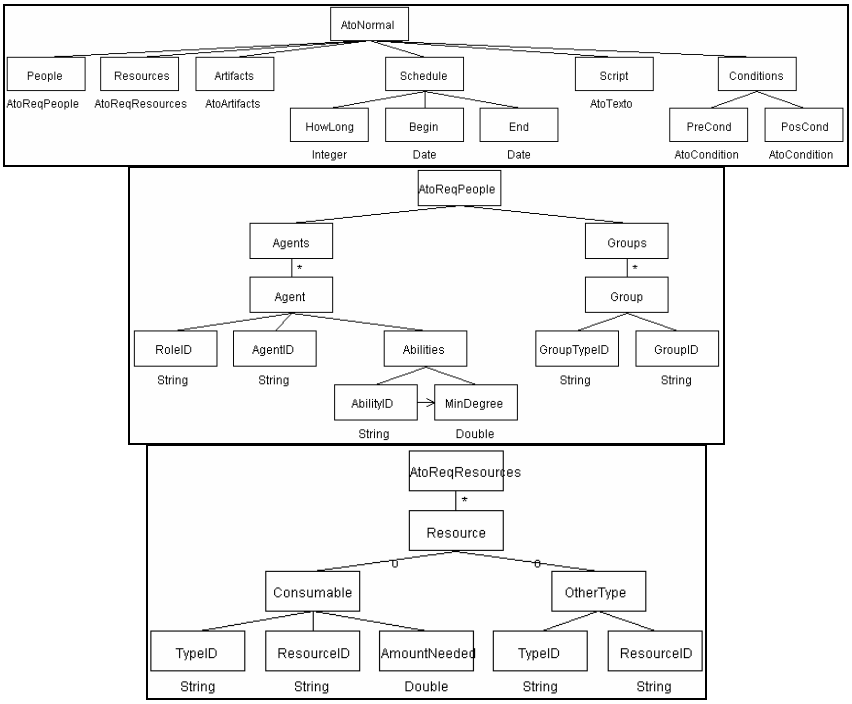


Figura A 15 As classes *Normal*, *ReqPeople* e *ReqResources*

A.5. Software Artifacts

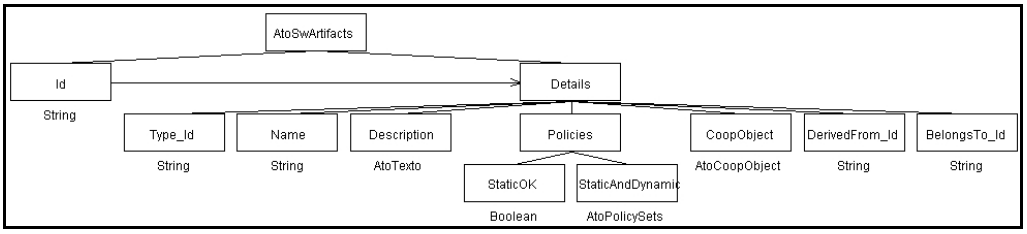


FIGURA A 16 - Classe *SwArtifacts*.

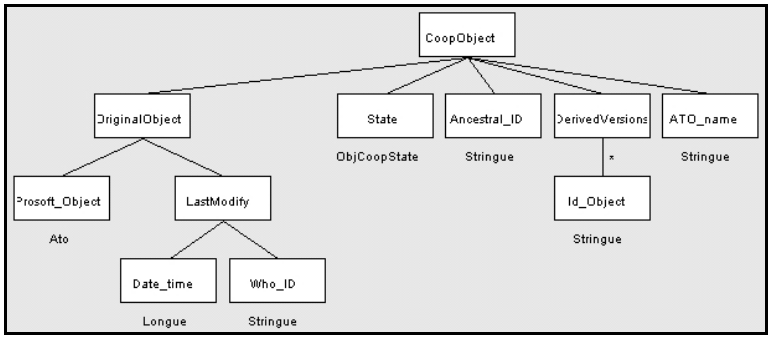


FIGURA A 17 - Classe *CoopObject*.

A.6. Tools

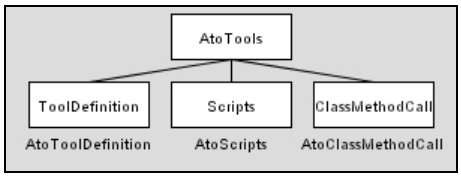


FIGURA A 18 - Classe *Tools*.

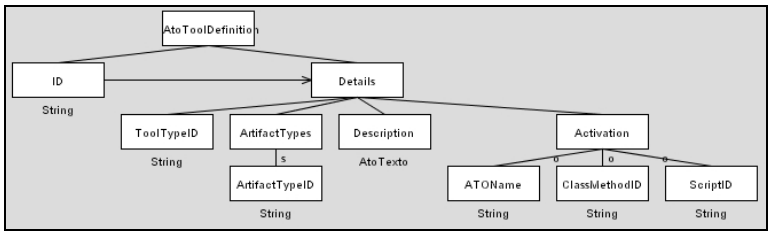


FIGURA A 19 - Classe *ToolDefinition*.

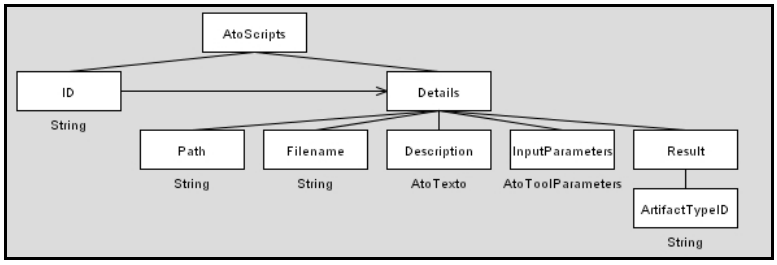


FIGURA A 20 - Classe *Scripts*.

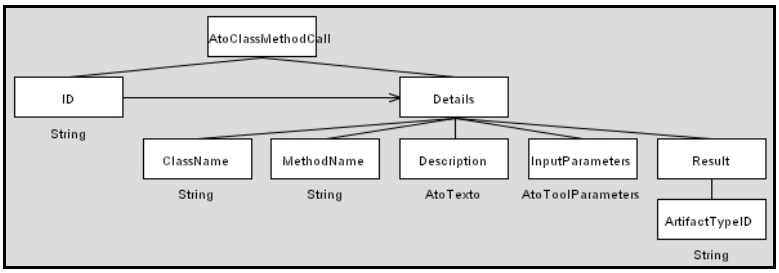
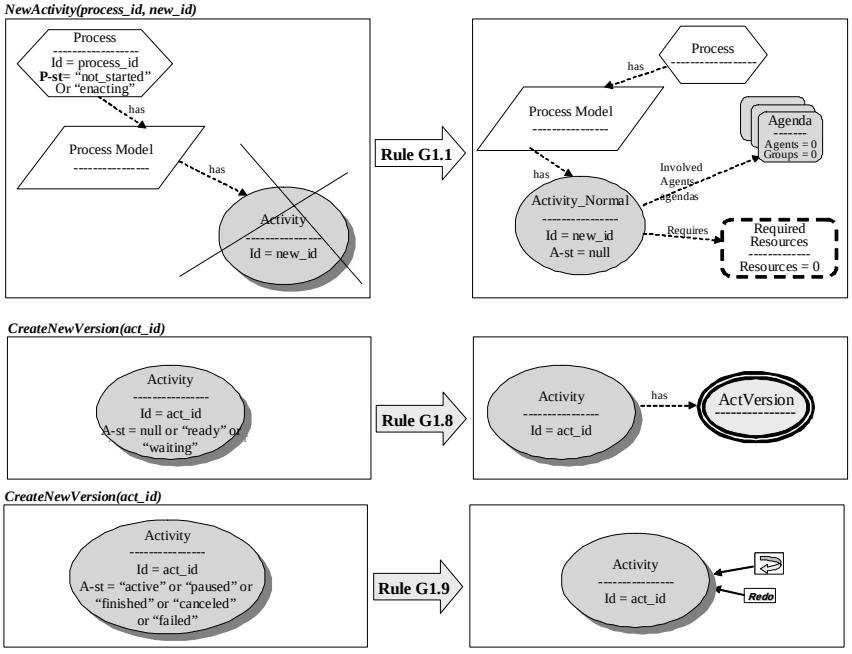


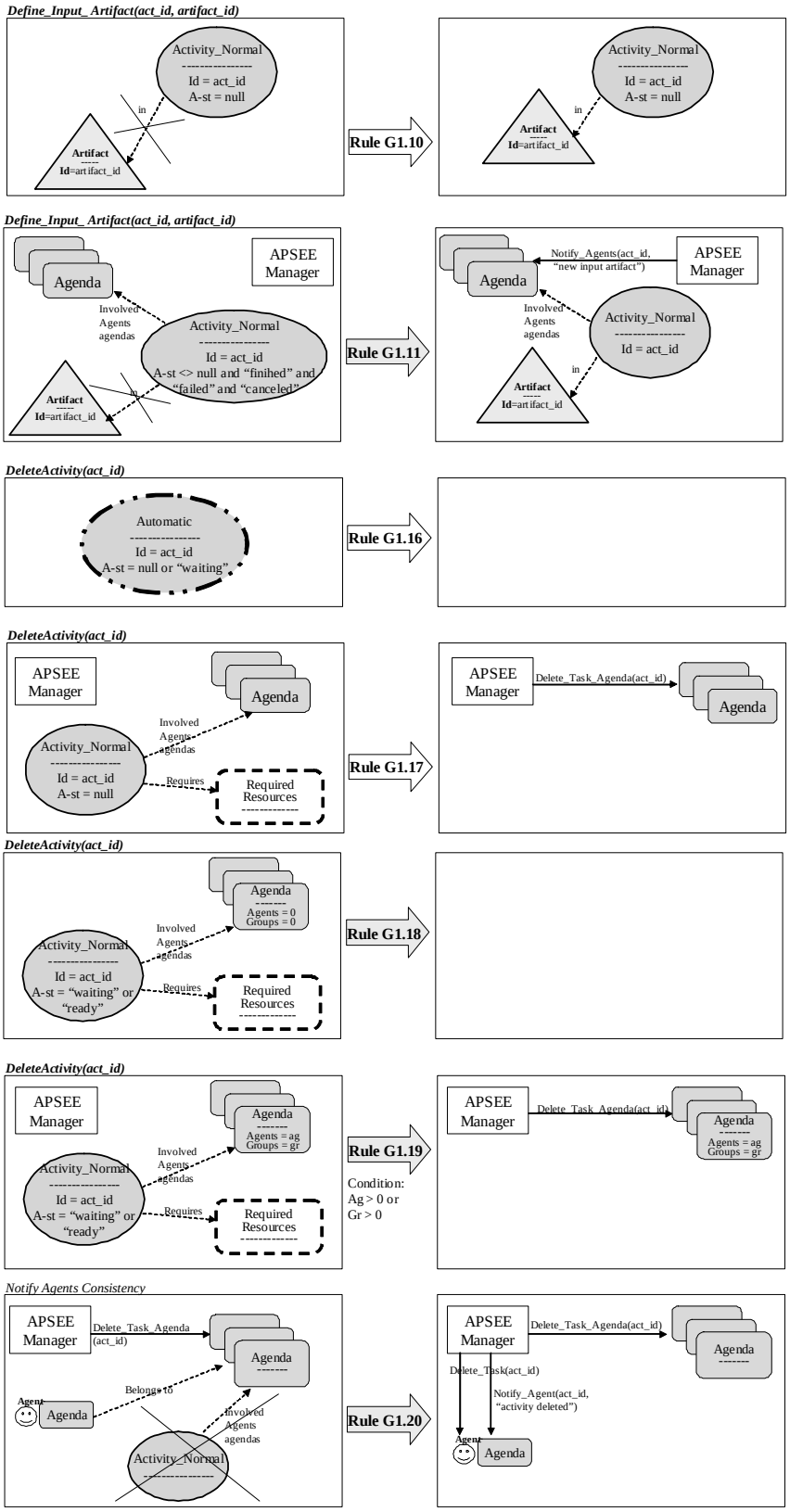
FIGURA A 21 - Classe *ClassMethodCall*.

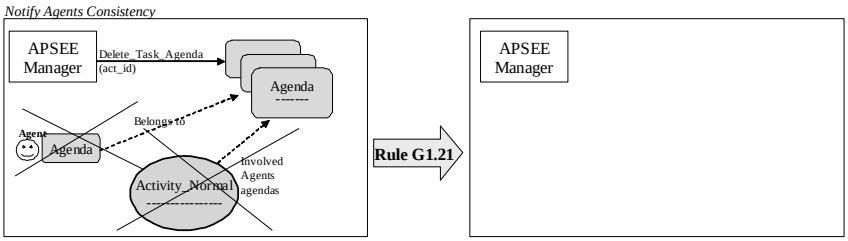
APÊNDICE B Regras de Sintaxe da APSEE-PML com Modificação Dinâmica

Este apêndice apresenta as principais regras de sintaxe da APSEE-PML que incluem as regras de modificação dinâmica. As regras são organizadas em grupos para facilitar a leitura. Algumas regras são omitidas pela sua simplicidade e por questões de espaço. O cabeçalho da regra define seu objetivo e seus argumentos. Normalmente esse objetivo é uma chamada do usuário no editor do APSEE. O estado das atividades e das conexões demonstra a situação do processo (em execução ou em modelagem). Além disso, algumas regras de consistência são necessárias para manter a integridade do modelo como resultado de uma ação do usuário.

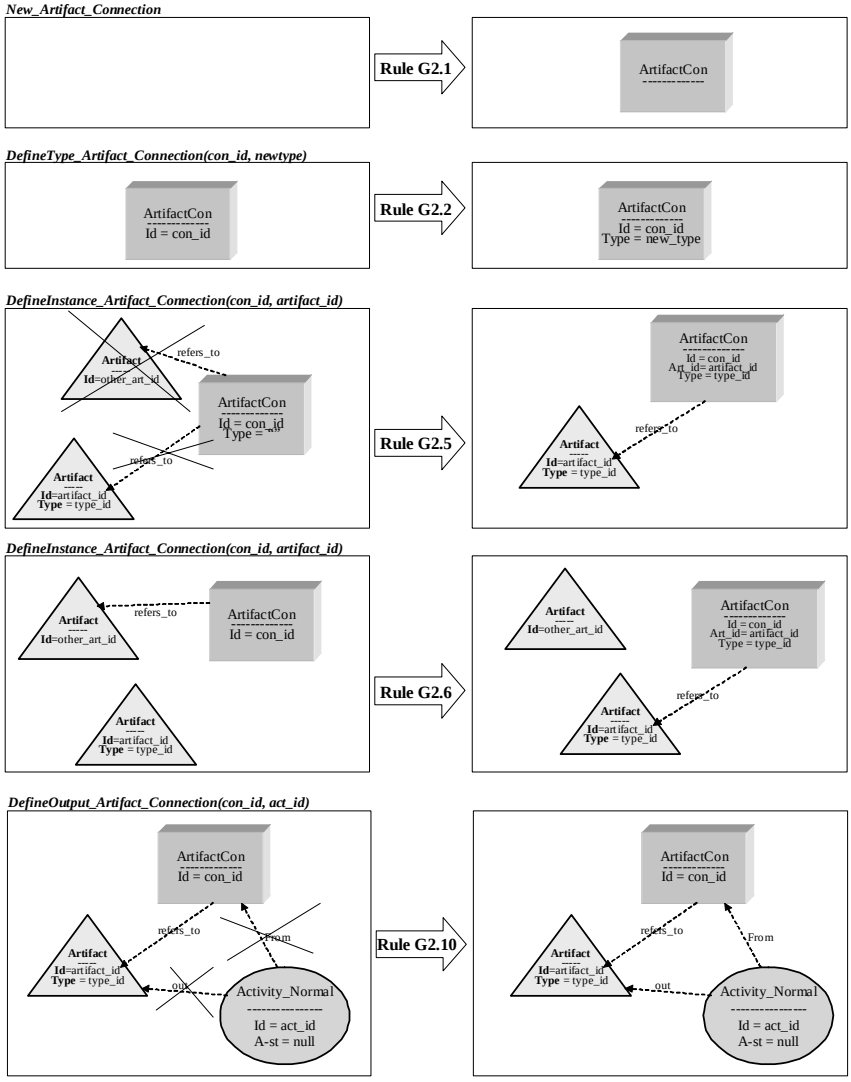
B.1. Algumas regras de Edição de Atividades

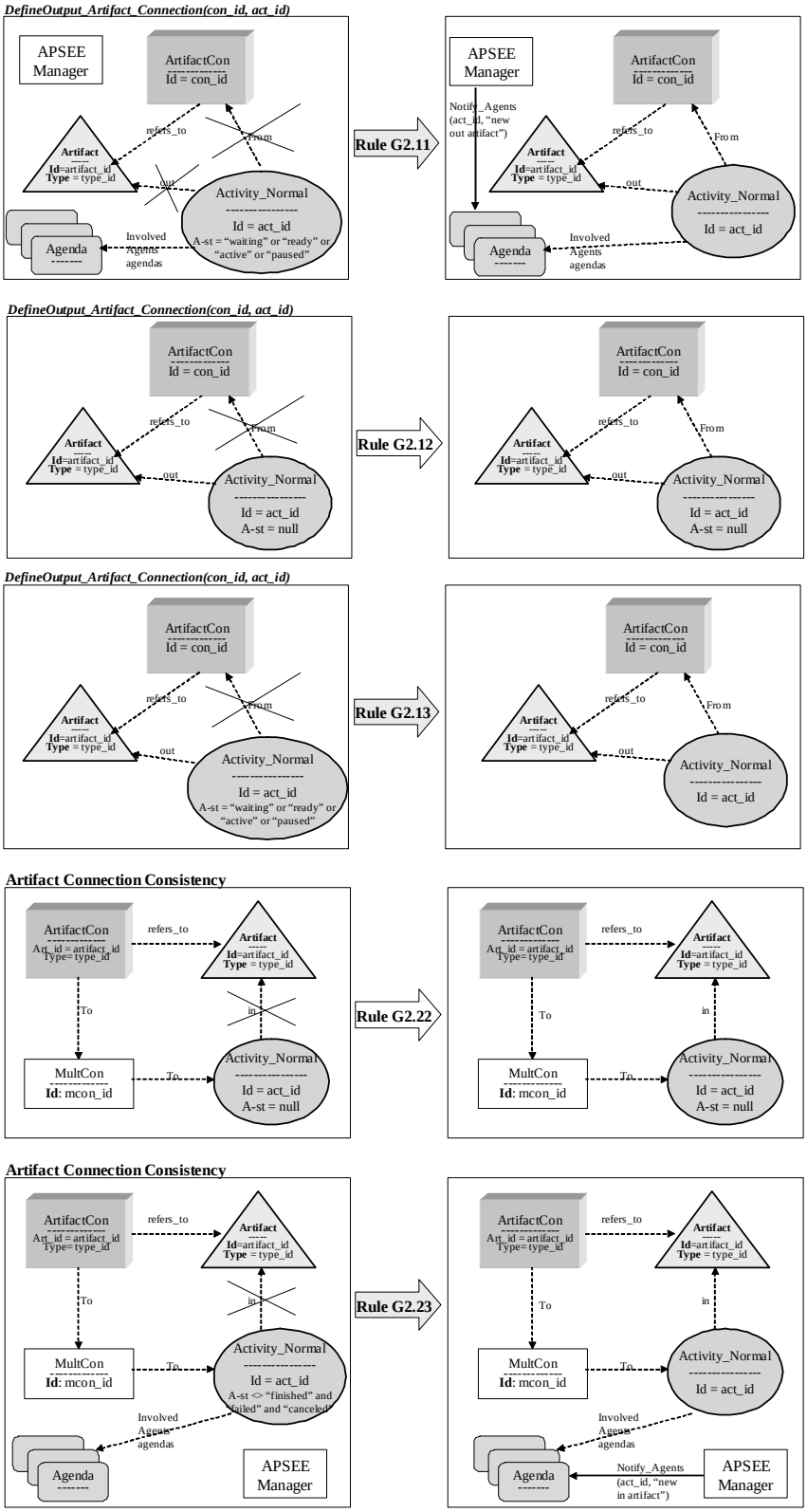


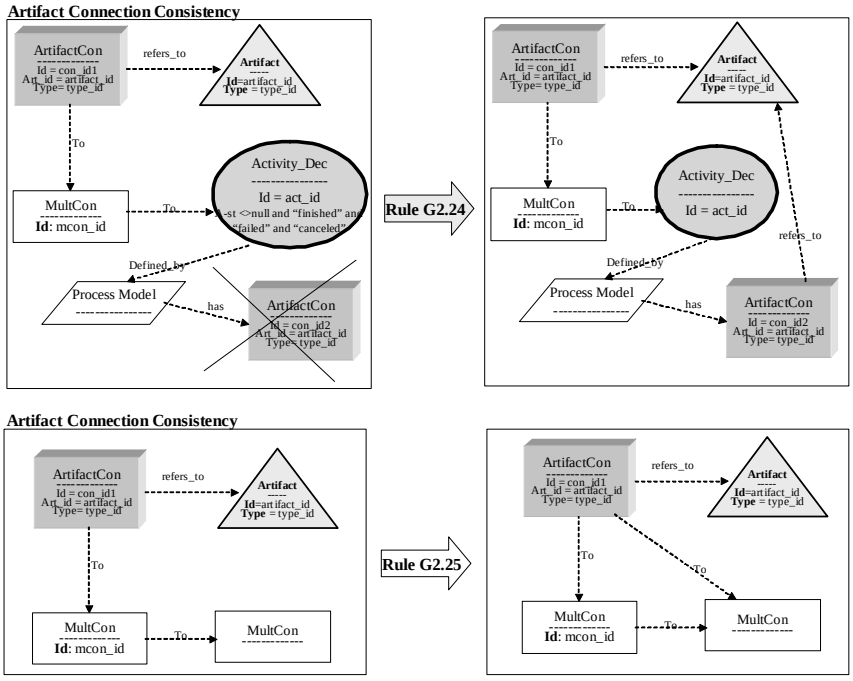

Define_Input_ Artifact(act_id, artifact_id)
Activity_Normal
Id = act_id
A-st = null
in
Artifact
Id=artifact_id
Rule G1.10
Activity_Normal
Id = act_id
A-st = null
in
Artifact
Id=artifact_id
Define_Input_ Artifact(act_id, artifact_id)
APSEE Manager
Agenda
Involved Agents agendas
Activity_Normal
Id = act_id
A-st <> null and "finihed" and "failed" and "canceled"
in
Artifact
Id=artifact_id
Rule G1.11
Notify_Agents(act_id, "new input artifact")
APSEE Manager
Agenda
Involved Agents agendas
Activity_Normal
Id = act_id
in
Artifact
Id=artifact_id
DeleteActivity(act_id)
Automatic
Id = act_id
A-st = null or "waiting"
Rule G1.16
DeleteActivity(act_id)
APSEE Manager
Agenda
Involved Agents agendas
Activity_Normal
Id = act_id
A-st = null
Requires
Required Resources
Rule G1.17
APSEE Manager
Delete_Task_Agenda(act_id)
Agenda
DeleteActivity(act_id)
Agenda
Agents = 0
Groups = 0
Involved Agents agendas
Activity_Normal
Id = act_id
A-st = "waiting" or "ready"
Requires
Required Resources
Rule G1.18
DeleteActivity(act_id)
APSEE Manager
Agenda
Agents = ag
Groups = gr
Involved Agents agendas
Activity_Normal
Id = act_id
A-st = "waiting" or "ready"
Requires
Required Resources
Rule G1.19
Condition:
Ag > 0 or
Gr > 0
APSEE Manager
Delete_Task_Agenda(act_id)
Agenda
Agents = ag
Groups = gr
Notify Agents Consistency
APSEE Manager
Delete_Task_Agenda (act_id)
Agenda
Agent
Agenda
Belongs to
Involved Agents agendas
Activity_Normal
Rule G1.20
APSEE Manager
Delete_Task_Agenda(act_id)
Agenda
Delete_Task(act_id)
Notify_Agent(act_id, "activity deleted")
Agent
Agenda



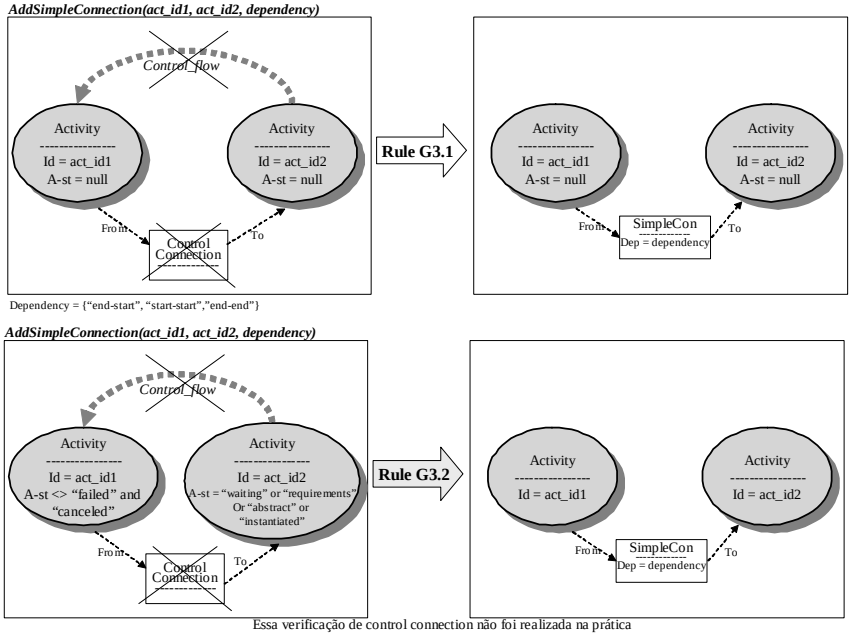
B.2. Definição de Conexões de Artefato

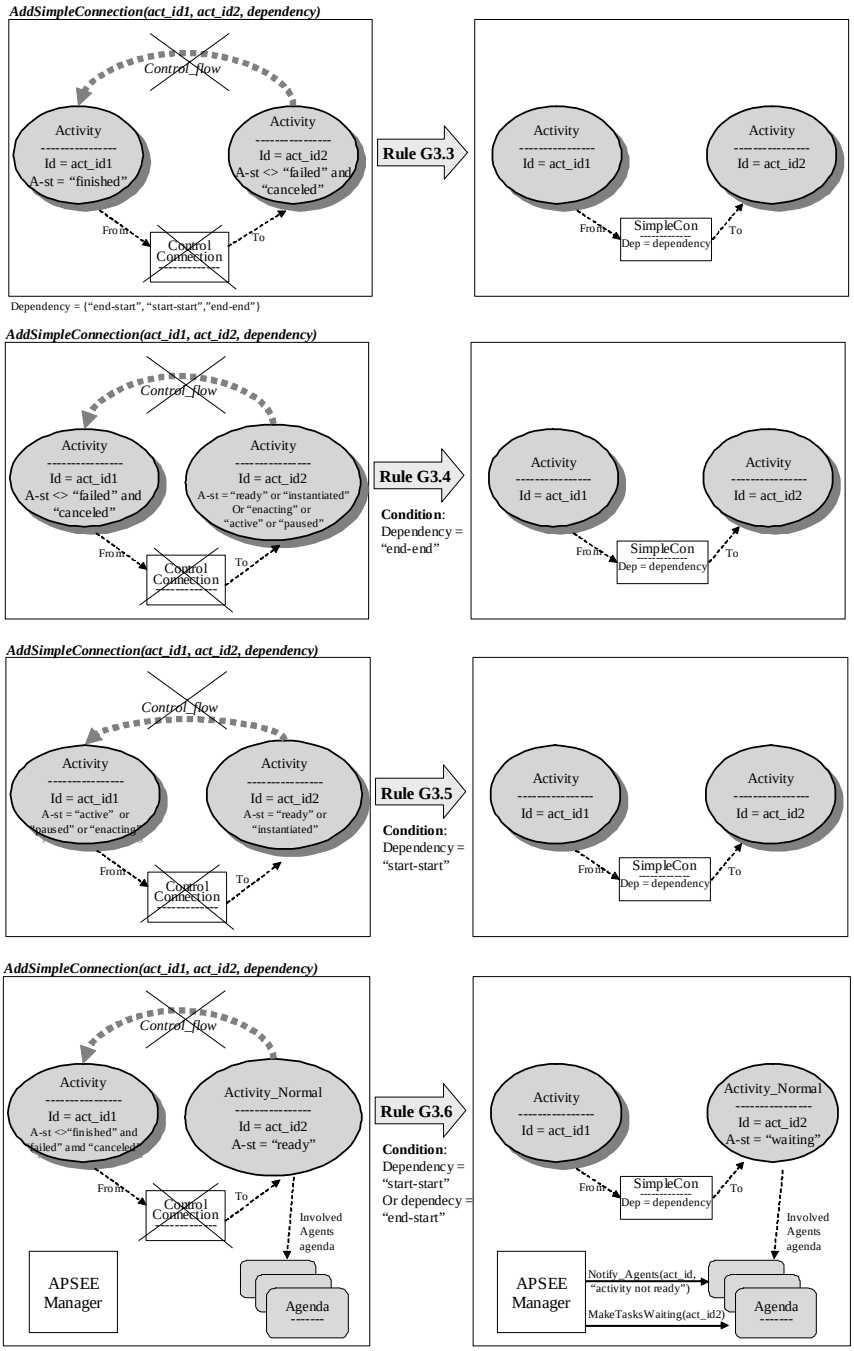


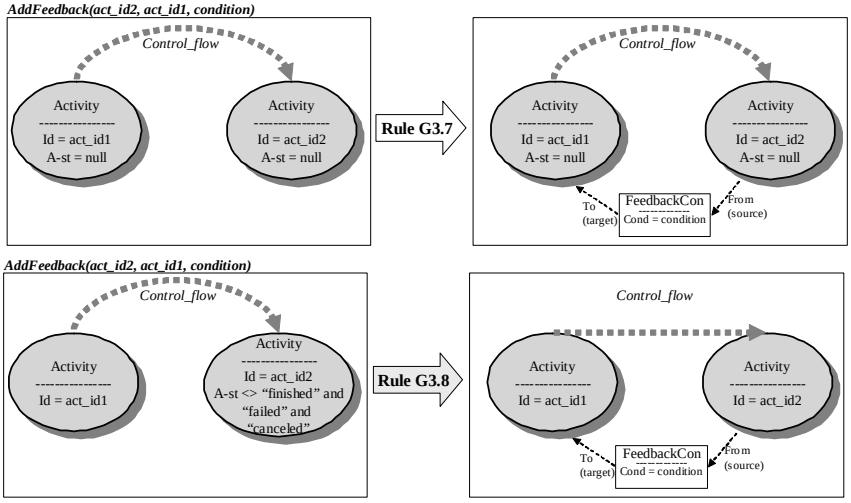

DefineOutput_Artifact_Connection(con_id, act_id)
APSEE Manager
ArtifactCon
Id = con_id
refers_to
From
Artifact
Id=artifact_id
Type = type_id
out
Activity_Normal
Id = act_id
A-st = "waiting" or "ready" or "active" or "paused"
Agenda
Involved Agents agendas
Rule G2.11
Notify_Agents (act_id, "new out artifact")
Activity_Normal
Id = act_id
Rule G2.12
A-st = null
DefineOutput_Artifact_Connection(con_id, act_id)
Rule G2.13
Artifact Connection Consistency
To
in
MultCon
Id: mcon_id
Rule G2.22
Rule G2.23
Notify_Agents (act_id, "new in artifact")



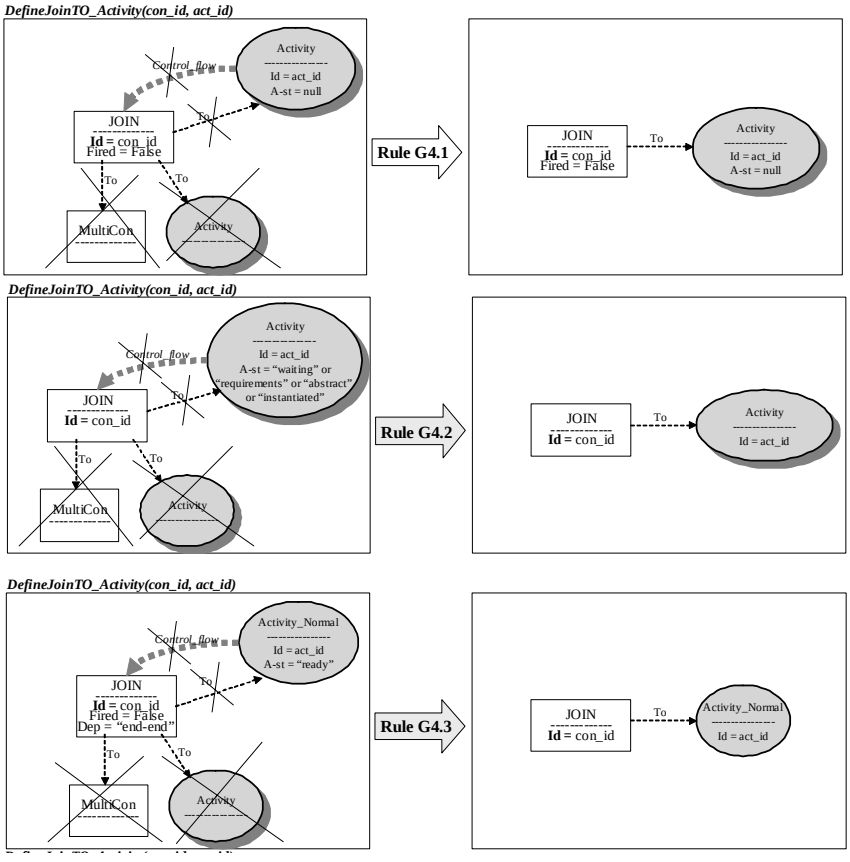
B.3. Definição de Conexões entre Atividades

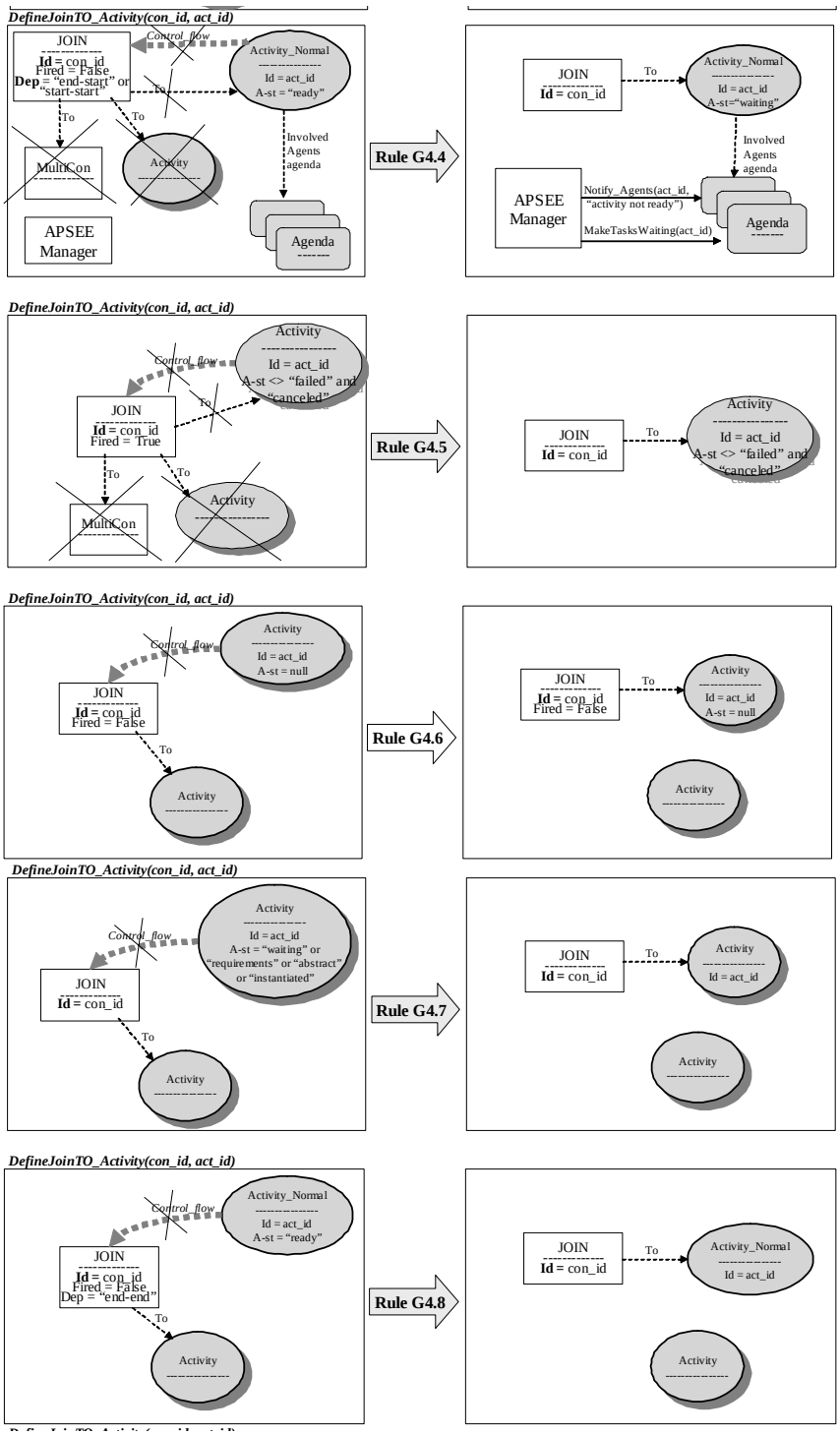


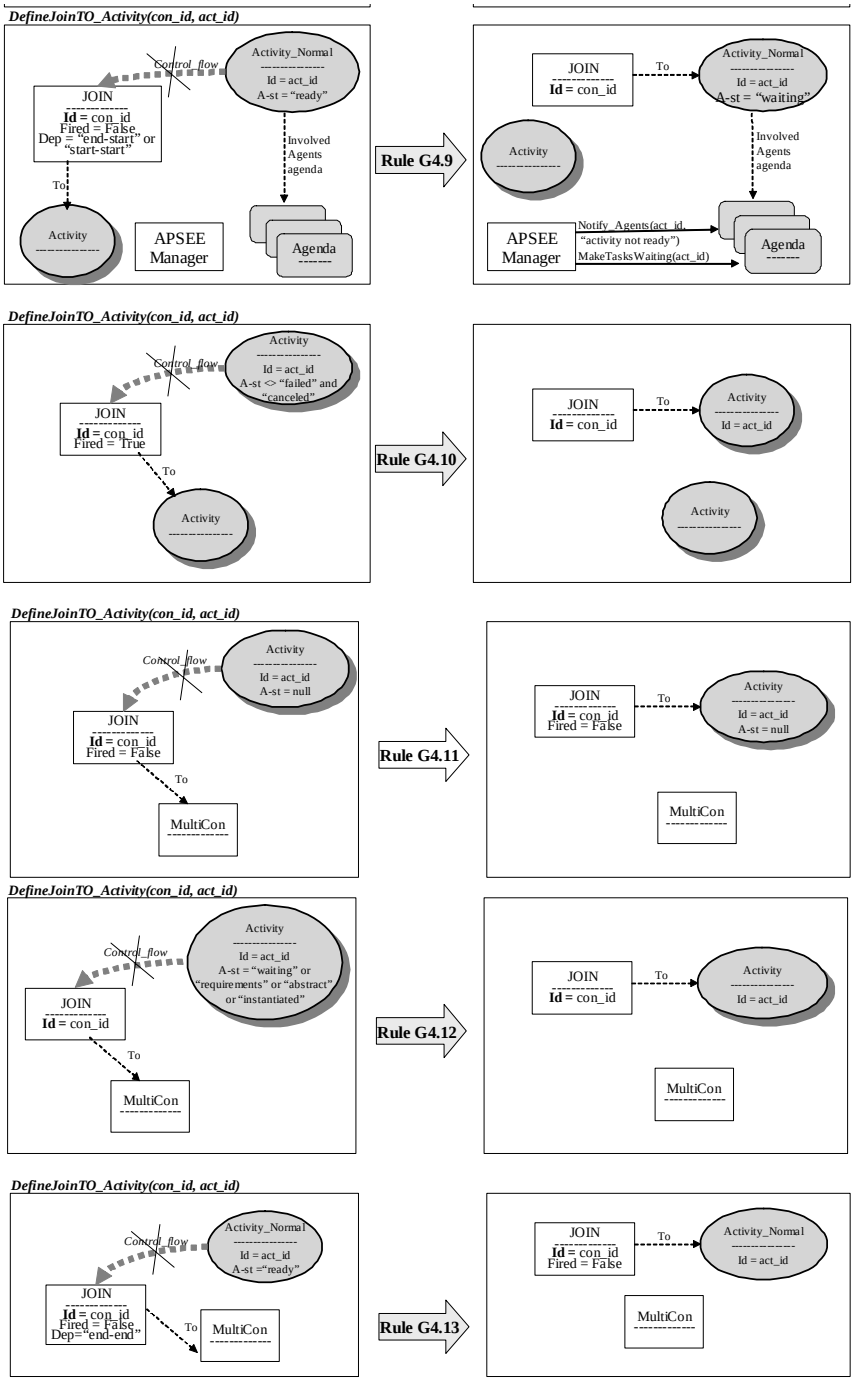

AddSimpleConnection(act_id1, act_id2, dependency)
Control_flow
Activity
Id = act_id1
A-st = "finished"
Activity
Id = act_id2
A-st <> "failed" and "canceled"
From
To
Control Connection
Rule G3.3
Activity
Id = act_id1
Activity
Id = act_id2
From
SimpleCon
Dep = dependency
To
Dependency = {"end-start", "start-start","end-end"}
AddSimpleConnection(act_id1, act_id2, dependency)
Control_flow
Activity
Id = act_id1
A-st <> "failed" and "canceled"
Activity
Id = act_id2
A-st = "ready" or "instantiated" Or "enacting" or "active" or "paused"
From
To
Control Connection
Rule G3.4
Condition:
Dependency = "end-end"
Activity
Id = act_id1
Activity
Id = act_id2
From
SimpleCon
Dep = dependency
To
AddSimpleConnection(act_id1, act_id2, dependency)
Control_flow
Activity
Id = act_id1
A-st = "active" or "paused" or "enacting"
Activity
Id = act_id2
A-st = "ready" or "instantiated"
From
To
Control Connection
Rule G3.5
Condition:
Dependency = "start-start"
Activity
Id = act_id1
Activity
Id = act_id2
From
SimpleCon
Dep = dependency
To
AddSimpleConnection(act_id1, act_id2, dependency)
Control_flow
Activity
Id = act_id1
A-st <>"finished" and "failed" and "canceled"
Activity_Normal
Id = act_id2
A-st = "ready"
From
To
Control Connection
Involved Agents agenda
APSEE Manager
Agenda
Rule G3.6
Condition:
Dependency = "start-start" Or dependecy = "end-start"
Activity
Id = act_id1
Activity_Normal
Id = act_id2
A-st = "waiting"
From
SimpleCon
Dep = dependency
To
Involved Agents agenda
APSEE Manager
Notify_Agents(act_id, "activity not ready")
MakeTasksWaiting(act_id2)
Agenda



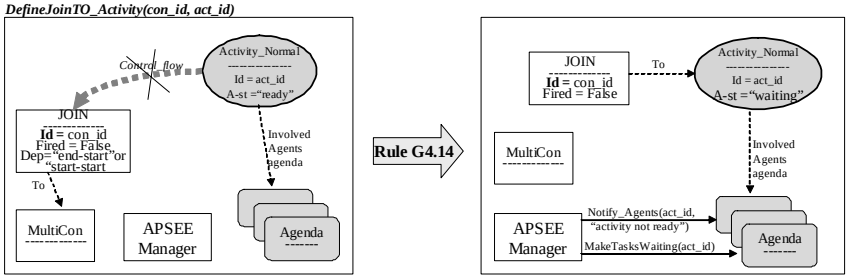
B.4. Definição de Conexões Join



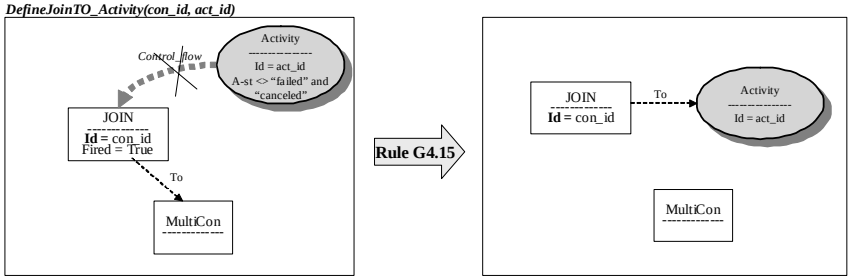

DefineJoinTO_Activity(con_id, act_id)
JOIN
Id = con_id
Fired = False
Dep = "end-start" or "start-start"
Control_flow
Activity_Normal
Id = act_id
A-st = "ready"
To
MultiCon
Activity
Involved Agents agenda
Agenda
APSEE Manager
Rule G4.4
JOIN
Id = con_id
Activity_Normal
Id = act_id
A-st="waiting"
Notify_Agents(act_id, "activity not ready")
MakeTasksWaiting(act_id)
DefineJoinTO_Activity(con_id, act_id)
Activity
Id = act_id
A-st <> "failed" and "canceled"
JOIN
Id = con_id
Fired = True
Rule G4.5
DefineJoinTO_Activity(con_id, act_id)
Activity
Id = act_id
A-st = null
JOIN
Id = con_id
Fired = False
Rule G4.6
DefineJoinTO_Activity(con_id, act_id)
Activity
Id = act_id
A-st = "waiting" or "requirements" or "abstract" or "instantiated"
JOIN
Id = con_id
Rule G4.7
DefineJoinTO_Activity(con_id, act_id)
Activity_Normal
Id = act_id
A-st = "ready"
JOIN
Id = con_id
Fired = False
Dep = "end-end"
Rule G4.8


DefineJoinTO_Activity(con_id, act_id)
Control_flow
Activity_Normal
Id = act_id
A-st = "ready"
JOIN
Id = con_id
Fired = False
Dep = "end-start" or "start-start"
To
Activity
Involved Agents agenda
Agenda
APSEE Manager
Rule G4.9
JOIN
Id = con_id
To
Activity_Normal
Id = act_id
A-st = "waiting"
Activity
Involved Agents agenda
Notify_Agents(act_id, "activity not ready")
MakeTasksWaiting(act_id)
DefineJoinTO_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st <> "failed" and "canceled"
JOIN
Id = con_id
Fired = True
To
Activity
Rule G4.10
JOIN
Id = con_id
To
Activity
Id = act_id
Activity
DefineJoinTO_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = null
JOIN
Id = con_id
Fired = False
To
MultiCon
Rule G4.11
JOIN
Id = con_id
Fired = False
To
Activity
Id = act_id
A-st = null
MultiCon
DefineJoinTO_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = "waiting" or "requirements" or "abstract" or "instantiated"
JOIN
Id = con_id
To
MultiCon
Rule G4.12
JOIN
Id = con_id
To
Activity
Id = act_id
MultiCon
DefineJoinTO_Activity(con_id, act_id)
Control_flow
Activity_Normal
Id = act_id
A-st ="ready"
JOIN
Id = con_id
Fired = False
Dep="end-end"
To
MultiCon
Rule G4.13
JOIN
Id = con_id
Fired = False
To
Activity_Normal
Id = act_id
MultiCon

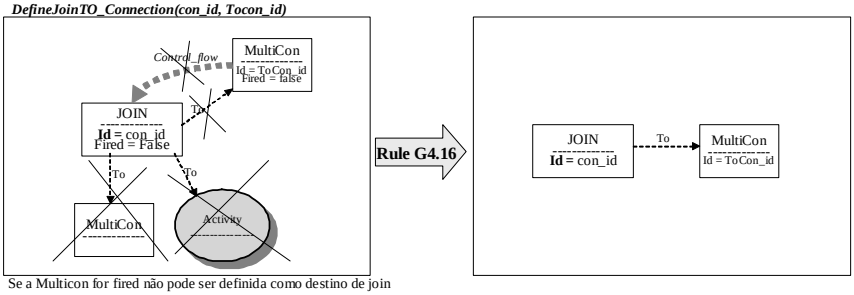
DefineJoinTO_Activity(con_id, act_id)



DefineJoinTO_Activity(con_id, act_id)



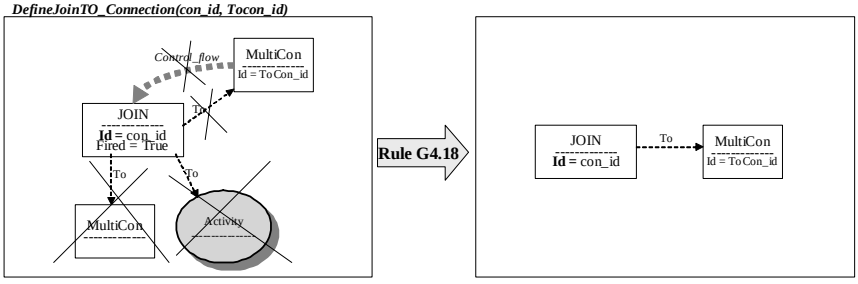
DefineJoinTO_Connection(con_id, Tocon_id)

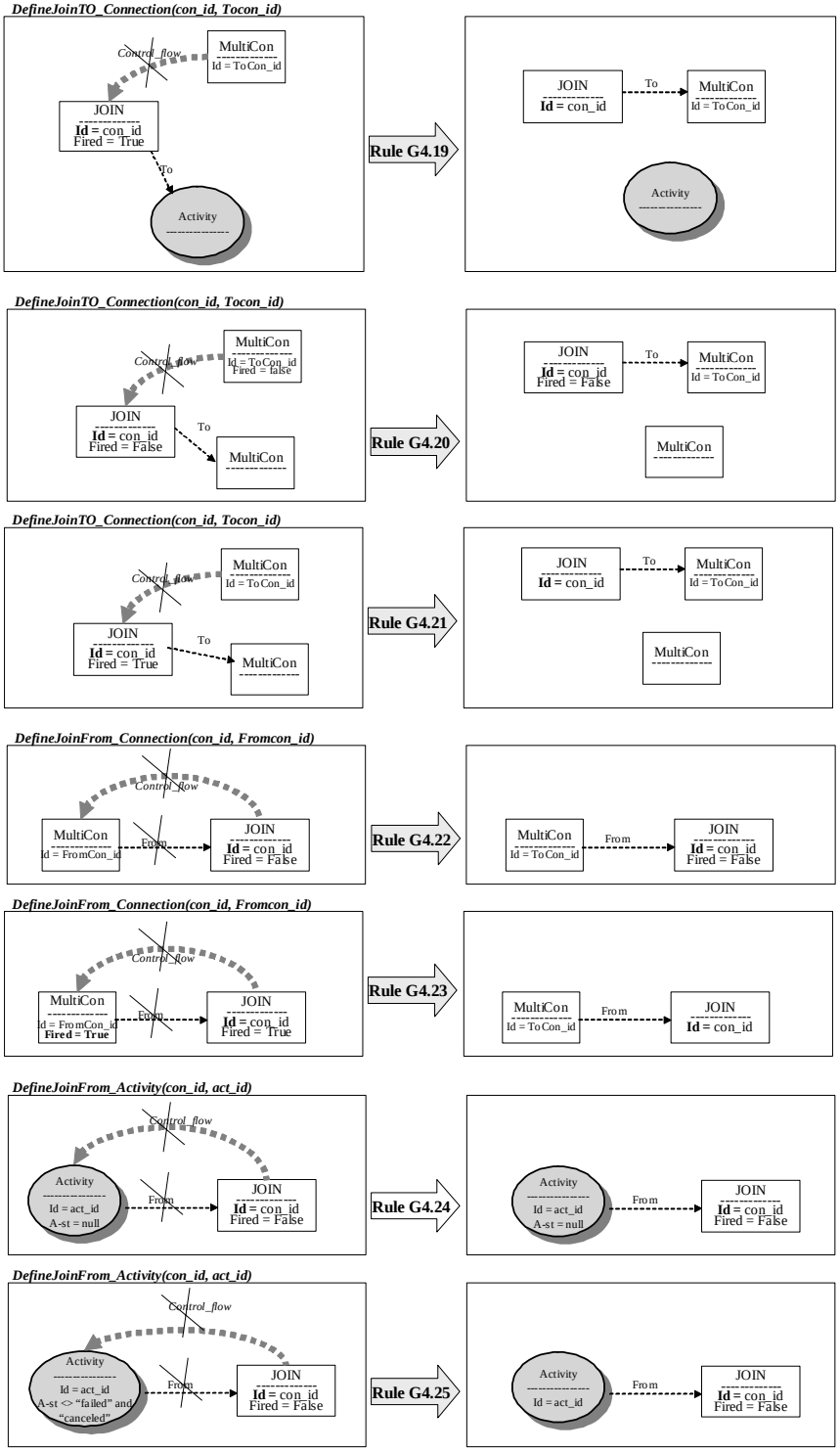


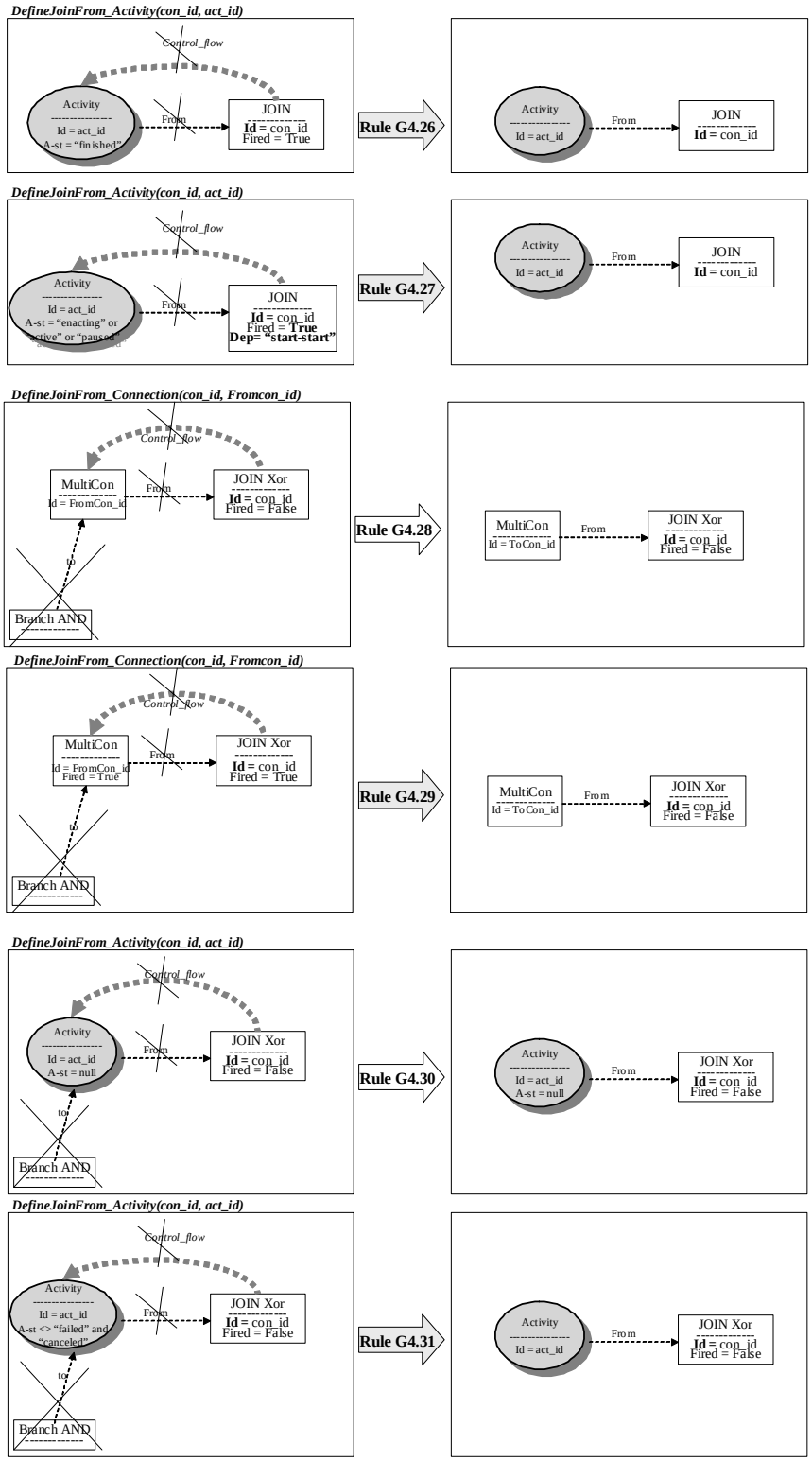
Se a Multicon for fired não pode ser definida como destino de join

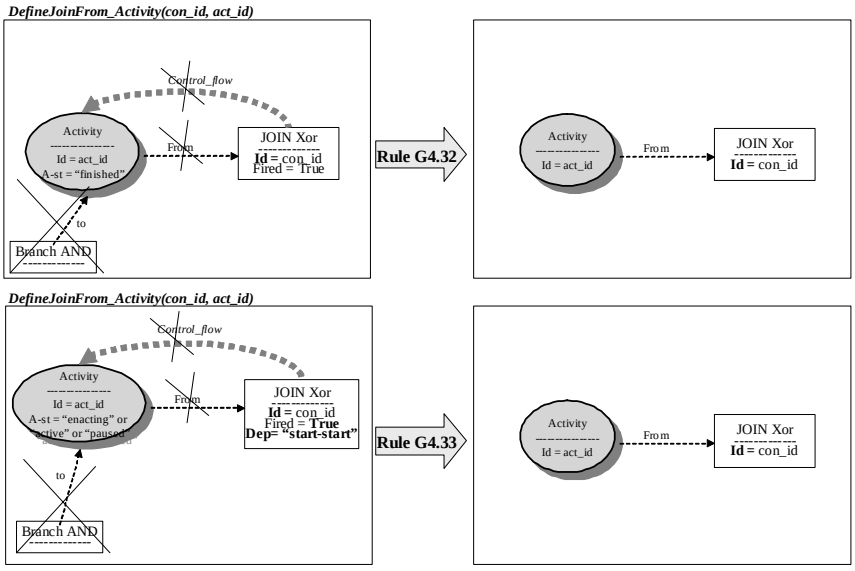
DefineJoinTO_Connection(con_id, Tocon_id)

DefineJoinTO_Connection(con_id, Tocon_id)




DefineJoinTO_Connection(con_id, Tocon_id)
Control_flow
MultiCon
Id = ToCon_id
JOIN
Id = con_id
Fired = True
To
Activity
Rule G4.19
JOIN
Id = con_id
To
MultiCon
Id = ToCon_id
Activity
DefineJoinTO_Connection(con_id, Tocon_id)
Control_flow
MultiCon
Id = ToCon_id
Fired = false
JOIN
Id = con_id
Fired = False
To
MultiCon
Rule G4.20
JOIN
Id = con_id
Fired = False
To
MultiCon
Id = ToCon_id
MultiCon
DefineJoinTO_Connection(con_id, Tocon_id)
Control_flow
MultiCon
Id = ToCon_id
JOIN
Id = con_id
Fired = True
To
MultiCon
Rule G4.21
JOIN
Id = con_id
To
MultiCon
Id = ToCon_id
MultiCon
DefineJoinFrom_Connection(con_id, Fromcon_id)
Control_flow
MultiCon
Id = FromCon_id
From
JOIN
Id = con_id
Fired = False
Rule G4.22
MultiCon
Id = ToCon_id
From
JOIN
Id = con_id
Fired = False
DefineJoinFrom_Connection(con_id, Fromcon_id)
Control_flow
MultiCon
Id = FromCon_id
Fired = True
From
JOIN
Id = con_id
Fired = True
Rule G4.23
MultiCon
Id = ToCon_id
From
JOIN
Id = con_id
DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = null
From
JOIN
Id = con_id
Fired = False
Rule G4.24
Activity
Id = act_id
A-st = null
From
JOIN
Id = con_id
Fired = False
DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st <> “failed” and
“canceled”
From
JOIN
Id = con_id
Fired = False
Rule G4.25
Activity
Id = act_id
From
JOIN
Id = con_id
Fired = False

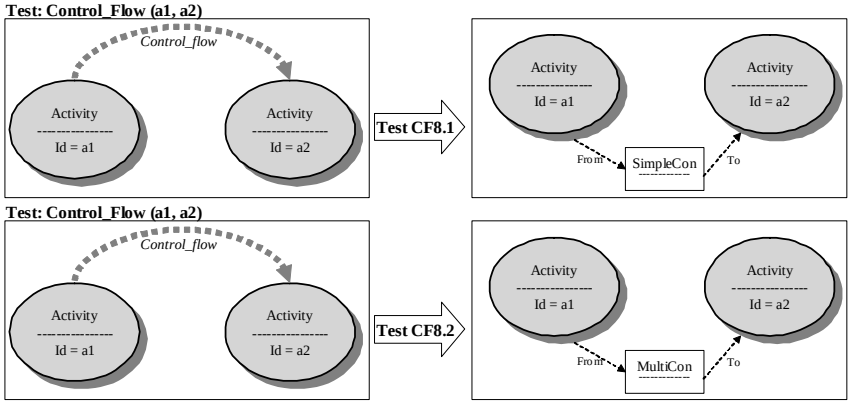

DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = "finished"
From
JOIN
Id = con_id
Fired = True
Rule G4.26
Activity
Id = act_id
From
JOIN
Id = con_id
DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = "enacting" or "active" or "paused"
From
JOIN
Id = con_id
Fired = True
Dep= "start-start"
Rule G4.27
Activity
Id = act_id
From
JOIN
Id = con_id
DefineJoinFrom_Connection(con_id, Fromcon_id)
Control_flow
MultiCon
Id = FromCon_id
From
JOIN Xor
Id = con_id
Fired = False
to
Branch AND
Rule G4.28
MultiCon
Id = ToCon_id
From
JOIN Xor
Id = con_id
Fired = False
DefineJoinFrom_Connection(con_id, Fromcon_id)
Control_flow
MultiCon
Id = FromCon_id
Fired = True
From
JOIN Xor
Id = con_id
Fired = True
to
Branch AND
Rule G4.29
MultiCon
Id = ToCon_id
From
JOIN Xor
Id = con_id
Fired = False
DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st = null
From
JOIN Xor
Id = con_id
Fired = False
to
Branch AND
Rule G4.30
Activity
Id = act_id
A-st = null
From
JOIN Xor
Id = con_id
Fired = False
DefineJoinFrom_Activity(con_id, act_id)
Control_flow
Activity
Id = act_id
A-st <> "failed" and "canceled"
From
JOIN Xor
Id = con_id
Fired = False
to
Branch AND
Rule G4.31
Activity
Id = act_id
From
JOIN Xor
Id = con_id
Fired = False

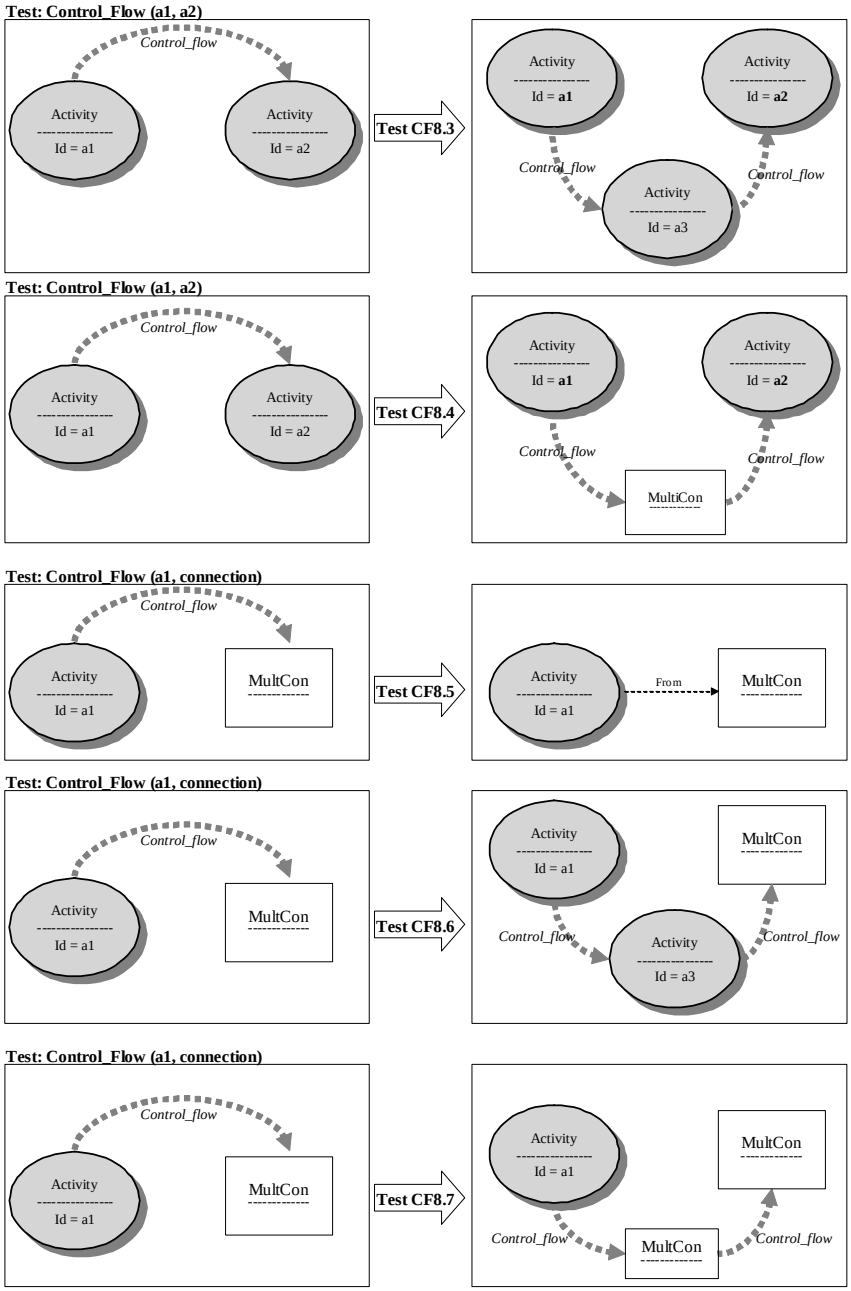


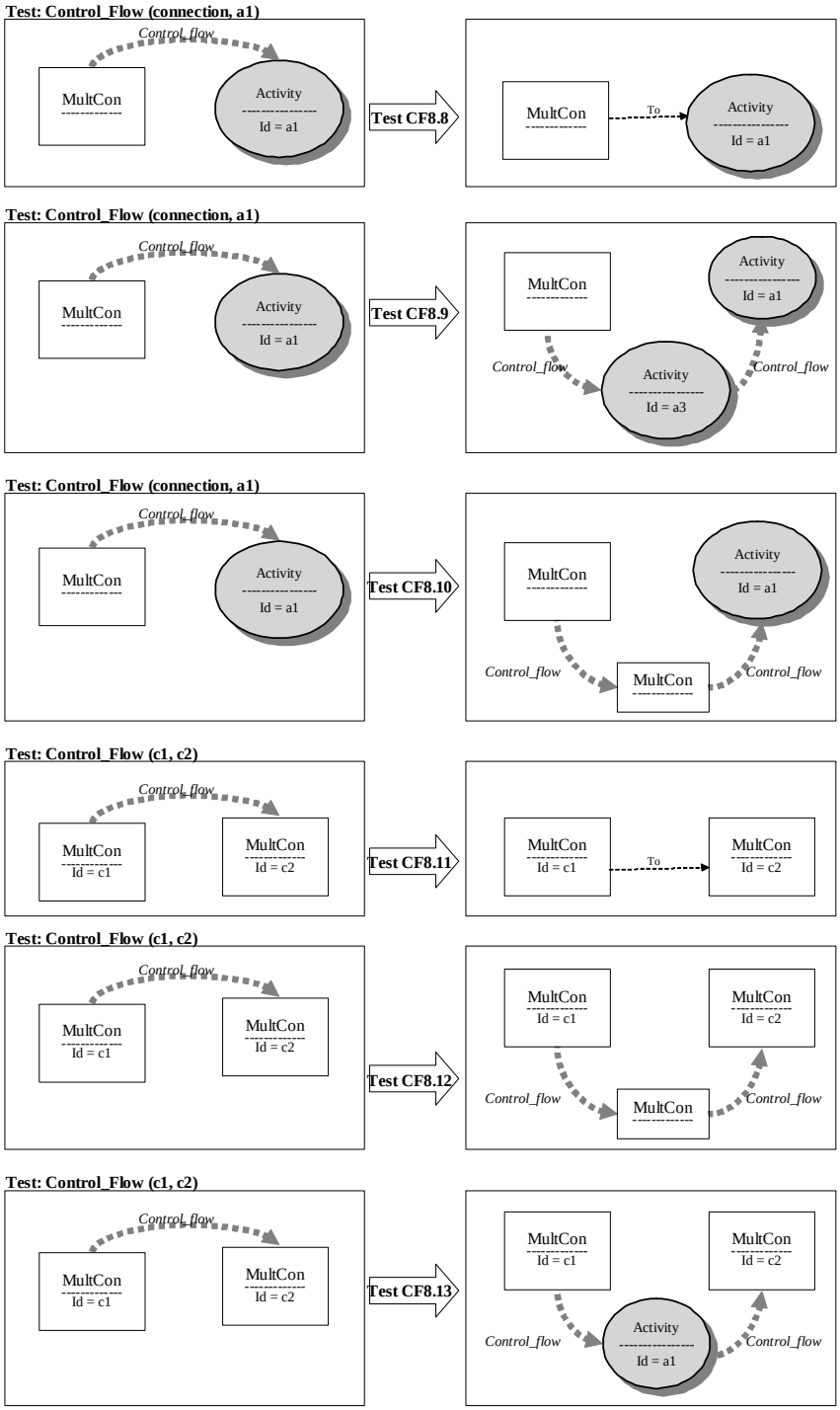
B.5. Definição de Conexões Branch

A definição de conexões Branch não será mostrada porque foi construída através de raciocínio análogo ao *Join*, considerando a inversão dos números de antecessores e predecessores.

B.6. Teste de Fluxo de Controle entre Atividades/Conexões



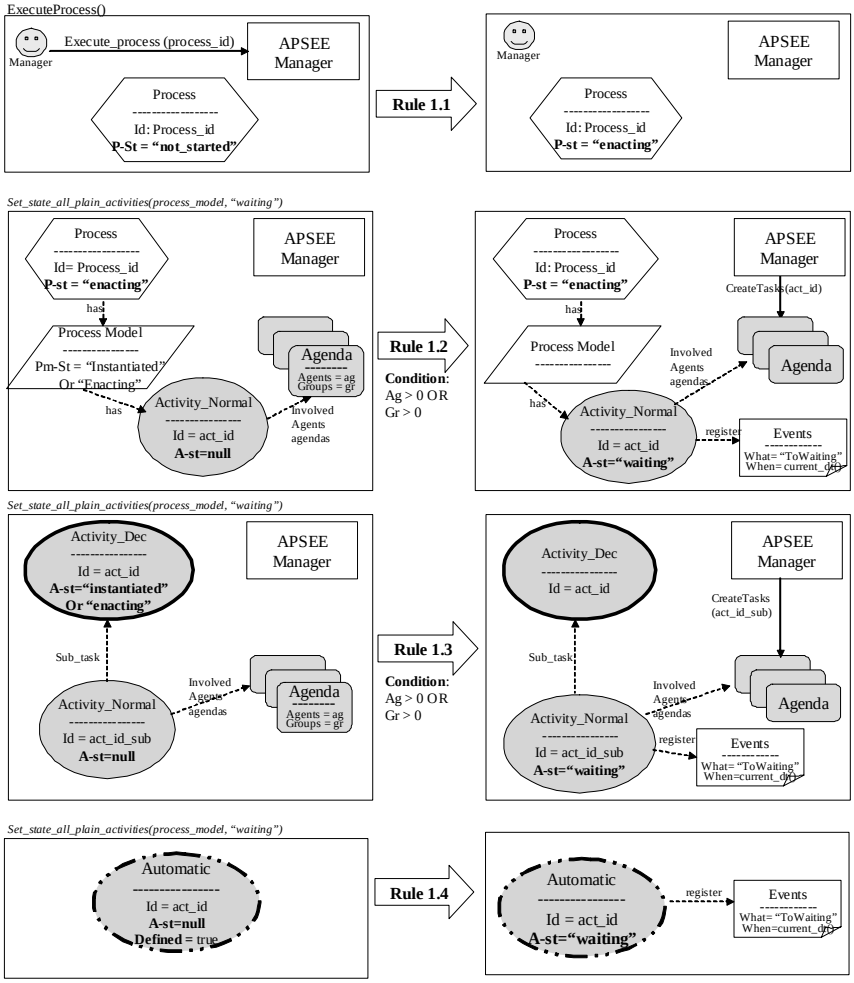

Test: Control_Flow (a1, a2)
Control_flow
Activity
Id = a1
Activity
Id = a2
Test CF8.3
Activity
Id = a1
Activity
Id = a2
Control_flow
Activity
Id = a3
Control_flow
Test: Control_Flow (a1, a2)
Control_flow
Activity
Id = a1
Activity
Id = a2
Test CF8.4
Activity
Id = a1
Activity
Id = a2
Control_flow
MultiCon
Control_flow
Test: Control_Flow (a1, connection)
Control_flow
Activity
Id = a1
MultCon
Test CF8.5
Activity
Id = a1
From
MultCon
Test: Control_Flow (a1, connection)
Control_flow
Activity
Id = a1
MultCon
Test CF8.6
Activity
Id = a1
MultCon
Control_flow
Activity
Id = a3
Control_flow
Test: Control_Flow (a1, connection)
Control_flow
Activity
Id = a1
MultCon
Test CF8.7
Activity
Id = a1
MultCon
Control_flow
MultCon
Control_flow


Test: Control_Flow (connection, a1)
Control_flow
MultCon
Activity
Id = a1
Test CF8.8
To
Test: Control_Flow (connection, a1)
Test CF8.9
Activity
Id = a3
Control_flow
Test: Control_Flow (connection, a1)
Test CF8.10
Test: Control_Flow (c1, c2)
MultCon
Id = c1
MultCon
Id = c2
Test CF8.11
Test: Control_Flow (c1, c2)
Test CF8.12
Test: Control_Flow (c1, c2)
Test CF8.13

APÊNDICE C Regras para Execução de Processos de Software

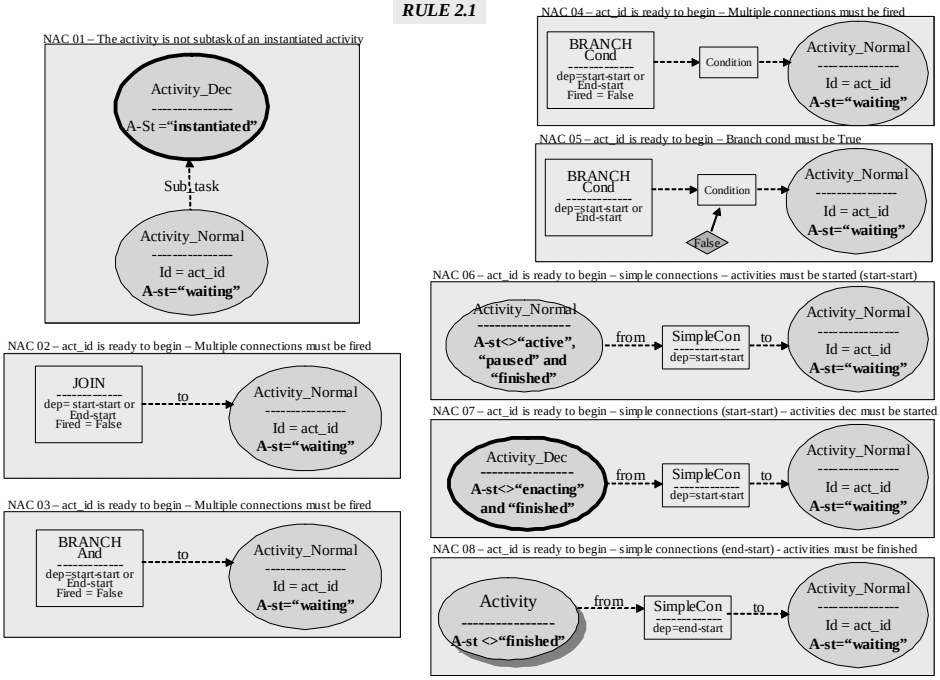
Este apêndice mostra as regras desenvolvidas para especificar o comportamento do mecanismo de execução do APSEE. Várias regras deste apêndice utilizam-se de NACs (condições negativas de aplicação) e foram divididas em grupos para facilitar sua leitura.

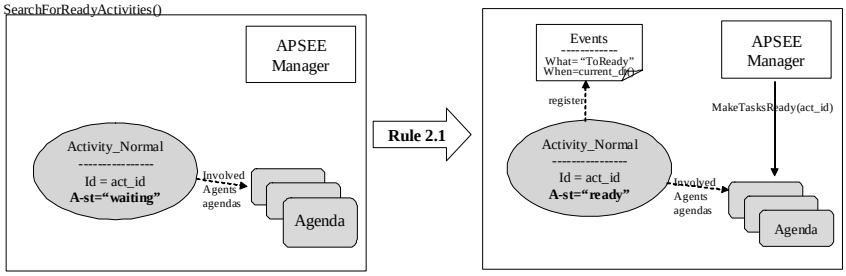
C.1. Regras do início da execução



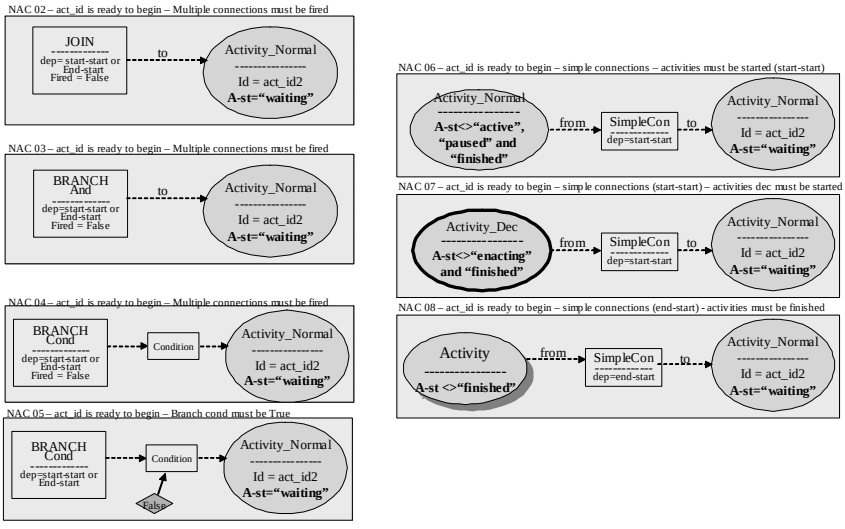
C.2. Regras que procuram atividades prontas para executar

RULE 2.1

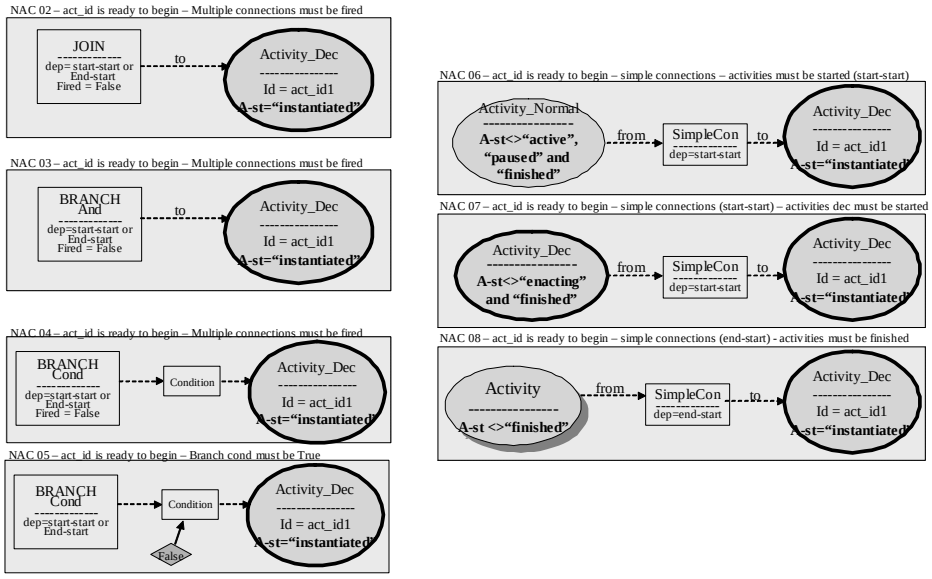


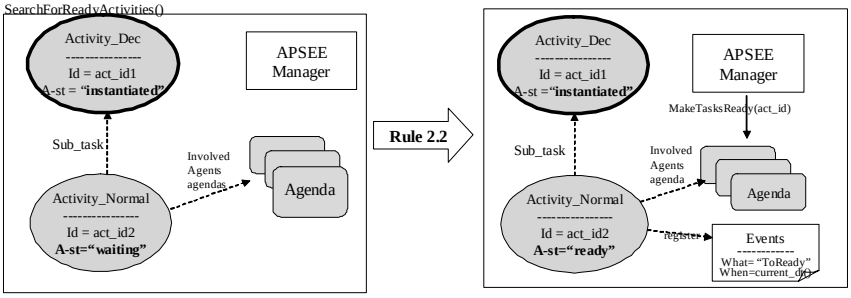


RULE 2.2

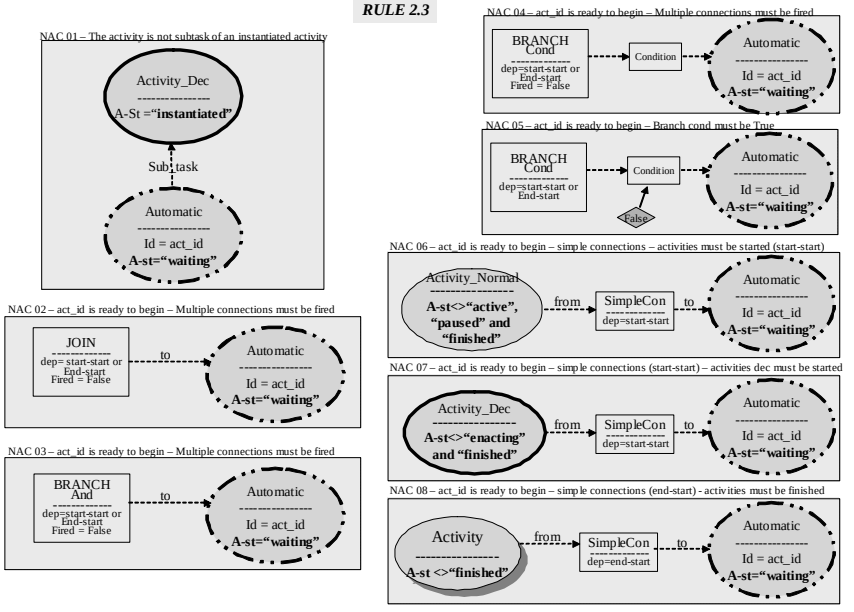


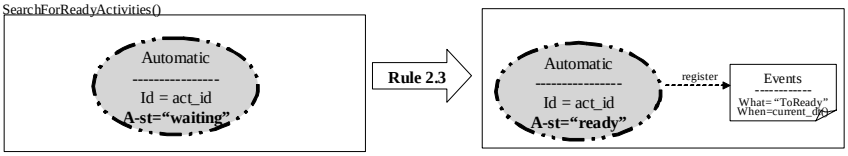
RULE 2.2



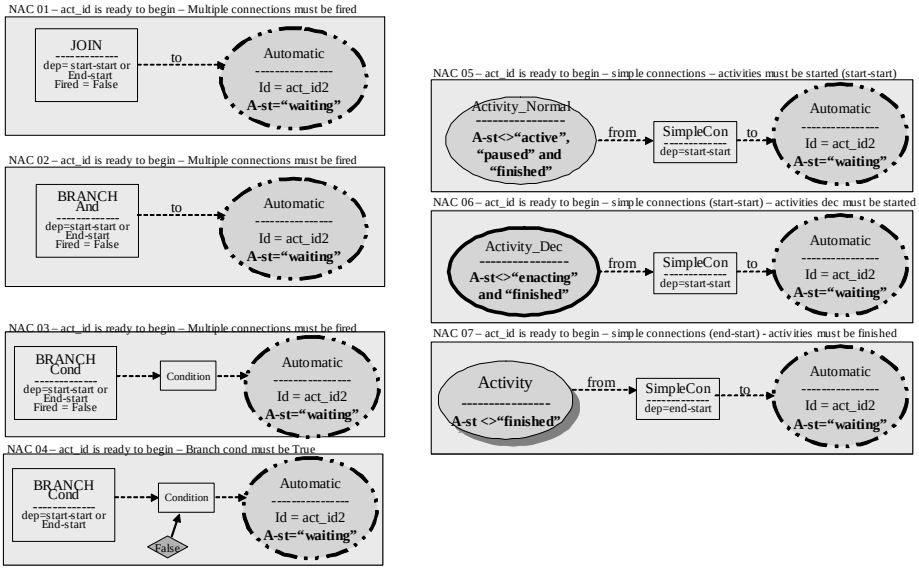


RULE 2.3

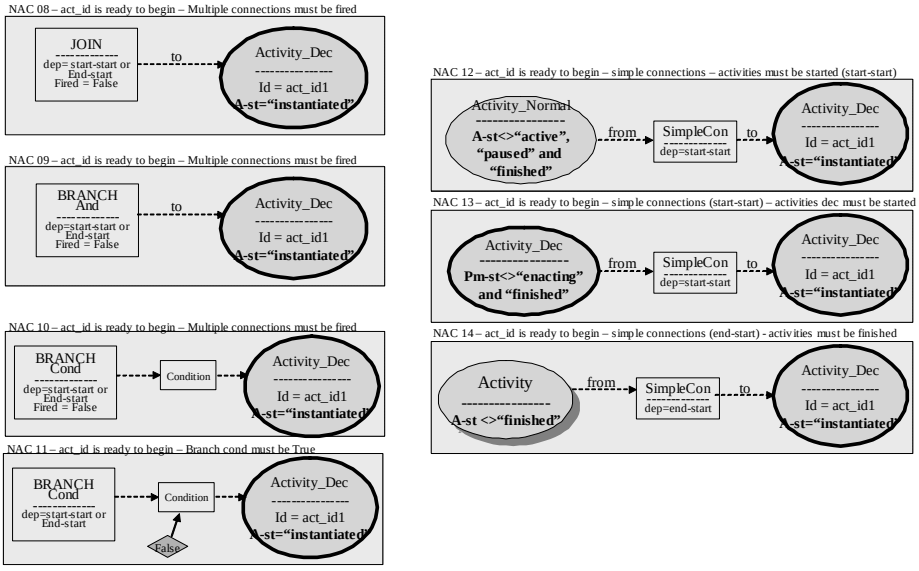


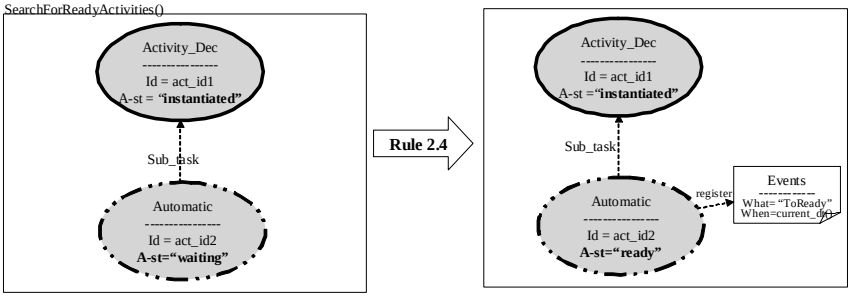


RULE 2.4

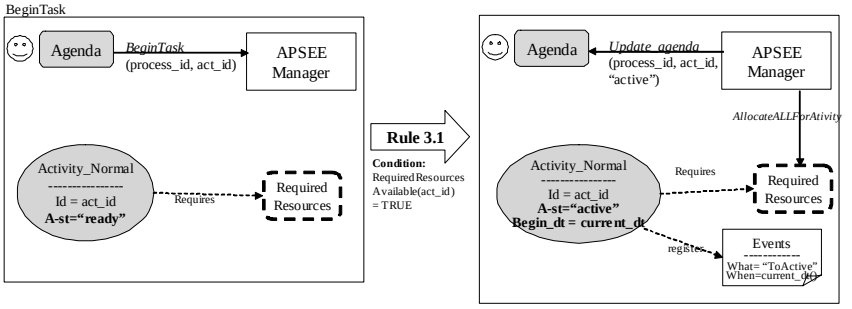


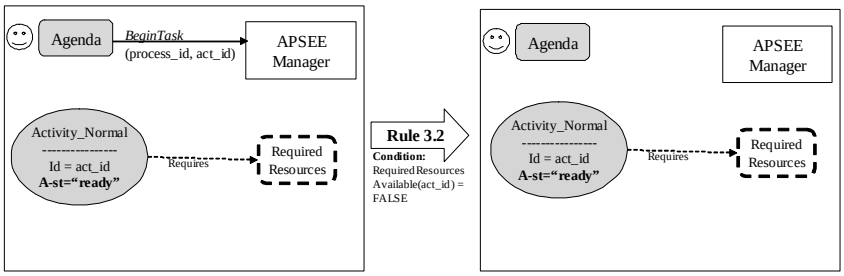
RULE 2.4

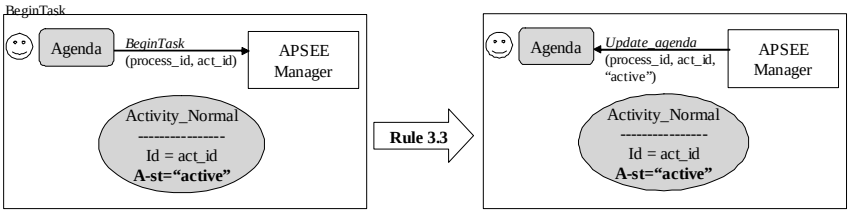


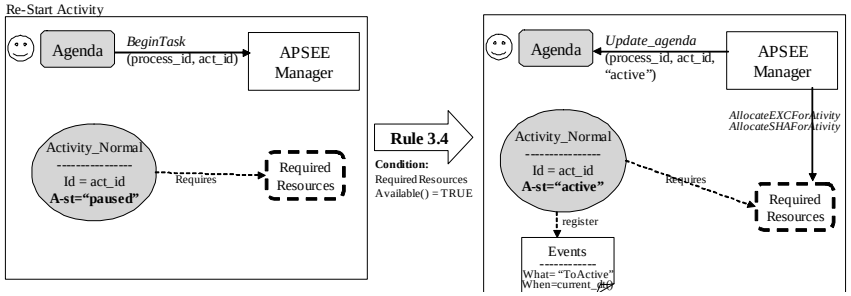


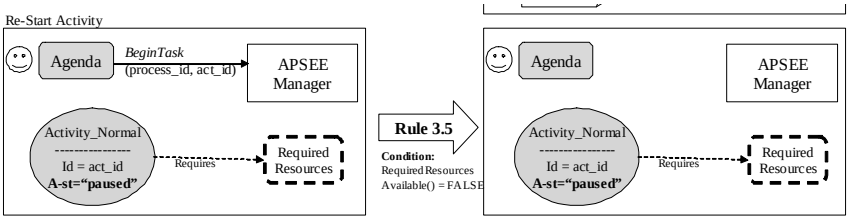
C.3. Regras para interação com o usuário



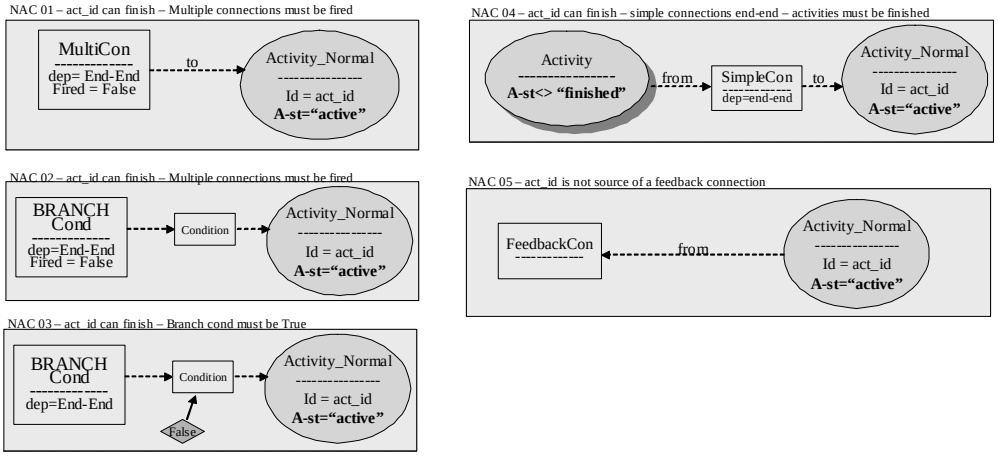


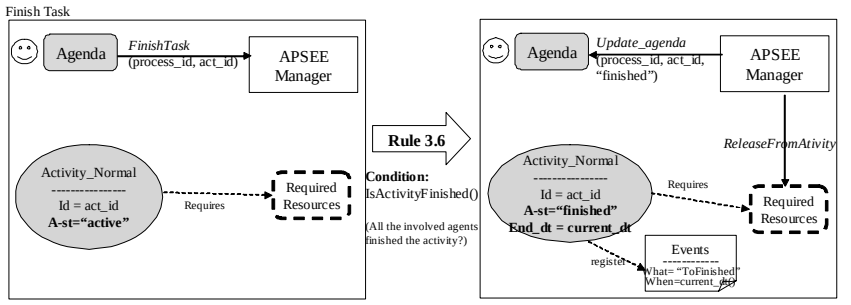


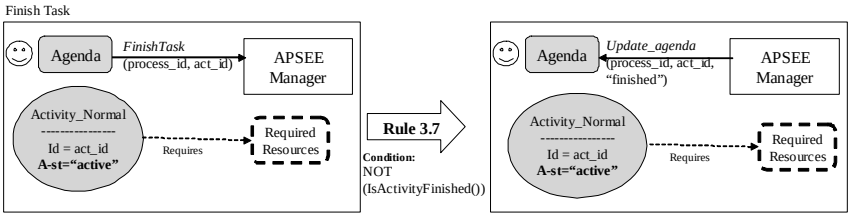


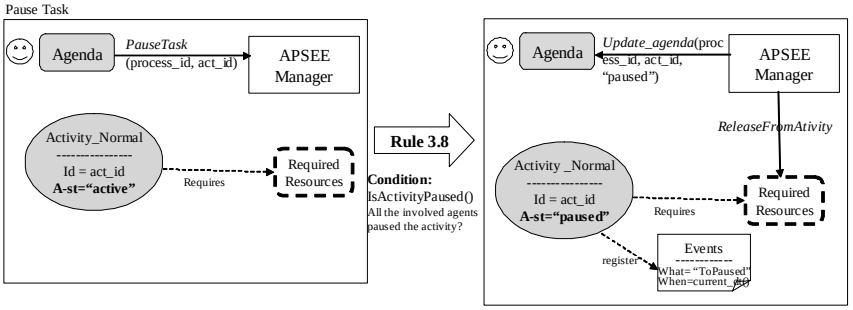

Re-Start Activity
Agenda
BeginTask
(process_id, act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st="paused"
Requires
Required Resources
Rule 3.5
Condition:
Required Resources Available() = FALSE
Agenda
APSEE Manager
Activity_Normal
Id = act_id
A-st="paused"
Requires
Required Resources

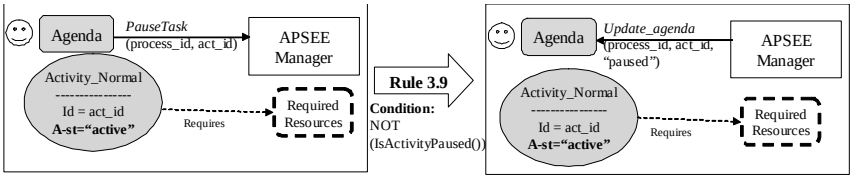
RULE 3.6


NAC 01 – act_id can finish – Multiple connections must be fired
MultiCon
dep= End-End
Fired = False
to
Activity_Normal
Id = act_id
A-st="active"
NAC 04 – act_id can finish – simple connections end-end – activities must be finished
Activity
A-st<> "finished"
from
SimpleCon
dep=end-end
to
Activity_Normal
Id = act_id
A-st="active"
NAC 02 – act_id can finish – Multiple connections must be fired
BRANCH Cond
dep=End-End
Fired = False
Condition
Activity_Normal
Id = act_id
A-st="active"
NAC 05 – act_id is not source of a feedback connection
FeedbackCon
from
Activity_Normal
Id = act_id
A-st="active"
NAC 03 – act_id can finish – Branch cond must be True
BRANCH Cond
dep=End-End
Condition
False
Activity_Normal
Id = act_id
A-st="active"


Finish Task
Agenda
FinishTask
(process_id, act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st="active"
Requires
Required Resources
Rule 3.6
Condition:
IsActivityFinished()
(All the involved agents finished the activity?)
Agenda
Update_agenda
(process_id, act_id, "finished")
APSEE Manager
ReleaseFromAtivity
Activity_Normal
Id = act_id
A-st="finished"
End_dt = current_dt
Requires
Required Resources
register
Events
What= "ToFinished"
When=current_dt

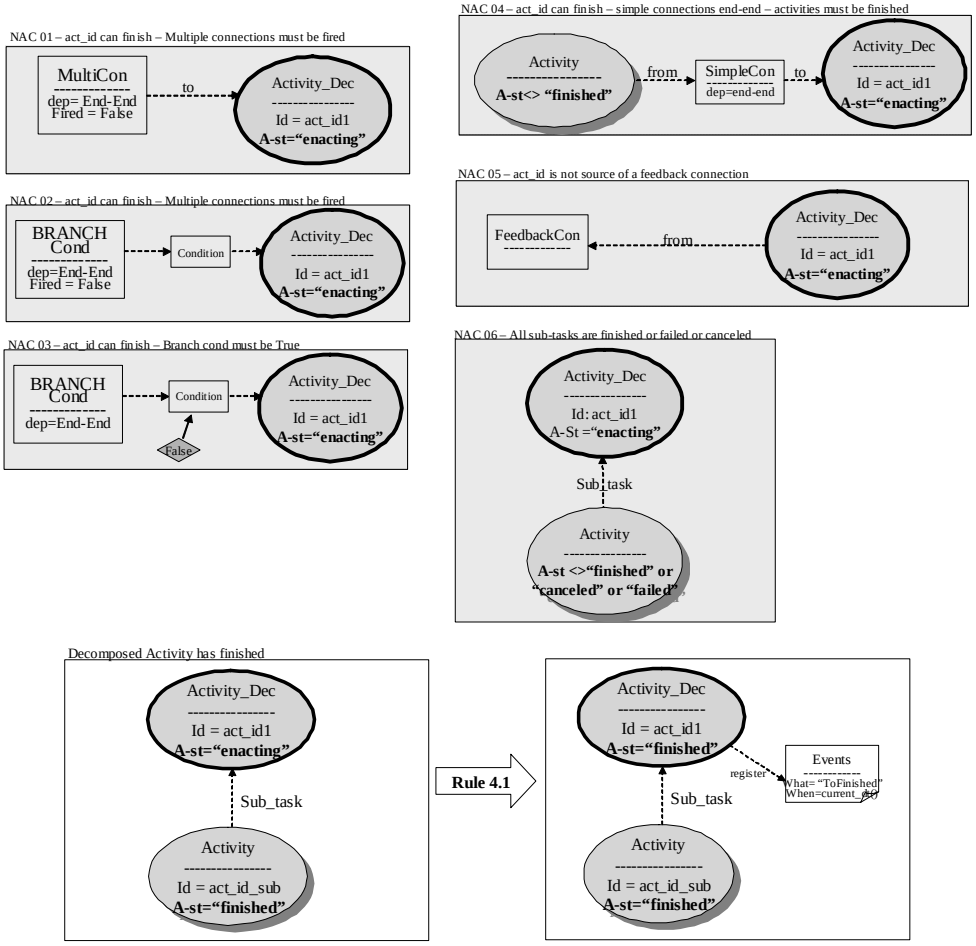

Finish Task
Agenda
FinishTask
(process_id, act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st="active"
Requires
Required Resources
Rule 3.7
Condition:
NOT (IsActivityFinished())
Agenda
Update_agenda
(process_id, act_id, "finished")
APSEE Manager
Activity_Normal
Id = act_id
A-st="active"
Requires
Required Resources


Pause Task
Agenda
PauseTask
(process_id, act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st="active"
Requires
Required Resources
Rule 3.8
Condition:
IsActivityPaused()
All the involved agents paused the activity?
Agenda
Update_agenda(process_id, act_id, "paused")
APSEE Manager
ReleaseFromAtivity
Activity _Normal
Id = act_id
A-st="paused"
Requires
Required Resources
register
Events
What= "ToPaused"
When=current_dt



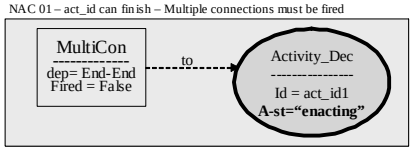
C.4. Regras que ajustam o estado de atividades decompostas

RULE 4.1

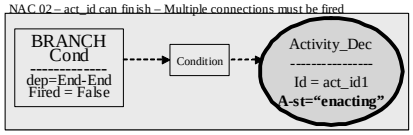


RULE 4.2

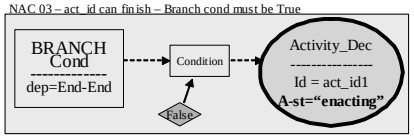
NAC 01 – act_id can finish – Multiple connections must be fired



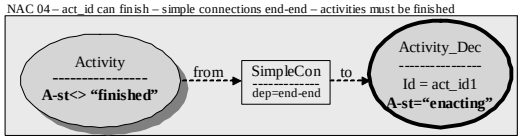
NAC 02 – act_id can finish – Multiple connections must be fired



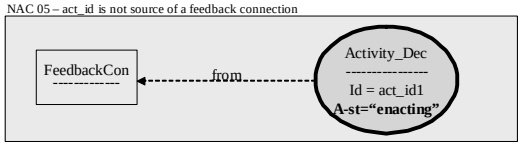
NAC 03 – act_id can finish – Branch cond must be True



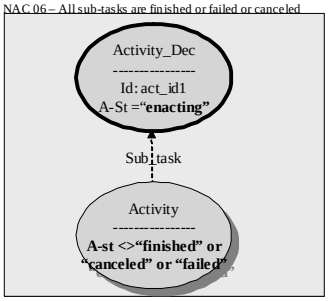
NAC 04 – act_id can finish – simple connections end-end – activities must be finished



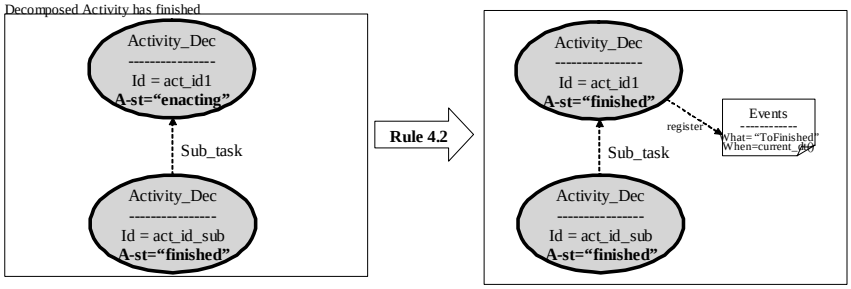
NAC 05 – act_id is not source of a feedback connection



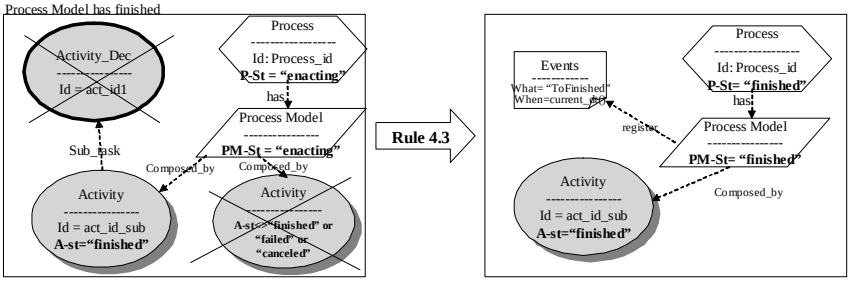
NAC 06 – All sub-tasks are finished or failed or canceled



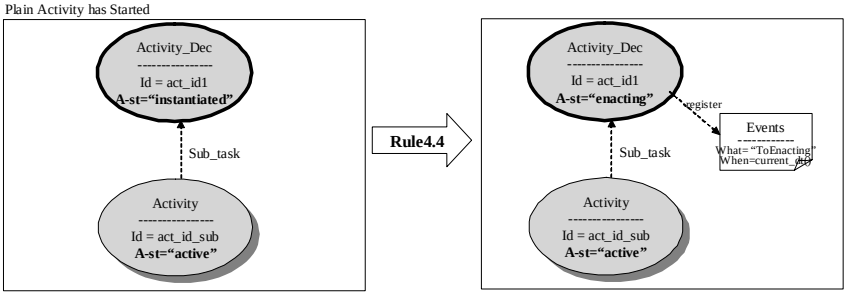
Decomposed Activity has finished

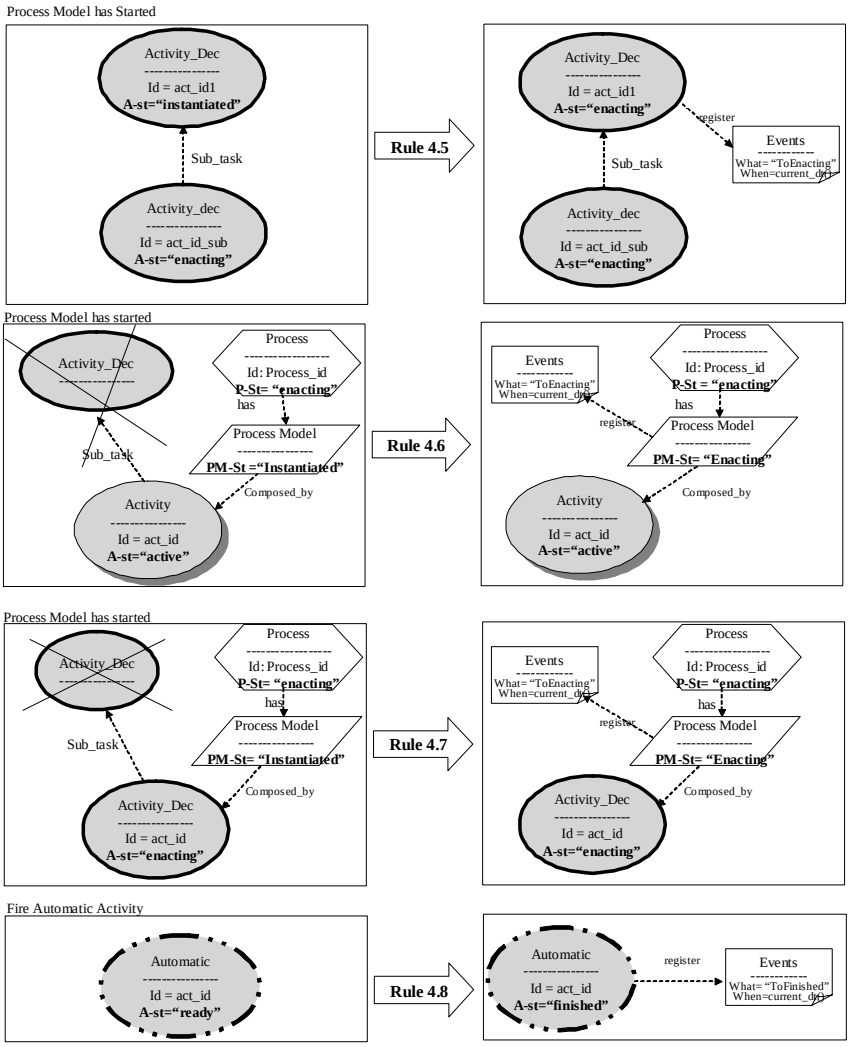


Process Model has finished

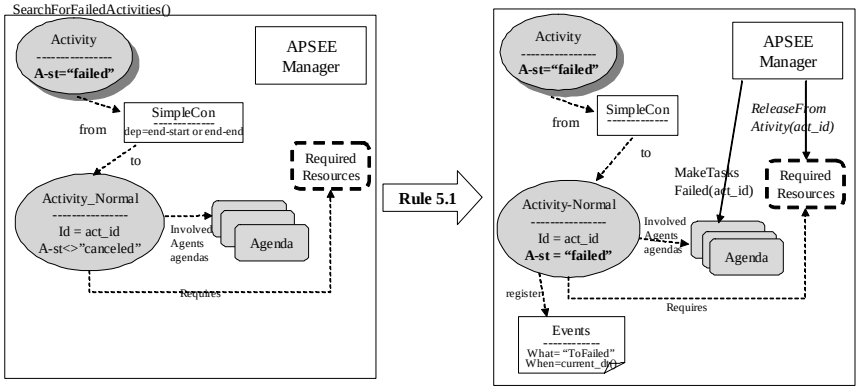


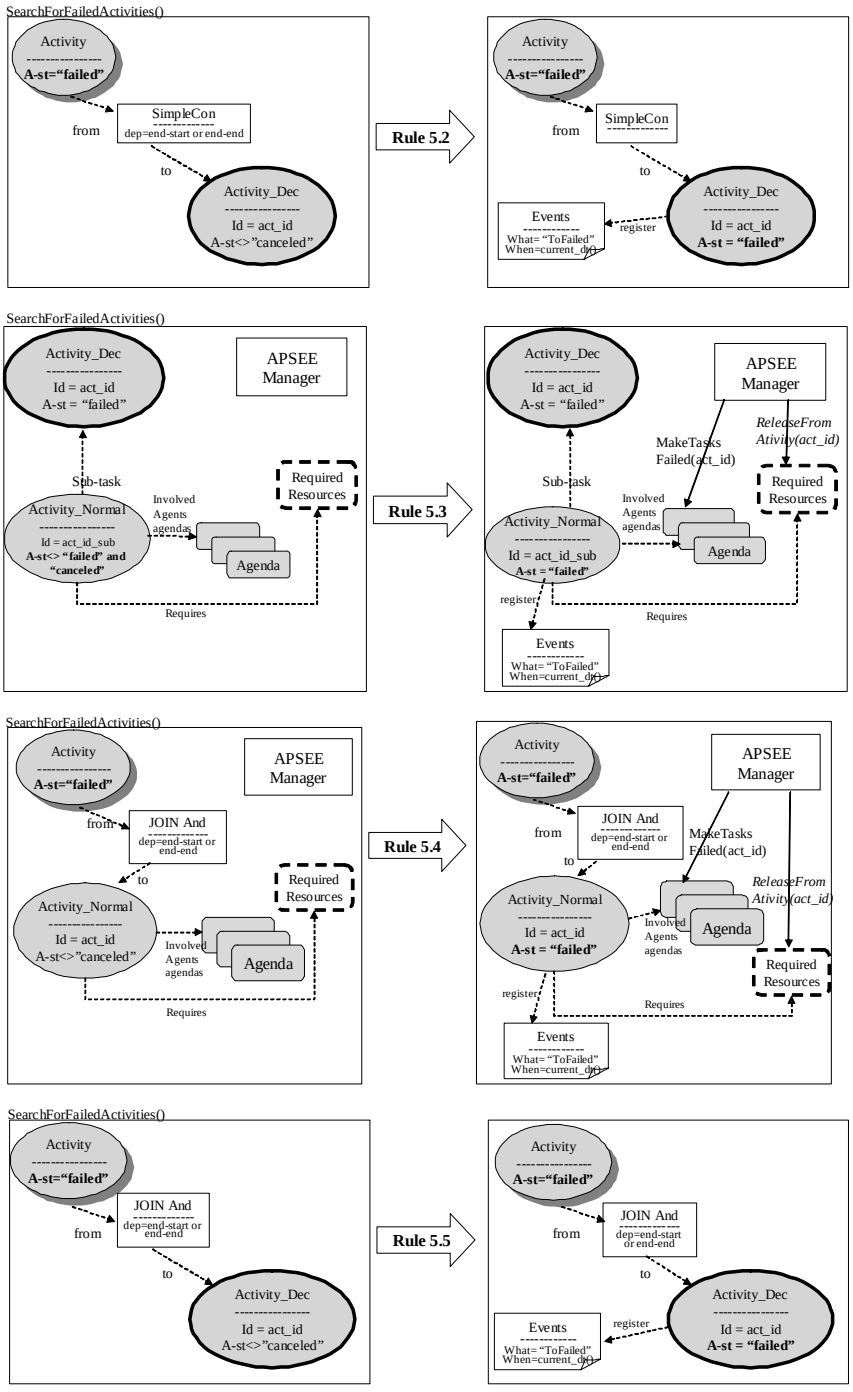
Plain Activity has Started

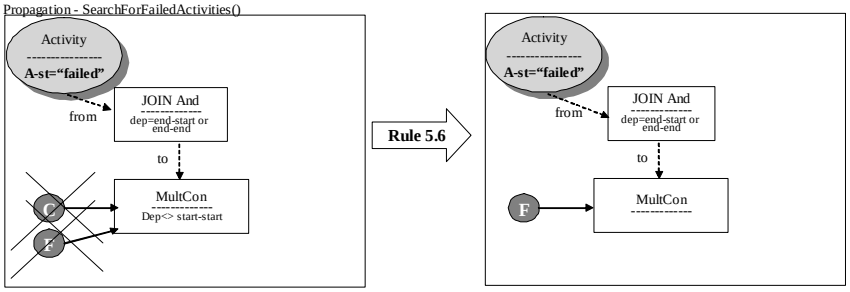


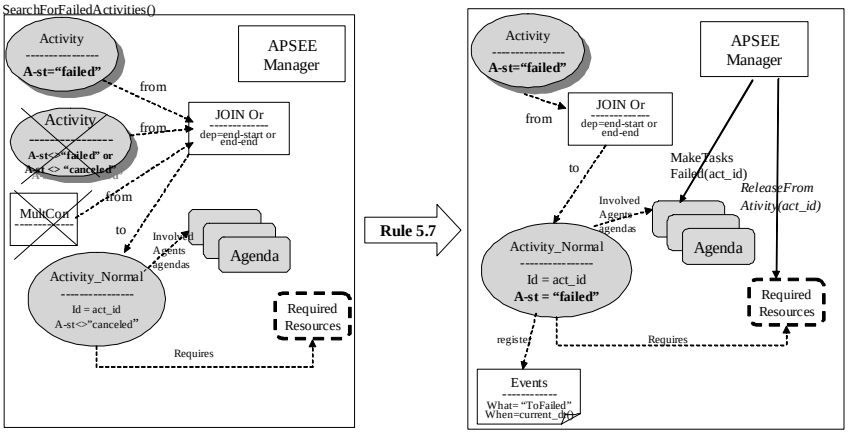


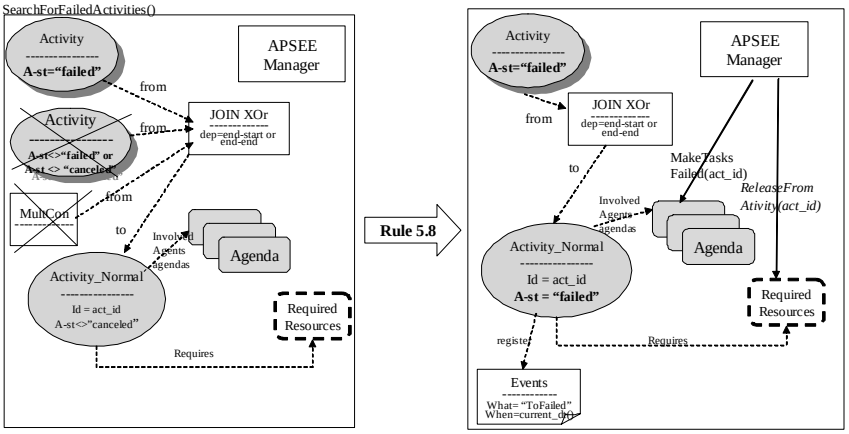
C.5. Regras que Propagam Falhas em atividades

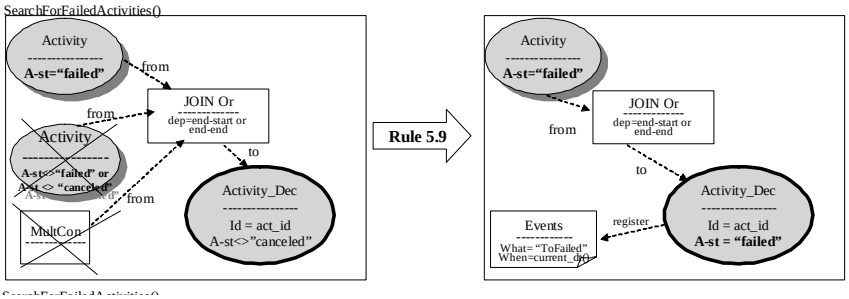


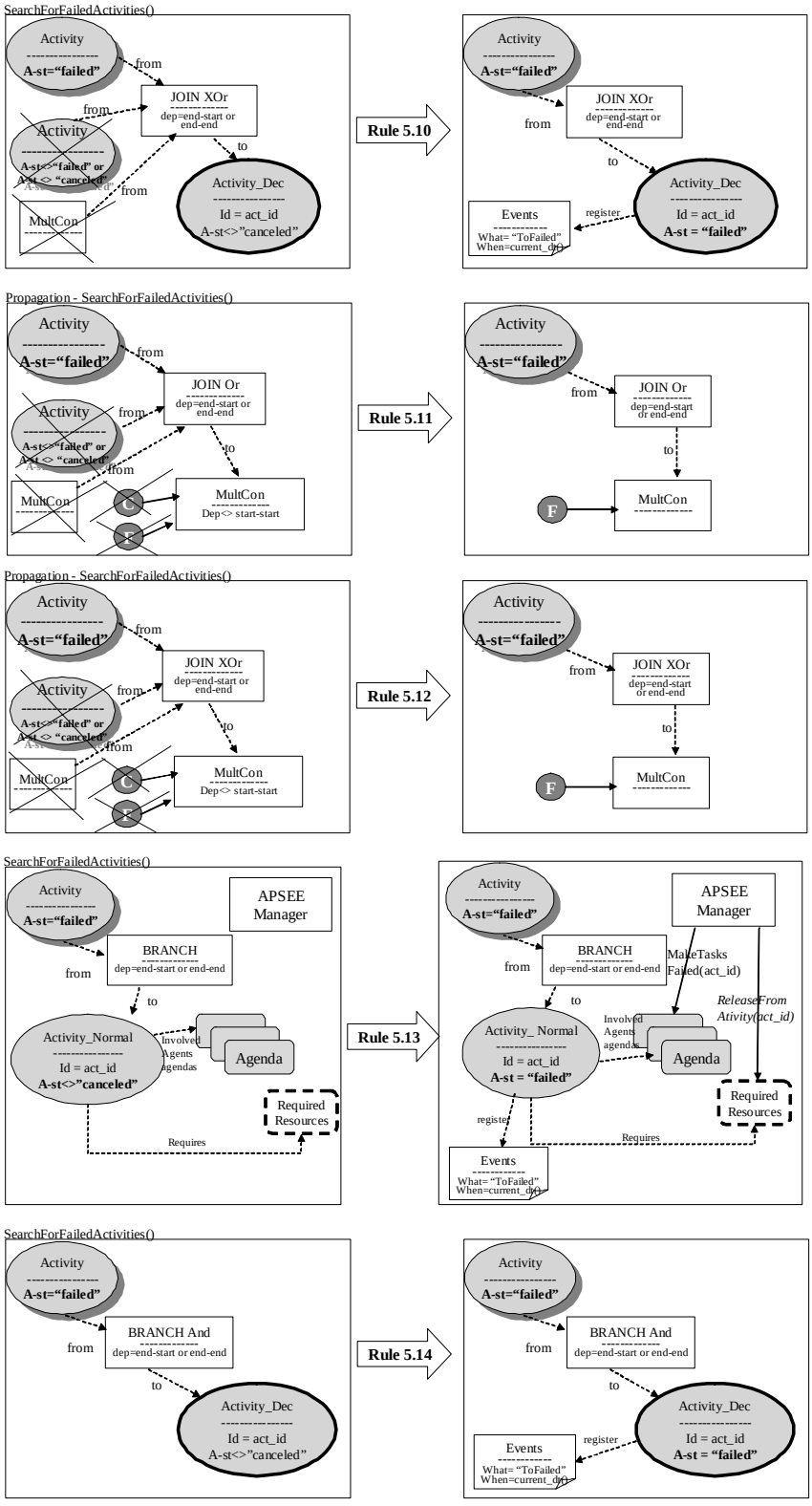

SearchForFailedActivities()
Activity
A-st="failed"
from
SimpleCon
dep=end-start or end-end
to
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.2
Activity
A-st="failed"
from
SimpleCon
to
Activity_Dec
Id = act_id
A-st = "failed"
Events
What= "ToFailed"
When=current_d()
register
SearchForFailedActivities()
Activity_Dec
Id = act_id
A-st = "failed"
APSEE Manager
Sub-task
Activity_Normal
Id = act_id_sub
A-st<> "failed" and "canceled"
Involved Agents agendas
Agenda
Required Resources
Requires
Rule 5.3
Activity_Dec
Id = act_id
A-st = "failed"
APSEE Manager
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
Sub-task
Activity_Normal
Id = act_id_sub
A-st = "failed"
Involved Agents agendas
Agenda
Required Resources
register
Requires
Events
What= "ToFailed"
When=current_d()
SearchForFailedActivities()
Activity
A-st="failed"
APSEE Manager
from
JOIN And
dep=end-start or end-end
to
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents agendas
Agenda
Required Resources
Requires
Rule 5.4
Activity
A-st="failed"
APSEE Manager
from
JOIN And
dep=end-start or end-end
MakeTasks Failed(act_id)
to
Activity_Normal
Id = act_id
A-st = "failed"
ReleaseFrom Ativity(act_id)
Involved Agents agendas
Agenda
Required Resources
register
Requires
Events
What= "ToFailed"
When=current_d()
SearchForFailedActivities()
Activity
A-st="failed"
from
JOIN And
dep=end-start or end-end
to
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.5
Activity
A-st="failed"
from
JOIN And
dep=end-start or end-end
to
Activity_Dec
Id = act_id
A-st = "failed"
Events
What= "ToFailed"
When=current_d()
register

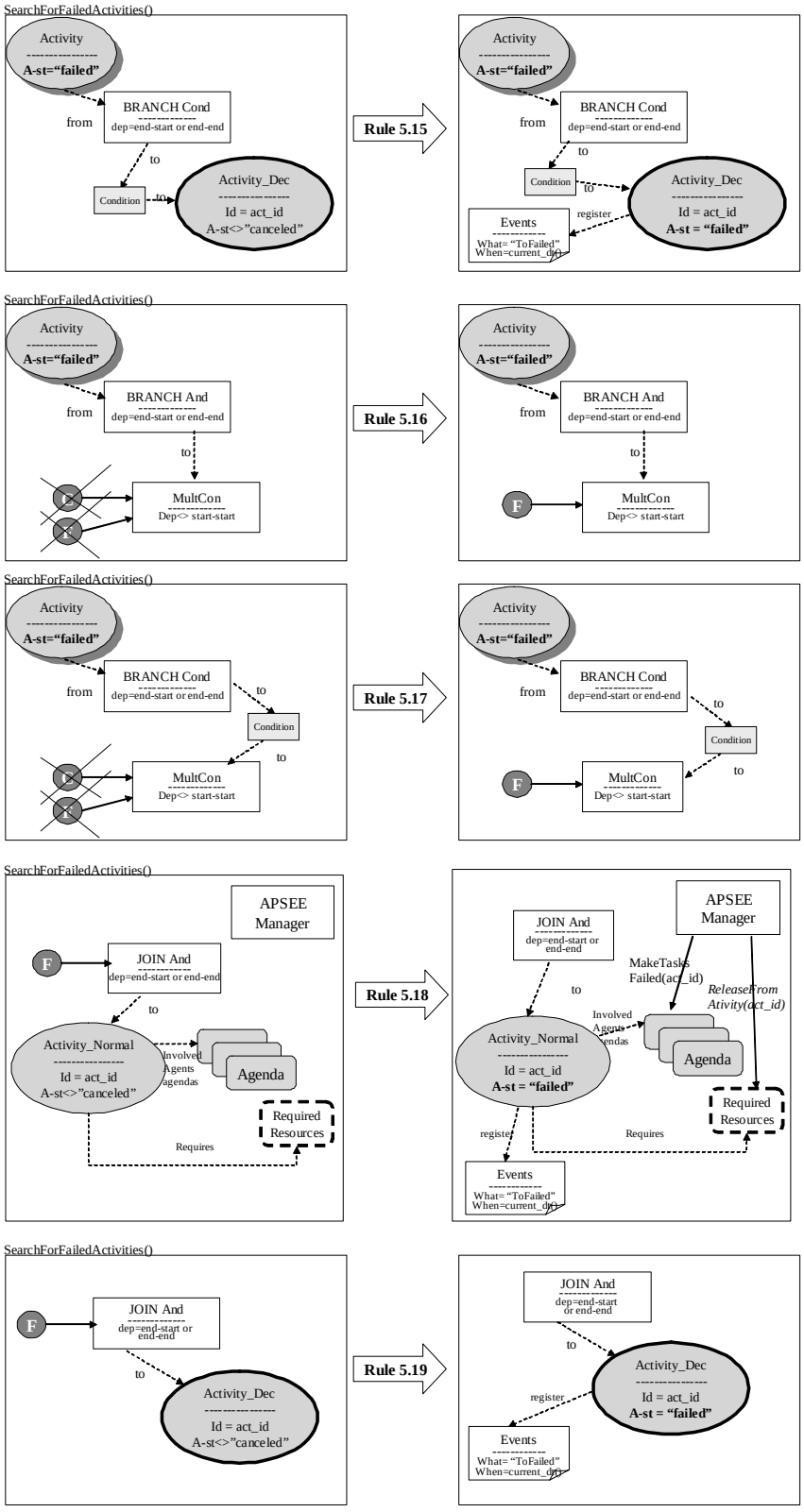

Propagation - SearchForFailedActivities()
Activity
A-st="failed"
from
JOIN And
dep=end-start or end-end
to
MultCon
Dep<> start-start
C
E
Rule 5.6
Activity
A-st="failed"
from
JOIN And
dep=end-start or end-end
to
MultCon
F

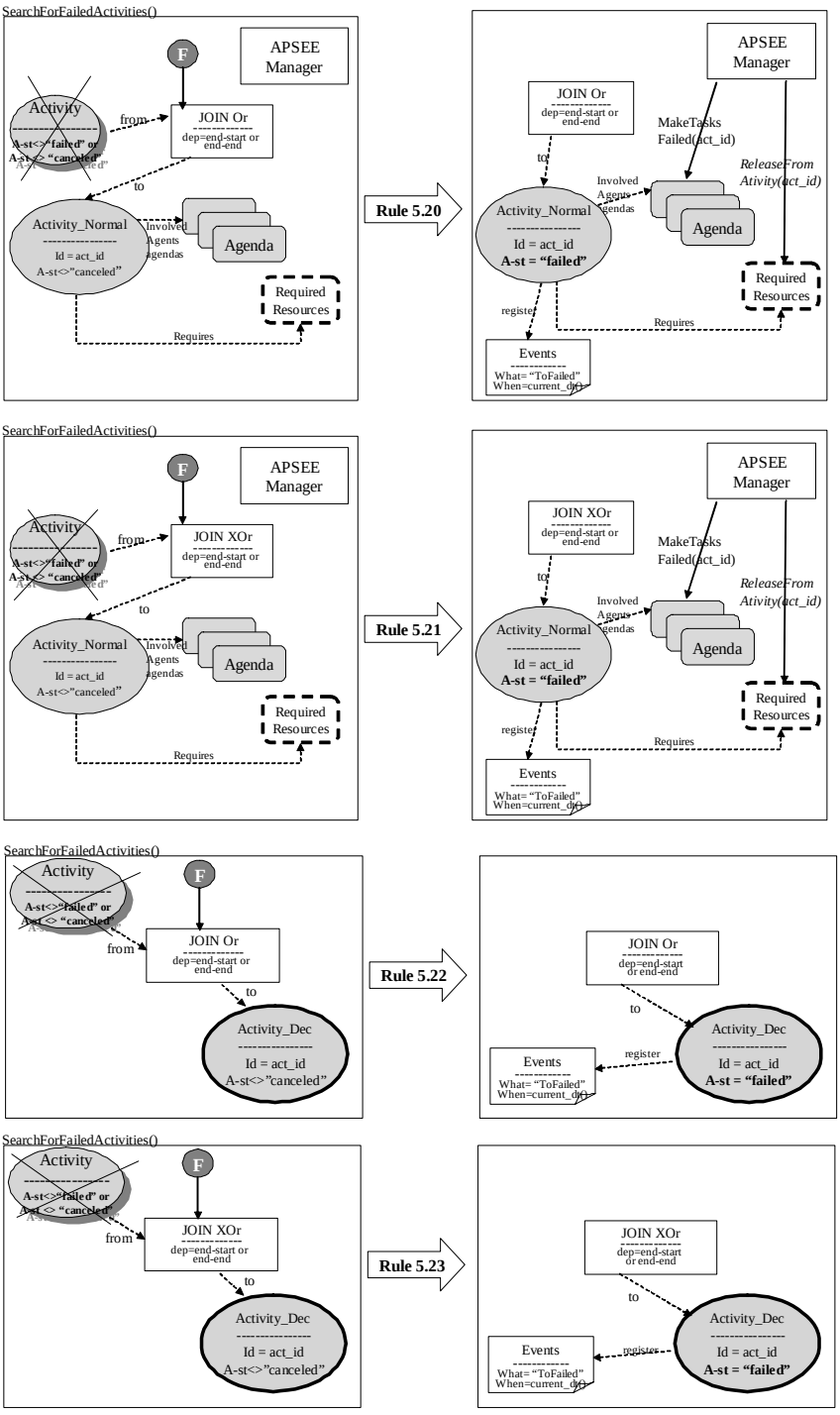

SearchForFailedActivities()
Activity
A-st="failed"
APSEE Manager
from
Activity
A-st<>"failed" or A-st <> "canceled"
JOIN Or
dep=end-start or end-end
MultCon
Involved Agents agendas
Agenda
Activity_Normal
Id = act_id
A-st<>"canceled"
Required Resources
Requires
Rule 5.7
Activity
A-st="failed"
APSEE Manager
JOIN Or
dep=end-start or end-end
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
Involved Agents agendas
Agenda
Activity_Normal
Id = act_id
A-st = "failed"
Required Resources
register
Requires
Events
What= "ToFailed"
When=current_dt()

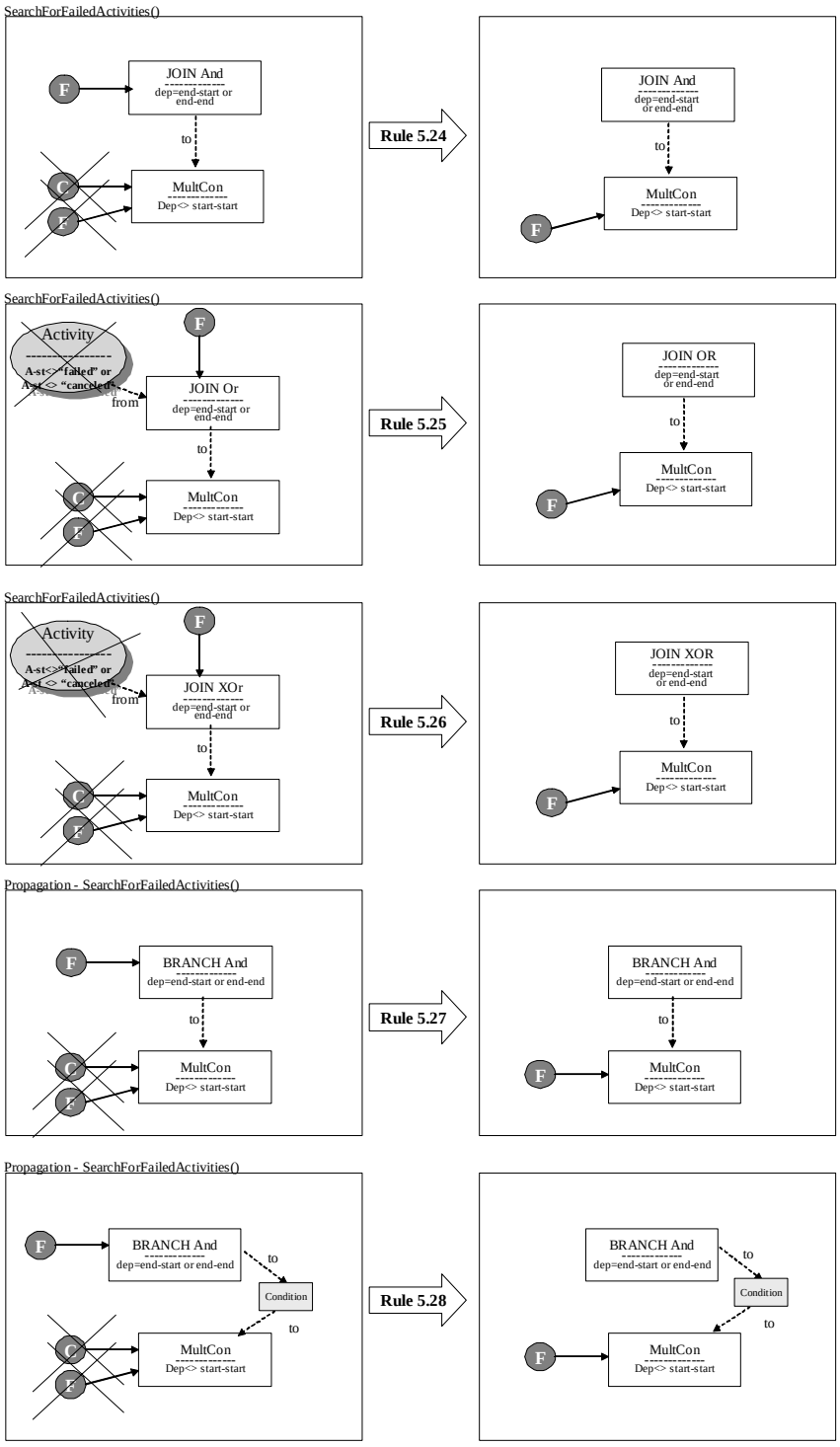

SearchForFailedActivities()
Activity
A-st="failed"
APSEE Manager
from
Activity
A-st<>"failed" or A-st <> "canceled"
JOIN XOr
dep=end-start or end-end
MultCon
Involved Agents agendas
Agenda
Activity_Normal
Id = act_id
A-st<>"canceled"
Required Resources
Requires
Rule 5.8
Activity
A-st="failed"
APSEE Manager
JOIN XOr
dep=end-start or end-end
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
Involved Agents agendas
Agenda
Activity_Normal
Id = act_id
A-st = "failed"
Required Resources
register
Requires
Events
What= "ToFailed"
When=current_dt()


SearchForFailedActivities()
Activity
A-st="failed"
from
JOIN Or
dep=end-start or end-end
Activity
A-st<>"failed" or A-st <> "canceled"
to
from
MultCon
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.9
Activity
A-st="failed"
JOIN Or
dep=end-start or end-end
from
to
Activity_Dec
Id = act_id
A-st = "failed"
Events
What= "ToFailed"
When=current_dt()
register


SearchForFailedActivities()
Activity
A-st="failed"
from
JOIN XOr
dep=end-start or end-end
to
Activity
A-st<>"failed" or A-st <> "canceled"
MultCon
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.10
Activity
A-st="failed"
JOIN XOr
dep=end-start or end-end
Activity_Dec
Id = act_id
A-st = "failed"
Events
What= "ToFailed"
When=current_d/t
register
Propagation - SearchForFailedActivities()
JOIN Or
dep=end-start or end-end
MultCon
Dep<> start-start
C
F
Rule 5.11
JOIN Or
dep=end-start or end-end
Propagation - SearchForFailedActivities()
JOIN XOr
dep=end-start or end-end
Rule 5.12
SearchForFailedActivities()
APSEE Manager
BRANCH
dep=end-start or end-end
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents agendas
Agenda
Required Resources
Requires
Rule 5.13
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
SearchForFailedActivities()
BRANCH And
dep=end-start or end-end
Rule 5.14


SearchForFailedActivities()
Activity
A-st="failed"
from
BRANCH Cond
dep=end-start or end-end
to
Condition
to
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.15
register
Events
What= "ToFailed"
When=current_dt()
Activity_Dec
Id = act_id
A-st = "failed"
SearchForFailedActivities()
BRANCH And
dep=end-start or end-end
MultCon
Dep<> start-start
Rule 5.16
F
SearchForFailedActivities()
Rule 5.17
SearchForFailedActivities()
APSEE Manager
JOIN And
dep=end-start or end-end
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents
Agenda
Required Resources
Requires
Rule 5.18
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
SearchForFailedActivities()
Rule 5.19

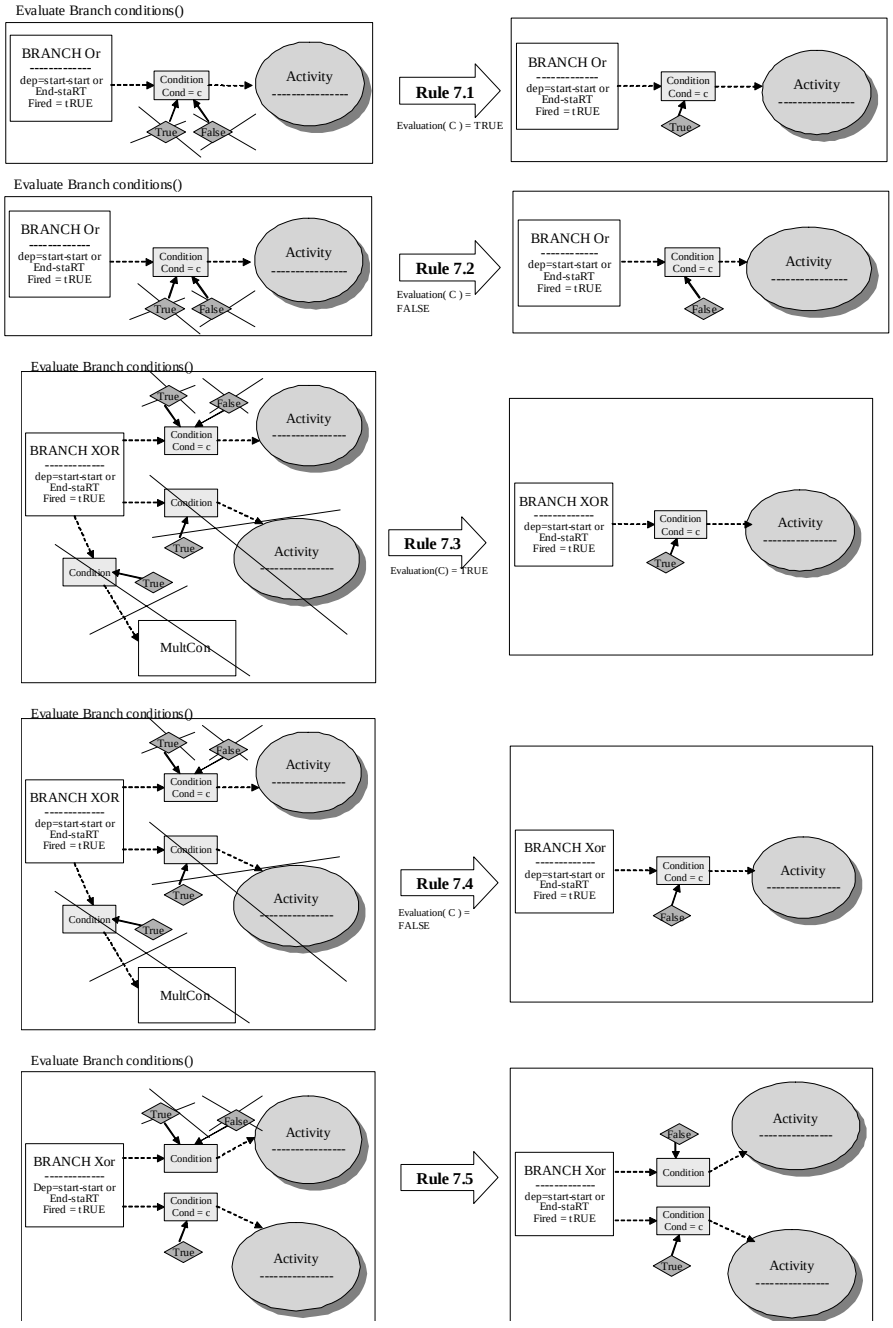

SearchForFailedActivities()
APSEE Manager
F
Activity
A-st<>"failed" or
A-st <> "canceled"
from
JOIN Or
dep=end-start or end-end
to
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents agendas
Agenda
Required Resources
Requires
Rule 5.20
JOIN Or
dep=end-start or end-end
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
Involved Agents agendas
Activity_Normal
Id = act_id
A-st = "failed"
register
Events
What= "ToFailed"
When=current_d()
SearchForFailedActivities()
JOIN XOr
dep=end-start or end-end
Rule 5.21
JOIN XOr
dep=end-start or end-end
SearchForFailedActivities()
Activity_Dec
Id = act_id
A-st<>"canceled"
Rule 5.22
Activity_Dec
Id = act_id
A-st = "failed"
SearchForFailedActivities()
Rule 5.23

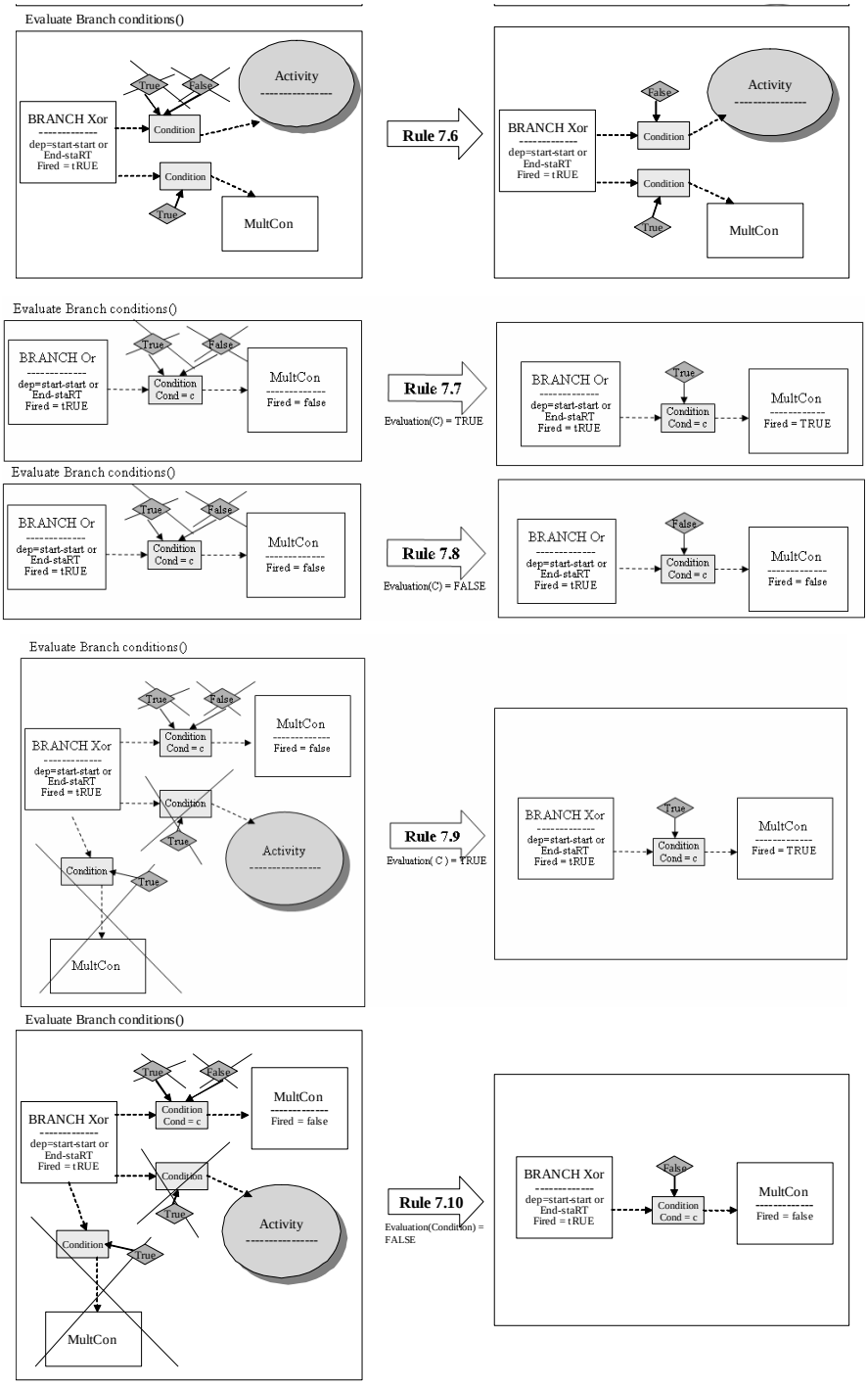


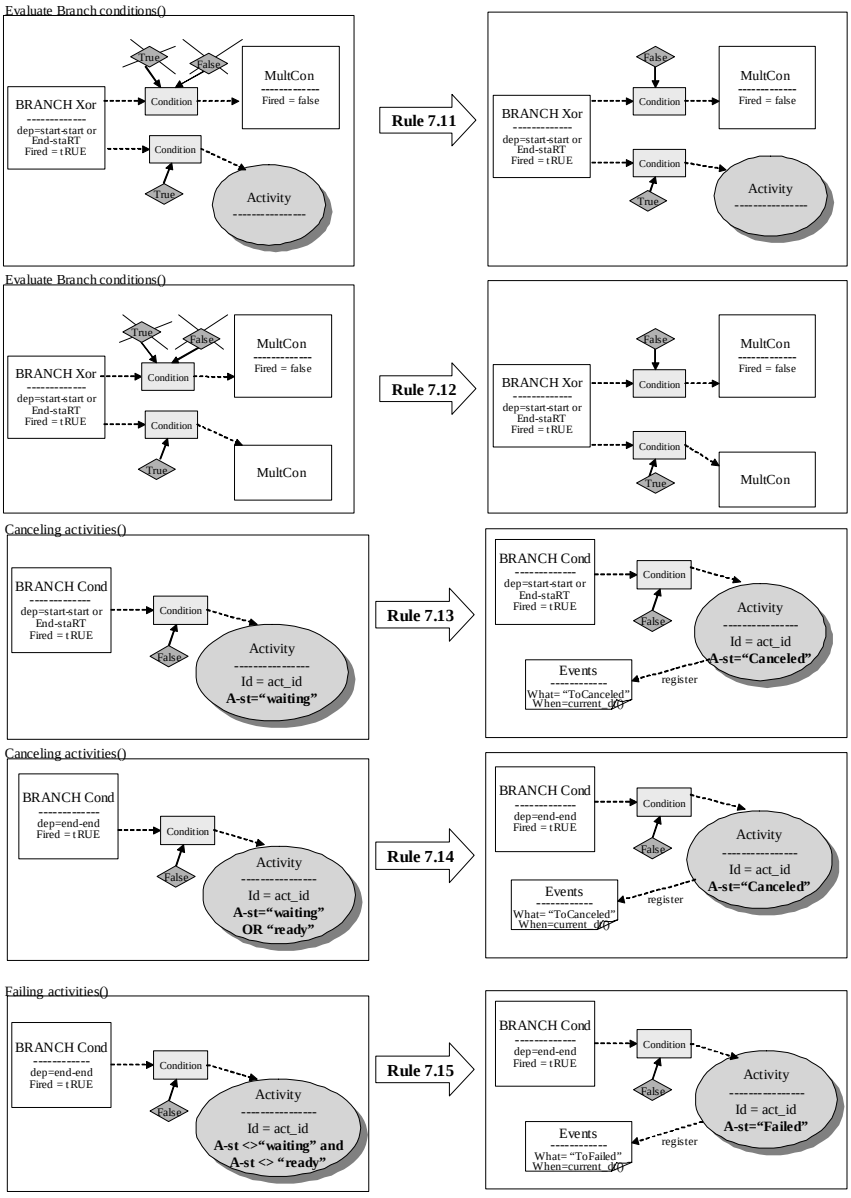
C.6. Regras que propagam Cancelamento entre atividades

As regras que propagam cancelamento foram construídas com raciocínio análogo às regras de propagação de falhas e por isso não serão apresentadas aqui. O cancelamento é marcado pelo símbolo C ligado a conexões e atividades.

C.7. Regras que tratam conexões Branch

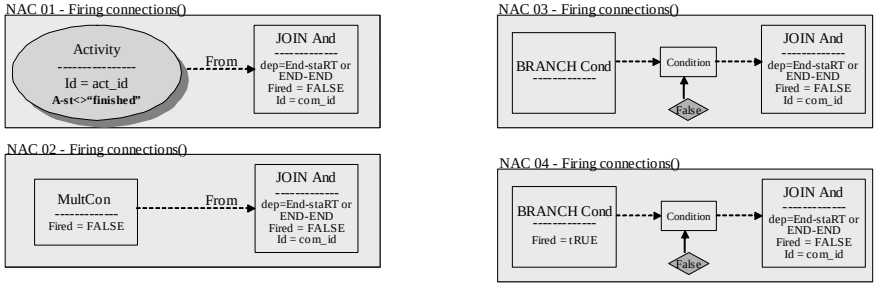


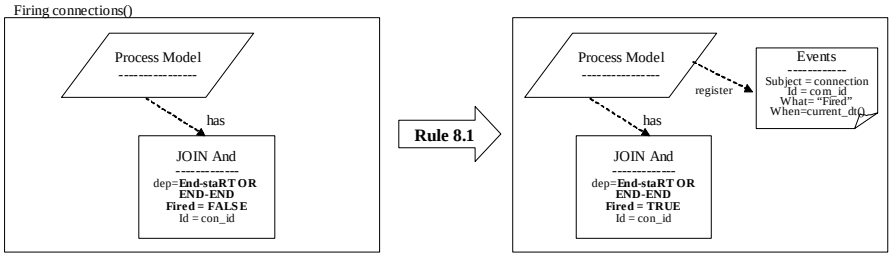

Evaluate Branch conditions()
BRANCH Xor
dep=start-start or End-staRT
Fired = tRUE
Condition
True
False
Activity
MultCon
Rule 7.6
Evaluate Branch conditions()
BRANCH Or
dep=start-start or End-staRT
Fired = tRUE
Condition
Cond = c
MultCon
Fired = false
Rule 7.7
Evaluation(C) = TRUE
Fired = TRUE
Evaluate Branch conditions()
Rule 7.8
Evaluation(C) = FALSE
Evaluate Branch conditions()
Rule 7.9
Evaluation(C) = TRUE
Evaluate Branch conditions()
Rule 7.10
Evaluation(Condition) = FALSE



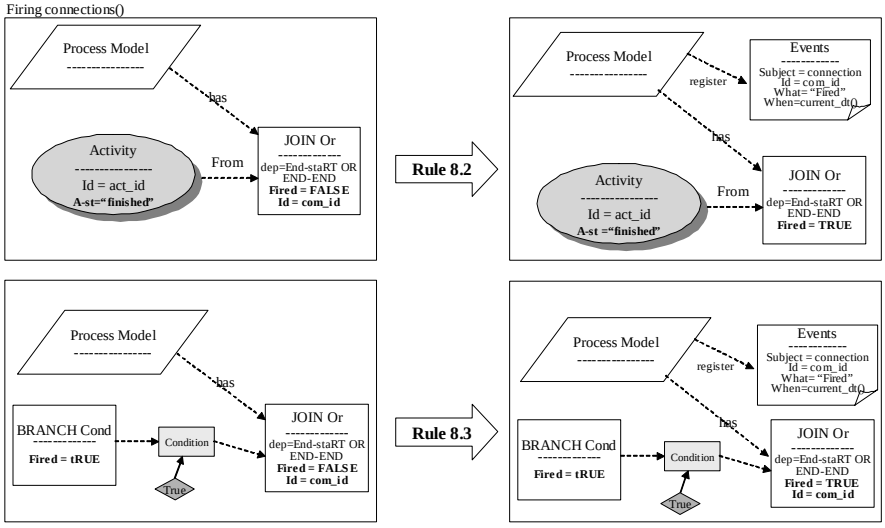
C.8. Regras que tratam ativação de conexões múltiplas

Regras para definição do atributo Fired em conexões múltiplas

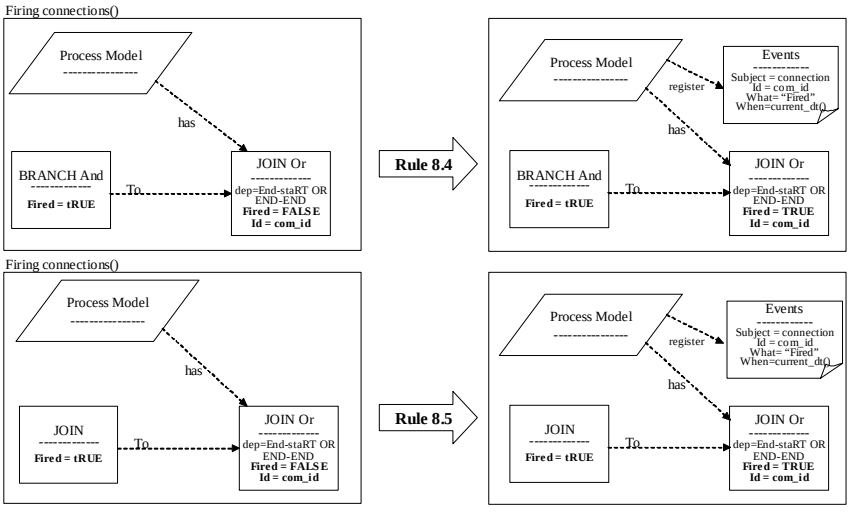


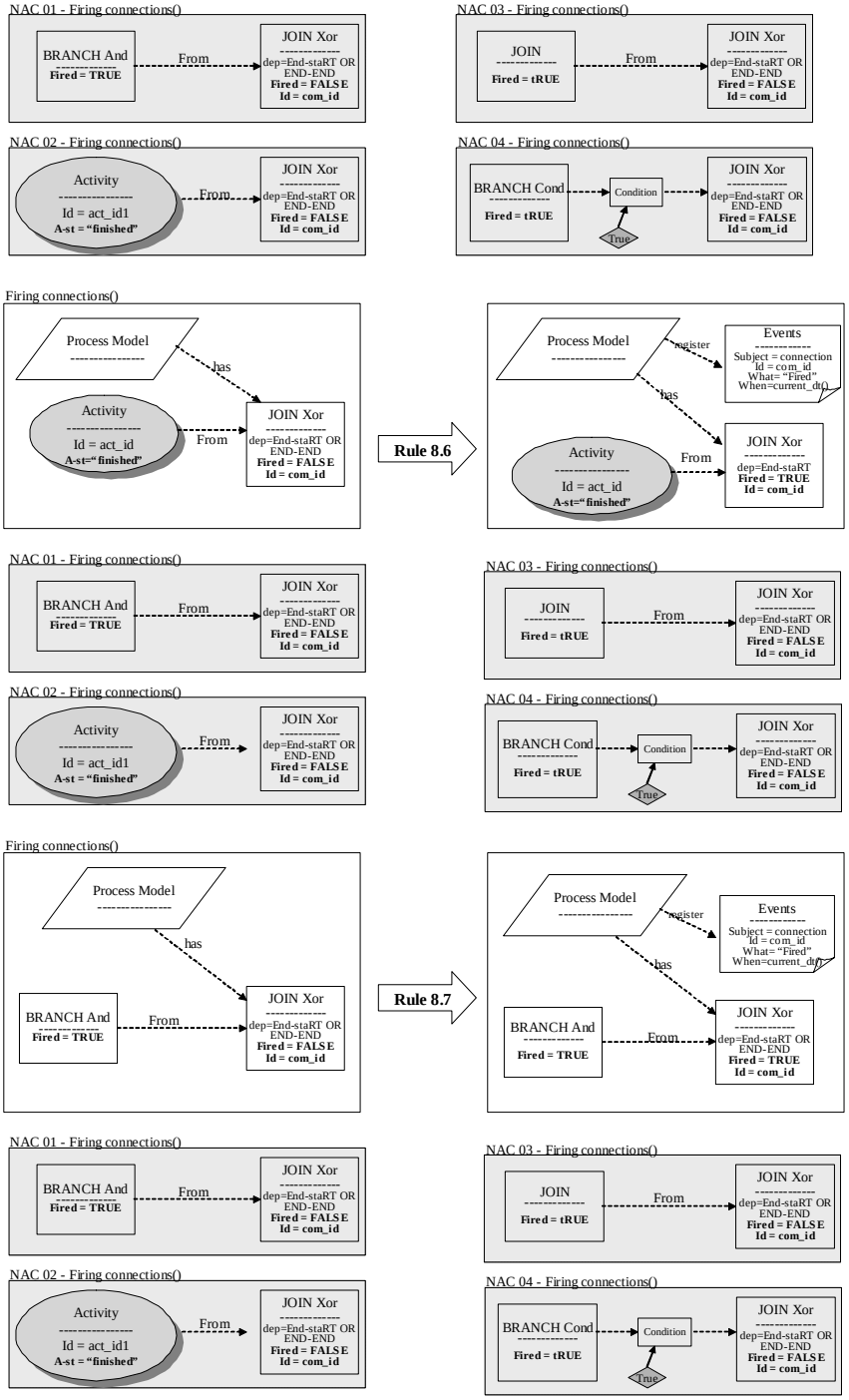


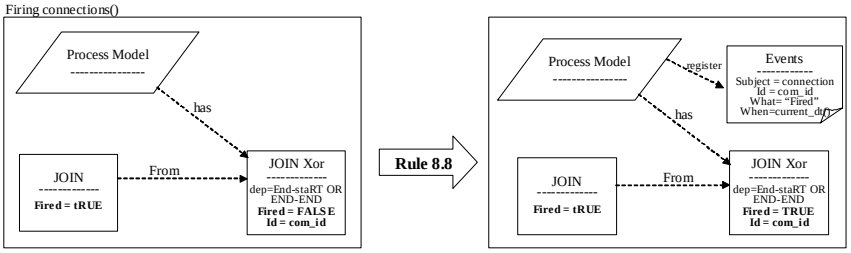
Regras para definição do atributo Fired em conexões múltiplas

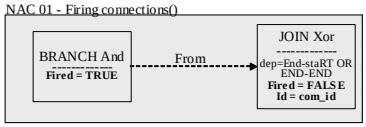


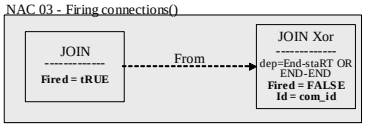
Regras para definição do atributo Fired em conexões múltiplas

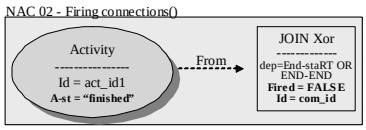


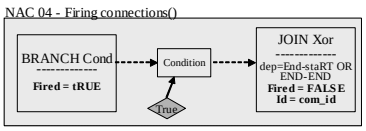

NAC 01 - Firing connections()
BRANCH And
Fired = TRUE
From
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
NAC 03 - Firing connections()
JOIN
Fired = tRUE
From
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
NAC 02 - Firing connections()
Activity
Id = act_id1
A-st = "finished"
From
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
NAC 04 - Firing connections()
BRANCH Cond
Fired = tRUE
Condition
True
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
Firing connections()
Process Model
has
Activity
Id = act_id
A-st="finished"
From
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
Rule 8.6
Process Model
register
Events
Subject = connection
Id = com_id
What= "Fired"
When=current_dt()
has
Activity
Id = act_id
A-st="finished"
From
JOIN Xor
dep=End-staRT
Fired = TRUE
Id = com_id
Firing connections()
Process Model
has
BRANCH And
Fired = TRUE
From
JOIN Xor
dep=End-staRT OR END-END
Fired = FALSE
Id = com_id
Rule 8.7
Process Model
register
Events
Subject = connection
Id = com_id
What= "Fired"
When=current_dt()
has
BRANCH And
Fired = TRUE
From
JOIN Xor
dep=End-staRT OR END-END
Fired = TRUE
Id = com_id

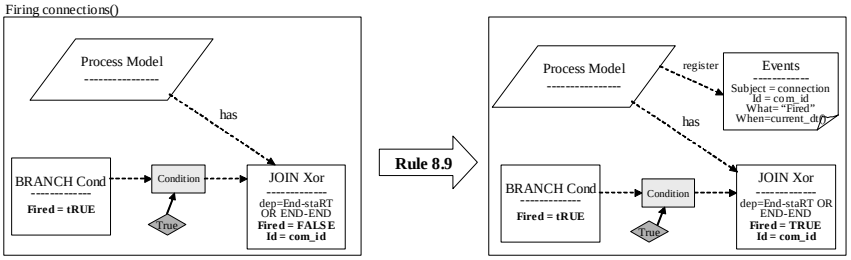




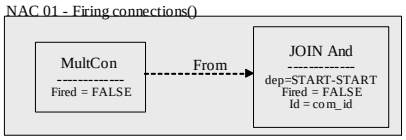


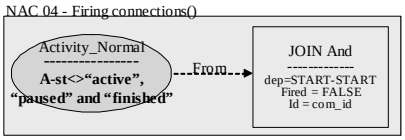


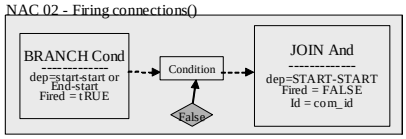


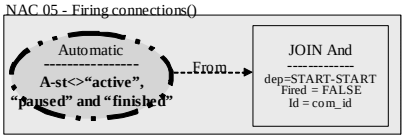


NACs para regra 8.10

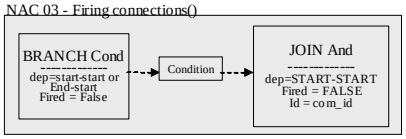


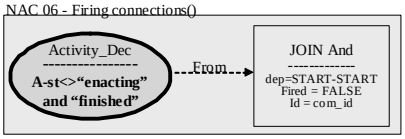


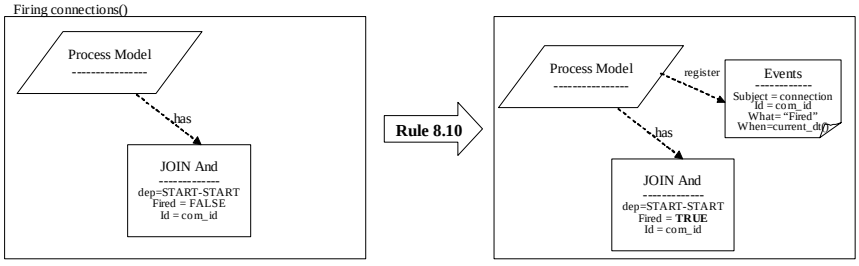


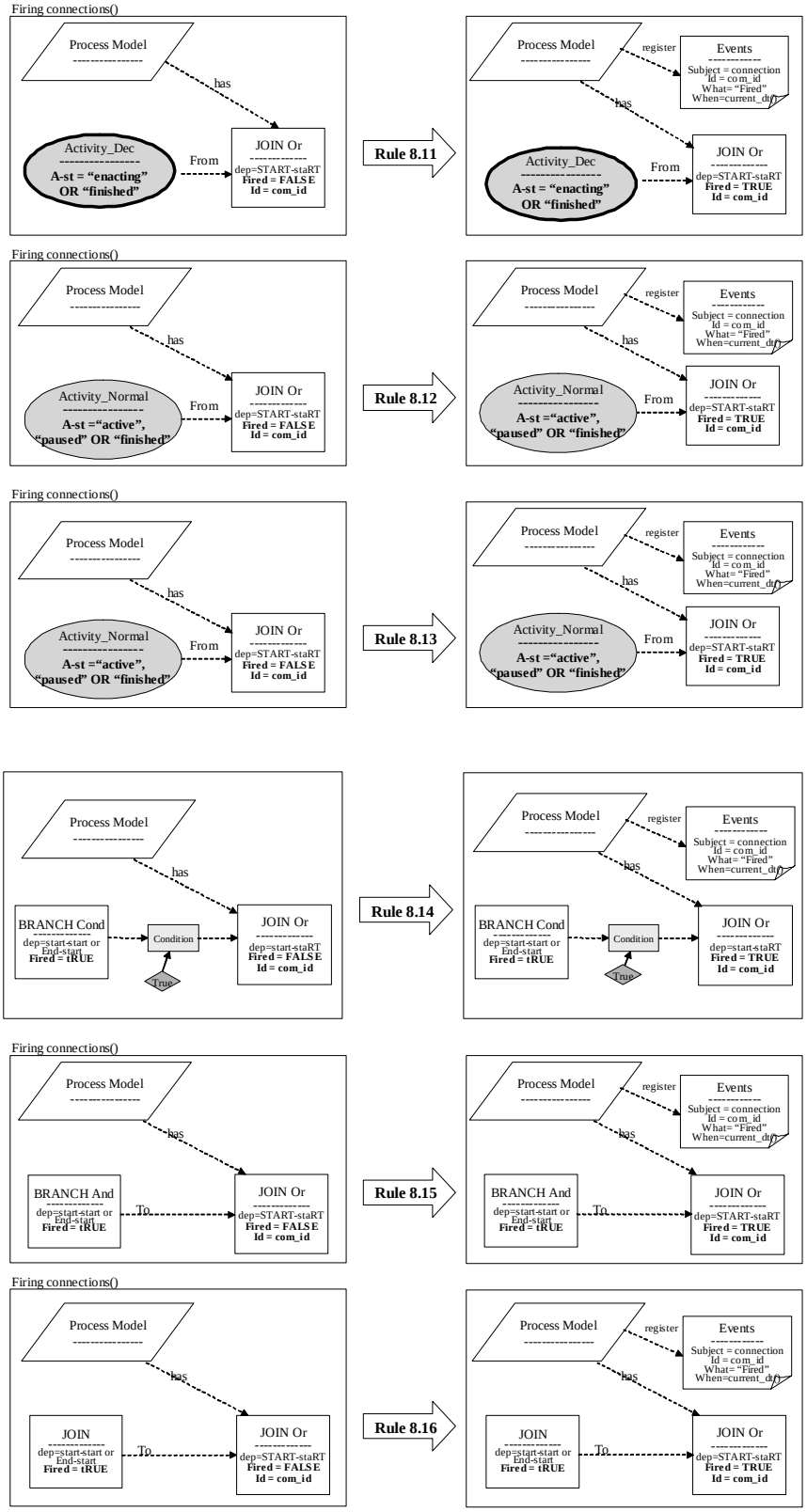


Atividade não pode estar ready, waiting ou canceled

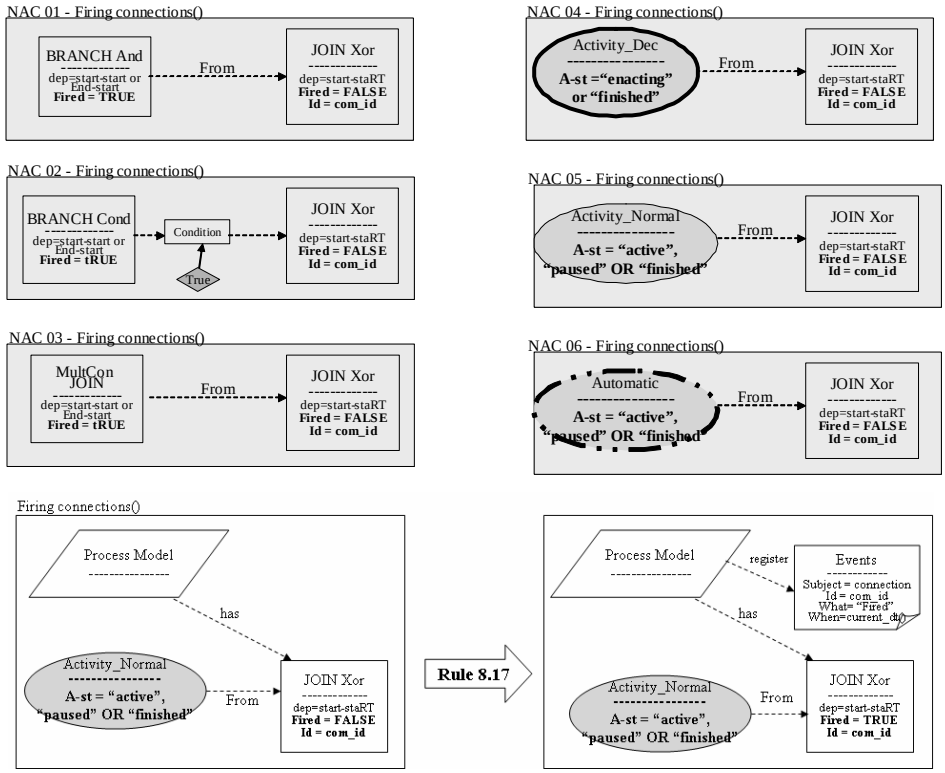




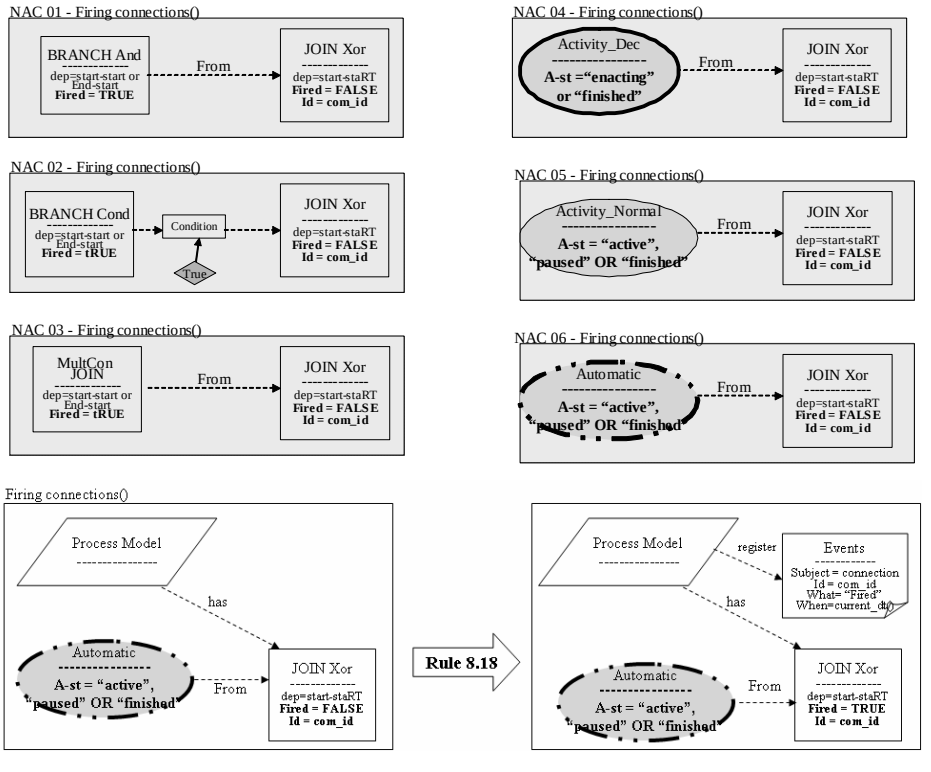



Firing connections()
Process Model
has
Activity_Dec
A-st = "enacting" OR "finished"
From
JOIN Or
dep=START-staRT
Fired = FALSE
Id = com_id
Rule 8.11
register
Events
Subject = connection
Id = com_id
What= "Fired"
When=current_dt()
Fired = TRUE
Firing connections()
Activity_Normal
A-st ="active", "paused" OR "finished"
Rule 8.12
Rule 8.13
BRANCH Cond
dep=start-start or End-start
Fired = tRUE
Condition
True
dep=start-staRT
Rule 8.14
BRANCH And
To
Rule 8.15
JOIN
Rule 8.16

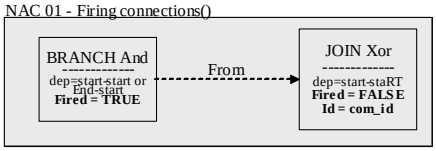
NACs para regra 8.17

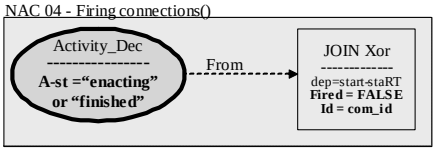


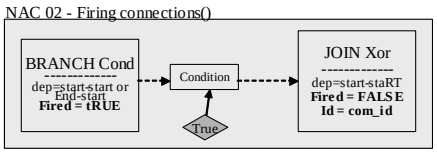
NACs para regra 8.18

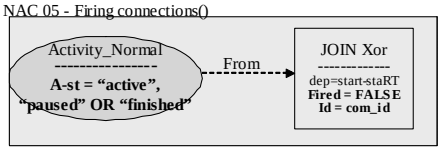


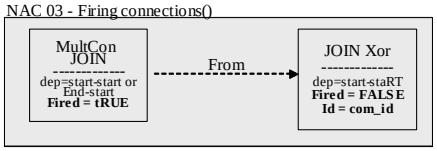
NACs para regra 8.19

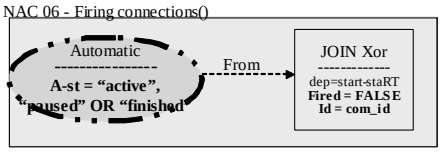


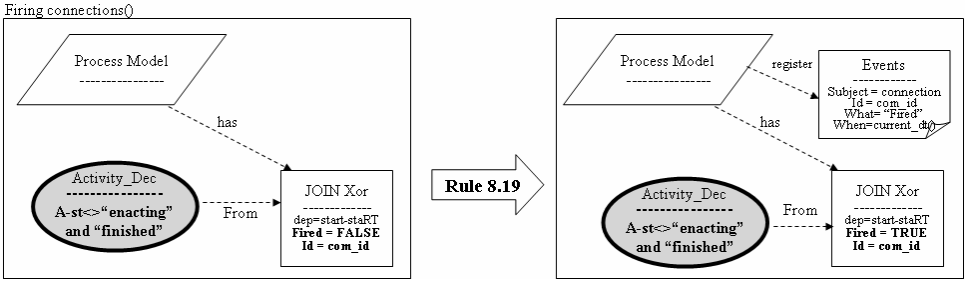




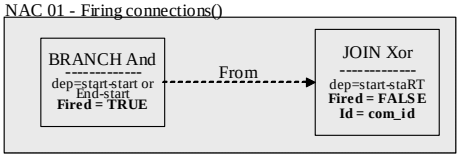


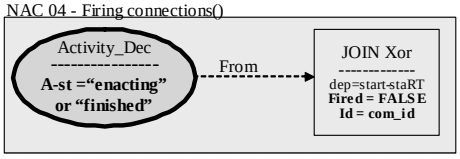


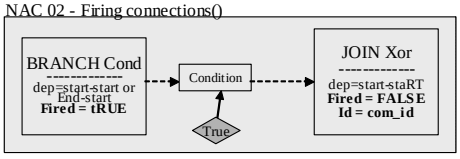


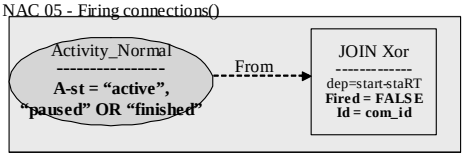


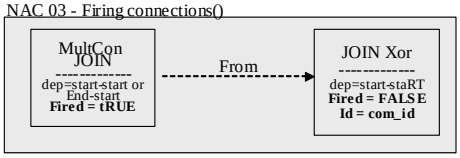
NACs para regra 8.20

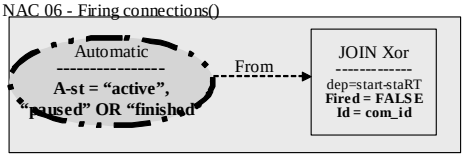


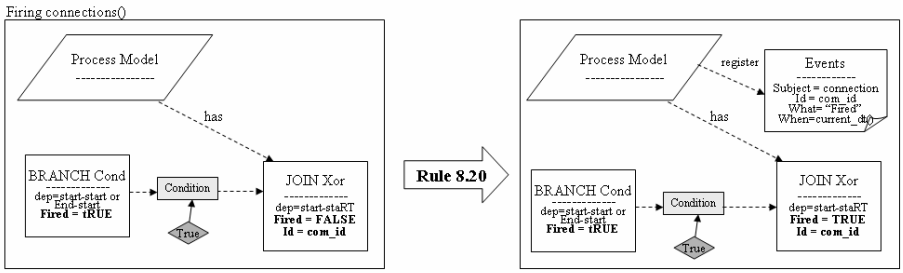




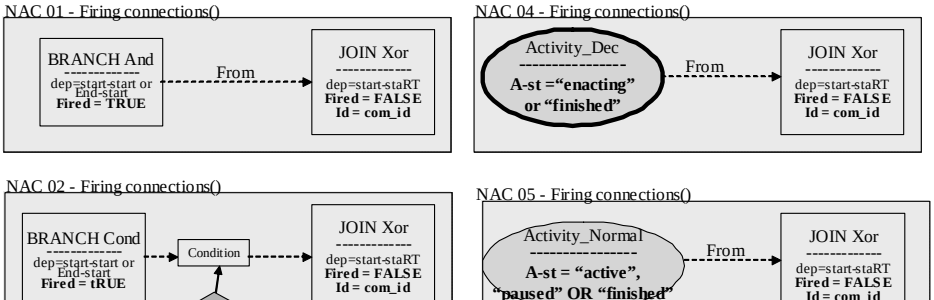


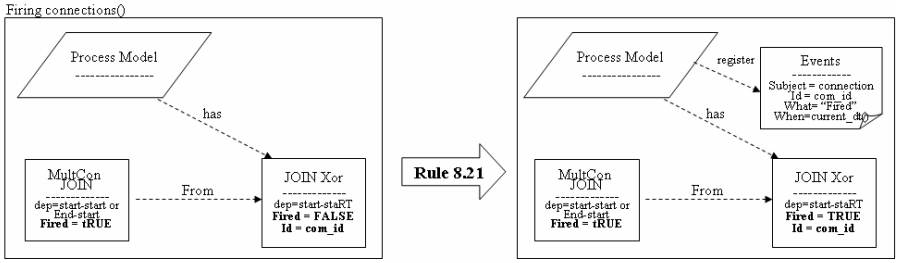




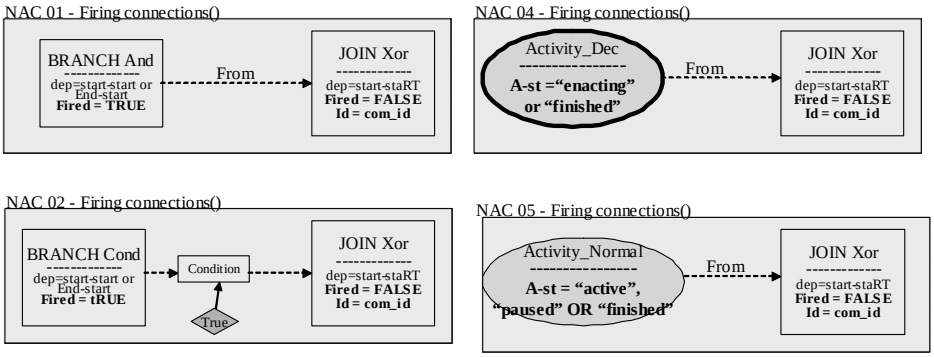


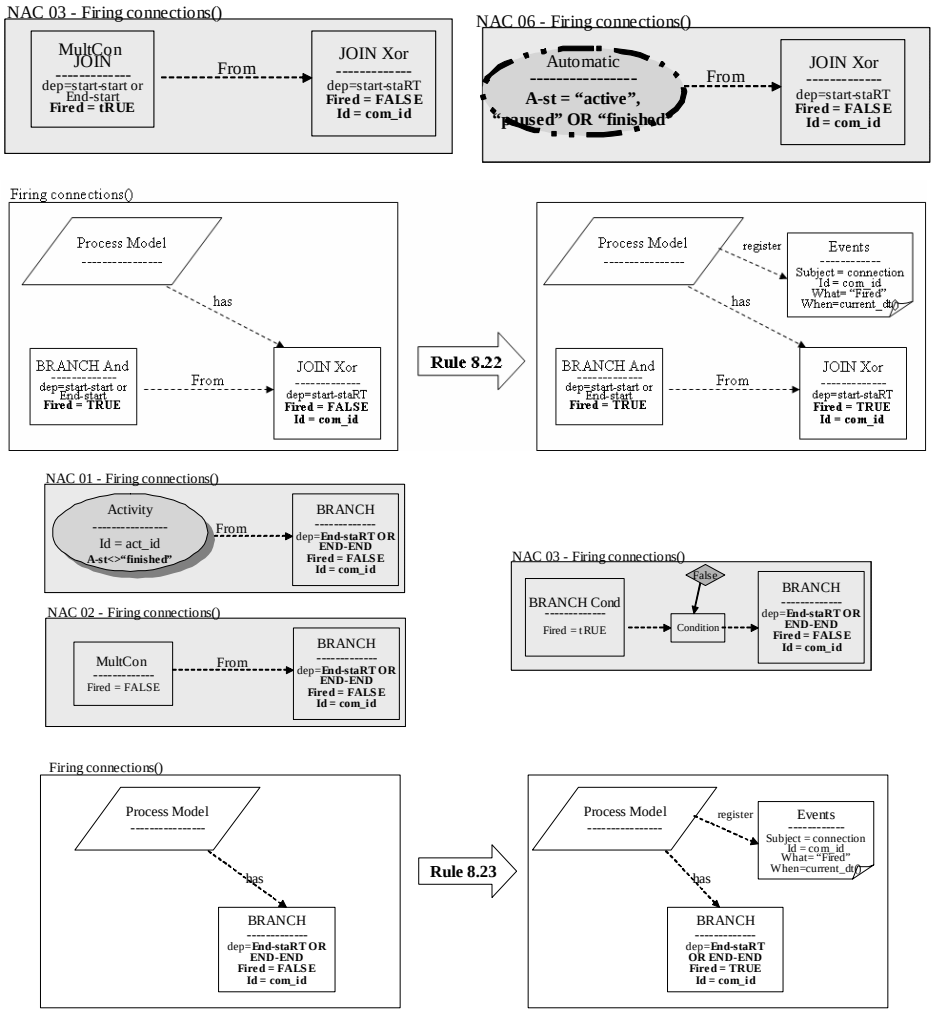
NACs para regra 8.21



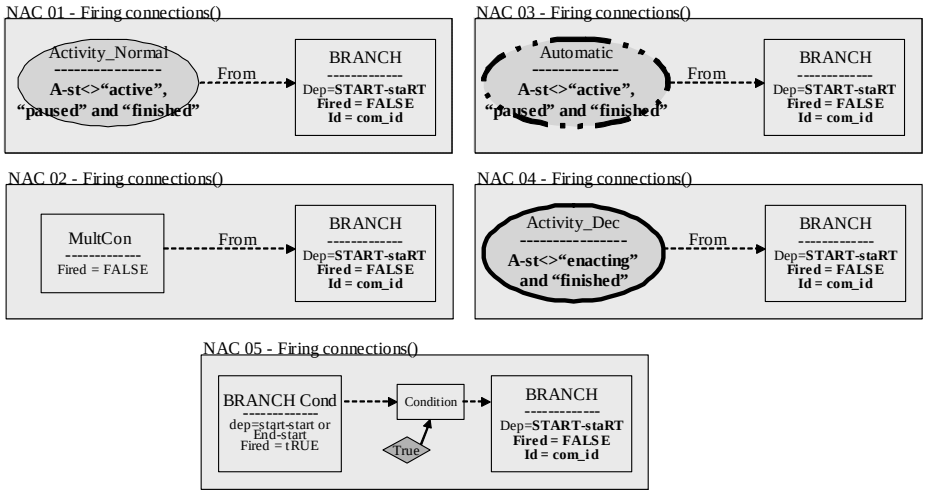


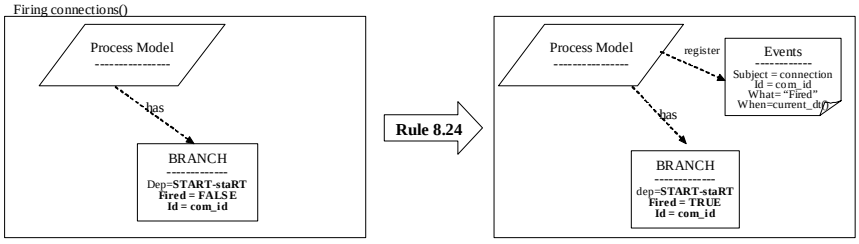
NACs para regra 8.22



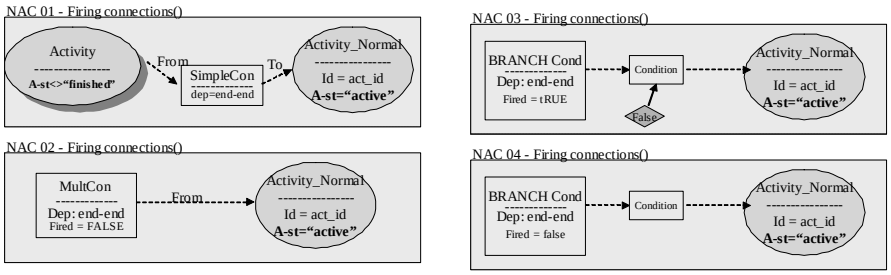


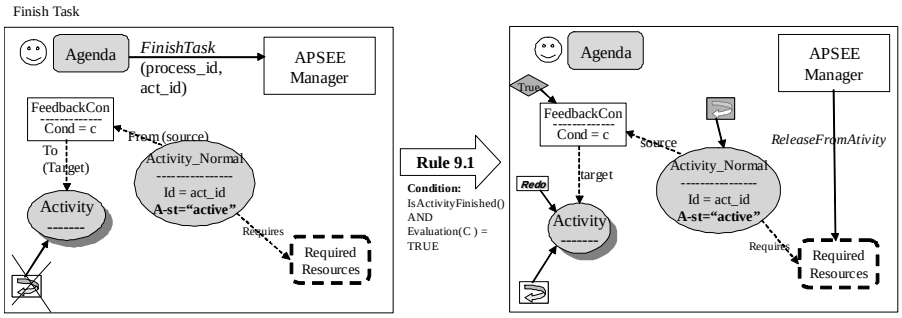
NACs para regra 8.24

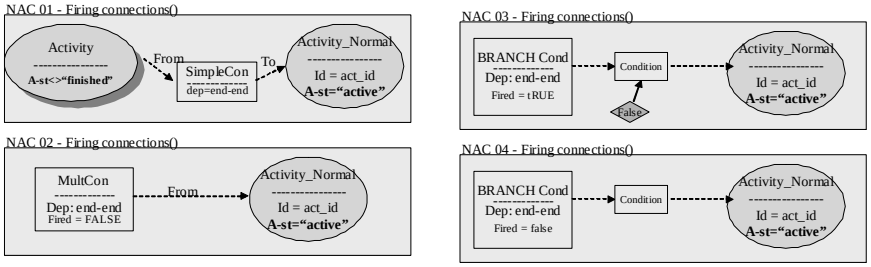


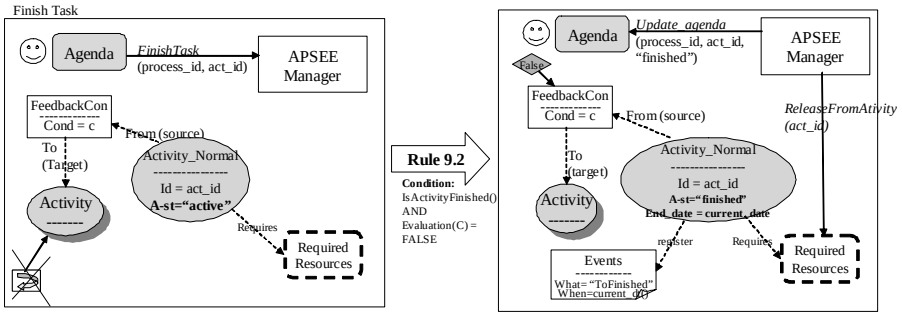


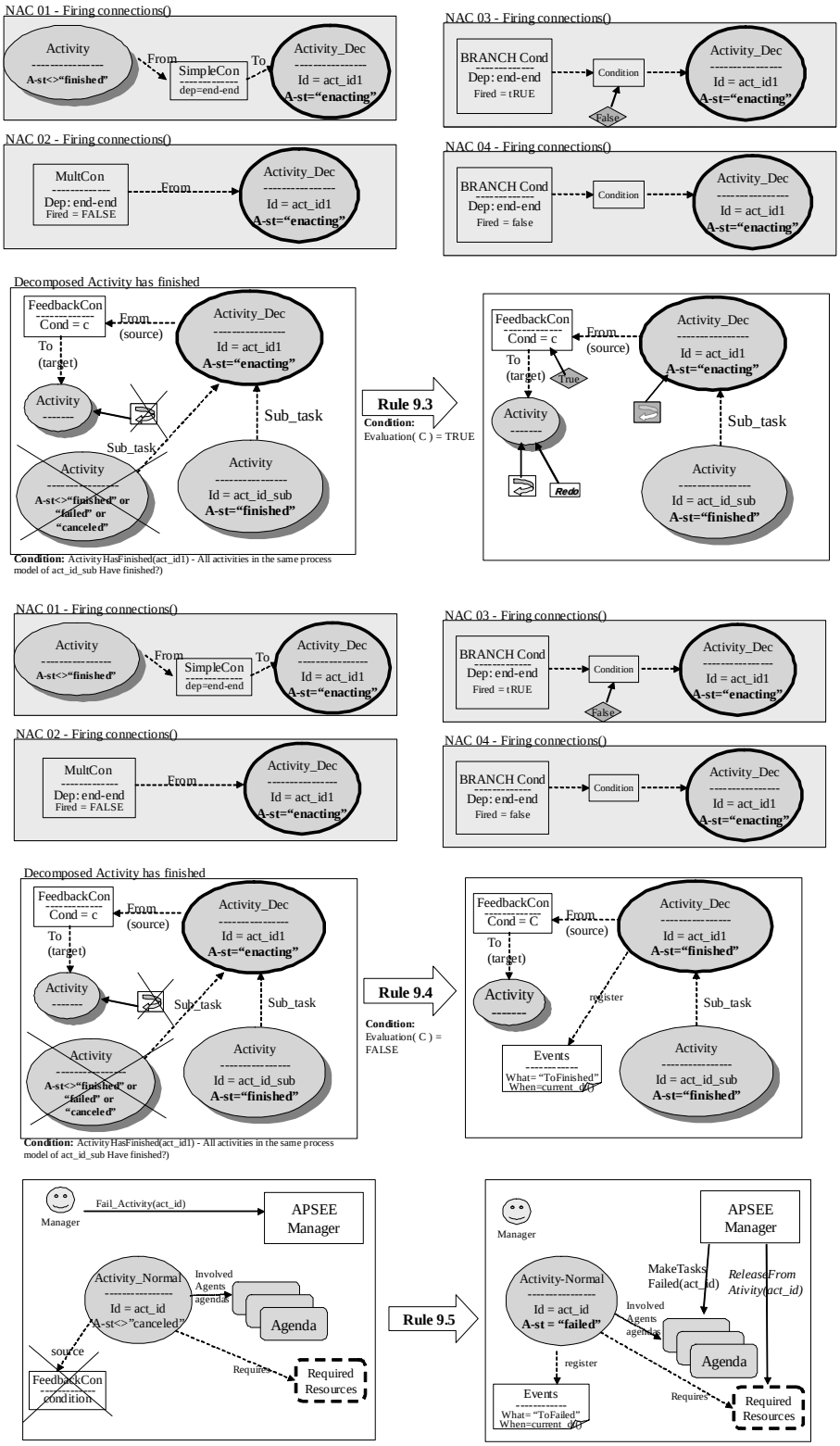
C.9. Regras que tratam propagação de *feedback*

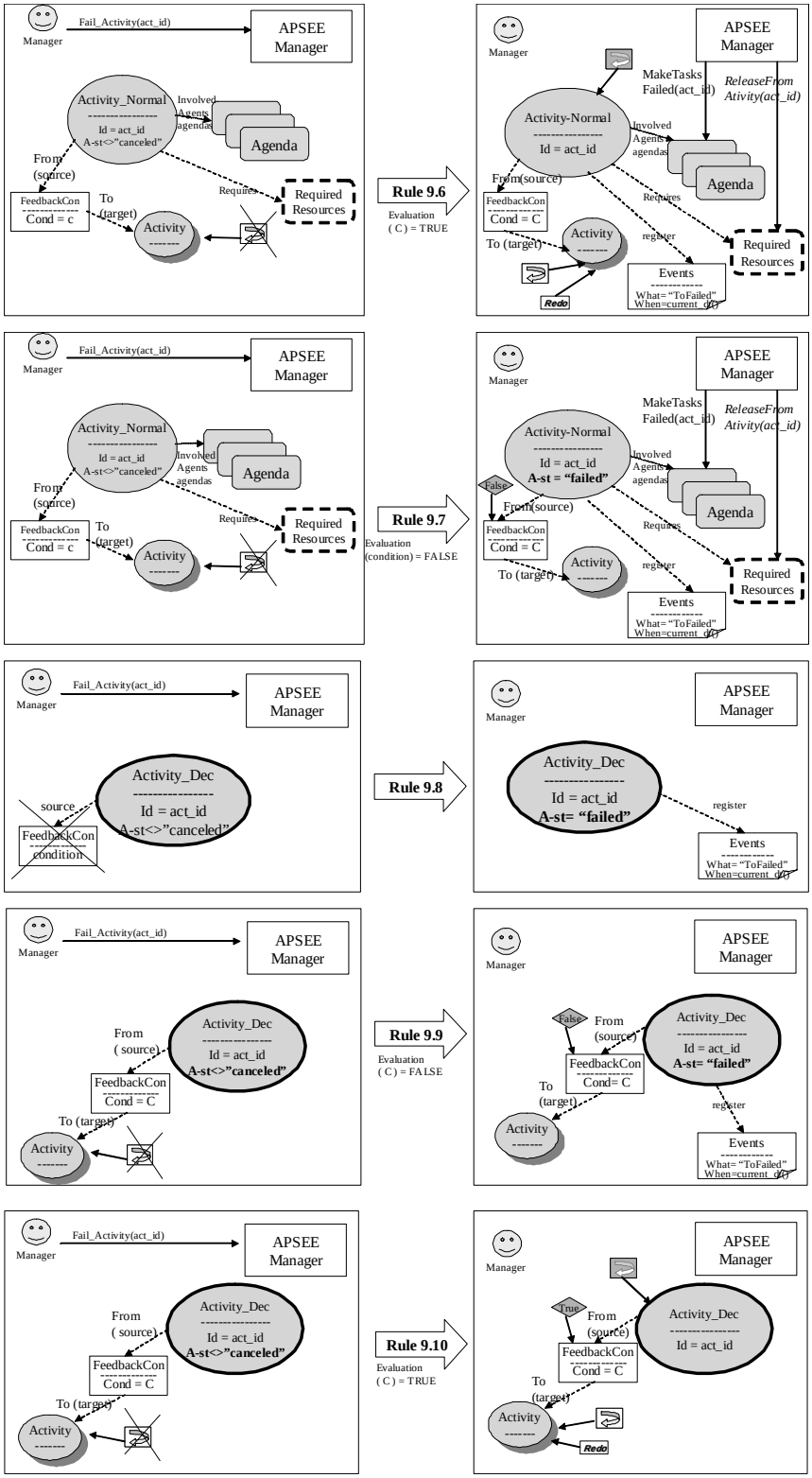


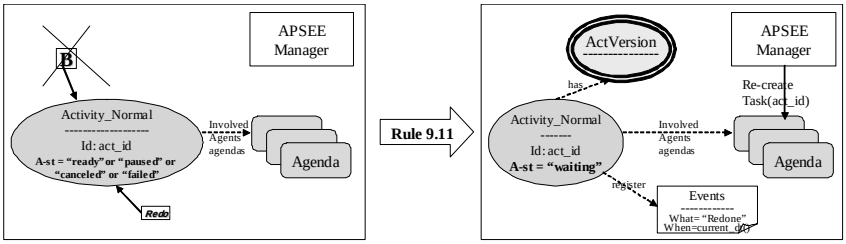


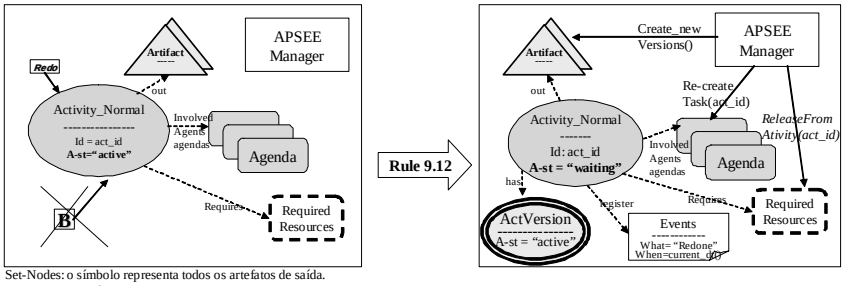




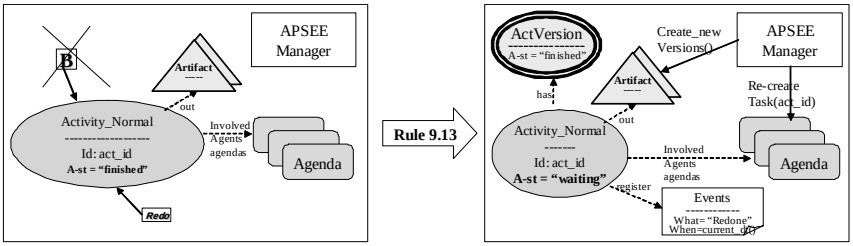

NAC 01 - Firing connections()
Activity
A-st<>"finished"
From
SimpleCon
dep=end-end
To
Activity_Dec
Id = act_id1
A-st="enacting"
NAC 03 - Firing connections()
BRANCH Cond
Dep: end-end
Fired = tRUE
Condition
False
Activity_Dec
Id = act_id1
A-st="enacting"
NAC 02 - Firing connections()
MultCon
Dep: end-end
Fired = FALSE
From
Activity_Dec
Id = act_id1
A-st="enacting"
NAC 04 - Firing connections()
BRANCH Cond
Dep: end-end
Fired = false
Condition
Activity_Dec
Id = act_id1
A-st="enacting"
Decomposed Activity has finished
FeedbackCon
Cond = c
From
(source)
To
(target)
Activity_Dec
Id = act_id1
A-st="enacting"
Activity
Sub_task
Activity
A-st<>"finished" or "failed" or "canceled"
Activity
Id = act_id_sub
A-st="finished"
Condition: ActivityHasFinished(act_id1) - All activities in the same process model of act_id_sub Have finished?)
Rule 9.3
Condition:
Evaluation(C) = TRUE
True
Redo
NAC 01 - Firing connections()
NAC 03 - Firing connections()
NAC 02 - Firing connections()
NAC 04 - Firing connections()
Decomposed Activity has finished
Rule 9.4
Condition:
Evaluation(C) = FALSE
register
Events
What= "ToFinished"
When=current_dt()
Condition: ActivityHasFinished(act_id1) - All activities in the same process model of act_id_sub Have finished?)
Manager
Fail_Activity(act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents agendas
Agenda
source
FeedbackCon
condition
Requires
Required Resources
Rule 9.5
Activity-Normal
Id = act_id
A-st = "failed"
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)


Manager
Fail_Activity(act_id)
APSEE Manager
Activity_Normal
Id = act_id
A-st<>"canceled"
Involved Agents agendas
Agenda
From (source)
FeedbackCon
Cond = c
To (target)
Activity
Requires
Required Resources
Rule 9.6
Evaluation (C) = TRUE
Activity-Normal
Id = act_id
MakeTasks Failed(act_id)
ReleaseFrom Ativity(act_id)
Cond = C
register
Events
What= "ToFailed"
When=current_dt()
Redo
Rule 9.7
Evaluation (condition) = FALSE
A-st = "failed"
False
Activity_Dec
source
condition
Rule 9.8
Rule 9.9
Evaluation (C) = FALSE
Rule 9.10
Evaluation (C) = TRUE
True

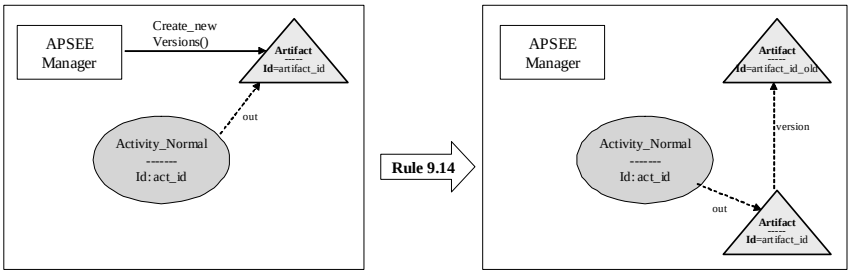


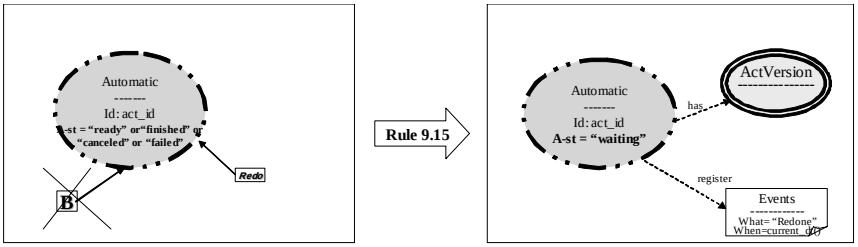


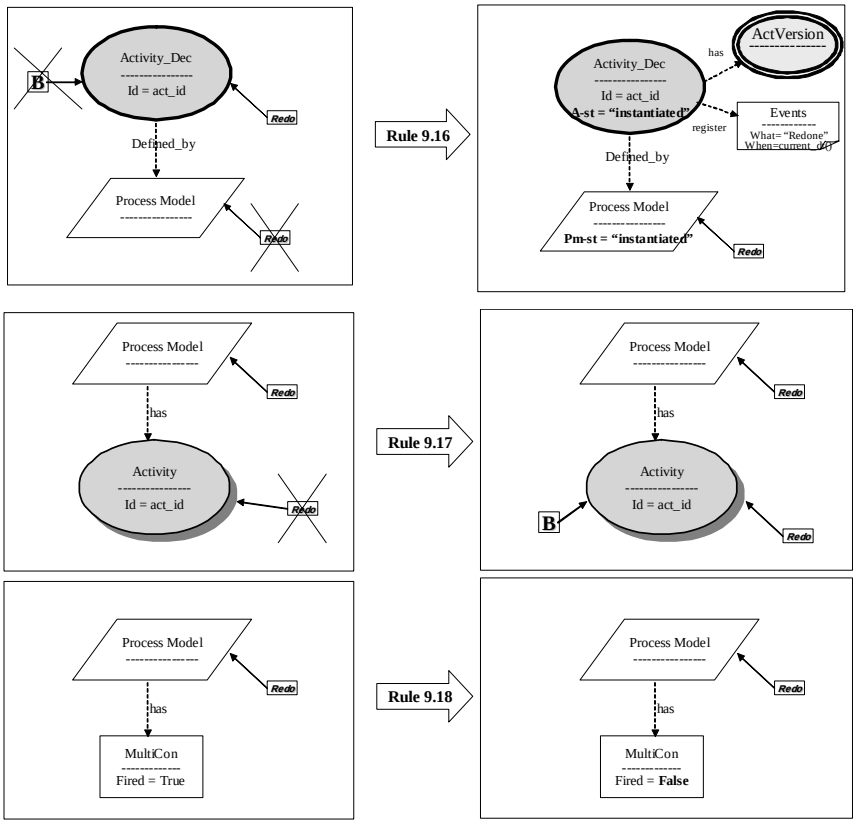
Set-Nodes: o símbolo representa todos os artefatos de saída.
Conjunto possivelmente vazio



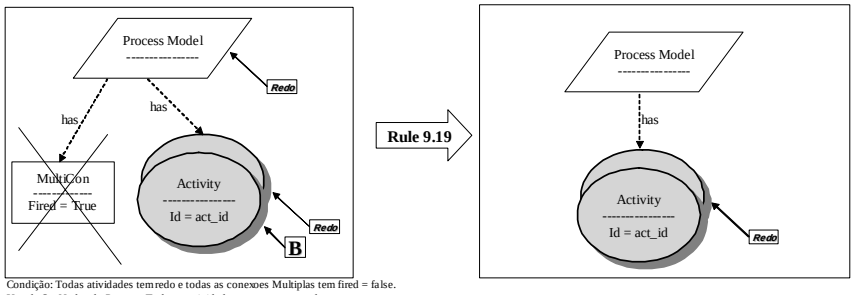
Set-Nodes: o símbolo representa todos os artefatos de saída.
Conjunto possivelmente vazio



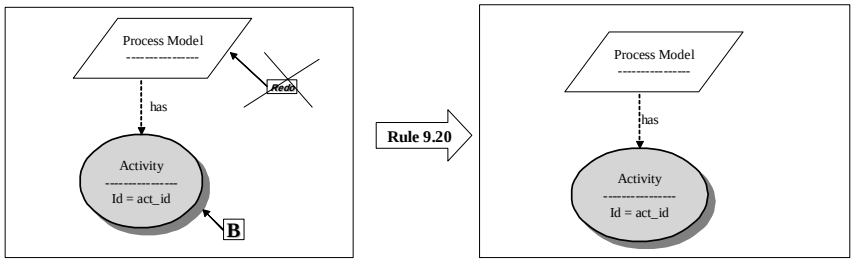


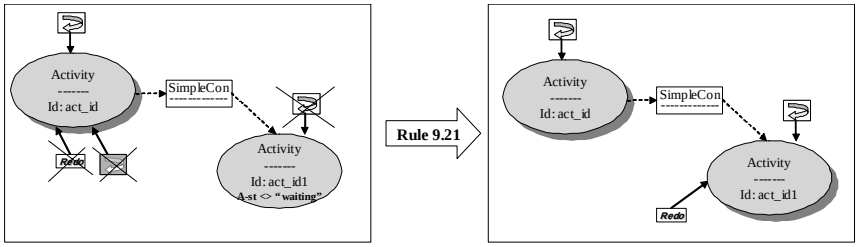


A necessidade de bloqueio (B) se explica porque a regra que insere Redo em atividades sem Redo (9.15) seria aplicada continuamente para sub-atividades, Gerando infinitas versões.

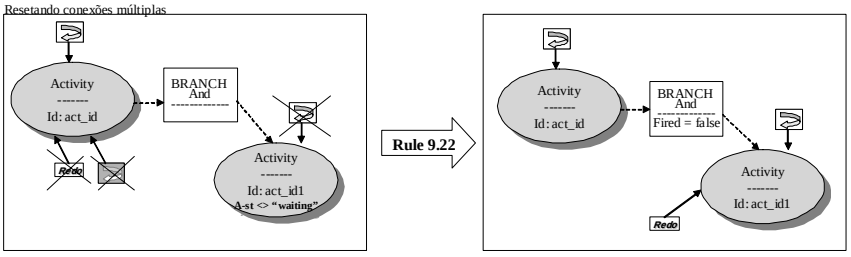


Condição: Todas atividades tem redo e todas as conexões Multiplas tem fired = false.
Uso de Set Nodes do Progres: Todas as atividades estao representadas na regra
Se todas as atividades do processo temos simbolos indicados entao a regra eh aplicada

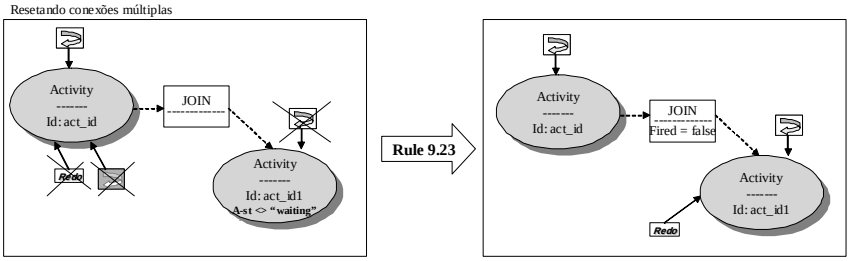




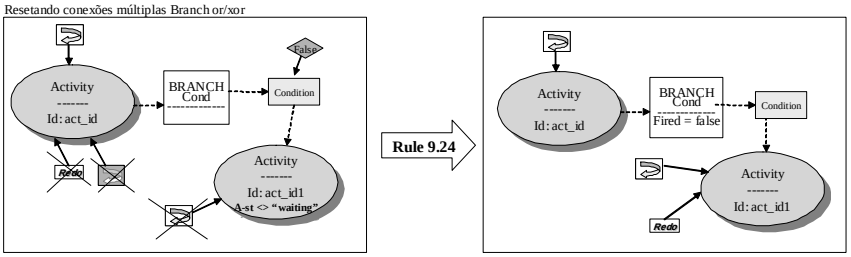
Resetando conexões múltiplas



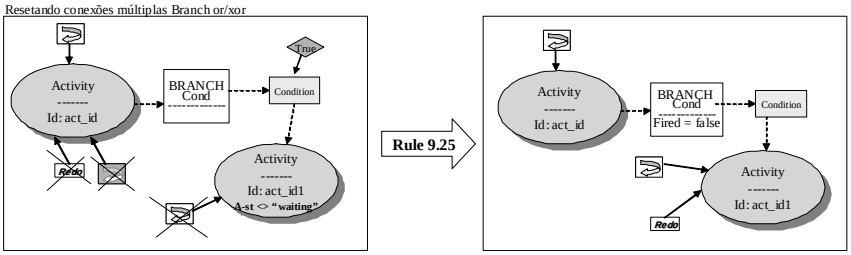
Resetando conexões múltiplas



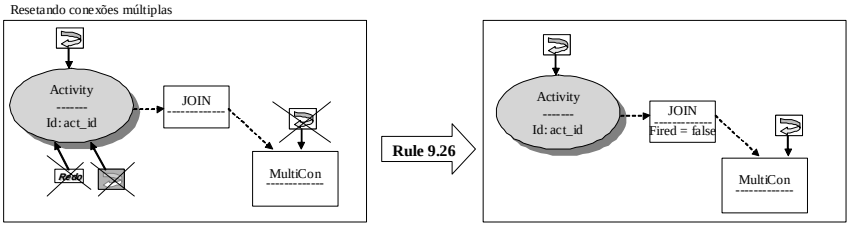
Resetando conexões múltiplas Branch or/xor



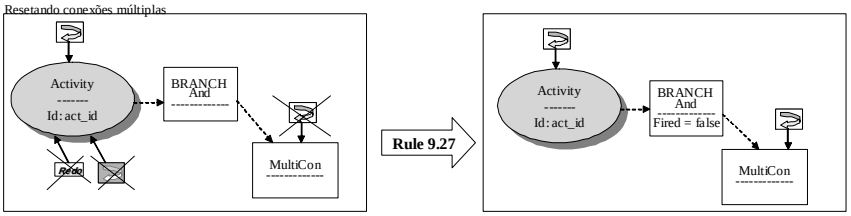
Resetando conexões múltiplas Branch or/xor



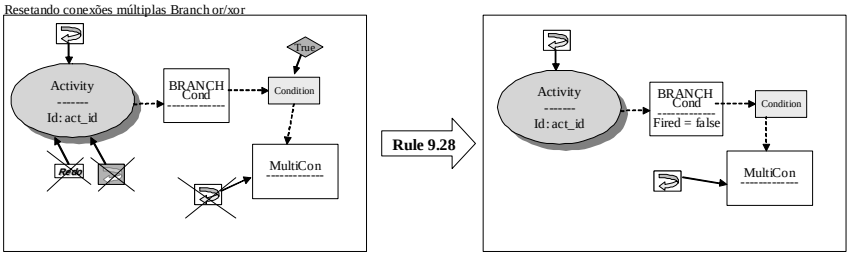
Resetando conexões múltiplas



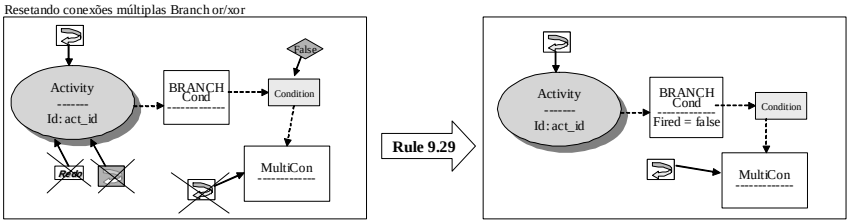
Resetando conexões múltiplas



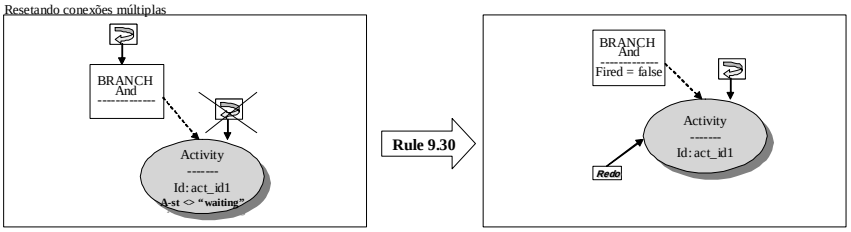
Resetando conexões múltiplas Branch or/xor



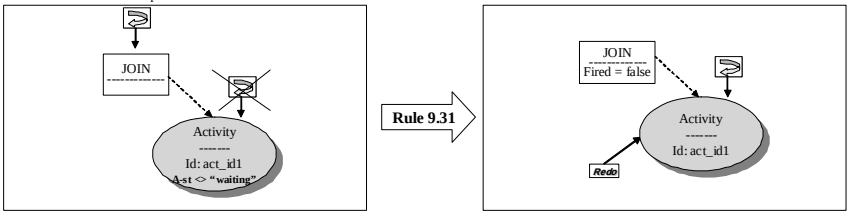
Resetando conexões múltiplas Branch or/xor



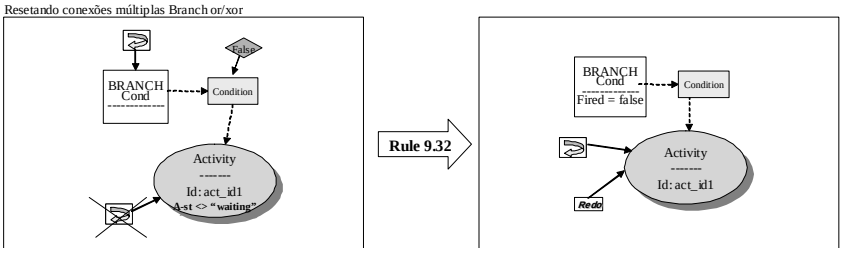
Resetando conexões múltiplas



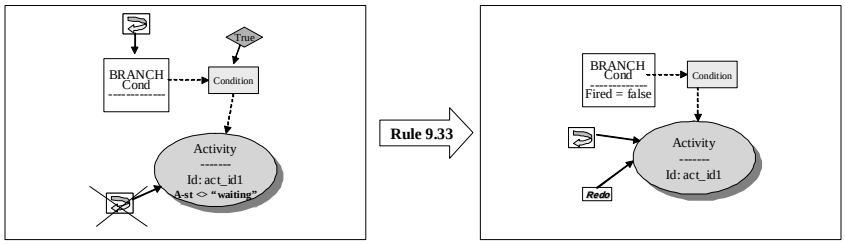
Resetando conexões múltiplas



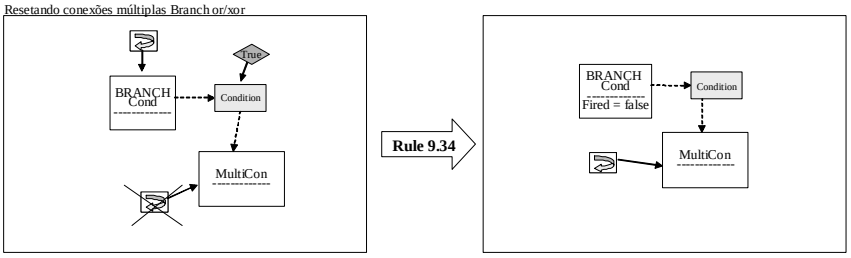
Resetando conexões múltiplas Branch or/xor



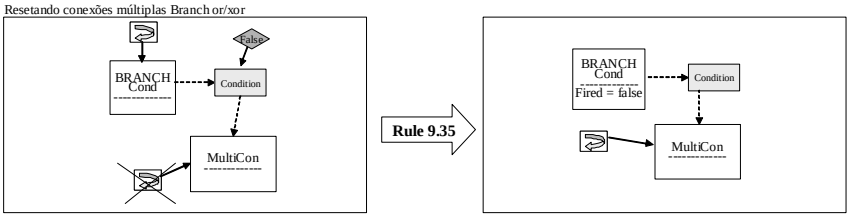
Resetando conexões múltiplas Branch or/xor



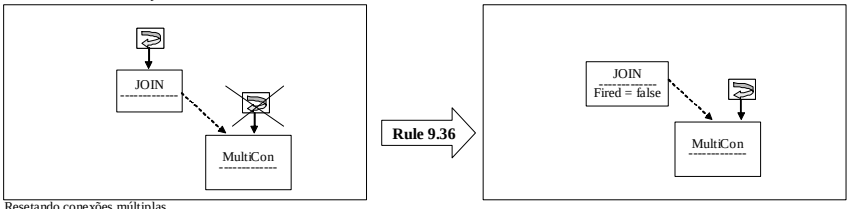
Resetando conexões múltiplas Branch or/xor



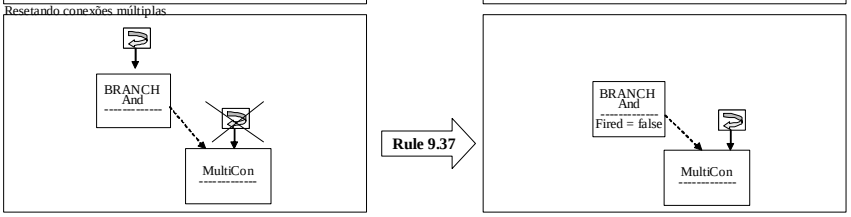
Resetando conexões múltiplas Branch or/xor



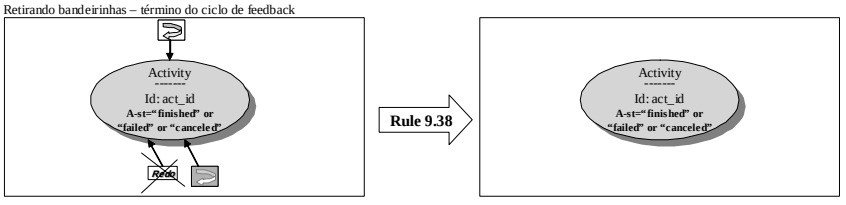
Resetando conexões múltiplas



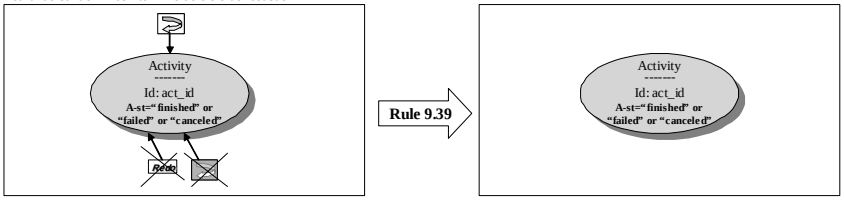
Resetando conexões múltiplas



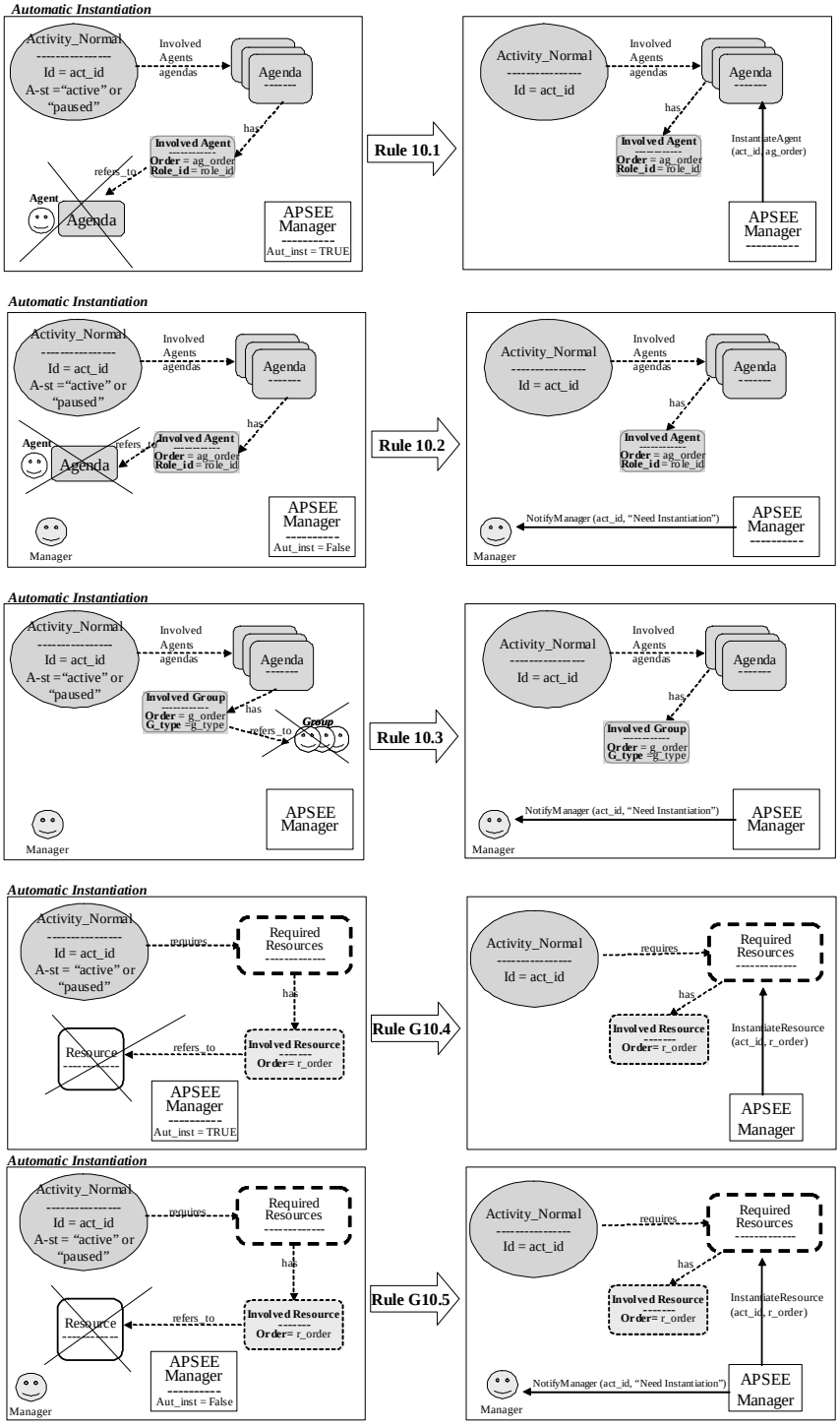
Retirando bandeirinhas – término do ciclo de feedback



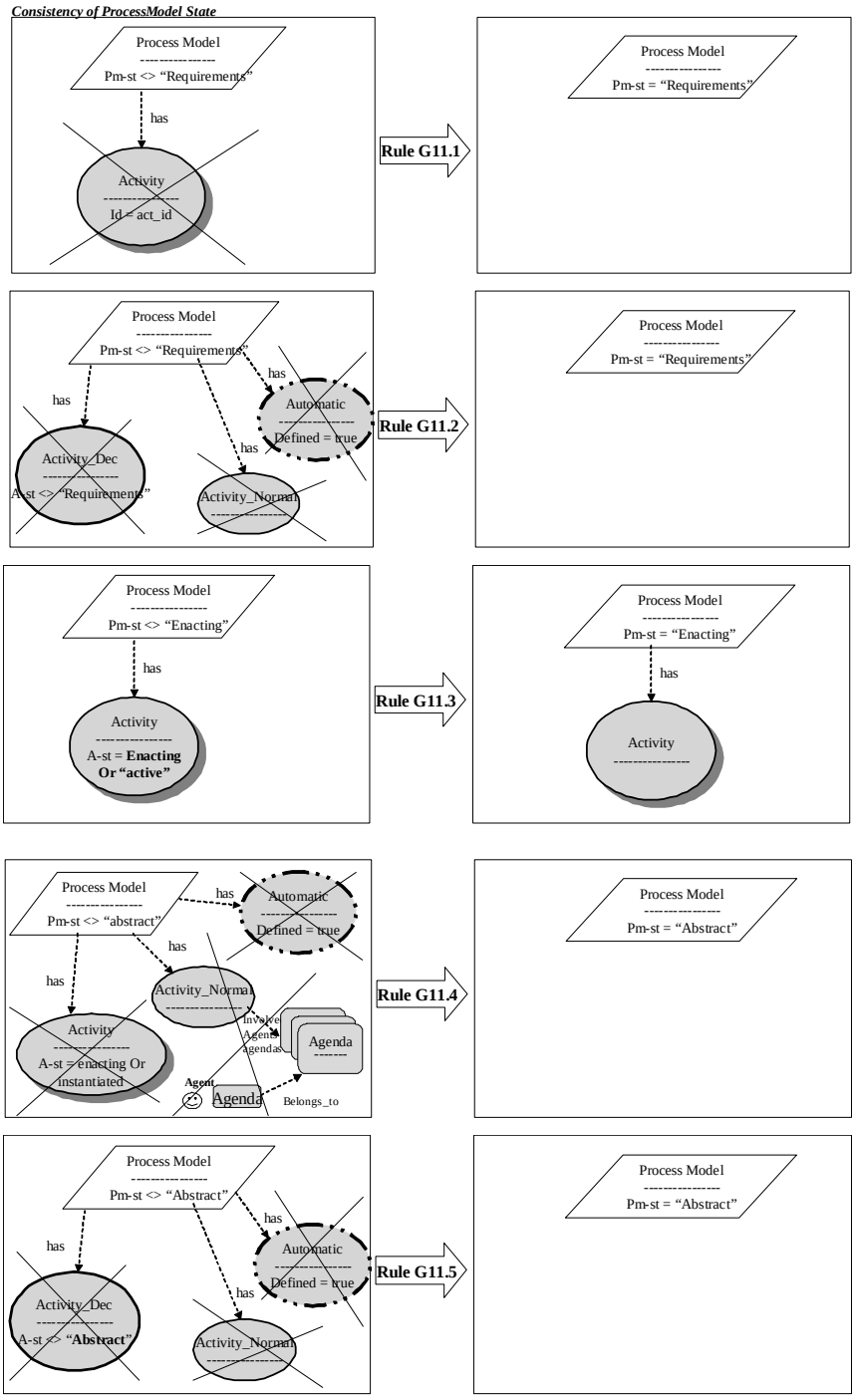
Retirando bandeirinhas - término do ciclo de feedback

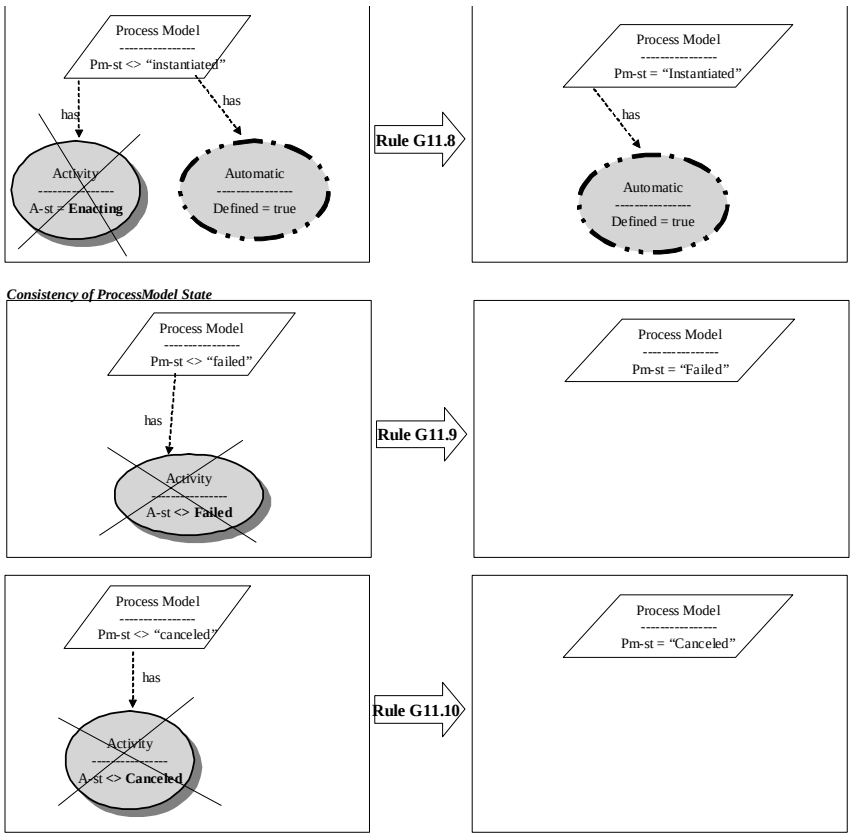


C.10. Regras para Instanciação de processos durante execução



C.11. Regras para Definição do estado de modelos de processo




Process Model
Pm-st <> “instantiated”
has
has
Activity
A-st = Enacting
Automatic
Defined = true
Rule G11.8
Process Model
Pm-st = “Instantiated”
has
Automatic
Defined = true
Consistency of ProcessModel State
Process Model
Pm-st <> “failed”
has
Activity
A-st <> Failed
Rule G11.9
Process Model
Pm-st = “Failed”
Process Model
Pm-st <> “canceled”
has
Activity
A-st <> Canceled
Rule G11.10
Process Model
Pm-st = “Canceled”

APÊNDICE D Especificação Algébrica para Políticas de Instanciação

D.1. Etapas para Instanciação de um recurso

As operações a seguir consideram que uma atividade de um processo requer um tipo específico de recurso (*req_type*) em determinada quantidade (*amount_needed*, se o recurso é consumível) e que possui datas de início e fim (*begin, end*) planejadas.

D.1.1 Etapa 1 - Obter lista inicial de recursos compatíveis

A operação do ato Apsee *initial_resources_list* obtém uma lista de identificadores de recursos que pertencem a um tipo (*req_type*) e estão disponíveis em um período. Esta operação solicita que o Ato Resources retorne a lista de recursos e envia uma mensagem ICS a este fornecendo os argumentos necessários.

Ato APSEE

initial_resources_list: APSEE, String, longue, longue, real → ListofString
initial_resources_list(Apsee, req_type, begin, end, amount_needed) =
= **ICS**(RESOURCES, get_needed_resources, get_resources*(apsee)*,
<*get_APSEE-Types*(apsee), req_type, begin, end, amount_needed>)

A operação do Ato Resources que obtém a lista de identificadores de recursos disponíveis é *get_needed_resources*. Esta operação verifica em cada recurso existente no mapeamento *resources* se o recurso é subtipo de req_type e está disponível (através de chamada à operação *is_available*). O id do recurso é colocado na lista resultante se essa verificação for verdadeira. A operação é recursiva para que todos os recursos existentes sejam verificados.

A operação *is_available* avalia a disponibilidade do recurso de acordo com seu tipo (exclusivo, consumível ou compartilhável). As operações *required_available* e *components_available* analisam a definição do recurso para obter a disponibilidade de seus recursos requeridos e componentes.

Ato Resources

get_needed_resources: Resources, APSEE-Types, String, longue, longue, real → listOfString
get_needed_resources(modify(id, details, resources), APSEE-Types, req_type, begin, end, amount_needed) =
If **ICS**(APSEE-TYPES, subtype_of, APSEE-Types, <select-Type-Id(Details), req_type>)
and *is_available*(modify(id, details, resources), id, begin, end, amount_needed)
Then cons(id, *get_needed_resources*(resources, req_type, begin, end, amount_needed))
Else *get_needed_resources*(resources, req_type, begin, end, amount_needed)
get_needed_resources(empty_list, _,_,_,_) = empty_list

is_available: Resources, String, Longue, Longue → Booleano
is_available(modify(id, (_, _, _, _, _, _, Requires required, Kind_Resource Exclusive exclusive), resources), id_resource, begin, end,_) =
if id_resource = id
then **ICS**(EXCLUSIVERES, no_reservation, exclusive, <begin, end>)
and ICS(EXCLUSIVERES, no_defect, exclusive)
and *required_available*(resources, required, begin, end)
and *components_available*(resources, id, begin, end)
else*is_available*(resources, id_resource, begin, end)

is_available(modify(id, (_,_,_,_,_,_, Requires required, Kind_Resource Shareable shareable), resources), id_resource, begin, end,_) =
if id_resource = id
then **ICS**(SHAREABLERES, available, shareable)
and *required_available*(resources, required, begin, end)
and *components_available*(resources, id, begin, end)
else*is_available*(resources, id_resource, begin, end,_)

is_available(modify(id, (_,_,_,_,_,_, Requires required, Kind_Resource Consumable consumable), resources), id_resource, begin, end, amount_needed) =
if id_resource = id
then **ICS**(CONSUMABLERES, available, consumable,<amount_needed>)
and *required_available*(resources, required, begin, end)
and *components_available*(resources, id, begin, end)
else*is_available*(resources, id_resource, begin, end, amount_needed)

is_available(empty_mapping, _,_,_) = false

** verifica se um determinado recurso está disponível para um período

D.1.2 Etapa 2 - Obter lista ordenada de políticas aplicáveis

Para a escolha da política a ser aplicada para o tipo requerido é gerada uma lista ordenada das políticas aplicáveis, priorizando as locais da atividade e colocando por último as da organização. Para ordenar as políticas compatíveis com o tipo requerido é necessário navegar na estrutura do modelo de processo para obter as políticas habilitadas no nível correto.

A operação principal desta etapa é *Get_ordered_InstR_policies*. Ela recebe o id da atividade e do processo que se quer instanciar, o tipo requerido e retona uma lista com as políticas ordenadas primeiro para o framento da atividade, depois para a organização e por último coloca a palavra "default" indicando que se nenhuma política for aplicável, então o algoritmo default deve ser acionado.

Ato Apsee

Get_ordered_InstR_policies: APSEE, String, String, String → Listof String
Get_ordered_InstR_policies(apsee, req_type, proc_id, act_id)
= *Get_ordered_InstR_policies_fragment*(apsee, req_type, proc_id, *get_activity_path*(act_id))
^
Order_inst_Policies(apsee, req_type,
Get_instR_Policies_organization(apsee),"resources")
^
cons("default", empty-list)

**Rastreia as políticas habilitadas na atividade, no processo e na organização e ordena-as colocando a palavra "default" como último item. (principal função da etapa2)
** get_activity_path recebe uma String com o id da atividade (no formato id_proc.id_act1.id_act2) e retorna uma lista com os endereços da atividade ([id_proc, id_act1, id_act2]

A operação a seguir ordena as políticas pela sua distância com relação ao tipo requerido de recurso.

Order_inst_policies: apsee, String, listofString, String → listofString
Order_inst_policies(apsee, req_type, policies, instpolicy_type)
= *Order*(*policy_distance_list*(apsee, req_type, policies, instpolicy_type),2, "inc")

** ordena as políticas de acordo com a distância do tipo requerido.

A operação a seguir obtém uma lista onde cada elemento contém o id da política e a distância dessa política para o tipo requerido. A distância é obtida através de chamada à função get_policy_distance.

policy_distance_list: apsee, String, listofString, String → PolicyDistList
Policy_distance_list(apsee, req_type, policies, instpolicy_type) =
if *compatible_policy*(apsee, head(policies), req_type, instpolicy_type)
then cons((Pol head(policies), Dist *Get_policy_distance* (apsee, head(policies),
req_type, instpolicy_type)),
policy_distance_list(apsee, req_type, policies, instpolicy_type)
else *policy_distance_list*(apsee, req_type, policies, instpolicy_type)

policy_distance_list(_,_,empty-set,_) = empty-list

**PolicyDistList = Lista de record (Pol String, Dist inteiro)

A operação a seguir obtém a lista ordenada de políticas aplicáveis ao processo em questão onde se encontra a atividade a ser instanciada. É feito um caminhamento na estrutura do processo para obter primeiro as políticas habilitadas na atividade-folha, depois as políticas habilitadas no processo (atividade decomposta) que contém essa atividade, desta forma, subindo de nível até chegar nas políticas habilitadas no processo de software.

Get_ordered_instRPolicies_fragment: Apsee, String, String, listofString → listofString
Get_ordered_instRPolicies_fragment(apsee, req_type, proc_id, ini_path ^ cons(act, empty-list))
= *Order_Inst_Policies*(apsee, req_type, *get_instRpolicies_activity*(
ICS(PROCESSES, get_activities_from_processes,
get_processes(apsee), <proc_id>), ini_path ^ cons(act, empty-list)), "resources")
^ *Order_Inst_Policies*(apsee, req_type, *get_instRpolicies_fragment*(
ICS(PROCESSES, get_processModel, *get_processes*(apsee), <proc_id>),
ini_path ^ cons(act, empty-list)), "resources")
^ *Get_ordered_instRPolicies_fragment*(apsee, req_type, proc_id, ini_path)

Get_ordered_instRPolicies_fragment(_, _, _, empty-list) = empty_list

**obtém lista ordenada de políticas para o processo em questão, percorre a árvore do modelo de processo e retorna da folha para a raíz as políticas de instanciação de recursos habilitadas.

D.1.3 Etapa 3 - Verificar Condição da Política

Tendo obtido uma lista de políticas de instanciação aplicáveis, é feita a verificação se a condição para a aplicação da primeira política é verdadeira através do campo *Conditions.* A operação que verifica se uma condição é verdadeira para uma atividade já foi definida em (REIS, 2002f).

Ato Apsee

(já definida)
verify_conditions: Apsee, Condition, String, String → boolean
chamada: verify_conditions(apsee, *get_condition_InstRpolicy*(apsee, policy_id), proc_id, act_id)

D.1.4 Etapa 4 - Aplicar critérios de restrição da política escolhida

Os critérios de restrição podem ter parâmetros. Na política cada critério de restrição possui um identificador do método a ser chamado mais uma lista de parâmetros. Os parâmetros podem ser: integer, real, String, booleano ou um operador relacional (<,>,=,<=,>=,<>). A classe PolCriteria é mostrada na FIGURA A 22.

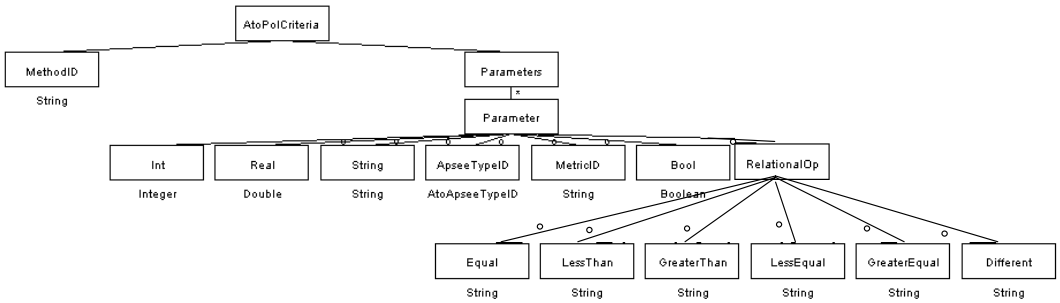


FIGURA A 22 - Classe *PolCriteria*(extraída de (REIS, 2002f)).

A principal operação desta etapa (*Restrict_Resources*) recebe uma lista de identificadores de recursos (vindos da etapa 1) e restringe esses recursos aplicando os métodos da lista de métodos da política.

Ato Apsee

Restrict_Resources: apsee, listofString, list → listofString
Restrict_resources(apsee, list_resources, list_methods)
= *restrict_resources* (apsee,
apply_restrict_method(apsee, list_resources, head(list_methods)),
tail(list_methods))

restrict_resources(_, list_resources, empty-list) = list_resources
restrict_resources(_, empty-list, _) = empty-list

** esta função aplica todos os critérios de restrição na ordem fornecida e retorna um novo conjunto de identificadores de recursos.

Para aplicar cada método de restrição foi criada a operação *apply_restrict_method* que recebe uma lista de recursos e um método com seus parâmetros e interpreta a aplicação do método. A especificação da função é apresentada para cada método de restrição possível (da Tabela 4.1).

apply_restrict_method: Apsee, listofString, PolCriteria → ListofString
apply_restrict_method(apsee, cons(id_resource, list_resources),
(Method_id **"Belongs_to"**, Parameters par))
= **If** **ICS** (APSEE-TYPES, super_type_of, select-APSEE-Types(apsee),
<head(par), **ICS**(RESOURCES, get_type_of_resource, *get_resources*(apsee),
<*get_belongs_to_resource_id*(apsee, id_resource), "resources>))
then cons(id_resource,
apply_restrict_method(apsee, list_resources,
(Method_id "Belongs_to", Parameters par)))
else *apply_restrict_method*(apsee, list_resources,
(Method_id "Belongs_to", Parameters par))
** obtém uma nova lista de recursos a partir da aplicação de um método
** O método belongs_to verifica se os recursos pertencem a recursos do tipo dado como parâmetro.

apply_restrict_method(apsee, cons(id_resource, list_resources),

```
                        (Method_id "NotBelongs_to", Parameters par))
=   If      ICS (APSEE-TYPES, super_type_of, select-APSEE-Types(apsee),
            <head(par),     ICS(RESOURCES, get_type_of_resource, get_resources(apsee),
                            <get_belongs_to_resource_id(apsee, id_resource), "resources>))
    then    apply_restrict_method(apsee, list_resources,
                            (Method_id "NotBelongs_to", Parameters par))
    elsecons(id_resource, apply_restrict_method(apsee, list_resources,
                                (Method_id "NotBelongs_to", Parameters par)))
** NotBelongs_to é o inverso de belongs_to

apply_restrict_method(apsee, cons(id_resource, list_resources),
                        (Method_id "NoRequirements", _))
=   If      ICS(RESOURCES, get_requires, get_resources(apsee), < id_resource>) = empty-set
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                                (Method_id "NoRequirements", _)))
    else    apply_restrict_method(apsee, list_resources, (Method_id "NoRequirements", _)))

** NoRequirements significa que o recurso não requer nenhum outro recurso para ser alocado.

apply_restrict_method(apsee, cons(id_resource, list_resources),
                        (Method_id "Requires", Parameters par))
=   If      head(par) ∈ (ICS(RESOURCES, get_types_of_resources, get_resources(apsee),
                    <ICS(RESOURCES, get_requires, get_resources(apsee), < id_resource>)>))
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                                (Method_id "Requires", Parameters par)))
    else    apply_restrict_method(apsee, list_resources,
                                        (Method_id "Requires", Parameters par)))

apply_restrict_method(apsee, cons(id_resource, list_resources),
                        (Method_id "NotRequires", Parameters par))
=   If      head(par) ∉ (ICS(RESOURCES, get_types_of_resources, get_resources(apsee),
                    <ICS(RESOURCES, get_requires, get_resources(apsee), < id_resource>)>))
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                                (Method_id "Requires", Parameters par)))
    else    apply_restrict_method(apsee, list_resources,
                                        (Method_id "Requires", Parameters par)))

** NotRequires: o recurso só permanece na lista se ele não requer recursos do tipo dado como parâmetro
apply_restrict_method(apsee, cons(id_resource, list_resources),
    (Method_id "GetMetric",
                        Parameters cons(id_metric, cons(op_rel, cons(value, empty-list))))
=   if      compare(get_metric_resource(apsee, id_metric, id_resource), op_rel, value)
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                (Method_id "GetMetric",
                Parameters cons(id_metric, cons(op_rel, cons(value, empty-list))))))
    else    apply_restrict_method(apsee, list_resources,
            (Method_id "GetMetric",
            Parameters cons(id_metric, cons(op_rel, cons(value, empty-list)))))
** GetMetric: o recurso só permanece na lista se ele tiver um valor para a métrica id_metric que atenda a comparacao
(op_rel) com o valor dado como parâmetro

apply_restrict_method(apsee, cons(id_resource, list_resources),
    (Method_id "Cost", Parameters cons(op_rel, cons(value, empty-list)))
=   if      compare(get_cost_resource(apsee, id_resource),    op_rel, value)
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
            (Method_id "Cost", Parameters cons(op_rel, cons(value, empty-list))))
    else    apply_restrict_method(apsee, list_resources,
            (Method_id "Cost", Parameters cons(op_rel, cons(value, empty-list))))

** Cost: o recurso só permanece na lista se ele tiver um valor de custo que atenda a comparacao (op_rel) com o valor
dado como parâmetro
```

```
apply_restrict_method(apsee, cons(id_resource, list_resources),
    (Method_id "MTBF", Parameters cons(op_rel, cons(value, cons(time-unit, empty-list))))
=   if      compare(get_MTBF_value_resource(apsee, id_resource, time_unit), op_rel, value)
    then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                (Method_id "MTBF",
                Parameters cons(op_rel, cons(value, cons(time-unit, empty-list)))))
    else    apply_restrict_method(apsee, list_resources,
            (Method_id "MTBF",
                Parameters cons(op_rel, cons(value, cons(time-unit, empty-list)))))
```

** MTBF: o recurso só permanece na lista se ele tiver um valor de mtbf que atenda a comparacao (op_rel) com o valor dado como parâmetro. O valor do mtbf a ser comparado é convertido dentro da operacao get_MTBF_value_resource para a unidade de tempo desejada.

```
apply_restrict_method(apsee, cons(id_resource, list_resources),
                    (Method_id "Consumable_New", _))
=   if          ICS(APSEE-TYPES, super_type_of, select-APSEE-Types(apsee),
                "Consumable",
                ICS(RESOURCES, get_type_of_resource, get_resources(apsee), <id_resource>),
                "Resources")
    then        if      ICS(RESOURCES, consumable_new, get_resources(apsee), <id_resource>)
                then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                            (Method_id "Consumable_New", _))
                else    apply_restrict_method(apsee, list_resources,
                            (Method_id "Consumable_New", _))
    else        cons(id_resource, apply_restrict_method(apsee, list_resources,
                            (Method_id "Consumable_New", _))
```

** Consumable_New: Este critério se aplica apenas aos recursos do tipo consumable. Estes permanecem na lista caso nunca tenham sido utilizados. Se o recurso em questão não é do tipo consumable, então ele permanece na lista.

```
apply_restrict_method(apsee, cons(id_resource, list_resources),
    (Method_id "Max_PercentUsed, Parameters cons(value, empty-list))
=   if          ICS(APSEE-TYPES, super_type_of, select-APSEE-Types(apsee),
                "Consumable",
                ICS(RESOURCES, get_type_of_resource, get_resources(apsee), <id_resource>),
                "Resources")
    then        if      ICS(RESOURCES, max_percent_used, get_resources(apsee),
                        <id_resource, value>)
                then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                            (Method_id "Max_PercentUsed, Parameters cons(value, empty-list))
                else    apply_restrict_method(apsee, list_resources,
                            (Method_id "Max_PercentUsed, Parameters cons(value, empty-list))
    else        cons(id_resource, apply_restrict_method(apsee, list_resources,
                            (Method_id "Max_PercentUsed, Parameters cons(value, empty-list))
```

** Max_PercentUsed: Apenas permanecem os recursos consumíveis que foram utilizados até um máximo percentual calculado dentro da operação max_percent_used em Resources

```
apply_restrict_method(apsee, cons(id_resource, list_resources),
    (Method_id "Amount_Available, Parameters cons(op_rel, cons(value, empty-list)))
=   if          ICS(APSEE-TYPES, super_type_of, select-APSEE-Types(apsee),
                "Consumable",
                ICS(RESOURCES, get_type_of_resource, get_resources(apsee), <id_resource>),
                "Resources")
    then        if      compare(ICS(RESOURCES, get_amount_available,
                            get_resources(apsee), <id_resource>), op_rel, value)
                then    cons(id_resource, apply_restrict_method(apsee, list_resources,
                        (Method_id "Amount_Available,
                            Parameters cons(op_rel, cons(value, empty-list)))
                else    apply_restrict_method(apsee, list_resources,
                        (Method_id "Amount_Available,
                            Parameters cons(op_rel, cons(value, empty-list)))
```

```
else        cons(id_resource, apply_restrict_method(apsee, list_resources,
                    (Method_id "Amount_Available",
                        Parameters cons(op_rel, cons(value, empty-list)))
```

** Amount_Available: Apenas permanecem os recursos consumíveis que que possuem uma determinada quantidade disponível

D.1.5 Etapa 5 - Aplicar critério de ordenação nos recursos selecionados

A quinta e última etapa utiliza o critério fornecido no campo *OrderBy* para ordenar o conjunto resultante da quarta função. O valor utilizado para ordenar é inserido no resultado desta etapa com o objetivo de facilitar a decisão.

A operação *apply_orderBy_Method* recebe uma lista de recursos (possivelmente da etapa 4) e o método de ordenação escolhido na política. Esta operação chama a operação get_list_orderby_value que obtém o valor correto para ordenação de cada recurso e o resultado é ordenado com um critério (crescente, decrescente) dependente do método escolhido).

```
apply_orderBy_method: Apsee, listofString, PolCriteria → ListSuggestions
apply_orderBy_method(apsee, list_resources, PolCriteria)
=  order(get_list_orderby_value(apsee, list_resources, PolCriteria),
          2, get_ordination_criteria_method(select-Method_ID(PolCriteria)))
```

** order é uma função de ordenação definida na etapa 2 que recebe uma lista, a posição do valor a ser ordenado e o critério de ordenação (inc ou dec) e devolve a lista ordenada

```
get_list_orderby_value: apsee, listofString, PolCriteria → ListSuggestions
get_list_orderby_value(apsee, cons(id_resource, list_resources), PolCriteria)
=  if      select-Method_id(PolCriteria) = "Low Cost" or select-Method_id(PolCriteria) = "High Cost"
   then    cons((RES_ID id_resource, ORDER_BY get_cost_resource(apsee, id_resource)),
           get_list_orderby_value(apsee, list_resources, PolCriteria))
   else    if   select-Method_id(PolCriteria) = "Low Mtbf" or
                select-Method_id(PolCriteria) = "High Mtbf"
           then    cons((RES_ID id_resource,
                   ORDER_BY get_mtbf_value_resource(apsee, id_resource, "")),
                   get_list_orderby_value(apsee, list_resources, PolCriteria))
           elseif      select-Method_id(PolCriteria) = "Metric_higher" or
                       select-Method_id(PolCriteria) = "Metric_lower"
                   then    cons((RES_ID id_resource,
                           ORDER_BY get_metric_resource(apsee,
                                   head(select-Parameters(PolCriteria)), id_resource)),
                           get_list_orderby_value(apsee, list_resources, PolCriteria))
                   else    cons((RES_ID id_resource, ORDER_BY
                           get_cost_resource(apsee, id_resource)),
                           get_list_orderby_value(apsee, list_resources, PolCriteria))
```

** Esta função obtém, para cada recurso, os valores a serem utilizados para ordenação dependendo do método de ordenação escolhido na política. Se nenhum método foi definido para ordenação, então o custo do recurso é obtido.

D.2. Operações Auxiliares

Existem diversas operações necessárias para a conclusão das etapas anteriores da instanciação. Essas operações foram classificadas como auxiliares e podem ser encontradas na íntegra em (LIMA REIS, 2001d).

APÊNDICE E Primitivas para Condições Lógicas

Tabela E.1 - Funções relacionadas com elementos estáticos do tipo *Activity*

Métodos relacionados com o tipo Activity - Parte Estática				
Tipo APSEE	**Nome da função**	**Parâmetros**	**Tipo do Resultado**	**Descrição**
Activity	Number_required_agents	-	Integer	Obtém a quantidade de agents requeridos para uma atividade
Activity	get_feedback_connection	-	Set of Connection	Obtém o conjunto de conexões do tipo *feedback* que uma atividade possui
Activity	get_feedback_connection	Activity	Connection	Obtém a conexão de tipo *feedback* entre a atividade destinatária e a passada por parâmetro (se houver)
Activity	get_input_artifacts	-	Set of Artifact	Obtém o conjunto de artefatos consumidos por uma atividade
Activity	get_output_artifacts	-	Set of Artifact	Obtém o conjunto de artefatos produzidos por uma atividade
Activity	get_successors	-	Set of Activity	Obtém a atividade sucessora (precedente) (se houver)
Activity	get_transformed_artifacts	-	Set of Artifact	Obtém o conjunto de artefatos transformados por uma atividade
Activity	get_type	-	String	Obtém o tipo da atividade na hierarquia de tipos
Activity	Number_of_sucessors	-	Integer	Obtém a quantidade de atividades sucessoras para uma atividade específica (diretamente conectadas por *simple connections* à atividade em questão).
Activity	get_roles	-	Set of Role	Obtém o conjunto de papéis que atuam em uma atividade
Activity	get_groups_types	-	Set of GroupType	Obtém o conjunto de tipos de grupos que uma atividade possui
Activity	get_types_required_abilities	-	Set of AbilityType	Obtém o conjunto de tipos de habilidades que uma atividade exige
Activity	get_types_input_artifacts	-	Set of ArtifactType	Obtém o conjunto de tipos dos artefatos de entrada de uma atividade
Activity	get_types_output_artifact	-	Set of ArtifactType	Obtém o conjunto de tipos dos artefatos de

	s			saída de uma atividade
Activity	get_estimated_duration	-	Integer	Obtém a duração estimada para uma atividade em dias
Activity	get_types_of_resouces	-	Set of ResourceType	Obtém um conjunto de tipos dos recursos de uma atividade
Activity	get_reqPeople	-	ReqPeople	Obtém informações sobre as habilidades requeridas por uma atividade

Tabela E.2 - Funções relacionadas com elementos Instanciados do tipo *Activity*

Métodos relacionados com o tipo Activity - Parte Instanciada				
Tipo APSEE	**Nome da função**	**Parâmetros**	**Tipo do Resultado**	**Descrição**
Activity	get_agents	-	Set of Agent	Obtém o conjunto de agentes que atuam em uma atividade
Activity	get_groups	-	Set of Group	Obtém um conjunto de grupos de uma atividade
Activity	get_allocated_resources	-	Set of Resource	Obtém um conjunto de recursos alocados para uma atividade
Activity	get_input_artifacts	-	Set of artifact	Obtém um conjunto de entrada de uma atividade
Activity	get_output_artifacts	-	Set of artifact	Obtém um conjunto de artefatos de saída
Activity	get_estimated_start_date	-	Date	Obtém a data estimada para o início de uma atividade
Activity	is_performed_by_agents	Set of Agent	Boolean	Obtém um resultado True se os agentes citados são responsáveis pela atividade
Activity	is_performed_by_groups	Set of Group	Boolean	Obtém um resultado True se o grupo citado é o responsável pela atividade

Tabela E.3 - Funções relacionadas com elementos em execução do tipo *Activity*.

Tipo APSEE	**Nome da função**	**Parâmetros**	**Tipo do Resultado**	**Descrição**
PARTE DINAMICA "Enacting"				
Activity	is_enacting	-	Boolean	Obtém true se a atividade estiver em execução
Activity	is_late_to_start	-	Boolean	Obtém true se o inicio da atividade estiver atrasada
Activity	is_late_to_finish	-	Boolean	Obtém true se a atividade estiver atrasada
Activity	is_paused	-	Boolean	Obtém true se a atividade estiver em pausa
Activity	is_ready	-	Boolean	Obtém true se a atividade estiver pronta
Activity	is_waiting	-	Boolean	Obtém true se a atividade estiver esperando
Acitivity	is_active	-	Boolean	Obtém true se a atividade estiver ativa

Activity	get_state	-	String	Obtém o estado da atividade (Ready, Paused, Waiting, Failed, Cancelled ou Active)
Activity	is_repeating	-	Boolean	Obtém True se a atividade estiver sendo re-executada.
CONEXÕES ENTRE ATIVIDADES				
Activity	get_sucessors	-	Set of Connection	Obtém as conexões do tipo *simple_connetion* para uma atividade
Activity	get_sucessors_of_type	Set of ConnectionType	Connection	Obtém as conexões de uma atividade que sejam de conjunto de tipos especificados
Acitivity	get_feedback_conn_from	Activity	Set of Connection	Obtém o conjunto de conexões do tipo *feedback* que uma atividade possui
Acitivty	get_feedback_conn_to	Activity	Set of Connection	Obtém um conjunto de conexões do tipo feedback que possui com destino a atividade definida
CONEXÕES MÚLTIPLAS				
Activity	get_multiple_conn_to	Mult_conn_type (branch, join, any) Typedep (end-start, start-start, end-end, any)	Set of Connection	Obtém o conjunto de conjunto de conexões múltiplas de uma atividade
Activity	get_multiple_conn_from	Mult_conn_type (branch, join, any) Typedep (end-start, start-start, end-end, any)	Set of Connection	Obtém o conjunto de conjunto de conexões múltiplas de uma atividade
CONEXÕES DE ARTEFATO				
Activity	get_artifact_conn_to	-	Set of Activity	Obtém um conjunto de atividades que recebem artefatos da atividade passada como parâmetro
Activity	get_artifact_conn_from	-	Set of Activity	Obtém um conjunto de atividades que enviam artefatos da atividade passada como parâmetro
Activity	get_artifact_conn_inout	-	Set of Artifact	Obtém um conjunto de conexões de artefatos de entrada e saída da atividade passada como parâmetro
OPERAÇÕES SOBRE CONEXÕES				
Connection	is_simple_conn	-	Boolean	Obtém true se a conexão for do tipo *simpleconnection*
Connection	is_multiple_conn	-	Boolean	Obtém true se a conxão for do tipo *multipleconnection*
Connection	is_artifact_conn	-	Boolean	Obtém true se a conexão for do tipo *artifactconnection*
Connection	get_type	-	Connection Type	Obtém o tipo de conexão (Artifact, Multiple, Simple)
CONEXÕES SIMPLES				

SimpleConn	get_from_activity	-	Activity	Obtém a atividade-origem de uma conexão
SimpleConn	get_to_activity	-	Activity	Obtém a atividade-destino de uma conexão
SimpleConn	is_feedback_connection	-	Boolean	Obtém true se a conexão é do tipo *feedback*
SimpleConn	is_sequence_connection	-	Boolean	Obtém true se a conexão é do tipo *sequence*
SimpleConn	is_sequence_end_start	-	Boolean	Obtém true se a conexão é do tipo *sequence end-start*
SimpleConn	is_sequence_start_start	-	Boolean	Obtém true se a conexão é do tipo *sequence start-start*
SimpleConn	is_sequence_end_end	-	Boolean	Obtém true se a conexão é do tipo *sequence end-end*
CONEXÕES MÚLTIPLAS				
MultipleConn	is_join	-	Boolean	Obtém true se uma conexão múltipla é do tipo *join*
MultipleConn	is_branch	-	Boolean	Obtém true se uma conexão múltipla é do tipo *branch*
MultipleConn	is_and_connection	-	Boolean	Obtém true se uma conexão múltipla é do tipo *and*
MultipleConn	is_xor_connection	-	Boolean	Obtém true se uma conexão múltipla é do tipo *xor*
MultipleConn	is_inor_conection	-	Boolean	Obtém true se uma conexão múltipla é do tipo *or*
MutipleConn	get_activity_join_from	-	Set of {Activity U MultipleConn}	Obtém um conjunto com os elementos (atividades ou conexões múltiplas) de origem de uma conexão múltipla do tipo *join*
MultipleConn	get_activity_join_to	-	Set of {Activity U MultipleConn}	Obtém um conjunto com os elementos (atividades ou conexões múltiplas) de destino de uma conexão múltipla do tipo *join*
MultipleConn	get_dependence_type	-	String	Obtém o tipo de dependência expresso em uma conexão múltipla (“End-start”, “Start-start” ou “End-end”)
CONEXÕES DE ARTEFATOS				
ArtifactConn	get_from_activity	-	Set of Activity	Obtém as atividades que produziram um artefato
ArtifactConn	get_to_activity	-	Set of Activity	Obtém as atividades que consomem um artefato
ArtifactConn	get_artifact	-	Artifact	Obtém o artefato associado a uma conexão de artefato.
OPERAÇÕES SOBRE ARTEFATOS				
Artifact	get_connections	-	Set of Connection	Obtém as conexões para um artefato
Artifact	get_type	-	ArtifactType	Obtém o tipo de um artefato
Artifact	is_produced_by	-	Set of Activity	Obtém as atividades que produziram o artefato

Artifact	is_consumed_by	-	Set of Activity	Obtém as atividades que consumiram o artefato
Artifact	is_derived_from	-	Artifact	Obtém o artefato que deu origem
Artifact	get_components	-	Set of Artifact	Obtém os artefatos componentes
Artifact	is_part_of	Artifact	Boolean	Obtém true se um artefato é componente de outro
ROLE				
Role	get_type	-	RoleType	Obtém o tipo de um cargo
Role	get_superior	-	Role	Obtém o cargo superior
Role	get_required_abilities	-	Set of Ability	Obtém as habilldades requeridas para um cargo
Role	is_subordinated_to	Role	Boolean	Obtém true se um cargo é subordinado de outro
ABILITIES				
Ability	get_type	-	AbilityType	Obtém o tipo de um objeto Ability